Offloading Raft distributed consensus protocol networking features to kernel space eBPF programs

Kellogg College
University of Oxford

A dissertation submitted for the
MSc in Software Engineering

Abstract

Distributed systems rely on computer networks and consensus algorithms to make multiple machines appear as a single, robust process. Raft is one of many consensus algorithms, designed to be robust, performant and simple. Extended Berkeley Packet Filter (eBPF) is a Linux operating system feature for developing custom programs that run on its kernel. eBPF allows for fast network packet processing, rich monitoring and an easy-to-use storage interface for sharing data across kernel and user contexts. This dissertation will focus on implementing Raft algorithm networking features in eBPF with the aim of exploring a more efficient way of communication between individual machines.

Contents

1 Introduction

This dissertation focuses on implementing Raft [1] consensus algorithm networking features using the Extended Berkeley Packet Filter (eBPF) framework [2]. This work will study Raft protocol networking requirements, the Linux operating system kernel and its features to integrate core distributed consensus features in an eBPF program.

eBPF is a sandbox environment that allows running custom programs inside the Linux kernel. eXpress Data Path (XDP) [3], one of the frameworks for eBPF, provides a high-performance network packet processing functionality, fully integrated in the Linux kernel. eBPF and its ecosystem enable building programs that take advantage of in-kernel processing performance. This project uses the Rust programming language [4] and Aya [5] — a Rust eBPF library for writing user and kernel space programs.

Raft is a leader-based consensus algorithm, where a single machine — a *leader* — is responsible for serving clients requests and maintaining authority for the cluster. It handles all log entries and coordinates with other servers when those entries can be committed locally. If a leader fails, it triggers an election process whereby other nodes — *followers* — vote among themselves to pick a leader. The majority vote wins, and the winning node — a *candidate* — assumes leader responsibilities for the cluster for the next arbitrary time period. If the previous leader returns after recovering from failure, it rejoins the cluster as a follower.

The main goal of this work is to create a working prototype for implementing distributed consensus functionality and showing performance improvements when offloading network packet processing to eBPF. Consensus algorithms heavily depend on node-to-node communication, and network packet processing forms a key part of this project. Each received network packet will trigger execution of the eBPF program, which will execute logic based on the information in the packet. To ensure overall system stability, all eBPF programs must adhere to Linux kernel verifier requirements [6] and its restrictions. These restrictions disallow unbounded loops, require special interfaces to communicate between kernel and user space code, and present a non-trivial challenge for the implementation of the project.

This dissertation will also aim to show the development process for creating an eBPF program. eBPF is actively developed and its foundation [7] "ensure[s] that the core of eBPF is well maintained and equipped with a clear roadmap and vision for the bright future ahead of eBPF." Regardless of the potential of the eBPF framework, there is no framework completely abstracting the complexity underneath, requiring new eBPF program developers to become familiar with Linux kernel code [8] and low-level system details. To achieve this, eBPF and its ecosystem will be viewed from the perspective of a developer with no previous Linux kernel and systems programming experience.

The detailed objectives of this work are shown in the table below.

Objective	Description
Objective 1	Implement core Raft algorithm networking features as eBPF program(s). These features include heartbeats, leader election and log replication.
Objective 2	Develop a working prototype which integrates eBPF features with a user space program.
Objective 3	Validate the performance characteristics of the eBPF-based Raft algorithm implementation.

Table 1.1: Thesis objectives.

This thesis has been inspired by the Electrode paper [9], published in 2023. It demonstrated a 30%+ latency and throughput improvements when offloading network features of a more complex, Paxos distributed consensus algorithm [10]. This dissertation attempts to achieve similar results by using a simpler and more widely used Raft algorithm.

2 Background

2.1 Raft

The Raft consensus algorithm was created by Diego Ongaro and John Ousterhout and is described in their paper "In Search of an Understandable Consensus Algorithm" [1]. The publication was first presented at the annual USENIX conference in 2014. It describes the state of consensus algorithms at a time, with Leslie Lamport's Paxos [10] being a standard reference for consensus algorithms, "almost synonymous with consensus" [1, p. 2], despite being difficult to understand and often impractical for building reliable systems. Paxos complexity argument was also echoed by Van Renesse and Altibunken, who conclude that their paper, "Paxos Made Moderately Complex", adds to the "long line of papers that describe the Paxos protocol, present the experience of implementing it, or make it easier to understand" [11, p. 24].

These drawbacks led to the creation of Raft consensus algorithm, a "practical system bear[ing] little resemblance to Paxos" [1, p. 3]. Its creators aimed to provide an understandable, safe and performant distributed consensus algorithm. Raft architecture is relatively simple, with a clear interaction interface and minimal states and rules that the servers must follow. As a result, the original Raft algorithm implementation contains "roughly 2000 lines of C++ code not including tests, comments or blank lines" [1, p. 13].

Over the years, the algorithm has become a popular choice for building distributed systems and has robust implementations in many programming languages [12]. It is being taught in dozens of university courses, including the popular Distributed Systems class at Massachusetts Institute of Technology, where students learn to build a fault-tolerant key-value storage system using Raft [13]. Kubernetes [14], a large open-source project, uses `etcd`[15] — a Raft-based database for storing all its critical data. Raft algorithm is used not only to build new databases (TiKV [16] and RethinkDB [17]), but also to extend existing synchronous databases, like MySQL [18] and PostgreSQL [19].

To ensure cluster and data integrity after a failure, the algorithm describes steps to maintain guaranteed safety properties. This includes a formal specification and proof [20, p. 112].

2.2 BPF and eBPF

2.2.1 Berkeley Packet Filter (BPF)

eBPF is an extended version of the original Berkley Packet Filter (BPF) which was first introduced as "The BSD Packet-Filter" [21]. The paper, published in 1993, describes a "register-based filter evaluator" which allows a more efficient network packet filtering and monitoring for BSD and BSD-based operating systems. At the time, BSD (Berkeley Software Distribution) was an open-source Unix-based operating system and would eventually become the foundation of many popular Unix distributions used to this day — FreeBSD [22], OpenBSD [23] and others.

Authors of the paper describe network packet processing challenges stemming from network tools being run as user space programs and requiring packet data to be copied between kernel and user space, significantly impacting performance. Their approach was to develop a lightweight in-kernel virtual machine ("packet filter"), designed to be protocol-independent, generalizable and fast. The resulting "programmable pseudo-machine" allowed building networking observability software such as `tcpdump` [24], a Unix-based command-line tool for real-time network packet capture and analysis.

2.2.2 Extended Berkeley Packet Filter (eBPF)

In 2013, BPF was rewritten by Alexei Starovoitov [25] to extend its functionality beyond network packet processing and became part of the Linux kernel in version 3.15 [26]. This evolution has since become known as eBPF [27]. It introduced numerous features over the previous version, now referred to as classic BPF (`cBPF`). For the remainder of the thesis, the terms eBPF and BPF will be used interchangeably, always referring to the extended version of the BPF.

Since Starovoitov's BPF rewrite, each kernel version added new features[28], but BPF has remained relatively niche. Recently, it was popularised by Brendan Gregg, an internationally renowned expert in computing performance[29]. Gregg was active in the Linux community, did many talks and wrote numerous blog posts describing methods and tools for finding performance bottlenecks in systems and devising ways of visualizing hierarchical performance data [30]. He published two books — *BPF Performance Tools* [31] and *Systems Performance* [32] — providing a set of useful BPF programs for observing system CPUs, memory, file systems and disk performance. His work was influential given the rapid growth of cloud computing and Moore's Law slowdown [33], requiring thorough understanding of Linux systems — from the CPU chips of individual servers to hypervisors running thousands of systems and virtualized applications — all crucial for maximising performance. Gregg describes the current iteration of BPF as being composed of an "instruction set, storage objects, and helper functions" [31, p.1], comparable to JavaScript, except the programs run not in the browser, but on the Linux kernel.

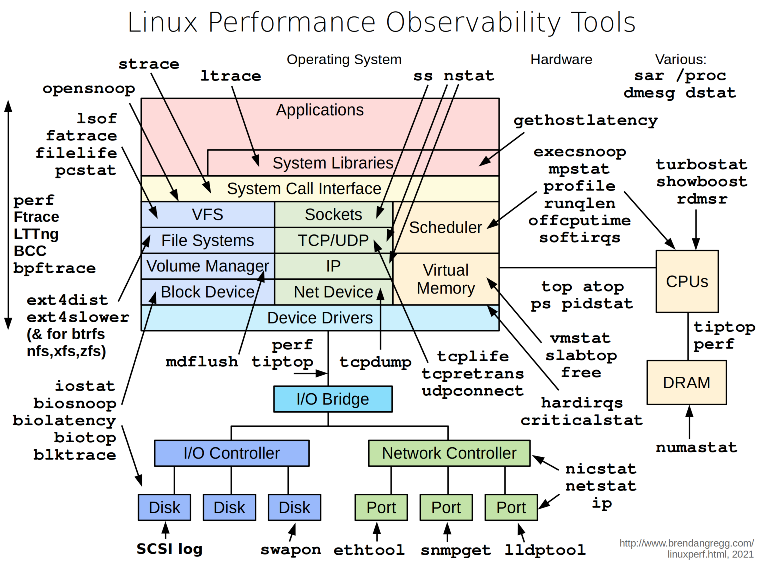


Figure 2.1: Various BPF-based performance observability tools for Linux [34].

Thanks to the widespread use of Linux as an operating system ("96.3% of the top one million web servers are running Linux" [35]), eBPF is actively developed, with features and improvements being released in almost every Linux kernel version. In 2021, a Meta, Google, Isovalent, Microsoft and Netflix as formed an eBPF Foundation [36]. Its goal was to "raise, budget and spend funds supporting the eBPF community" and fund development of "Technical Projects" [37], such as tools for kernel tracing (`bcc`, `bpftrace`), eBPF runtime in Linux kernel as well as widely actively used Software Defined Networking (SDN) projects Calico and Cilium [38].

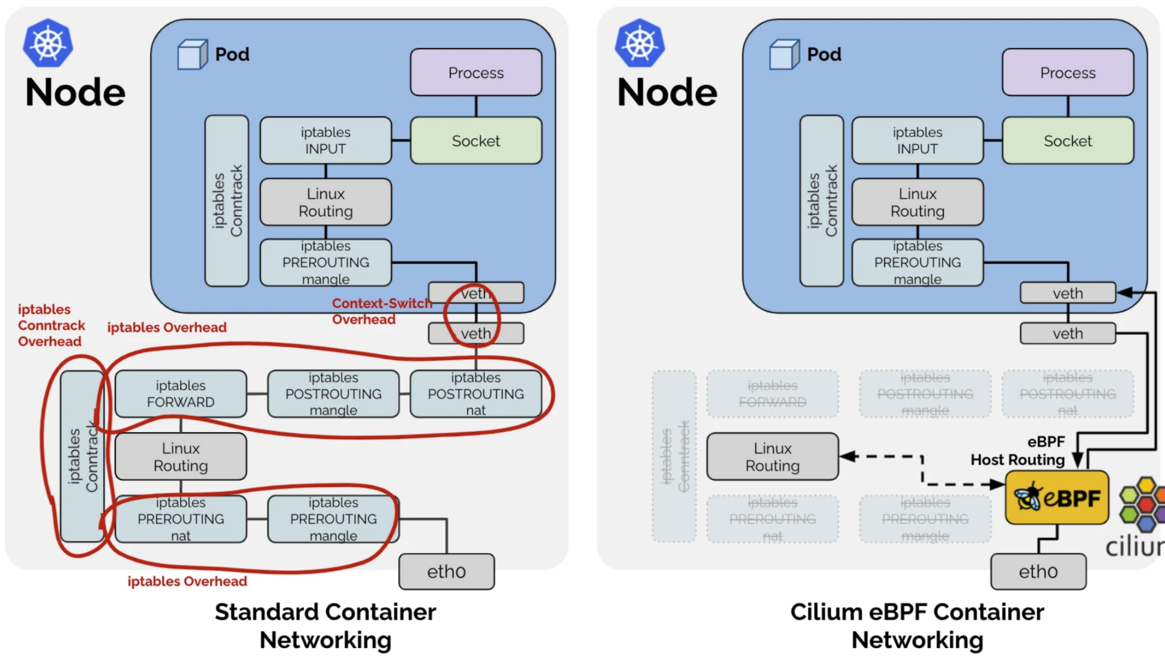


Figure 2.2: Cilium [39], a project replacing previous `iptables` network packet filtering with eBPF.

2.2.3 eBPF runtime

eBPF runtime consists of an interpreter, a Just-In-Time (JIT) compiler, various helper functions and a verifier [6]. Interpreter allows translating eBPF bytecode into executable actions and JIT compilers such as `rbpf`, `uBPF` allow translating bytecode into native machine code directly [40].

Helper functions provide a programming interface to interact with program resources within its context [41], from printing debug messages to interacting with built-in storage system [42]. Helper functions will be explored in more detail later on.

The most crucial part of the runtime is the verifier. It ensures that the bytecode loaded in the kernel adheres to eBPF runtime control flow requirements. These requirements disallow unbounded loops by constructing a direct acyclic graph (DAG) and evaluating all possible paths the program can take. The verifier needs to be able to complete its analysis within the "limits of the configured upper complexity limit" [43], therefore disallowing complex and large eBPF programs. As a result, invalid instructions and out-of-bounds memory access patterns are forbidden to ensure stability of the system.

2.2.4 Kernel modules vs eBPF

Why use eBPF programs if Linux kernel modules allow dynamic loading and unloading of custom code similar to eBPF [44]? eBPF does not directly replace kernel modules, but instead provides a safer method for extending kernel functionality without risks to system stability. Custom kernel modules may introduce a risk of a system failure if the underlying module crashes, therefore becoming a viable path only to those with significant kernel or systems development experience. Kernel modules are also tied to the specific kernel version they are built for, requiring additional maintenance overhead when upgrading systems.

Liz Rice, author of a book on eBPF, describes [45] a hypothetical example where a maliciously crafted internet packet can crash a system ("packet of death"). With eBPF, she explains, in a manner of minutes, this vulnerability can be mitigated by writing and distributing the eBPF program across the production network without waiting weeks or months for a security patch to be released and systems updated.

2.2.5 Compile Once, Run Everywhere (CO-RE)

CO-RE is an eBPF feature for compiling applications that "can run on multiple kernel versions and configurations without modifications and runtime source code compilation on the target machine" [46]. This functionality is achieved by integrating multiple components [47]:

- `BTF Type Format` [48] allows expressing type information for C (BPF) programs.
- The compiler embeds additional metadata, known as *BPF relocations*, to map recorded relocations against a number of target system relocations [49].
- BPF loader, `libbpf` [50], processes the outputs produced above and adapts the BPF program to the specifics of a system kernel.

The components described above ensure that the eBPF program will pass the verification on the target system and run correctly — the same as if it were originally compiled on the target host.

2.2.6 Dynamic and static tracing

A key distinction between eBPF and other observability tools is that eBPF is agentless [51] — i.e. it does not require installing additional software on the host to be able to extract system information. This provides a non-intrusive way of monitoring the operating system by loading eBPF programs to capture real-time system information. Tracing capabilities provided by eBPF allow capturing application data without explicitly instrumenting application code.

eBPF has two distinct ways of tracing, dynamic and static. Dynamic tracing is done by attaching special probes (`Kprobes` [52] and `Uprobes` [53]) to kernel and user space code respectively and observing system behaviour. Dynamic tracing works based on the function names which may change over the course of program lifetime and static tracing encodes stable event names using tracepoints for kernel events and user-level statistically defined tracing (USDT) configuration [54]. Both approaches allow for different ways of extracting system information and summarising recoded information using counters and gauges in response to various events in the system.

Tracing implementation is outside the scope of this thesis, instead, existing open-source eBPF tools will be used.

2.2.7 Linux modes

While eBPF support for Windows is being actively developed [55] and Rust binaries can be cross-compiled to work on different operating systems, there is a strong historical link between BPF programs and the Linux kernel, fully understanding one without the other is not possible.

All Linux programs operate in either of two modes, kernel space or user space. The purpose of kernel space is to protect the system from unauthorized access by reserving memory for itself [56, p. 23] and to shield developers from the low-level details of the underlying hardware interface. The kernel runs in a privileged mode, and it has unrestricted access to the hardware of the system [57, p. 53], free to perform process scheduling, allocate memory to programs and manage devices. At the same time, it provides an interface for user-space programs to perform privileged operations, known as “system calls” or “syscalls” [58, p. 217]. When a user-space application needs to execute a privileged instruction, it includes a special “trap” instruction which allows it to escalate the privilege level and perform any of the privileged work required [57, p. 83]. Once finished, the “return-from-trap” instruction returns back to user-space with the original program privileges.

Michael Kerrisk provides [56, p. 234] an example which illustrates performance impact when using syscalls. He says, a single write of 1000 bytes is preferable over 1000 writes of a single byte, even as the kernel performs the same number of disk accesses. This is because the kernel must trap the call, verify its arguments and transfer data between user and kernel space. Context switch is the overhead that McCanne and Jacobson identified when building the original version of BPF. Despite the Linux kernel consistently introducing features which improve performance, eBPF remains a powerful method for tasks where context switch is identified as a bottleneck.

2.2.8 Aya eBPF library

This project uses the Aya eBPF library for Rust [59]. The library was created in 2021 and is still actively maintained with a vibrant Discord community and a website which includes numerous tutorials on using the library [5]. The library is used to develop `Blixt` [60], an experimental load balancer for Kubernetes, `bpfman` [61], a tool for simplifying the deployment and administration of eBPF programs, and other smaller projects.

Aya library uses foreign function interface (FFI) bindings to interoperate with C code, allowing integration of eBPF functionality into Rust applications without relying on `libbpf` and `bcc` libraries [62]. Aya implements CO-RE and large part of eBPF features. Aya BPF environment requires using `unsafe`{} blocks for dereferencing raw pointer objects and forbids standard library via `#![no_std]` to use a “platform-agnostic subset of the `std` crate which makes no assumptions about the system the program will run on” [63]. As a result, general purpose Rust data structures — vectors and linked lists — cannot be used when writing kernel space code.

The main motivator for using Rust over other languages is safety guarantees and memory management, important when writing both kernel and user space programs.

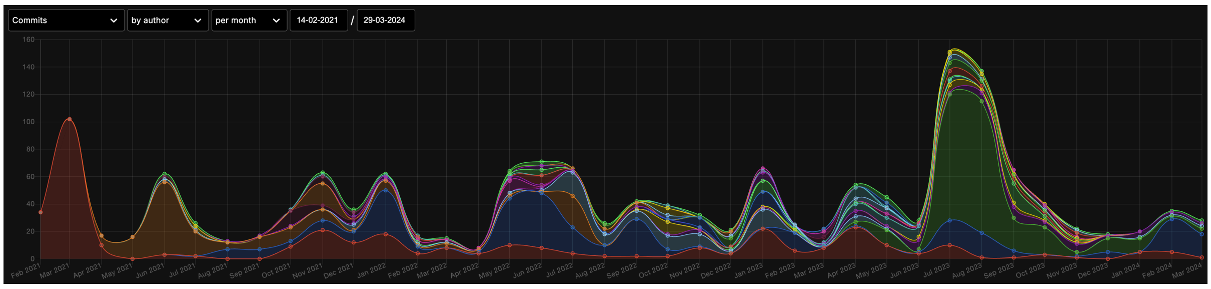


Figure 2.3: `aya-rs/aya` git repostitory commits between 2021 and 2024, by author (via gitstat.com).

2.2.9 Aya project layout

Aya allows generating a project template for an eBPF application with multiple components. The template includes dependencies as well as sample user and kernel space code to create a valid, but empty, eBPF program.

The folder structure used for this project is shown below:

```
raft/
  raft/          # user space code
  raft-ebpf/     # kernel space code
  raft-commmon/  # shared data structures, variables, etc.
```

Programs can be built and started with a single `xtask` [64] command. `xtask` is a library for building and executing multiple Rust libraries, referred to as “crates” in the Rust ecosystem.

When starting a user space program, a compiled BPF object file is included at compile-time and loaded at run-time [65]. After the user space program initialises and attaches the BPF program, the rest of the user space control flow continues. Console log output, by default, includes prefixed messages from both, kernel and user space code to allow distinguishing between events emitted by each program.

2.2.10 Development environment

The development environment consists of multiple virtual machines, provisioned using OrbStack [66] on an Apple Mac M1 Max laptop. All virtual machines run the same operating system, Debian 12 (`Bookworm`, aarch64) with Kernel version 6.7.10.

One of the virtual machines is pre-installed with relevant Rust dependencies to compile kernel and user space programs (see Appendix A). Other virtual machines share only application source code and compiled binary target directory, taking advantage of eBPF CO-RE feature and not require any additional dependency installation.

All virtual machines run the same binary to simulate a working Raft cluster, where any machine can be in any of possible Raft node states. Nodes persist their configuration in memory and after stopping the program, all configuration is lost.

The application development loop consists of writing and compiling the program on a single machine and starting the resulting binary on other machines.

2.2.11 Networking

eXpress Data Path (XDP) and Traffic Control (TC)

XDP [3] and TC [67] are eBPF frameworks for processing network packets. XDP supports processing incoming packets, whereas TC processes both incoming and outgoing packets. XDP was introduced in Linux kernel version 4.8 [68] and this thesis will use XDP due to its performance characteristics [69].

XDP achieves its performance due to its implementation at the network device driver level [70], allowing it to run earlier in the Linux networking stack. XDP provides the ability to manipulate incoming network packets and its general uses include network routing, load balancing and distributed denial-of-service (DDoS) mitigation [3, p. 54].

eBPF is an event-based system, and XDP programs are triggered by incoming network packets. XDP allows intercepting all packets on a given network interface. Network packet data can be read, rewritten and packet redirected before it reaches the kernel networking stack. Packet data includes access to protocol, source and destination ports, packet length, checksums, and payload.

To control packet flow, XDP defines multiple return codes:

```c
enum xdp_action {
 XDP_ABORTED = 0,
 XDP_DROP,
 XDP_PASS,
 XDP_TX,
 XDP_REDIRECT,
};
```

Listing 2.1: XDP return codes (Linux kernel v6.5.7) [71].

The return codes allow controlling flow based on the information contained in the network packet headers. `XDP_PASS` allows the packet to continue through the operating system networking stack. `XDP_DROP` drops the packet, resulting in a "`Connection Refused`" message for TCP network traffic. `XDP_TX` allows sending the packet back through the same interface as it came in. `XDP_REDIRECT` allows redirecting the packet to a different interface, and `XDP_ABORTED` drops the packet and raises an exception.

Tools like `tcpdump` [24] are run at higher layers of the networking stack and are unable to capture packets returned via `XDP_TX` and `XDP_REDIRECT`. However, there are eBPF-based observability tools like `pwru` [72] and `xdpdump` [73], the latter of which does not work with Aya as some BTF functionality is not yet supported.

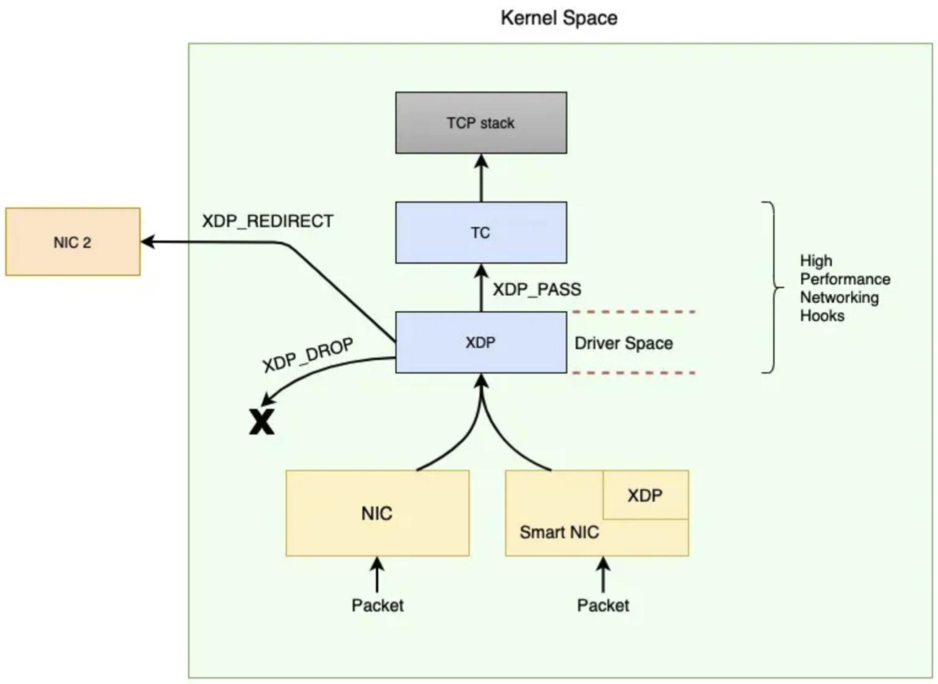


Figure 2.4: High-level overview of eXpress Data Path (XDP) [74].

Aya documentation includes a tutorial [75] which demonstrates a simple XDP program for blocking network traffic. The program defines a list of "blocked" IP addresses in user space and uses a BPF data structure to access the data from the kernel program. For each received packet, it parses the headers, extracts source IP information, looks it up in the data structure and drops the packet if there is a match. The packet processing foundation demonstrated in this tutorial will form a foundation for subsequent work on implementing various Raft protocol features.

XDP also supports offloading packet processing to physical network interface card, allowing to use hardware-based acceleration for executing eBPF code and reducing host CPU usage [76]. It is a topic for highly specialised tasks and an area of active research [77], outside the scope of this project.

IP packet modification

Raft algorithm implementation in eBPF will rely on parsing and modifying incoming packets. An example flow is described below:

1. Read header data to extract relevant information (source IP address, port).
2. Based on the extracted information, decide what to do with the packet.
   (a) Allow packet to continue (via `XDP_PASS`).
   (b) Modify packet data and header fields.
      i. Recalculate packet checksum.
      ii. Send modified packet back through the network interface using `XPD_TX`.

Internet checksum [78] is a crucial element for detecting corruption in packet traffic and its recalculation is important after modifying packet data, as packets with invalid checksums may be discarded at any point. During early stages of development using Cloud-based virtual machines in Google Cloud Platform, invalid checksums led to dropped packets and checksum validation could not be disabled. At the time of the writing, there is no community library for recalculating checksums within the eBPF program [79]. Since no community solution has shown to work consistently across multiple environments, the development environment was modified to use locally provisioned Linux virtual machines with disabled checksum validation.

```
$ ethtool -K eth0 rx-checksumming off tx-checksumming off
Actual changes:
tx-checksum-ip-generic: off
...
tx-checksum-sctp: off
rx-checksum: off
```

Listing 2.2: Disabling ethernet interface checksums using `ethtool`.

Disabling checksum validation works around the limitations and shortens the development loop for the project. The community recommends using virtual ethernet (`veth`) interface pairs when developing XDP programs [80]. Similarly, packet modification must include correct MAC addresses of the Ethernet header, as "[s]ome NICs [Network Interface Cards] especially virtualized ones, will drop outgoing packets if the source MAC does not match the configured MAC of the NIC" [81].

Internet protocols use big-endian order for consistent data representation across different hardware [82]. In big-endian order, the most significant byte comes first. To ensure correct correct header modifications, Rust provides helper functions to convert from little-endian and big-endian format and vice versa. Parsing the packet, retrieving the source IP address (`198.19.249.40`) and representing it in the 32-bit unsigned integer in both formats produces the following output:

```
[INFO  raft_main] Source IP (original - big-endian): 687412166
[INFO  raft_main] Source IP (converted to little-endian): 3323197736
[INFO  raft_main] Source IP (converted to big-endian): 687412166
```

2.2.12 Raft eBPF program architecture

Raft eBPF programs will build upon layers of functionality introduced over the next chapters. The main focus will be on implementing Raft heartbeats and leader election functionality. The scope of this project will not include the data safety and consistency guarantees provided by the complete algorithm.

The resulting program will be a single identical binary, deployed on all nodes, running kernel and user space code simultaneously, serving, initiating and responding to requests over the network.

3 Implementing heartbeats

3.1 Failure detection

Failure detection plays an important role in building fault-tolerant distributed systems. Failures are unpredictable, can occur at any moment and may exhibit non-deterministic behaviour, resulting in a system that seemingly responds to some requests while ignoring others [83, p. 275]. In this case, remaining non-faulty members must decide what steps to limit reconcile discrepancies and minimise disruption when serving client requests.

Steen and Tannenbaum [84, p. 464] describe two methods of detecting failures. First method is to actively send heartbeat messages ("Are you alive?") and expect an answer. The second is to wait for other nodes to initiate communication. Fundamentally, both methods rely on using timeouts to control the time window during which a successful communication must happen. Authors provide on a formal definition of "eventually perfect failure detectors", in which "`P` will suspect another process `Q` to have crashed after `t` time units have elapsed and still `Q` did not respond to `P`'s probe". If `Q` eventually recovers, comes back online and sends a message to `P`, it is no longer considered suspected, and the operations can resume. Otherwise, it is presumed to be unreachable.

At the same time, exclusive reliance on probes can be problematic. Very short timeouts can destabilise the system when nodes are saturated and are unable to respond within the required timeout. Timeout can also be adjusted dynamically, based on the ew network conditions. Dynamic timeouts may lead to unpredictable behaviour as the timeout keeps increasing. Similarly, a connectivity loss between two nodes `A` and `B` might have no impact on communication between `A` and `C`, and `B` and `C`. As a result, `A` can unilaterally decide the fate of `B` and remove it from cluster membership, despite `B`'s ability to successfully communicate with all other cluster members. Problems like these can be solved using a gossip-based approach, where the state of all systems is disseminated through the network, allowing to make more informed decisions about cluster membership in an event of a failure. Gossip-based protocols "maintain relaxed consistency requirements amongst a very large group of nodes" [85] and have been used in distributed databases, Cassandra [86] and Riak [87].

Failure detection is the first indicator which may trigger other processes - cluster membership changes, data rebalancing or leader elections.

3.2 `AppendEntries`

In the Raft paper, failure detection probes are referred to as "heartbeats". Heartbeats allow the leader to "maintain [its] authority and prevent new elections" [1, p. 6]. From the follower's perspective, absence of any heartbeats from the leader in a given time window triggers a new leader election process among the remaining nodes.

Leader node makes use of `AppendEntries` remote procedure calls (RPCs) to send heartbeats to the followers. RPCs can either carry an empty or a non-empty payload. RPCs with an empty payload are explicit heartbeats. RPCs with a non-empty payload are implicit and serve a dual purpose - they function as a data replication mechanism. Followers must respond to all `AppendEntries` RPCs to maintain their membership in the cluster. In the absence of incoming messages, followers initiate a new leader election process.

Heartbeats are also used when servicing client requests. To avoid returning stale data when a leader is suddenly deposed, Raft requires having the "leader exchange heartbeat messages with a majority of the cluster before responding to read-only requests." [1, p. 13] As a result, Raft cluster deployments work best in low-latency networks, like those within a single data centre. Raft cluster sizes are recommended to be in odd sizes - 3, 5, 7 nodes allowing to tolerate 1, 2 and 3 node failures respectively [88]:

$$\text{Fault toleration} = \text{node count} - \left\lceil \frac{\text{node count}}{2} \right\rceil$$

Message exchange over a network is a key feature of Raft heartbeat mechanism and the remainder of this chapter will focus on implementing heartbeat mechanism using eBPF. Additional Raft features - leader election and recovery from failures - will be described in Chapters 4 and 5.

3.3 Raft heartbeats using XDP

eBPF programs can be attached to various parts of the system. One such part, a *hook point* [89], is eXpress Data Path (XDP). XDP allows attaching an eBPF program to control the flow of incoming (ingress) network packets. XDP has been designed for fast packet processing and uses a lightweight kernel data structure (`xdp_md`), optimised for speed [90]. An XDP program gets invoked for each networking packet and can only access data only for a current packet. It is therefore not suitable for processing complete network requests as those are often spread over multiple packets, complicating access to payload data contained in each `AppendEntries` RPC.

At the same time, each incoming packet contains network metadata — source and destination addresses, protocol and packet length. This metadata can be used to decode (or encode) application-specific information and to design an eBPF program which allows differentiating between different types of traffic - heartbeat requests and responses, leader election and other Raft features. XDP packet processing actions (`XDP_DROP`, `XDP_TX` and `XDP_PASS`) allow controlling the direction of the packet and build an initial implementation for eBPF-based heartbeat mechanism.

Open Systems Interconnection model [91] defines two commonly used network protocols for data transmission at the transport layer ("Layer 4") - Transmission Control Protocol (TCP) and Unified Datagram Protocol (UDP). TCP provides a reliable and ordered stream of data between applications on a IP network. TCP is widely used and is the protocol of choice for all mature Raft algorithm implementations. TCP implements a three-way handshake (SYN, SYN-ACK, ACK) [92] to establish a connection between two hosts, allowing reliable message exchange. Since even an empty `AppendEntries` RPC may be split across multiple packets and connection establishment would result in multiple packets, there is little advantage in using TCP to implement Raft heartbeat mechanism in eBPF. Compared to TCP, UDP does not require connection establishment, it is connectionless [93]. It comes at a cost of reliability - packets can be lost or may not be deduplicated, leading to reliability issues and potential network congestion [94]. UDP is often used for applications where some data packet loss can be tolerated - video streaming, Voice over Internet Protocol (VoIP) and others.

UDP, therefore, is suitable for implemnting low-bandwidth communication protocol over a low-latency internal network. A small UDP with an empty payload can be a substitue for an empty `AppendEntries` RPC, compatible with Raft algorithm specification. A single UDP packet can be processed by XDP program without ever receiving the packet in user space.

A UDP-based heartbeat mechanism can be implemented as follows:

1. `user space`: Leader sends a single UDP packet to the follower.
2. `kernel space`: Follower receives the UDP packet, swaps source and destination addresses, and sends it back to the leader via `XDP_TX`.
3. `kernel space`: Leader receives the rewritten UDP packet, updates eBPF data structure and uses `XDP_DROP` to discard the packet to prevent further processing.

Implementing UDP-based heartbeats is more complex than TCP equivalent, due to communication logic being split across user and kernel space. eBPF programs can only respond to events, not initiate them, so initial packet must be generated in user space. At the same time, once a packet is sent, XDP allows for complete control over packet direction, enabling faster packet processing, compared to operating system networking stack or Transmission Control (TC) programs in eBPF. Speed is crucial when detecting unresponsive followers and quickly converging to a desired state after any cluster event.

3.4 Reading and updating packet metadata

Reading network packet metadata within an eBPF program requires accessing data at specific memory offsets. Individual headers are retrieved using the `ptr_at` helper function, which returns a cast of a specific type at an offset within the XDP context (`ctx`). The example below shows accessing Ethernet header at offset `0`, and IPv4 header at offset `14`, defined as `EthHdr::LEN`.

```
let ethhdr: *const EthHdr = ptr_at(&ctx, 0)?;
let ipv4hdr: *const Ipv4Hdr = ptr_at(&ctx, EthHdr::LEN)?;
let source_addr = u32::from_be(*ipv4hdr.src_addr);
```

Once header data is parsed, individual fields within the headers can be updated. The example below shows updating packet destination address to `198.19.249.160` IP address, represented as an integer in big-endian byte order:

```
let client_ip = 2700678086 as u32;
(*ipv4hdr).dst_addr = client_ip;
```

Header value modification allows for packet redirection when used with XDP_TX and XDP_REDIRECT return codes. As mentioned in Section 2.2.11, packet content rewrite requires header checksum recalculation to avoid packets being discarded by the network interface. In addition to previously disabled checksum validation on the interface, XDP program will explicitly be attached in socket buffer mode (SKB_MODE), which provides a generic, non-hardware optimized implementation of XDP [95]:

```
prog.attach("eth0", XdpFlags::SKB_MODE)?;
```

3.5 Data storage

Kernel space programs need to be aware of all cluster members and membership cannot be static - nodes may be removed or added at any time. Furthremore, data needs to be populated at runtime without recompiling or restarting the application. eBPF provides a feature to allow sharing data between user and kernel space - BPF maps [42]. BPF maps enable efficient data exchange between different parts of the program and are important for successful Raft algorithm implementation in eBPF.

There are more than a dozen different BPF storage types. All data structures have fixed size, some support per-CPU and Least Recently Used (LRU) variants. Some of the most common data structures are:

- BPF_MAP_TYPE_ARRAY - generic array storage with index lookup.
- BPF_MAP_TYPE_HASH - generic hash map storage with key lookup.
- BPF_MAP_TYPE_QUEUE and BPF_MAP_TYPE_STACK - *first-in, first-out* (FIFO) and *last-in, first-out* (LIFO) storage for accessing items in a specific order.

This project will make active use of BPF_MAP_TYPE_ARRAY and BPF_MAP_TYPE_HASH map types for storing data.

In Aya library, BPF maps are defined by specifying a map outer attribute and defining a static reference for a specific BPF map type with required type information.

```
#[map]
static MY_MAP: HashMap<i32, i32> = HashMap::with_max_entries(1024, 0);
```

In addition to primitive types - integers, strings - BPF hash maps support using composite values (structs) as both keys and values [96].

User space map initialization is done after loading the eBPF program bytecode:

```
let myMap: HashMap<_, u32, u32> = HashMap::try_from(
    bpf.take_map("MY_MAP").unwrap(),
)?;
```

After maps are loaded, user and kernel space programs can perform lookup, add, update and deletions on a shared datastructure. BPF maps are optimised for lookup speed, with atomic element update operations and the kernel managing memory allocation [97]. Using BPF maps for Raft heartbeat implementation is a practical choice which allows associating lookup keys with additional information tracking the state of each follower.

3.5.1 Heartbeat maps

The first hash map, `FOLLOWERS`, contains data about individual followers and the last sent heartbeats. Map type signature is `u32,u64` with Internet Protocol version 4 (IPv4) address as the key. IPv4 address `10.0.0.1` becomes `167772161` by converting four dot-delimited eight-bit octets to a 32-bit integer. To consistently represent IPv4 addresses in the system, user-space code uses Rust library package (`std::net::Ipv4Addr`) to perform necessary conversions to match data format used in kernel space. The key value (`u64`) is initialised to `0` when a key is inserted. Each time the leader sends a new heartbeat, it records the timestamp and updates the map from the user space.

The second map, `HEARTBEAT_LATENCY`, uses the same type signature and contains data about each follower's latency.

3.5.2 Time representation

Accurate time measurements play an important role in evaluating the health of each follower. Lack of standard library support in BPF programs prevent using rich ecosystem of time conversion libraries [98]. However, BPF provides various helper functions [41], one of which is `bpf_ktime_get_ns()`, returning a monotonically increasing integer representing the time elapsed since system boot - useful for providing reliable timestamps and measuring short time intervals [99]. Boot time is not incremented while the system is in suspended (sleep) state, but this limitation does not apply to this project.

User space programs have many libraries for reading Linux system time to read eBPF-equivalent values:

```
let ts_nanos = Duration::from(
    clock_gettime(nix::time::ClockId::CLOCK_MONOTONIC).unwrap()
).as_nanos();
```

Consistent time representations allows measuring follower latency from both user and kernel space programs. eBPF programs have limited facility to perform complex calculations, however, BPF maps allow delegating data transformation tasks to user space.

All Raft heartbeat latency measurements will be recorded on a single node, eliminating any potential clock drift introduced by using data from other nodes.

3.6 Heartbeat flow

3.6.1 Leader: sending the heartbeat packet

Raft eBPF heartbeat mechanism requires all machines to run the same kernel and user space code, with one machine being manually designated as the leader by populating its `FOLLOWERS` map with IP addresses of other followers. Map is used by a background user space process which periodically iterates through map keys (IP addresses), sends a single UDP packet to each address and records the current timestamp. Other nodes - those with local `FOLLOWERS` map empty - are configured to be idle and only respond to incoming requests via eBPF.

The UDP packet is sent to network port `27001` on each machine. It has been selected as port for receiving and parsing the heartbeats from the leader. User space program is not configured to receive traffic on any UDP ports - packet is received only by the eBPF program.

3.6.2 Follower: receiving the heartbeat packet

eBPF program is configured to parse each incoming packet and pattern match UDP traffic on specific ports. If no matches exist, the network packet is allowed to continue via `XDP_PASS` return code.

When follower receives a UDP packet on port `27001`, it swaps source and destination addresses in the packet and changes the destination port to `27000`, a port used for sending heartbeat responses. `XDP_TX` return code is used to send the packet back through the receiving network interface.

3.6.3 Leader: receiving the heartbeat packet

The rewritten packet - heartbeat response from a follower - is then received by eBPF program on port 27000. eBPF program looks up packet source IP address in the FOLLOWERS map, retrieves the original timestamp and calculates time elapsed time since the original heartbeat request was sent. The resulting data point is written to the HEARTBEAT_LATENCY map, with follower's IP address as the key. Timing information is also printed to the console:

```
[user] heartbeat req: Sent to 198.19.249.4:27000 (3323197736)
[kernel] heartbeat resp: Got packet from 3323197736 (latency 277842 ns)
```

After receiving the packet and calculating the latency, user space program has access to latency information for all followers. Data in HEARTBEAT_LATENCY can be used to create additional user space data structures to track and expose follower latency over longer periods. This information can eventually be used to detect a non-responsive follower and remove it from the cluster membership.

Current heartbeat implementation does not have a mechanism to confirm that traffic originates from a valid Raft cluster member. Additional enchancments for Raft eBPF communication protocol will be explored in Sections 4 and 5.

3.7 Map data injection at runtime

Injecting map data at runtime allows to avoid defining follower IP addresses at compile-time. It also allows managing cluster membership changes dynamically. Given that a significant portion of the cluster configuration resides in maps, having an interface to read and modify map values simplifies operational tasks.

This feature is implemented by building an HTTP API that enables interaction with data in BPF maps. tokio-rs/axum [100] is a popular and extensible web framework for Rust, built on top of Tokio (asynchronous Rust runtime), Tower (a library of networking abstractions) and Hyper (Rust HTTP implementation).

The API uses `JSON` as a data interchange format and exposes multiple endpoints with appropriate HTTP methods:

- `/follower/add [POST]`
- `/follower/delete [POST]`
- `/followers/list [GET]`

Command below demonstrates process for adding a new follower:

```
$ curl -X POST -d '{"ip": "5.6.7.8"}' 1.2.3.4:8888/follower/add
```

Endpoints `/follower/add` and `/follower/delete` validate incoming payloads, extract relevant data and modify the `FOLLOWERS` map initialised at startup. The background task, responsible for sending heartbeats, continuously sends heartbeat messages to all IP addresses found in the map.

Endpoint `/followers/list` reads the `HEARTBEAT_LATENCY` and returns latency for configured followers in a human-readable format.

Follower data is persisted in memory and is lost once the program terminates. Modern distributed systems use automated service discovery mechanisms to manage node information dynamically. For the purposes of this project, IP addresses will be populated manually.

3.7.1 Rust ownership model

Rust documentation defines *ownership* as a feature which "enables Rust to make memory safety guarantees without needing a garbage collector" [101]. Ownership is enforced through the use of *borrow checker* [102] and *lifetimes* [103], components that allow the compiler to track and validate the value usage across different functions with the aim of preventing dangling references. This method enforces a strict development workflow, compared to other system programming languages like C or Go.

Rust compiler requires adjustments when using BFP maps as long-lived objects, shared across different parts of the user space application. Rust standard library does not support creating globally mutable maps [104], however, `Arc` [105] library enables creating a mutable reference which can be shared across different functions. This requires encapsulating a BPF map reference in an `Arc` and adding it to the data structure shared between all HTTP endpoints in the `axium` router.

```
// load BPF program as bpf
// define BPF map as myMap
let myMap: HashMap<_, u32, u32> = HashMap::try_from(
    bpf.take_map("MY_MAP").unwrap(),
)?;

// define state using a custom struct
let state = AppState {
    my_map: Arc::new(Mutex::new(myMap)),
};

// define app
let app = Router::new()
        .route("/list", get(list_something))
        .route("/add", post(add_something))
        .with_state(state); // state defined above

// start app
```

`Arc` (“Atomic Reference Counted”) is a thread-safe reference counter, and it allows cloning references to a single data object without additional duplication. `Arc` “ensur[es] the data will last as long as there are any active references” [106]. `Arc` requires the use of `Mutex`, a Rust mutual exclusion primitive, to control access to data using locks. `Axum` router requires explicit `Clone` trait implementation on any data structure shared between its routes. Since `Arc` always clones references, it ensures the HTTP API has safe concurrent access to read and modify BPF map data.

User space logs capture successful BPF map modification using the HTTP API:

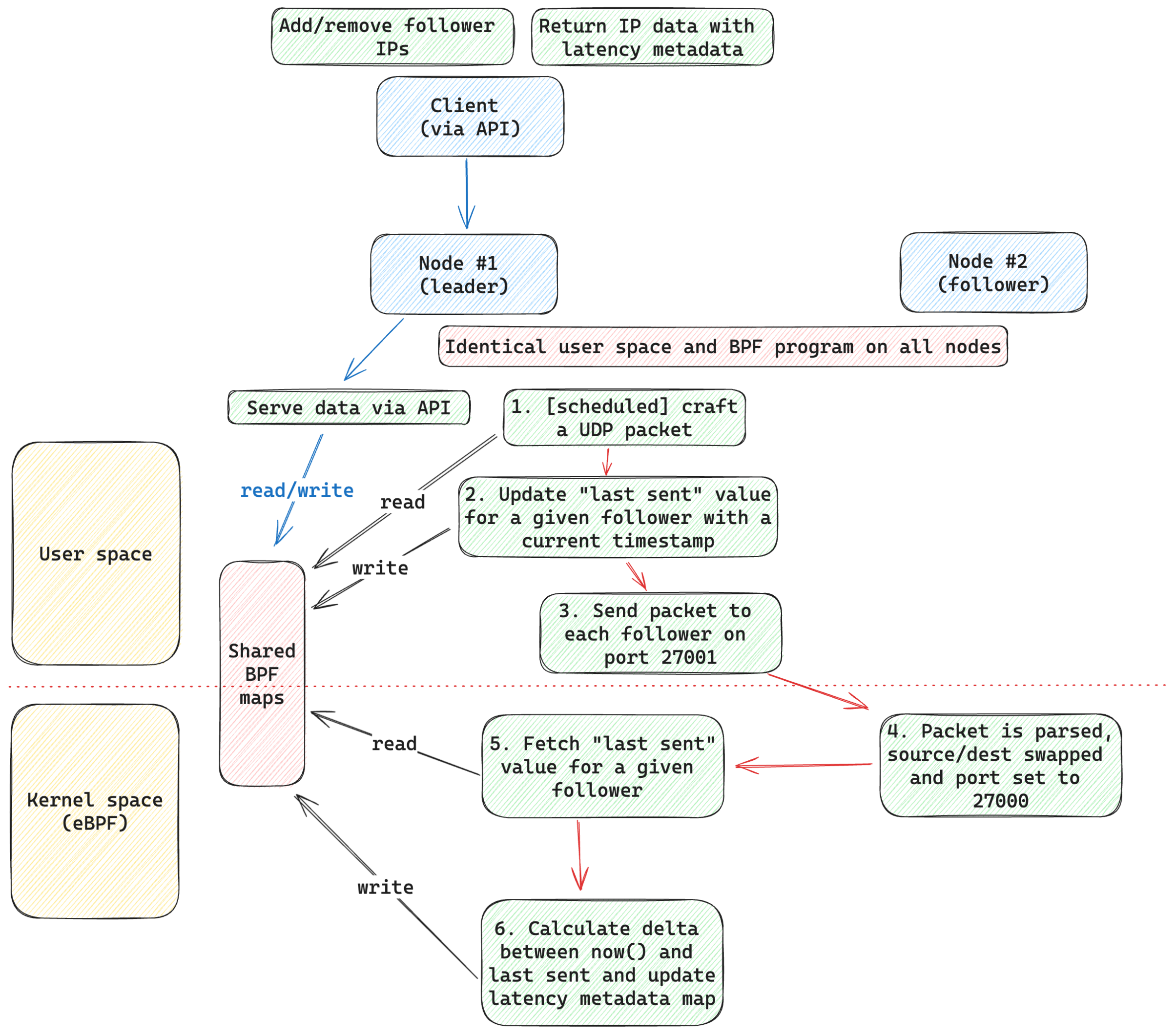
```
[INFO raft] Listening on...0.0.0.0:8888
[INFO raft] add_follower: successfully inserted IP address: 1.1.1.1
[INFO raft] list_followers: successfully returned list of followers (
   total: 1)
[INFO raft] delete_follower: successfully removed IP address: 1.1.1.1
[INFO raft] list_followers: successfully returned list of followers (
   total: 0)
```

3.8 Summary

This chapter demonstrated a rudimental but functional eBPF-based heartbeat implementation. The current approach does not yet include Raft algorithm functionality, but allows recording basic data about follower response times in a single cluster. Cluster membership can be managed dynamically, and current map data accessed using an HTTP API. All machines run identical code in user and kernel space, which allows any follower to become a leader and vice versa.

The work described in this chapter lays the groundwork for building additional Raft algorithm features, explored in subsequent chapters.

The diagram below visualises the current eBPF heartbeat flow implementation:


Add/remove follower IPs
Return IP data with latency metadata
Client (via API)
Node #1 (leader)
Node #2 (follower)
Identical user space and BPF program on all nodes
Serve data via API
1. [scheduled] craft a UDP packet
read/write
read
2. Update "last sent" value for a given follower with a current timestamp
write
User space
Shared BPF maps
3. Send packet to each follower on port 27001
Kernel space (eBPF)
read
5. Fetch "last sent" value for a given follower
4. Packet is parsed, source/dest swapped and port set to 27000
write
6. Calculate delta between now() and last sent and update latency metadata map

4 Implementing consensus and leader election

4.1 Consensus and consistency

In modern computing environments, distributed systems are often highly dynamic due to automated capacity adjustments, which respond to changes in system load. Similar to single-node systems, distributed systems are also susceptible to physical and software failures. Unresponsive, slow or crashing nodes, and mismatching system clocks, require mechanisms to handle and recover from failures, complicating the task of building reliable systems where network is the primary of way of communication. Raft, Paxos and ZooKeeper Atomic Broadcast (ZAB) consensus algorithms provide a method for ensuring data consistency and reliability in distributed systems. While individual implementations may differ, all algorithms attempt to make multiple machines appear as a single and robust process.

Distributed consensus algorithms provide different levels of consistency guarantees. Strongly consistent systems guarantee that most or all nodes hold the same data at all times. Raft and other strongly consistent algorithms prioritize consistency over performance by requiring the majority to agree on each change to the data [83, p. 323]. In contrast, eventually consistent systems may return stale data until the overall system converges on the new value [83, p. 322]. This characteristic allows achieving higher throughput at the cost of temporary inconsistencies.

Raft algorithm uses “linearizable semantics” [20, p. 70], a strong form consistency, which makes “a system appear as if there were only one copy of the data and all operations on it are atomic” [83, p. 324]. Atomic commits ensure that each change is either committed to the log or aborted — never partially applied. Atomicity in Raft is maintained by filtering duplicate client requests. A leader may receive a request and commit the entry but crash before responding, forcing the client to retry the request. To avoid duplicate commits of the same data, Raft assigns each client a unique identifier and generates a serial number for every request. A client session is maintained on each server and its expiration is coordinated between servers. This approach, implemented using distributed consensus, allows filtering duplicate requests and applying all changes sequentially across all nodes.

Data consistency semantics are outside the scope of this thesis. However, it illustrates scenarios which the algorithm must handle to present a consistent view of the system.

4.2 Leader election

Many consensus algorithms require “one process to act as coordinator, initiator, or otherwise perform some special role” [84, p. 330]. It ensures all peers are synchronized, despite each system not sharing a centralised clock. In 1978, Leslie Lamport introduced a method for sequencing events using a system of logical clocks, focusing on the relative order of events instead of using absolute time [107].

Raft protocol implements logical clocks using “terms” of arbitrary length [20, p. 14]. Term number is monotonically increasing and is exchanged whenever servers communicate. There can be only one leader in any given term, ensuring leader autonomy. Additionally, term numbers allow detecting stale leaders.

When a new Raft node starts up, it immediately becomes a follower and remains in this state as long there is a leader who maintains its authority by communicating with all followers. If followers do not receive any communication during a specific time, they become a candidate, increment their local term number and send vote requests to other cluster members. More than one follower can become a candidate and request votes, causing the leadership position to become contested. Raft randomizes timeouts to decrease the chance of split votes. The candidate wins the election and becomes the leader after receiving votes from the majority of the servers. It then moves to establish its authority by sending heartbeats to the followers. If another a candidate or a previous leader discovers that its term is out of date, it becomes a follower. Communication between the leader and followers ensures that the current term is always propagated to all cluster members [20, p. 15].

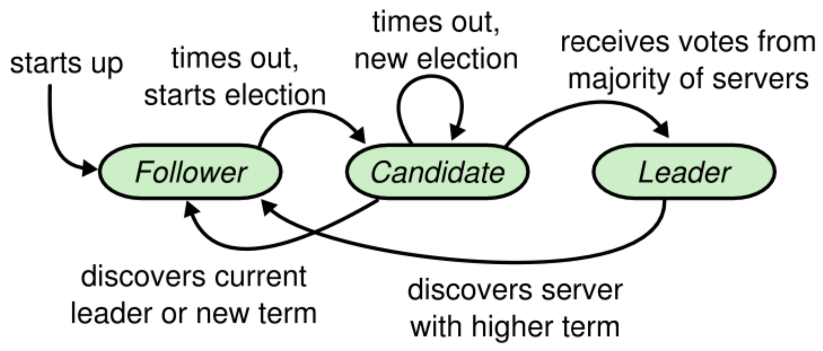


Figure 4.1: *Leader election process [20, p. 15].*

Raft safety mechanisms restrict leadership only to those candidates with the most up-to-date data [20, p. 22]. Raft uses a stronger form of leadership to maintain authority over the followers and is a conscious design choice. One of the authors argues that implementing an alternative multi-leader approach would conflict with the existing approach. It would likely result in a new algorithm, one which is much harder to understand [20, p. 164].

The remainder of this chapter will focus on implementing functionality of the leader election process by introducing node states, terms and additional communication methods to the application described in Chapter 3.

4.3 States and terms

The existing heartbeat mechanism does not yet have any Raft-specific features — other nodes simply respond to incoming heartbeat packets and take no action in the absence of communication from the leader. To ensure all nodes can function as valid Raft cluster members, each node must maintain its own internal state. Each state would allow a node to make decisions about the state of the cluster, perform specific tasks, and transition between states if necessary. As before, BPF maps will be used as a storage interface for storing additional data structures.

The following sections extend the heartbeat implementation described in Chapter 3 by incorporating term numbers and adding different node states.

4.3.1 Node state

eBPF provides an array storage to store a constant size map. `BPF_MAP_TYPE_ARRAY` uses integers as keys and allows storing primitive and composite data types as entries.

On start, each Raft node is initialised in a follower state. `Node` composite data type (struct) encodes relevant information about the current node and the cluster:

- `state`: `Follower | Candidate | Leader` enum.
- `term`: local term number.
- `peers`: IP addresses for all cluster members.
- `vote`: composite data type for storing vote metadata during the election process (`Candidate` state only):
  - `in_progress`: indicates vote status.
  - `started_ts`: last vote start time.
  - `ended_ts`: last vote end time (used for measurements only).
  - `election_timeout`: randomized election timeout.

In addition to maintaining state about itself, each node must record the state of the leader. `Leader` is a composite data type containing basic information about the leader, populated by nodes in follower state after receiving a heartbeat. It contains the following fields:

- `last_seen`: timestamp when the last communication from the leader was received.
- `source_addr`: IP address of the current leader.
- `term`: current leader term.

`Node` and `Leader` are kept separate for simplicity and are defined as a singleton arrays:

```
// eBPF
pub static NODE: Array<Node> = Array::with_max_entries(1, 0);
pub static LEADER: Array<Leader> = Array::with_max_entries(1, 0);
```

Interacting with array-based maps in an eBPF program is similar to hash maps. Array-based maps require retrieving a mutable pointer and updating values directly, rather than inserting data. There are two additional considerations when using composite types in BPF maps:

1. `#[repr(C)]` Rust macro [108] for performing necessary conversions for memory alignment to ensure interoperability with C language.
2. `aya::Pod` trait for types that can safely be converted to and from byte slices [109].

Additional maps extend previous functionality and contain relevant data to be used by user and kernel space programs.

4.3.2 State machines

Each Raft node starts three background user space processes to allow defining functionality for each of three possible node states — `Follower`, `Candidate` and `Leader`. All processes use the same mutable shared data structure to interact with shared state. This allows easy transitioning between different states and accessing data structure concurrently, from user and kernel space code.

Background processes are configured to run in parallel, in an infinite loop, configured to only handle actions for the specific state of the node.

```
loop {
  if !state.am_i(NodeState::Leader) {
    continue; // skip, if current node is not a leader
  }
  // perform leader related activities
}
```

4.3.3 Terms

As described previously, the term is represented as a monotonically increasing integer. The term is represented as a 64-bit integer, and its fixed length allows encoding and decoding the value from the UDP packet payload.

In the eBPF program, the UDP packet payload can be retrieved by reading data after the Ethernet, IPv4 and UDP headers, which are 14, 20, and 8 bytes long, respectively. The length of the payload is 8 bytes, representing the size of an unsigned 64-bit integer.

The function below is used to verify packet boundary:

```
// eBPF
if helpers_xdp::ptr_exists::<[u8;8]>(
    ctx,
    EthHdr::LEN + Ipv4Hdr::LEN + UdpHdr::LEN,
) {
    return true;
}
```

Additional helper functions are defined to facilitate data conversion to match term representation defined in `Node` and `Leader` data structures:

```
// eBPF
let incoming_term_number: u64 = match helpers_raft::
   parse_term_in_payload(&ctx) {
    Ok(x) => x,
    Err(_) => return Ok(xdp_action::XDP_DROP)
 };
```

UDP packets, which either have no payload or have it larger than 8 bytes, are discarded using `XDP_DROP` return code. Since the eBPF program uses incoming packet pattern matching on protocol and port, packet parsing functionality exists in few places, limiting chances of rejecting legitimate network traffic. There are also risks associated when relying on encoding term numbers in such a way, since 8 bytes can hold completely arbitrary data. For the purposes of this project, it is assumed that term numbers are only sent by a candidate requesting a vote or a leader sending heartbeats.

Encoding the term number in the UDP packet allows eBPF program to decode the incoming term, compare it with its own and perform necessary actions as part of the algorithm.

4.4 `RequestVote`

Raft algorithm has only two ways of communicating between servers. The first method is initiated by the leader and involves sending `AppendEntries` RPCs to other members. So far, this chapter focused on extending the initial heartbeat functionality by adding a term number to each UDP heartbeat packet.

The second communication method involves sending `RequestVote` RPCs. These calls are initiated by candidates when requesting votes from other members. Similar to `AppendEntries`, `RequestVote` RPCs contain information about the candidate state — its current term and last log index. This information allows other nodes (voters) to decide whether to vote for a particular candidate.

To differentiate between different types of RPCs, eBPF program pattern-matches on two additional ports. Within the context of this project, these ports are referred to "vote ports" and have the following specifications:

- port `28000` is used to receive `RequestVote` *requests* RPCs on all nodes.
- ports `29000` and `29001` are used to `RequestVote` *responses* on nodes in the candidate state. Each port records "no" and "yes" vote response, respectively.

Request and response ports require additional data structures to allow implementing desired Raft protocol features:

1. `VOTE_TERMS` hash map records terms for which a particular node has voted for and uses the term number as a lookup key.
2. `VOTE_RESULTS` hash map records the responses for each voting round, using voter IP as a lookup key. The user space program iterates through keys and values in the map to determine whether a quorum is reached, and map data is reset after each voting round.

The current voting mechanism does not incorporate additional Raft data in the packet payload. It exclusively relies on term numbers present in packet payload and this information is sufficient to allow demonstrating eBPF-based leader election process. Log replication and validation is outside the scope of this project, but the existing implementation can be further extended to incorporate additional checks when receiving both `RequestVote` and `AppendEntries` remote procedure calls.

4.5 First election

This section describes a scenario when starting a leader election in an uninitialised Raft cluster with 3 nodes. For clarity, user and kernel space behaviour is described separately and focuses on a single node which transitions from being a follower to a candidate, before finally becoming a leader.

The previous follower tracking implementation is extended to allow populating peer data on the current node with a list of all IP addresses of the cluster. During initialisation, the node's own IP is excluded from the peer list.

4.5.1 User space

Follower. On startup, each node gets initialised to a follower state and starts state-specific background processes. In this case, the cluster does not have a leader — heartbeat packets are absent and the `Leader` data structure is empty. After reaching a randomized communication timeout, the follower transitions to a candidate state and begins a new election.

Candidate. The candidate process starts a new vote. It increments its term number and votes for itself by inserting a "yes" vote in the `VOTE_RESULTS` map. It then sets a randomized election timeout and sends `VoteRequest` RPCs to other peers' vote request port (`28000`). The candidate remains in this state until one of the following conditions is met:

- The candidate's election timeout expires, triggering a new election with incremented term number.
- The candidate receives majority votes from other nodes by counting "yes" votes in the `VOTE_RESULTS` map.

In this scenario, the candidate only needs a single vote from either of the two nodes, which allows it to reach a quorum (2 out of 3) and transition to the leader state.

Leader. As a newly minted leader, it establishes its authority by sending heartbeat packets to other peers. Other nodes, regardless of their state, transition to being followers. Other nodes update their term number to match one received in leader's heartbeat packets.

The leader continues sending heartbeats until it crashes. Failover behaviour will be explored in detail in Chapter 5.

4.5.2 Kernel space

Follower. Followers receive `VoteRequest` packets on their voting port `28000`. It decodes the term number present in the packet and determines voting behaviour by sending a corresponding packet to an appropriate “no” (`29000`) or “yes” (`29001`) port using a `XDP_TX` return code.

Candidate. The candidate receives incoming vote responses and updates `VOTE_RESULTS` map with appropriate values — `0` for “no”, `1` for “yes”.

Leader. A node in the leader state only records follower latency metrics. Since UDP payloads are sent from the user space, cluster state updates take place on follower and candidate nodes.

If followers or candidates receive heartbeat packets with a higher term than their own, they either remain in or transition to the follower state.

4.6 Summary

The implementation described so far extends the original heartbeat functionality to include term numbers and adds additional communication methods (`RequestVote`) to implement the leader election process. The resulting eBPF-based heartbeat and leader election mechanism takes place without using user space web servers and can exist alongside Raft and non-Raft networking traffic between nodes. While the current approach purposefully excludes log replication verification checks, it demonstrates various mechanisms through which such checks could be added in the future.

The following diagram offers a visual description of the leader election process as described in this chapter, integrating with existing heartbeat functionality.

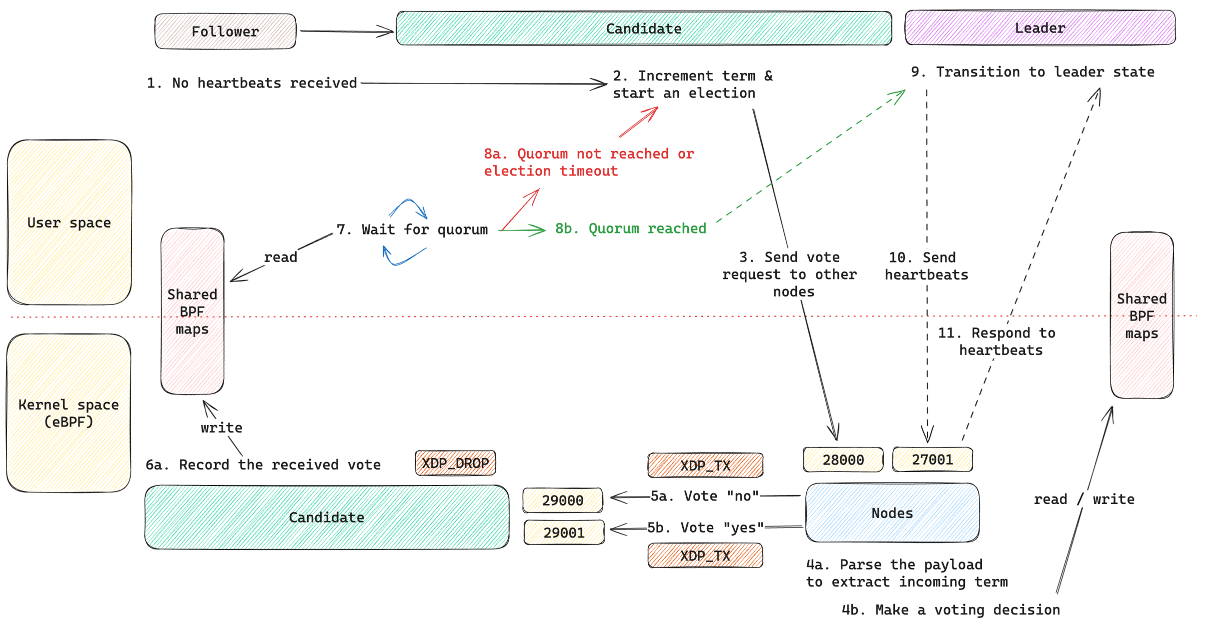

Follower
Candidate
Leader
1. No heartbeats received
2. Increment term & start an election
9. Transition to leader state
User space
8a. Quorum not reached or election timeout
7. Wait for quorum
8b. Quorum reached
read
Shared BPF maps
3. Send vote request to other nodes
10. Send heartbeats
11. Respond to heartbeats
Shared BPF maps
Kernel space (eBPF)
write
6a. Record the received vote
XDP_DROP
XDP_TX
28000
27001
Candidate
29000
5a. Vote "no"
Nodes
read / write
29001
5b. Vote "yes"
XDP_TX
4a. Parse the payload to extract incoming term
4b. Make a voting decision

Figure 4.2: eBPF-based Raft leader election process.

5 Failovers

The previous chapter has shown a functional leader election process. This section will describe additional guardrails and functionality required for tolerating faults in the process. Additional guardrails will help to resolve split vote, avoiding multiple leaders, and mention challenges synchronizing behaviour between user and kernel space code.

5.1 Timeouts

5.1.1 Communication timeout

The node in the follower state is configured to expect receiving heartbeats every 50 milliseconds. It randomizes the communication timeout between 100 and 150 milliseconds to prevent race conditions with other nodes. This ensures the node with the lowest communication timeout in each follower loop iteration will be the first to detect the absence of communication and initiate a new election.

To detect a failed leader, the follower relies on data recorded by the eBPF program receiving heartbeat packets on the heartbeat port (`27001`). The eBPF program updates the shared `Leader` data structure and allows the user space program to detect absence of heartbeat packets:

```
if state.leader_last_seen() > LEADER_COMMUNICATION_TIMEOUT_MS {
    state.become_candidate();
    continue
}
```

In practice, some followers may never transition to a candidate state, as the election finishes before their communication timeout expires.

5.1.2 Election timeout

Each election starts by randomizing election timeout value. The resulting timeout value is between 1 and 5 milliseconds, to ensure the election outcome is reached quickly. Election start time is recorded using the monotonically increasing system boot clock and used by user space functions to determine if the vote has timed out:

```
pub fn election_timed_out(&self) -> bool {
    let node = self.get_current_node();
    let time_elapsed = get_current_clock_ns() - node.vote.started_ts;
    if node.vote.in_progress && (time_elapsed > node.vote.
   election_timeout) {
        return true
    }
    return false
}
```

Ongaro [1, p. 17] recommends setting leader election timeout between 150 and 300 milliseconds, however, most modern Raft implementations use much lower values. The relatively fast connectivity between virtual machines allows for these timeouts to be lowered, aborting any split elections and starting new ones within a shorter period of time.

5.2 Voting

Vote request port 28000, described in Section 4.4, serves as a key part of the election process. Vote requests are received and parsed by other nodes to determine their voting decision. This decision is based on the incoming term number and local state of the node.

Raft algorithm requires each node to vote for at most one candidate in a given term [20, p. 16]. This condition is met by tracking vote terms in VOTE_TERMS map and discarding votes (packets) for terms that the node has already voted on.

Having each node vote only for a single term prevents situations when multiple candidates simultaneously increment their term numbers and send vote requests, leading to a split vote for multiple consecutive election cycles.

Assuming the node has not yet voted for a given term, its voting decision is as follows:

- It votes "**yes**" if the incoming term number is greater than the term number currently present on the node.
- It votes "**no**" if the incoming term number is less or equal than the number currently present on the node.
- It **drops** the incoming packet if it has received `VoteRequest` while being a leader.

No node state transitions are performed in voting ports. Aya lacks full BTF support [110] which prevents iterating through map items. This requires delegating vote counting functionality to a user space program. With future support for BTF, it would be possible to transition from a candidate to a leader state after the final vote that meets the quorum.

5.3 Accepting a new leader

State transitions, explicitly defined in the user space program, are unidirectional. Node starts as a follower, then becomes a candidate and upon quorum, becomes a leader.

As previously mentioned, the leader establishes its authority by sending heartbeat packets to other nodes. After a new leader is elected, some nodes might still be in the candidate state, in the process of sending `RequestVote` requests to other nodes. To prevent an instance when a newly elected leader receives a vote request from another node with a higher term number, two additional eBPF program restrictions are introduced:

1. The newly elected leader will not respond to any vote requests for the duration of its term.
2. Nodes in the cluster will revert to the follower state when receiving the first heartbeat packet.

The state transition within the eBPF program initiates activity in the follower background process. Followers implicitly trust any node which sends traffic to the heartbeat port `27001` and use received packets to update leader metadata (`Leader.last_seen`) and match the incoming term number with their own.

5.4 Troubleshooting

Synchronizing kernel and user space program behaviour presents a non-trivial challenge, as the election process and subsequent state transitions happen in less than a millisecond. Command-line messaged produced by user space programs are often displayed before the logs produced by the kernel space program, further complicating the debugging process:

```
[user] [candidate] Starting vote (term: 38), election timeout: 1314323719 ns
[user] [candidate] Quorum (2/3) reached after 1673677 ns, becoming leader with term: 38
[kernel] [candidate] [<-] Received 'YES' vote from 3323197736
[kernel] [candidate] [<-] Received 'YES' vote from 3323197789
```

To aid troubleshooting, all log lines produced by the eBPF program have been prefixed with a timestamp produced by the `bpf_ktime_get_ns()` function. This serves as an *execution ID* for each eBPF program invocation and allows for ordering of eBPF events.

5.5 Summary

eBPF-based leader elections ensure an orderly transfer of power after each election. It provides an efficient convergence mechanism that exists alongside other network traffic between nodes. Offloading Raft algorithm features to eBPF programs enables transitioning between different node states much faster, using data captured at the first operating system entry point of the network packet. BPF maps provide efficient data storage for multiple programs on the same system, allowing user and kernel space applications to react to changes in shared data.

The current leader election implementation does not strictly adhere to Raft specification as it lacks additional information, for instance, candidate log state, which is necessary for an existing leader to be successfully deposed [20, p. 22]. Extending the UDP payload to include additional Raft node information is reserved for future work.

6 Tests and observations

This chapter evaluates the performance of the currently implemented Raft algorithm features: heartbeats and leader election. Each test was performed multiple times and the final result was selected for representation. Data is represented in two formats: graphs visualize results, while the accompanying tables provide a summary.

Visualization shows data metric readings over time (x), aligning data where necessary to compare data from separate test runs. Latency (y) is represented as nanoseconds, shown on a logarithmic scale.

Tables provide latency summary, converted to milliseconds. The summary includes data for multiple percentiles (50th, 95th, 99th), as well as minimum, average and maximum values.

6.1 Heartbeats

The first group of tests measure time needed to send a heartbeat request to a follower and receive a response back. As described in Chapter 3, the heartbeat flow is as follows: a leader sends a UDP packet, a follower receives it, rewrites source and destination fields in eBPF and finally returns the packet through the same network interface using `XDP_TX` return code.

Applications run in two distinct scenarios, low and high system load, with the aim of observing heartbeat latency performance under different operational conditions.

6.1.1 Methodology

Initial measurement is taken after the heartbeat packet is sent, whereas the final measurement is taken when the packet gets received by the eBPF program. Each data point represents the difference between both measurements, capturing response latency. Response latency for each heartbeat response is calculated and printed by the leader's eBPF program, using data recorded in HEARTBEAT_LATENCY map introduced in Section 3.5.1.

An example log entry shows a received heartbeat response from 198.19.249.93 IP address (represented as 3323197789), with latency of 32209 nanoseconds or ≈ 0.03 milliseconds:

```
[2024-04-05T18:50:00Z INFO  raft_main] [leader] :27000, Received a
    heartbeat response from 3323197789,32209
```

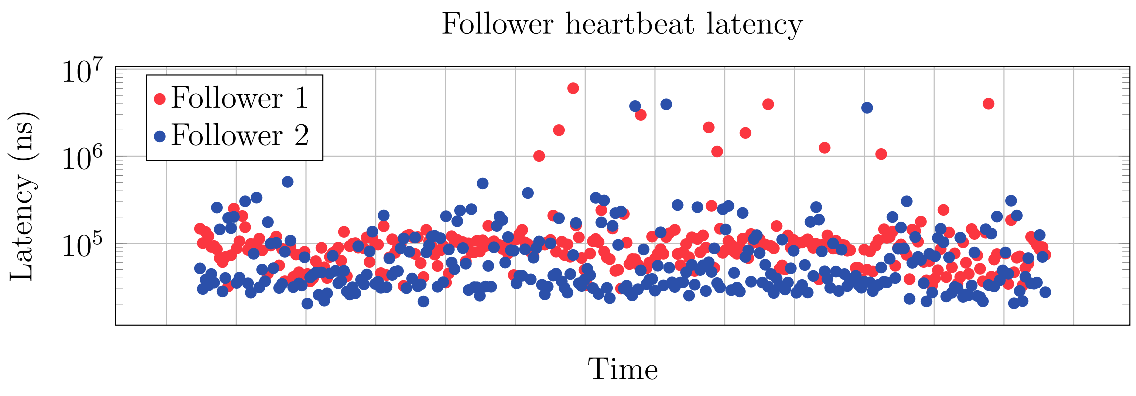
Over the test period, the cluster topology does not change and includes a single leader and two followers. The leader sends heartbeats every 200 milliseconds, and the test duration is approximately a minute. This results in about 300 data points for each follower.

High system load is simulated using stress [111] utility, targeting all available cores of the development machine:

```
$ stress --cpu 10 --vm-bytes 128M --timeout 900s
```

6.1.2 Low system load

Measurements taken during a low system load attempt to capture baseline performance of the heartbeat flow. During the test interval, each follower received and responded to exactly 300 heartbeat requests.



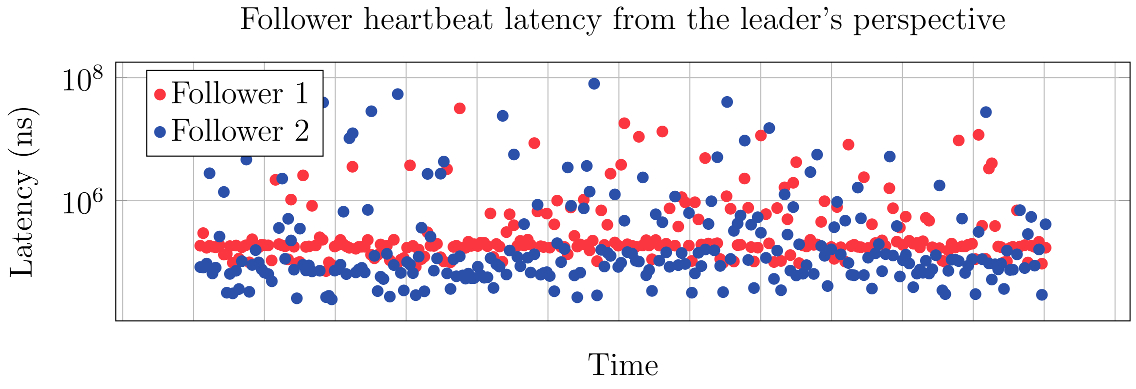
Machine	Min	Average	p50	p95	p99	Max
Follower 1	0.03	0.19	0.09	0.24	3.95	6.02
Follower 2	0.02	0.13	0.04	0.28	3.60	3.94

Table 6.1: Low system load measurement summary (in milliseconds).

Data shows similar performance for both followers, with occasional spikes. These spikes are between 50 to 100 times slower than the median (p50) value. Outliers indicate a reliability problem which can impact operations of the cluster.

6.1.3 High system load

High system load simulates operations under a high CPU load, that affects the entire system. Each follower responded to exactly 257 heartbeat messages, suggesting a congestion when sending UDP packets through the user space network socket.



Machine	Min	Average	p50	p95	p99	Max
Follower 1	0.07	0.90	0.19	3.78	12.46	31.69
Follower 2	0.02	1.89	0.10	6.14	39.85	79.80

Table 6.2: High system load measurement summary (in milliseconds).

Measurements show a significant increase in 95th and 99th percentile. Outliers continue to be present, with system load likely exacerbating the underlying root cause.

Compared to measurements taken during low system load, 50th percentile data shows similar results, indicating that half of the time, followers process and return packets rapidly.

6.1.4 Possible outlier sources

Intermittent latency increases seen in the previous measurements can be caused by any layer of the system, from underlying virtualization software and its networking fabric implementation to the kernel and user space programs running the application. eBPF and application code will be examined in detail.

eBPF

XDP is optimised for performance; however, its current configuration might introduce additional latency in the processing path, caused by inefficient packet processing logic or slow map item retrieval.

`bpftop` tool allows viewing real-time performance information about all loaded eBPF programs. It uses `BPF_ENABLE_STATS` system call to enable global runtime measurement and refreshes statistics at one-second intervals. The output produced by the program confirms XDP as an unlikely source of the increased latency, as average eBPF program runtime is less than 0.02 milliseconds.

Machine	Total avg. runtime (ns)	Events per second	Total CPU
Leader	10555	10	0.01%
Follower 1	16385	5	0.008%
Follower 2	16610	5	0.009%

Table 6.3: Output from the `bpftop` utility.

Measurements produced by `bpftop` show average runtime, event count and CPU usage. eBPF program on the leader node gets invoked 10 events per second, matching the expected behaviour of receiving 5 heartbeat responses from each follower in a 200 millisecond intervals. Similarly, eBPF program on each follower gets invoked 5 times. The difference in CPU usage between leader and followers is likely explained by the overhead introduced by performing additional map lookups after receiving the heartbeat packet.

Rust

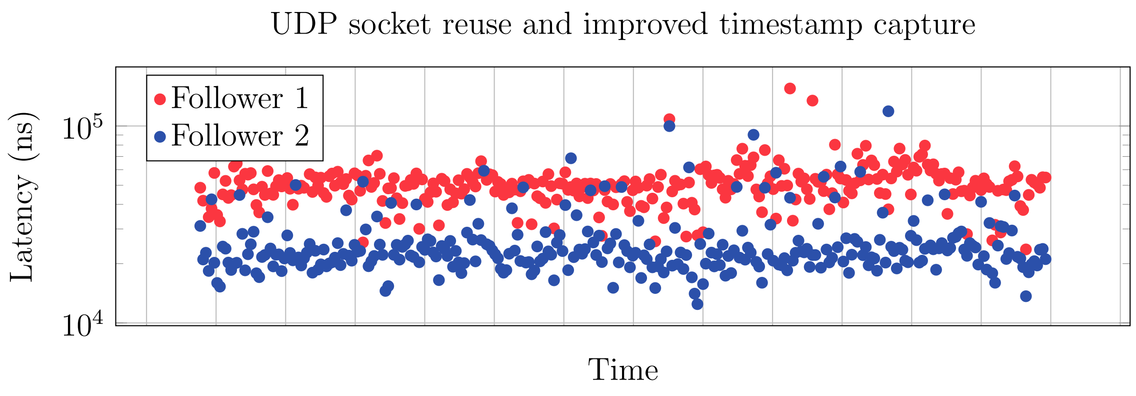
Another source of slowness may be the implementation of user space communication logic. The current approach iterates through a list of peers, records the current timestamp, and sends the packet to the follower. Between timestamp capture and network call, there are a number of additional function calls which may slow down the processing and introduce slowness. These include calls to create network sockets using `UdpSocket::bind()` and a socket address structs via `SocketAddr::new()`.

Internally, network socket creation requires making a `socket()` *syscall*, which returns a socket descriptor. This descriptor allows sending and receiving network packets using special socket calls [112]. A socket address data structure is instantiated for each destination host, referencing the socket object and populating additional fields, for instance, IP address and port number.

Socket creation requires allocating system resources. User space logic has been updated to initialise a single shared socket to be used across all functions of the program. Timestamp capture now gets recorded just before the packet gets sent, ensuring that any other function calls (socket address struct creation, payload creation) do not interfere with recording the timestamp.

6.1.5 Low system load (with UDP socket and timing optimizations)

UDP socket reuse shows a significant improvement over previous measurements. Outliers are no longer present, and the maximum heartbeat response is only up to 3 times compared to the median value.



Machine	Min	Average	p50	p95	p99	Max
Follower 1	0.02	0.05	0.05	0.06	0.10	0.15
Follower 2	0.01	0.03	0.02	0.05	0.07	0.12

Table 6.4: Application optimization measurement summary (in milliseconds).

Optimizations expose a latency gap between each follower, as the data shows one of the followers being slower than the other. Multiple rounds of repeated tests show similar results.

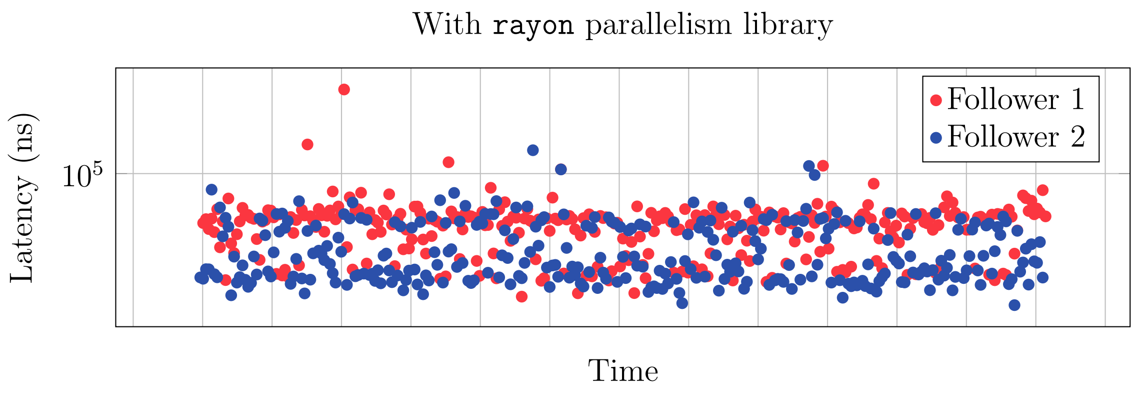
The gap between followers may be explained by the UDP communication protocol, which does not provide any ordering guarantees. Operating system network socket buffers may also introduce additional latencies when sending small UDP payloads.

A helpful clue is provided in the original Raft paper, which specifies that the Raft servers issue RCPs in parallel to achieve better throughput [1, p. 17] and avoid performance degradation caused by head-of-line blocking. `rayon` a data parallelism library for Rust, allows converting sequential code into parallel with minimal changes. The parallel implementation of the original heartbeat request functionality can be seen below:

```
let followers = state.get_current_followers();
followers.par_iter().for_each(|&ip| {
    state.send_heartbeat_rpc(&socket, ip);
});
```

6.1.6 Low system load (with UDP socket, timing optimizations and parallelism)

The switch to sending heartbeat requests in parallel has decreased the variance between different followers, resulting in a consistently performant heartbeat exchange.

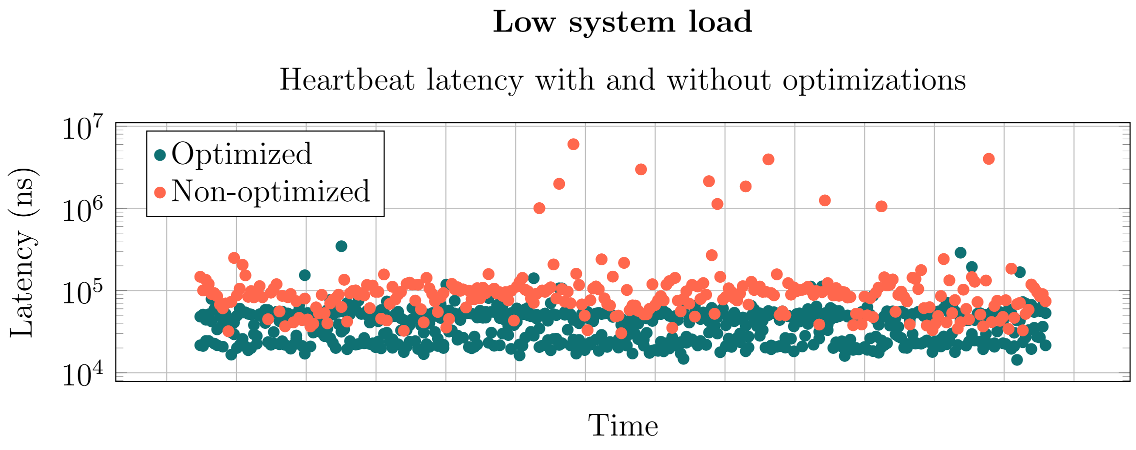


Machine	Min	Average	p50	p95	p99	Max
Follower 1	0.02	0.05	0.05	0.07	0.16	0.34
Follower 2	0.01	0.03	0.03	0.06	0.10	0.14

Table 6.5: Parallel optimization measurement summary (in milliseconds).

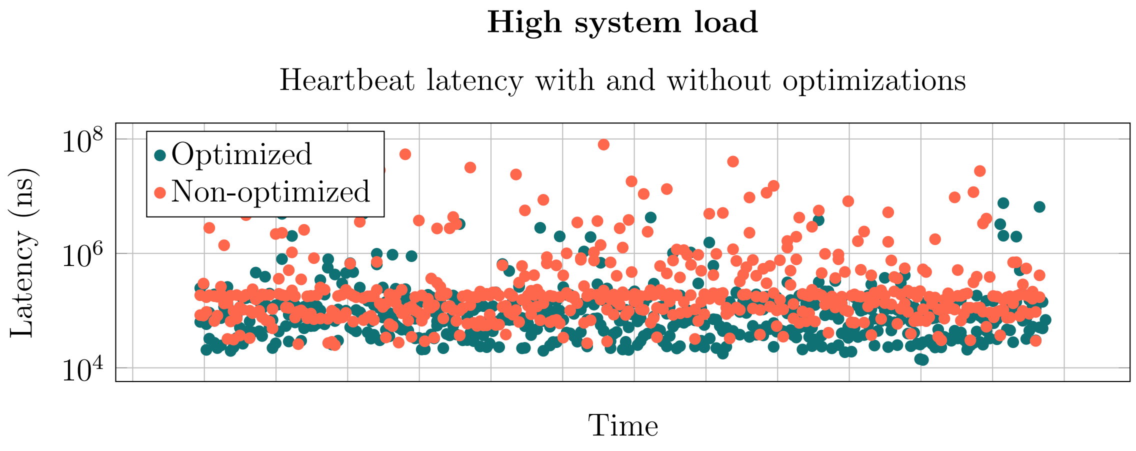
6.1.7 Summary

Application improvements have addressed latency spikes and discrepancies between individual nodes. Comparison between optimized and non-optimized versions of the application show overall improvements, during low and high system loads. In particular, the optimized version has a reduced tail latency and a more predictable latency response during low and high system load. Meanwhile, the median response time across all four categories ranges from 0.05 to 0.18 milliseconds, indicating that high system load has minimal impact on half of the heartbeat requests.



Machine	Min	Average	p50	p95	p99	Max
Optimized	0.01	0.04	0.05	0.07	0.17	0.35
Non-optimized	0.03	0.20	0.08	0.24	3.94	6.02

Table 6.6: Optimized vs. non-optimized measurements: low system load (in milliseconds).



Machine	Min	Average	p50	p95	p99	Max
Optimized	0.01	0.21	0.07	0.51	3.80	7.56
Non-optimized	0.03	1.30	0.18	4.94	18.27	79.80

Table 6.7: Optimized vs. non-optimized measurements: high system load (in milliseconds).

This section has demonstrated eBPF-based heartbeat performance and has shown sub-millisecond response times during measurements under low system load. It also underscored the importance of troubleshooting, implementing gradual improvements and gathering data at each step of the process.

6.2 Leader election

The second group of tests measures the time required to successfully reach a quorum during the leader election process. The quorum is reached after a candidate receives a majority of “yes” votes from other nodes. Evaluation is done using different Raft cluster sizes to measure possible performance degradation with increased cluster size.

6.2.1 Methodology

Initial measurement is taken once a new election starts, and the final measurement is taken after a candidate reaches the quorum needed to become a leader. Both measurements are recorded in user space, with kernel space program processing vote responses and updating `VOTE_RESULTS` map accordingly (Section 4.4).

An example output of a successful leader election after 60709 nanoseconds (0.06 ms) for one of the candidates with IP address `198.19.249.40`) is shown below:

```
[candidate] 198.19.249.40,60709, becoming leader with term: 55
```

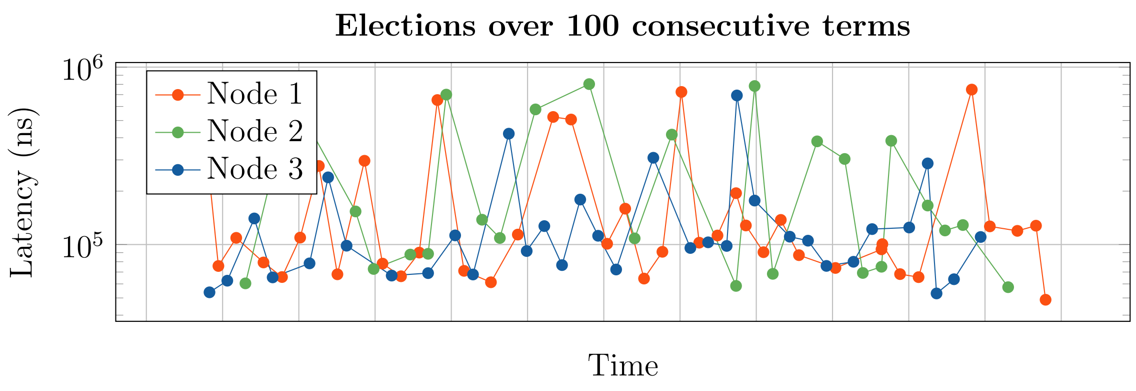
To minimize split votes, each follower and candidate randomizes its communication timeouts as described in Section 5.1. Leader failure is simulated by stopping communication between the currently active leader and other followers after the leader has sent 20 heartbeat RPCs. “Failed leader” transitions to a follower state, allowing other nodes to detect the absence of communication and start an election.

Tests use 3 and 5-node clusters, with appropriate quorum configuration. In a 3-node cluster, quorum is set to 2 nodes (self and one follower); in a 5-node cluster — to 3 nodes (self and two followers).

Tests stop after any node reaching 100th term, which lasts between one and two minutes, depending on cluster size.

6.2.2 Leader election within a 3 node cluster

The first test records leader election duration in a three node cluster. Data shows similar performance characteristics for all nodes.



Machine	Terms won	Min	Average	p50	p95	p99	Max
Node 1	40	0.05	0.19	0.10	0.72	0.75	0.75
Node 2	25	0.06	0.26	0.13	0.77	0.80	0.80
Node 3	34	0.05	0.14	0.10	0.34	0.60	0.70
	99	0.04	0.20	0.10	0.70	0.79	0.80

Table 6.8: Leader election duration (in milliseconds).

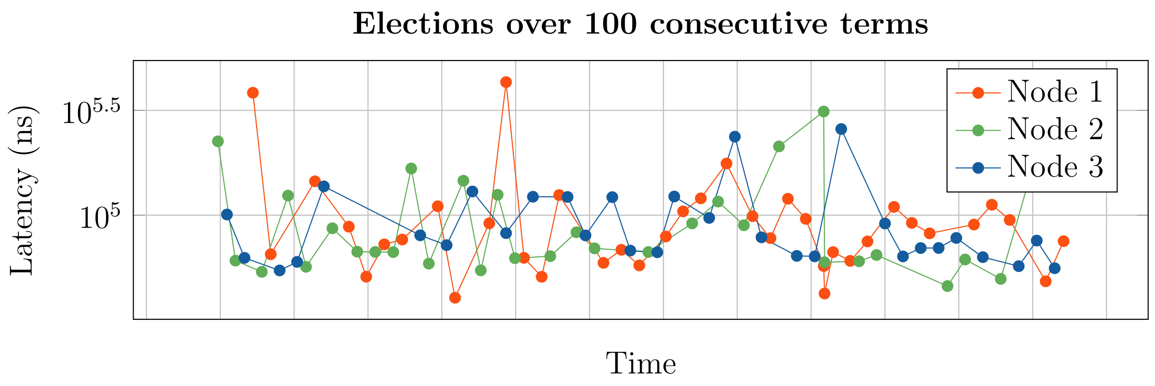
The data shows 99% leader election success in each of the 100 terms, somewhat evenly distributed between the nodes. Measurements also show several slow elections — up to 7 times slower than the median.

As mentioned in Section 4.3.2, the user space application relies on three background processes, each running within its thread. Each thread runs in an infinite loop, and each iteration requires it to read from a shared data structure to determine the state it is in. Data structure access uses a `RwLock`, a read-write lock, which allows read-only or write-only access. Compared to `Mutex`, `RwLock` does not create an exclusive lock for reads [113]. While designed for concurrent data access across multiple threads, the operating system context switch mechanism may be introducing additional delays when accessing data.

To verify this hypothesis, user space code has been rewritten to use a single background process. The process performs a single node state lookup on each loop iteration and allows pattern matching relevant functions for a specific node state.

6.2.3 Leader election within a 3 node cluster: Single thread

Tests using a single background process show an overall decrease in leader election duration, with significantly reduced tail latency.



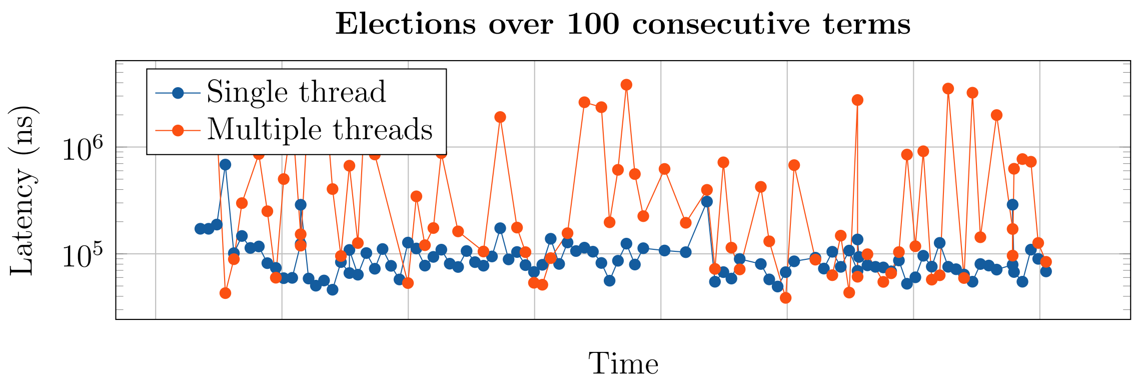
Machine	Terms won	Min	Average	p50	p95	p99	Max
Node 1	38	0.04	0.10	0.08	0.20	0.39	0.43
Node 2	31	0.05	0.09	0.07	0.22	0.28	0.31
Node 3	31	0.05	0.09	0.07	0.18	0.25	0.25
	100	0.04	0.10	0.07	0.23	0.41	0.43

Table 6.9: Leader election duration (in milliseconds).

Single thread measurements show between 2 and 3.5 faster election times 99% of the time, compared to multiple thread measurements. Data confirms that non-blocking read operations incur additional penalty, resulting in slower leader elections.

6.2.4 Leader election within a 5 node cluster: Single vs multi-thread

The following test uses a 5 node cluster to compare performance between using single and multiple threads.



Method	Terms won	Min	Average	p50	p95	p99	Max
Single thread	100	0.05	0.09	0.08	0.13	0.29	0.30
Multiple threads	80	0.04	0.75	0.17	3.4	3.9	4.0

Table 6.10: Leader election duration (in milliseconds).

Single thread performance a for a five-node cluster shows similar performance to a three-node cluster. Higher node count results in significantly increased tail latency and 20% of terms with no leader when running application with multiple threads.

6.2.5 Summary

Measurements described in this section show an efficient Raft leader election mechanism implemented using eBPF. Improvements further decreased election duration and led to having an elected leader in 100% of the rounds.

Compared to previously measured heartbeat latency under low system load, shows the additional overhead incurred by additional user space quorum processing:

Action	Min	Average	p50	p95	p99	Max
Heartbeats	0.01	0.03	0.03	0.06	0.10	0.14
Leader election	0.04	0.10	0.07	0.23	0.41	0.43

Table 6.11: Additional latency overhead incurred during the leader election process (in milliseconds).

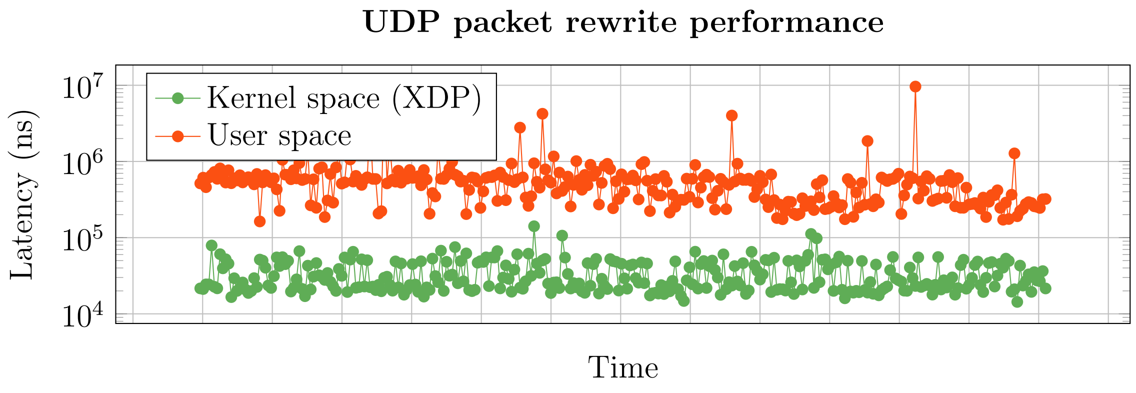
One way of decreasing the overhead is by offloading quorum handling functionality to eBPF program, as mentioned in Section 5.2. This would enable faster transitioning to a leader state after receiving the final decisive vote. Such an approach would remove the need of performing comparatively expensive user space lookups to determine voting outcome. The user space overhead can be further reduced by changing data access patterns or using *channels* [114], a feature which enables efficient communication in multithreaded Rust applications.

Timeout logic can be further extended to allow Raft clusters to operate across variable latency environments. Modern Raft implementations use back-off mechanisms to gradually increase communication intervals to limit network congestion.

6.3 XDP vs UDP

Thorough this thesis, performance characteristics of eXpress Data Path (XDP) framework have been mentioned multiple times. XDP allows rewriting the network packets at a lower level in the network stack, without requiring packs to traverse the full networking stack. This frees up system resources and reduces overall latency for internode communication.

The last test compares XDP with a native UDP solution where all packet processing occurs in user space. A native UDP solution consists of two user space applications, running on two separate nodes. It is configured to expose a UDP port to which to receive heartbeat traffic. Once it receives a packet, it rewrites it and sends it back through the same socket. This behaviour replicates the existing heartbeat functionality described in Chapter 3 and used in subsequent chapters.



Machine	Min	Average	p50	p95	p99	Max
XDP	0.01	0.03	0.03	0.06	0.10	0.14
Native UDP	0.16	0.61	0.53	1.05	4.00	9.62

Table 6.12: Heartbeat measurement comparing kernel and user space packet rewrite performance (in milliseconds).

Results show XDP delivering a much faster and consistent performance due to requiring only one user space traversal when sending and receiving UDP packets. In contrast, native UDP requires four traversals for each packet, resulting in a significantly increased latency and at least 15 times slower heartbeat response across all observed metrics.

7 Conclusion and Reflection

This project has been a valuable learning experience that has provided exposure to several distinct branches of study. Learning about the Linux kernel and eBPF ecosystem contributed to understanding operating systems, and Unified Datagram Protocol (UDP) allowed exploring details of computer networking. Studying Raft consensus algorithm through its original paper [1] and the Diego Ongaro's excellent PhD thesis [20] provided necessary background for understanding the algorithm and learning about different distributed consensus problems in detail. The Rust programming language and Aya library allowed working in a modern, type and memory-safe environment. The eBPF-based Raft communication protocol implementation and the original measurement data can be found in Appendix A.

Raft is an appropriate algorithm for implementing some of its functionality in eBPF. The relative simplicity of its communication protocol, as well as its widespread use, provided access to numerous practical examples in different programming languages. Time spent learning the algorithm and researching eBPF ultimately laid a foundation for developing required functionality.

This project has successfully demonstrated the feasibility of offloading core Raft network features to an eBPF program and enabling performant heartbeat and leader election mechanisms. XDP and Aya provide allow building applications for high-performance network packet processing in the Linux kernel.

The objectives, described in the Introduction, have been largely achieved:

- **Objective 1**: *Implement core Raft algorithm networking features as eBPF program(s). These features include heartbeats, leader election and log replication.*
  - This objective has been mostly met by implementing heartbeat (Chapter 3) and leader election (Chapters 4 and 5) functionality. Developing a log replication mechanism would require extending the existing eBPF-based communication implementation to include additional data consistency checks, as those briefly mentioned in Section 4.1. This objective is achievable; however, it is left for future work due to time and space constraints.
- **Objective 2**: *Develop a working prototype which integrates eBPF features with a user space program.*
  - This objective has been met by integrating all relevant user and kernel space code features into a single program. This prototype made extensive use of BPF maps for bidirectional communication across the operating system stack and provided a foundation for extending the application(s) in the future.
- **Objective 3**: *Validate the performance characteristics of the eBPF-based Raft algorithm implementation.*
  - This objective has been met by measuring the heartbeat and leader election performance, making additional optimizations and comparing the results with non-eBPF implementation. This comparison revealed significant performance improvements when using eBPF for packet processing.

Raft and eBPF are substantial areas of study in their own right, and doing them both adequate justice would require additional work. As such, this work has described only a small part of the eBPF ecosystem applied to a distributed consensus problem.

Sections below will reflect on each of the key parts of the thesis.

7.1 Rust

Starting a project in a new programming language can often be overwhelming, and the first weeks working with Rust were no different. It required understanding the Rust programming model, object lifetimes to build an application using appropriate language primitives. Rust borrow checker strictness led to many frustrations when modifying BPF map data via an HTTP API, but ultimately allowed discovering the established practices when working with shared resources.

Thanks to the aforementioned borrow checker, the program had almost no run-time crashes. During months of development, the program crashed only a handful of times, always when incorrectly dereferencing raw pointers in the eBPF program. The Rust compiler (`rustc`) ensured that, once compiled, a program would be unlikely to crash unexpectedly. Using Rust was the right decision, as using a programming language like C would have likely introduced subtle bugs which would be harder to troubleshoot. As a result, Rust allowed focusing on developing functionality rather than tracking runtime errors.

Rust borrow checker and eBPF verifier, exert strict control over the permitted behaviour. After the initial learning curve and time spent deciphering various compilation errors, working with user and kernel space program became an enjoyable experience. Rust dependency manager (`cargo`) allowed quickly adding high-quality libraries to user space code, and linter (`clippy`) provided various tips for improving the readability of the program.

At the same time, existing programs can be significantly improved by adding tests and additional error handling. Any improvements in this area are left for the future.

7.2 Linux kernel and eBPF

For someone with limited exposure to Linux kernel internals, publicly available documentation allows detailed tracing of the history of kernel feature development. Linux, the largest open-source project [115], provides extensive manual (`man`) pages for all its features and its pull request discussions provided invaluable context for introducing various kernel features over the years. In particular, blog posts on lwn.net, have been particularly accessible, providing an overview and impact of changes introduced in the kernel.

eBPF can be understood by reading technical blog posts written by individuals working with eBPF professionally. The high quality of these posts [46] made working with eBPF less daunting. eBPF was further made approachable through interactions with the Aya community, where members and maintainers helped understand cryptic eBPF verifier errors and the Rust programming language.

At its core, eBPF programs allow loading custom programs to be executed in the Linux kernel. This requires privileged access to the system to load an eBPF program. It is possible to write an XDP-based program where incorrect application logic would cause all incoming networking traffic to be discarded, locking the user out of the system and requiring a restart of the machine. This may limit adoption of eBPF as a general purpose platform for building applications, as loading eBPF programs distributed by third parties carries security and reliability risks [116].

This project explored only a small fraction of the features available in eBPF. The project can be further improved by using additional data structures (stacks, queues, ring buffers) to streamline communication between user and kernel space programs. Subscribing to map changes would be a welcome addition to the eBPF, as it would allow saving CPU cycles when polling for data.

eBPF program invocation is always an external event — a network packet or a function execution. It currently does not support triggering eBPF code execution on its own. eBPF supports "chaining", enabling calling multiple eBPF sequence, however, XDP currently supports attaching only one program per network interface. This limitation restricts XDP programs to being used only for specific tasks.

Traffic Control (TC) framework mentioned in Section 2.2.11 can further improve packet processing capabilities by controlling outgoing egress packets. Its features, such as `bpf_clone_redirect` allows cloning network packets and sending them through different interfaces, streamlining the process for sending identical network packets to multiple hosts.

The eBPF framework is still under active development. Similar to Linux, it has an active community of supporters who advocate for technology use by writing blog posts, hosting podcasts and even making an official documentary [117]. There are already numerous well-established eBPF projects [118], so the question is not whether the technology will succeed, but rather how it will be shaped by the future of Linux development.

7.3 Networking

Network packet processing, rewriting data and receiving traffic on other machines proved more complex than initially thought. It involved troubleshooting Internet checksum validation issues, network byte order and understanding IP packet structure. The initial decision to use Google Cloud Platform for development, in retrospect, was a costly mistake in terms of time and money spent on compute resources.

UDP, the network communication protocol choice for this project, enabled building a program which does not depend on establishing a network connection, unlike TCP. This significantly simplified developing required functionality and proved to be sufficient to implement basic Raft algorithm features. There have been past attempts at using UDP for Raft communication. One experiment uses UDP segments [119] to handle leader election, node failures and log replication. Another project, Aeron [120], uses UDP multicast and a custom, binary-based communication protocol to build a highly available and fault-tolerant Raft clusters.

Nearly all Raft algorithm implementations use the Transmission Control Protocol (TCP) which provides reliability guarantees (sequence numbers, acknowledgements) absent from UDP. As of 2024, there are no publications identifying TCP as a communication bottleneck in Raft deployments. Offloading networking features to eBPF can improve performance, but can also introduce novel issues.

XDP allows for rudimentary packet processing, but is inadequate to handle higher level protocols like HTTP or gRPC. There are examples of tracing HTTP requests by pattern-matching HTTP verbs (`GET`, `PUT`, etc.) in individual packets [121]. Similarly, there is an example of decoding encrypted data by using user space probes and using hooks attached to OpenSSL's `ssl_read()` and `ssl_write()` functions [122]. XDP excels in handling lower layer networking traffic. CloudFlare, a global content delivery company, demonstrated an XDP-based distributed denial-of-service (DDoS) mitigation technique, using `XDP_DROP` return code to discard around 10 million network packets per second [123].

A UDP packet, including both the header and payload, can be up to 1500 bytes in size. This corresponds to the Maximum Transmission Unit (MTU), a value which indicates the largest possible packet a network device will accept [124]. This limit leaves additional headroom to encode additional information, to provide full parity with the Raft algorithm specification. For the time being, this task will be left for the future.

7.4 Raft

This project has successfully demonstrated a novel method for exchanging heartbeats and performing leader elections for Raft consensus algorithm. The resulting project focuses on a narrow set of functionality and deliberately excludes parts related to handling data consistency. The project required making deliberate architectural choices to adapt the algorithm functionality to the eBPF environment.

At the beginning of this project, one of the options considered was to extend the existing Raft library for Rust [125] to include additional eBPF functionality. After brief evaluation, this approach was abandoned in favour of building basic functionality from scratch. Extending the existing library would have been significantly more work, due to its reliance on the TCP communication protocol. As of April 2024, there are no UDP-based Raft algorithm implementations in Rust programming language.

Raft consensus algorithm is built from few, well-defined components, and confirms the algorithm authors' intention to build an understandable consensus algorithm. At the same time, implementing additional features in eBPF is unlikely to make it simpler. It introduces additional failure modes, requiring substantially more work to reach the reliability of existing Raft open-source implementations. It is an open question if performance improvements demonstrated in Chapter 6 justify an increase in program complexity introduced by running kernel space program. Lowering election timeouts can lead to faster recovery, but, in practice, the decrease may go unnoticed, as clients can simply retry the request if the cluster is unavailable.

As such, the Raft implementation developed as part of this project remains in an experimental state and should be used for educational purposes.

Bibliography

[1] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[2] eBPF Foundation, “What is eBPF? An Introduction and Deep Dive into the eBPF Technology — ebpf.io,” https://ebpf.io/what-is-ebpf/, [Accessed 29-03-2024].

[3] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, “The express data path: Fast programmable packet processing in the operating system kernel,” in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.

[4] W. Bugden and A. Alahmar, “Rust: The programming language for safety and performance,” *arXiv preprint arXiv:2206.05503*, 2022.

[5] “Getting Started - Aya — aya-rs.dev,” https://aya-rs.dev/book/, [Accessed 30-03-2024].

[6] “eBPF verifier; The Linux Kernel documentation — docs.kernel.org,” https://docs.kernel.org/bpf/verifier.html, [Accessed 30-03-2024].

[7] “eBPF Linux Foundation Project — ebpf.foundation,” https://ebpf.foundation/, [Accessed 30-03-2024].

[8] “linux/include/uapi/linux/bpf.h at master · torvalds/linux — github.com,” https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h, [Accessed 30-03-2024].

[9] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating distributed protocols with eBPF," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23).* Boston, MA: USENIX Association, Apr. 2023, pp. 1391–1407. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/zhou

[10] L. Leslie, "The part-time parliament," *ACM Trans. on Computer Systems*, vol. 16, pp. 133–169, 1998.

[11] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 1–36, 2015.

[12] "Raft Consensus Algorithm — raft.github.io," https://raft.github.io/#implementations, [Accessed 30-03-2024].

[13] "6.5840 Lab 3: Raft — pdos.csail.mit.edu," https://pdos.csail.mit.edu/6.824/labs/lab-raft.html, [Accessed 30-03-2024].

[14] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[15] "GitHub - etcd-io/raft: Raft library for maintaining a replicated state machine — github.com," https://github.com/etcd-io/raft, [Accessed 16-04-2024].

[16] "Deep Dive — tikv.org," https://tikv.org/deep-dive/introduction/, [Accessed 30-03-2024].

[17] T. Pappas, "Raft & RethinkDB: A thorough Examination — cs.uoi.gr," https://www.cs.uoi.gr/~apappas/projects/Raft&Rethinkdb/, [Accessed 30-03-2024].

[18] "Building and deploying MySQL Raft at Meta — engineering.fb.com," https://engineering.fb.com/2023/05/16/data-infrastructure/mysql-raft-meta/, [Accessed 30-03-2024].

[19] "EDB Postgres Distributed (PGD) v5 - Raft elections in depth — enterprisedb.com," https://www.enterprisedb.com/docs/pgd/latest/routing/raft/04_raft_elections_in_depth/, [Accessed 30-03-2024].

[20] D. Ongaro, *Consensus: Bridging theory and practice.* Stanford University, 2014.

[21] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993, pp. 259–270.

[22] "About FreeBSD — freebsd.org," https://www.freebsd.org/about/, [Accessed 30-03-2024].

[23] "OpenBSD — openbsd.org," https://www.openbsd.org/, [Accessed 30-03-2024].

[24] "Home — TCPDUMP & LIBPCAP — tcpdump.org," https://www.tcpdump.org/index.html#documentation, [Accessed 30-03-2024].

[25] "LKML: Alexei Starovoitov: [PATCH net-next] extended BPF — lkml.org," https://lkml.org/lkml/2013/9/30/627, [Accessed 30-03-2024].

[26] "BPF: the universal in-kernel virtual machine [LWN.net] — lwn.net," https://lwn.net/Articles/599755/, [Accessed 30-03-2024].

[27] "Classic BPF vs eBPF; The Linux Kernel documentation — kernel.org," https://www.kernel.org/doc/html/v6.7/bpf/classic_vs_extended.html, [Accessed 30-03-2024].

[28] "BPF Features by Linux Kernel Version — android.googlesource.com," https://android.googlesource.com/platform/external/bcc/+/master/docs/kernel-versions.md, [Accessed 30-03-2024].

[29] B. Gregg, "Brendan Gregg: Bio — brendangregg.com," https://www.brendangregg.com/bio.html, [Accessed 30-03-2024].

[30] "Flame Graphs — brendangregg.com," https://www.brendangregg.com/flamegraphs.html, [Accessed 30-03-2024].

[31] B. Gregg, *BPF performance tools*, ser. Addison-Wesley Professional Computing Series. Boston, MA: Addison Wesley, Mar. 2020.

[32] ——, *Systems performance*, 2nd ed., ser. Addison-Wesley Professional Computing Series. Upper Saddle River, NJ: Pearson, Jan. 2021.

[33] "Q&A: Neil Thompson on computing power and innovation — news.mit.edu," https://news.mit.edu/2022/neil-thompson-computing-power-innovation-0624, [Accessed 16-04-2024].

[34] "Linux Performance — brendangregg.com," https://www.brendangregg.com/linuxperf.html, [Accessed 25-04-2024].

[35] "Linux Statistics 2024 - TrueList — truelist.co," https://truelist.co/blog/linux-statistics/, [Accessed 30-03-2024].

[36] "Meta, Google, Isovalent, Microsoft and Netflix Launch eBPF Foundation as Part of the Linux Foundation - Linux Foundation — linuxfoundation.org," https://tinyurl.com/4bua4h6b, [Accessed 30-03-2024].

[37] "Charter; eBPF — ebpf.foundation," https://ebpf.foundation/charter/, [Accessed 30-03-2024].

[38] "Projects; eBPF — ebpf.foundation," https://ebpf.foundation/projects/, [Accessed 30-03-2024].

[39] "Cilium - Cloud Native, eBPF-based Networking, Observability, and Security — cilium.io," https://cilium.io/, [Accessed 30-03-2024].

[40] "What Is eBPF? — IBM — ibm.com," https://www.ibm.com/topics/ebpf, [Accessed 30-03-2024].

[41] "bpf-helpers(7) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man7/bpf-helpers.7.html, [Accessed 30-03-2024].

[42] "BPF maps &x2014; The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/bpf/maps.html, [Accessed 30-03-2024].

[43] "What is eBPF? eBPF — ebpf.foundation," https://ebpf.foundation/what-is-ebpf/, [Accessed 30-03-2024].

[44] "Kernel modules; The Linux Kernel documentation — linux-kernel-labs.github.io," https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html, [Accessed 30-03-2024].

[45] "Liz Rice on Programming the Linux Kernel with eBPF, Cilium and Service Meshes — infoq.com," https://www.infoq.com/podcasts/liz-rice-ebpf/, [Accessed 30-03-2024].

[46] "BPF CO-RE reference guide — nakryiko.com," https://nakryiko.com/posts/bpf-core-reference-guide/, [Accessed 30-03-2024].

[47] "BPF Portability and CO-RE · BPF — facebookmicrosites.github.io," https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html#high-level-bpf-co-re-mechanics, [Accessed 30-03-2024].

[48] "BPF Type Format (BTF); The Linux Kernel documentation — kernel.org," https://www.kernel.org/doc/html/latest/bpf/btf.html, [Accessed 30-03-2024].

[49] "[v6,bpf-next,04/14] libbpf: implement BPF CO-RE offset relocation algorithm - Patchwork — fb.com," https://patchwork.ozlabs.org/project/netdev/patch/20190807214001.872988-5-andriin@fb.com/#2233012, [Accessed 30-03-2024].

[50] "libbpf Overview; The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/bpf/libbpf/libbpf_overview.html, [Accessed 30-03-2024].

[51] "ebpf: Revolutionizing security and observability in 2023 — f5.com," https://www.f5.com/company/blog/ebpf-revolutionizing-security-and-observability-in-2023, [Accessed 30-03-2024].

[52] "Kernel Probes (Kprobes); The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/trace/kprobes.html, [Accessed 30-03-2024].

[53] "Uprobe-tracer: Uprobe-based Event Tracing; The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/trace/uprobetracer.html, [Accessed 30-03-2024].

[54] "Using user-space tracepoints with BPF [LWN.net] — lwn.net," https://lwn.net/Articles/753601/, [Accessed 30-03-2024].

[55] "eBPF for Windows: Main Page — microsoft.github.io," https://microsoft.github.io/ebpf-for-windows/, [Accessed 30-03-2024].

[56] M. Kerrisk, *The Linux Programming Interface.* San Francisco, CA: No Starch Press, Jan. 2010.

[57] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems.* North Charleston, SC: Createspace Independent Publishing Platform, Sep. 2018.

[58] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA: O'Reilly Media, Nov. 2005.

[59] "Home - Aya — aya-rs.dev," https://aya-rs.dev/, [Accessed 30-03-2024].

[60] "Blixt - A load-balancer written in Rust, using eBPF, born from Gateway API — kubernetes.dev," https://www.kubernetes.dev/blog/2024/01/08/blixt-load-balancer-rust-ebpf-gateway-api/, [Accessed 30-03-2024].

[61] "bpfman — bpfman.io," https://bpfman.io/main/, [Accessed 30-03-2024].

[62] "GitHub - aya-rs/aya: Aya is an eBPF library for the Rust programming language, built with a focus on developer experience and operability. — github.com," https://github.com/aya-rs/aya?tab=readme-ov-file#overview, [Accessed 30-03-2024].

[63] "no_std - The Embedded Rust Book — docs.rust-embedded.org," https://docs.rust-embedded.org/book/intro/no-std.html#summary, [Accessed 30-03-2024].

[64] "GitHub - matklad/cargo-xtask — github.com," https://github.com/matklad/cargo-xtask, [Accessed 30-03-2024].

[65] "Hello XDP! - Aya — aya-rs.dev," https://aya-rs.dev/book/start/hello-xdp/#starting-out, [Accessed 30-03-2024].

[66] "Architecture; OrbStack Docs — docs.orbstack.dev," https://docs.orbstack.dev/architecture, [Accessed 30-03-2024].

[67] "tc-bpf(8) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man8/tc-bpf.8.html, [Accessed 30-03-2024].

[68] "Linux_4.8 - Linux Kernel Newbies — kernelnewbies.org," https://kernelnewbies.org/Linux_4.8#Support_for_eXpress_Data_Path, [Accessed 30-03-2024].

[69] H. Liu, "tc/BPF and XDP/BPF — liuhangbin.netlify.app," https://liuhangbin.netlify.app/post/ebpf-and-xdp/, [Accessed 30-03-2024].

[70] "eBPF XDP — tigera.io," https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/, [Accessed 30-03-2024].

[71] “bpf.h - include/uapi/linux/bpf.h - Linux source code (v6.5.7) - Bootlin — elixir.bootlin.com,” https://elixir.bootlin.com/linux/v6.5.7/source/include/uapi/linux/bpf.h#L6203, [Accessed 30-03-2024].

[72] “Going from Packet Where Aren’t You to pwru — cilium.io,” https://cilium.io/blog/2023/03/22/packet-where-are-you/, [Accessed 30-03-2024].

[73] “xdpdump: a simple tcpdump like tool for capturing packets at the XDP layer — xdp-tools System Administration — Man Pages — ManKier — mankier.com,” https://www.mankier.com/8/xdpdump, [Accessed 30-03-2024].

[74] K. Dalal, “Introducing Walmart’s L3AF Project: XDP based packet processing at scale — medium.com,” https://medium.com/walmartglobaltech/introducing-walmarts-l3af-project-xdp-based-packet-processing-at-scale-81a13ff49572, [Accessed 30-03-2024].

[75] “Parsing packets - Aya — aya-rs.dev,” https://aya-rs.dev/book/start/parsing-packets/, [Accessed 30-03-2024].

[76] P. Salva-Garcia, R. Ricart-Sanchez, E. Chirivella-Perez, Q. Wang, and J. M. Alcaraz-Calero, “Xdp-based smartnic hardware performance acceleration for next-generation networks,” *Journal of Network and Systems Management*, vol. 30, no. 4, p. 75, 2022.

[77] P. P. Waskiewicz Jr and N. Parikh, “Accelerating xdp programs using hw-based hints,” 2018.

[78] R. Braden, D. Borman, and C. Partridge, “Rfc1071: Computing the internet checksum,” 1988.

[79] “Discord - A New Way to Chat with Friends & Communities — discord.com,” https://discord.com/channels/855676609003651072/855676609003651075/1208849958971908186, [Accessed 07-04-2024].

[80] Tuetuopay, “Use veth pairs to develop xdp programs, because you can then tcpdump the other end of the pipe,” https://discord.com/channels/855676609003651072/855676609003651075/1150507802733842593, [Accessed 30-03-2024].

[81] "I'm not receiving packets using XDP_TX — stackoverflow.com," https://stackoverflow.com/questions/72120362/im-not-receiving-packets-using-xdp-tx/72159940#72159940, [Accessed 30-03-2024].

[82] "RFC 1700: Assigned Numbers — rfc-editor.org," https://www.rfc-editor.org/rfc/rfc1700, [Accessed 17-04-2024].

[83] M. Kleppmann, *Designing data-intensive applications.* Sebastopol, CA: O'Reilly Media, Mar. 2017.

[84] M. R. van Steen, *Distributed systems*, 2017.

[85] "Using Gossip Protocols for Failure Detection, Monitoring, Messaging and Other Good Things - High Scalability - — highscalability.com," https://highscalability.com/using-gossip-protocols-for-failure-detection-monitoring-mess/, [Accessed 17-04-2024].

[86] "Apache Cassandra — Apache Cassandra Documentation — cassandra.apache.org," https://cassandra.apache.org/_/index.html, [Accessed 17-04-2024].

[87] "Home Page — riak.com," https://riak.com/index.html, [Accessed 17-04-2024].

[88] "Consensus Protocol — Raft — Consul — HashiCorp Developer — developer.hashicorp.com," https://developer.hashicorp.com/consul/docs/architecture/consensus, [Accessed 31-03-2024].

[89] "XDP Tutorial - bpfman — bpfman.io," https://bpfman.io/v0.4.0/developer-guide/xdp-overview/, [Accessed 17-04-2024].

[90] Datadog, "A gentle introduction to XDP — datadoghq.com," https://www.datadoghq.com/blog/xdp-intro/, [Accessed 31-03-2024].

[91] https://www.cloudflare.com/en-gb/learning/ddos/glossary/open-systems-interconnection-model-osi/, [Accessed 31-03-2024].

[92] Deland-Han, "The three-way handshake via TCP/IP - Windows Server — learn.microsoft.com," https://learn.microsoft.com/en-us/troubleshoot/

windows-server/networking/three-way-handshake-via-tcpip, [Accessed 31-03-2024].

[93] "Everything you ever wanted to know about UDP sockets but were afraid to ask, part 1 — blog.cloudflare.com," https://blog.cloudflare.com/everything-you-ever-wanted-to-know-about-udp-sockets-but-were-afraid-to-ask-part-1/, [Accessed 31-03-2024].

[94] david wong, "Problems that UDP and only UDP has — cryptologie.net," https://www.cryptologie.net/article/449/problems-that-udp-and-only-udp-has/, [Accessed 31-03-2024].

[95] "Generic XDP [LWN.net] — lwn.net," https://lwn.net/Articles/720072/, [Accessed 17-04-2024].

[96] "BPF_MAP_TYPE_HASH, with PERCPU and LRU Variants; The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/bpf/map_hash.html, [Accessed 18-04-2024].

[97] "bpf(2) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man2/bpf.2.html, [Accessed 31-03-2024].

[98] "icu_calendar - Rust — docs.rs," https://docs.rs/icu_calendar/latest/icu_calendar/, [Accessed 31-03-2024].

[99] "ktime accessors; The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/core-api/timekeeping.html, [Accessed 31-03-2024].

[100] "Announcing Axum — Tokio - An asynchronous Rust runtime — tokio.rs," https://tokio.rs/blog/2021-07-announcing-axum, [Accessed 31-03-2024].

[101] "Understanding Ownership - The Rust Programming Language — doc.rust-lang.org," https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html, [Accessed 31-03-2024].

[102] "References and Borrowing — doc.rust-lang.org," https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html, [Accessed 31-03-2024].

[103] “Validating References with Lifetimes - The Rust Programming Language — doc.rust-lang.org,” https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html, [Accessed 31-03-2024].

[104] “Global mutable HashMap in a library — stackoverflow.com,” https://stackoverflow.com/a/34832819/19067718, [Accessed 31-03-2024].

[105] “Arc in std::sync - Rust — doc.rust-lang.org,” https://doc.rust-lang.org/std/sync/struct.Arc.html, [Accessed 31-03-2024].

[106] “Rust’s Rules Are Made to Be Broken — warp.dev,” https://www.warp.dev/blog/rules-are-made-to-be-broken#reference-counting-to-the-rescue, [Accessed 31-03-2024].

[107] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[108] “Other reprs - The Rustonomicon — doc.rust-lang.org,” https://doc.rust-lang.org/nomicon/other-reprs.html#reprc, [Accessed 31-03-2024].

[109] “Pod in aya - Rust — docs.aya-rs.dev,” https://docs.aya-rs.dev/aya/trait.pod, [Accessed 31-03-2024].

[110] “missing btf func_info whilst using bpf_loop() · Issue 521 · aya-rs/aya — github.com,” https://github.com/aya-rs/aya/issues/521, [Accessed 07-04-2024].

[111] https://linux.die.net/man/1/stress, [Accessed 05-04-2024].

[112] “Beej’s Guide to Network Programming — beej.us,” https://beej.us/guide/bgnet/html/#what-is-a-socket, [Accessed 06-04-2024].

[113] “RwLock in std::sync - Rust — doc.rust-lang.org,” https://doc.rust-lang.org/std/sync/struct.RwLock.html, [Accessed 13-04-2024].

[114] “Channels - Rust By Example — doc.rust-lang.org,” https://doc.rust-lang.org/rust-by-example/std_misc/channels.html, [Accessed 21-04-2024].

[115] G. V. Vlasselaer, “Linux: The world’s largest open-source project — mrwaggel.be,” https://mrwaggel.be/post/Linux-The-Worlds-Largest-Open-Source-Project, [Accessed 01-04-2024].

[116] "The good, bad, and compromisable aspects of linux ebpf — pentera.io," https://pentera.io/blog/the-good-bad-and-compromisable-aspects-of-linux-ebpf/, [Accessed 01-04-2024].

[117] "eBPF Documentary - Speakeasy Productions — ebpfdocumentary.com," https://ebpfdocumentary.com/, [Accessed 01-04-2024].

[118] "eBPF Applications Landscape — ebpf.io," https://ebpf.io/applications/, [Accessed 01-04-2024].

[119] "GitHub - EmanueleGallone/RaftUDP: Distributed Raft Consensus algorithm implementation using UDP segments. — github.com," https://github.com/EmanueleGallone/RaftUDP, [Accessed 01-04-2024].

[120] "aeron/aeron-cluster at master · real-logic/aeron — github.com," https://github.com/real-logic/aeron/tree/master/aeron-cluster, [Accessed 01-04-2024].

[121] "L7 Tracing with eBPF: HTTP and Beyond via Socket Filters and Syscall Tracepoints - eunomia — eunomia.dev," https://eunomia.dev/tutorials/23-http/#capturing-http-traffic-with-ebpf-socket-filter, [Accessed 01-04-2024].

[122] yunwei37, "eBPF Practical Tutorial: Capturing SSL/TLS Plain Text Using uprobe — yunwei356," https://medium.com/@yunwei356/ebpf-practical-tutorial-capturing-ssl-tls-plain-text-using-uprobe-fccb010cfd64, [Accessed 01-04-2024].

[123] "How to drop 10 million packets per second — blog.cloudflare.com," https://blog.cloudflare.com/how-to-drop-10-million-packets, [Accessed 01-04-2024].

[124] "What is mtu (maximum transmission unit)?" https://www.cloudflare.com/en-gb/learning/network-layer/what-is-mtu/, [Accessed 21-04-2024].

[125] "GitHub - tikv/raft-rs: Raft distributed consensus algorithm implemented in Rust. — github.com," https://github.com/tikv/raft-rs, [Accessed 01-04-2024].

[126] "GitHub - NaurisSadovskis/raft-ebpf-experiment — github.com," https://github.com/NaurisSadovskis/raft-ebpf-experiment, [Accessed 13-04-2024].

A Project code

The final project code and test data are available in the following public GitHub.com `git` repository:

- `NaurisSadovskis/raft-ebpf-experiment` [126].

Relevant data can be found in the following directories:

1. `raft/` - eBPF and user-space code for Chapters 3, 4 & 5.
2. `measurements/heartbeats/` - test data for Chapter 6.
3. `diagrams/` - application architecture diagrams.