Ciena Field Session XDP Report

Vincent Morgan
Fisher Darling
Casey Turner
Ethan Bond

Introduction

Ciena

Ciena is a networking company based in North America that produces hardware products to employ many various networking technologies as well as software products to assist in the deployment and maintenance of the hardware. Ciena's products and services are used by some of the largest telecommunications companies in the world.

Product Vision

Traditionally, much of networking maintenance requires dedicated hardware to monitor the frames and their reliability. With the latest improvements in low cost computing, it's possible to do this with servers running linux. Ciena would like to explore the possibilities of ethernet frame processing software that can run on an x86 processor. The primary goal of this project was, using eBPFs (extended Berkeley Packet Filters) and XDP (eXpress Data Path), to implement various Organization, Administration and Management protocols, or OAM protocols, to perform measurements in a network that can be run on various linux systems/containers.

Technical Jargon:
Throughout this technical report, there are many instances of technical jargon, such as OAM, eBPF, XDP, etc. Please refer to the appendix for definitions of common terms.

Requirements

Functional

Primarily, the software must provide an implementation of various OAM protocols that will report back to the user statistics about the network. The software must be interactable in some way during execution, either directly through the Command-Line Interface (CLI) or an Application Programming Interface (API) for the running process.

Specific requirements of the project are as follows:

- Must implement ping, Bi-directional Forward Detection (BFD), and other protocols detailed in Y.1731.
- With the implemented protocols, statistics such as RTT, latency (jitter), and Uptime/Downtime should be reported to the user

Non-functional

- Software must be a fairly high-performance networking application allowing for millions of packets per second per CPU thread.
- Software will be tested in multiple network environments (virtual, hardware, container)
- Software will be implemented in language that provides BPF Compiler Collection (BCC) package along with C code for the BPF filters.
- This software must be executable in a linux kernel of version 4.3 or higher.
- The software ultimately be a compiled binary that is executable on most/all x86 processors
- Software should be open source.

Definition of Done

- Code has been written
- Inline comments in code
- Documentation written outside of code
- Unit tests passed
- Implementation tests passed

System Architecture

The requirements of this project naturally split our design into two distinct sub-projects operating separately within kernel-space and user-space. The user-space application, written in Go, is used to manage and interface with the user. The kernel-space application, written in C, is responsible for directly interfacing with incoming network packets and sending the information to the user-space program. This communication takes place using a special data structure called an eBPF map and the user can use the command line to interact with the program.

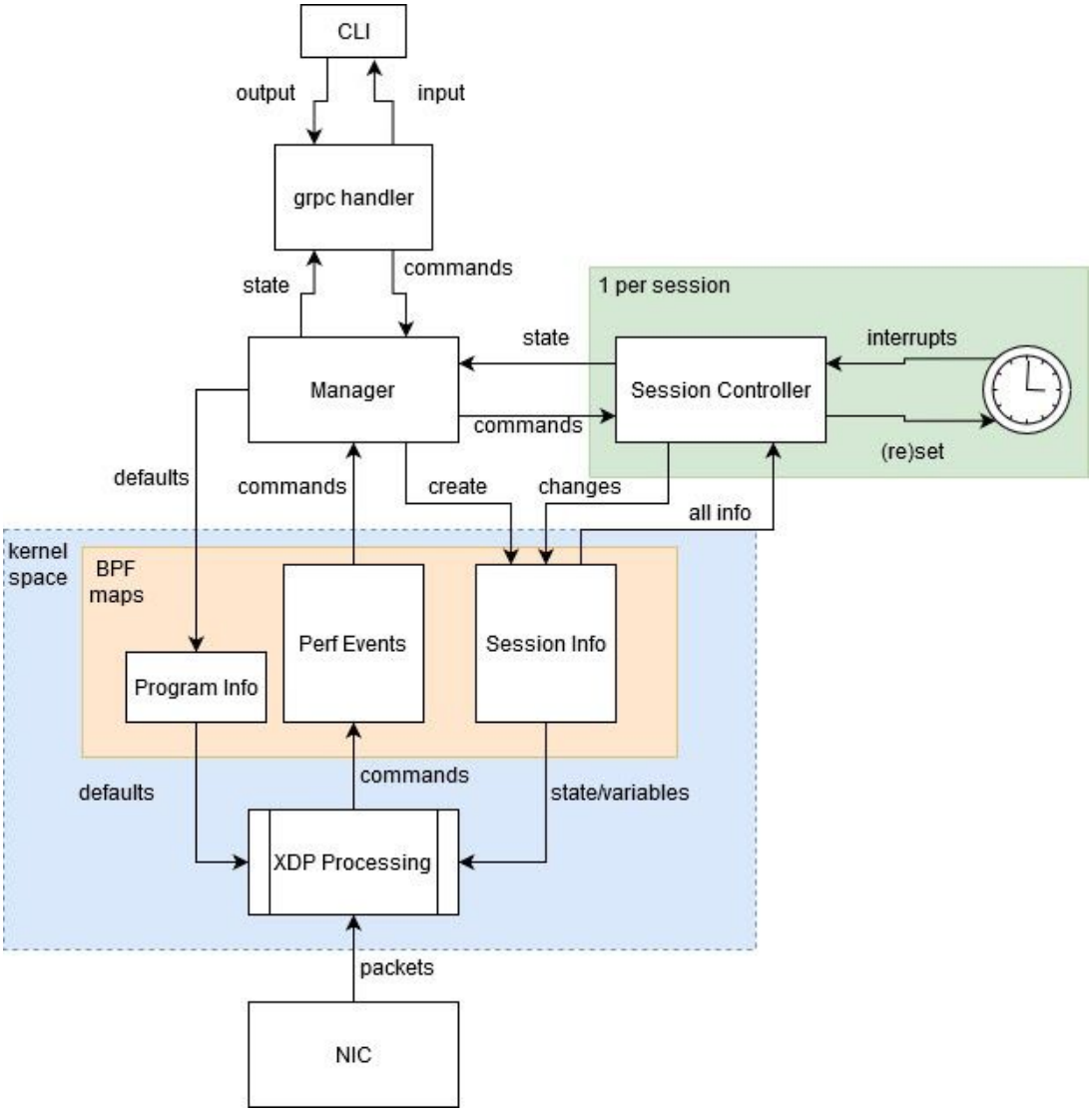

Figure 1: Design Overview

Kernel Space

The kernel program is an eBPF program that runs on a virtual machine inside the kernel. The eBPF program is written in a restricted subset of C (unable to use certain aspects such as loops) and is then compiled into BPF bytecode. This program is responsible for filtering packets and then reporting data from them to the userspace. Using XDP, the program can drop/redirect packets before they reach the kernel or just pass the packets to the kernel to be processed normally. Using BPF Maps and Perf Events, the eBPF program can communicate directly with the userspace to pass data about incoming packets. BPF maps are also used to implement stateful BPF programs, which allows for stateful network protocols, such as Bi-directional Forwarding Detection (BFD) to be used.

Using this method for computer networking, it allows arbitrary code to run inside the kernel so that direct data coming from the interface can be used as well as achieving high performance while running on any Linux computer instead of dedicated hardware.

Userspace

The core component of the Ciena OAM architecture is the userspace Go program. This Go program receives various commands for controlling the OAM protocols running on its associated Network Interface Cards (NICs). For example, one can command the “manager” to start pinging a specific IP address every X milliseconds. The manager will then collect the associated data with the ping protocol, such as Time to Live (TTL) and Round-Trip Time (RTT). Additionally the BFD protocol has been implemented in the program.

Technical Design

Design of the eBPF XDP Program

The eBPF XDP program, referred to as the kernel program above, has a couple different purposes. It needs to parse the packet and make a decision based on the information it finds and respond to any packets it should in an efficient manner.First does a series of checks when a packet is received. These checks are to avoid segmentation faults and help with parsing the packets. The first checks done are on the header sizes, making sure that the ethernet headers, IP headers, and UDP headers/ports are correct. The next large check the program does is on the size of the UDP header, which determines whether the packet is a control packet or an echo packet. If the packet is an echo packet, another check ensues, this time checking if the packet is a request or a reply. If the echo packet is a reply, additional checks are done and a perf event notifies the manager. If the echo packet is a request, the program flips all necessary headers/ports and redirects the packet back to the other host. If the original packet was a control packet instead of an echo packet, a different set of checks occurs. The control packet must be checked if it is either a poll packet or a final packet. If the control is a final packet, the manager is notified via perfevent and the packet is dropped. If the packet is a poll packet, the program must determine what has been changed, report it to the manager program, then redirect the packet.

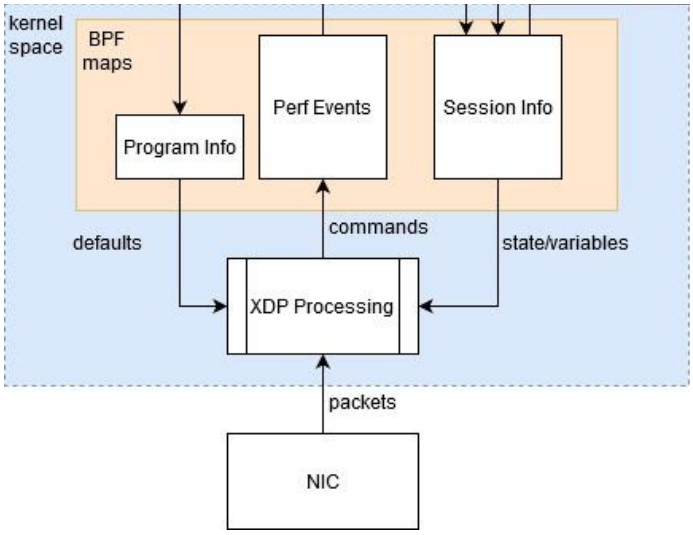

Figure 2: eBPF XDP Design Diagram

Design of the Manager Program

The manager program is the core communication and handler program of the Ciena OAM architecture. The manager controls the ebpf kernel program, running sessions, and reporting of

session data and status to users. The manager has three main components that it controls and communicates with. The first component is the communication with the client and server over a gRPC server. The second component is handling the lifecycle of a Session through a SessionController. The final component is the management of the eBPF program. Communication between the gRPC server and the manager, and also communication between individual SessionControllers are done through Go's channel type.

gRPC Server

gRPC is ***G***oogle's implementation of a ***R***emote ***P***rocedure ***C***all framework, described as a "A high-performance, open source universal RPC framework" that allows the sending and receiving of commands from one server to another. We use gRPC combined with ProtoBufs to describe the communication between the manager program and clients. ProtoBufs allow us to describe the API (Application Programming Interface) between the client and server with a file. This file can then be compiled into a language of the client's choice, allowing the client to write code that can communicate with the server in whatever language they would like. The client cli that we have created, which can create, read, and change the mode of a session, is also written with Go.

The manager program spins up a gRPC server that listens on port 5555. This server responds to requests that are described in the `./proto/bfd/bfd.proto` file. When the server receives a CreateSession request, the server locks the session map and initializes a new SessionController with the appropriate session data.

SessionController

The SessionController (SC) is the core userspace state-machine for the BFD Protocol. When a new BFD session is requested, a SC go-routine is created that begins to listen for PerfEvents that are sent from the server. These PerfEvents are events that are generated from the eBPF and denote state change for the BFD session. The active SC (the controller for the half that begins the session) begins to send the initial control packets to the server and awaits a response on a timeout. SC's also sends SessionInfo events back to the server on any state change. These events can then be further sent to a client that is listening to a stream of information for a session. The supports the use-case of being immediately notified if a session is down.

eBPF XDP Program Communication

The perf events sent by the XDP program are sent through a go channel to the user space server. Once the event has been received, the event's local discriminator in the perfevent is then used to route the event to the proper SC. The SC then uses that event to advance its internal state machine. For example, if the SC receives a perfevent with the final bit set, it knows to reset the receive timeout, since the other side successfully responded to a poll packet.

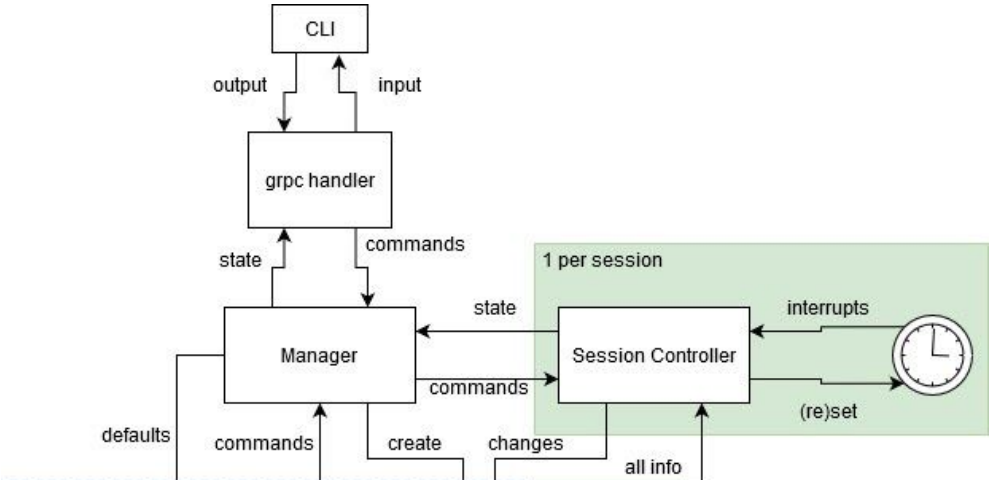

Figure 3: Manager Program Design Diagram

Bidirectional Forwarding Detection (BFD)

BFD is a common protocol used to determine if two links are connected and that connection is up and running. It begins with a starting handshake packet from the lead server. If the client server that receives the handshake wants to continue the session, it sends a response handshake packet that is interpreted as a PerfEvent in our Go program. The session begins in async mode, meaning that the client server repeatedly sends packets to the target server, asking the target server for a reply. If no reply is received after x requests, the connection is considered down. The other mode, demand mode, is much simpler. In this mode both servers repeatedly send echo packets to each other. If x packets are not received from one of the servers, the link is considered down. All of the timeouts and send timeouts are on the scale of a few milliseconds.

Quality Assurance

Unit Testing

- Verified user space program works
- Verified kernel space program
    - Test loading and passing verification
    - Check if programs are loaded with correct values
    - Verified maps are stored/loaded correctly
- Verified packets are crafted properly
    - Wireshark was used to view packets transferred between computers and verify their structure based on RFCs.
        - Ping - refer to RFC 792
        - BFD - refer to RFC 5880

Integration Testing

Overall, our integration testing framework required us to send and receive packets for an integration test. We used two different computers to run instances of our server and client programs respectively. From there, the programs reported perf events over the gRPC stream that were then verified.

Static and Dynamic Program Analysis

- Linting
    - Go lint
    - Clang-format
- Fuzzing for detecting parsing errors:
    - Fuzzing provides a method of discovering logic flaws with arbitrary input. For a program that deals with packet capturing and parsing, fuzzing is especially useful since there are a large number of possible packets and errors. Since this is an open-source program, there are programs from Microsoft and Google that provide free 24/7 fuzz testing.
- Valgrind for detecting memory leaks:
    - Long running programs need to be efficient and have an upper bound on memory usage. If we were to have any undetected memory leaks, then our program stability would be worsened. Even though Go is a garbage collected language, there can still be memory leaks.

Results

The primary purpose of our project was to implement network protocols using extended Berkeley Packet Filter (eBPFs) and eXpress Data Path (XDP) on a Linux system. We restricted this to Ping and BFD protocol, and implemented them utilizing the kernel space and user space design required by the technology. This necessitated the use of eBPF maps, which were successfully implemented into our project. Further, we also implemented the user interface using gRPC and a command line client. Overall our current project satisfies both functional requirements, the remaining non-functional requirements we initially laid out.

Testing Results

Following our testing plan, we have verified that multiple aspects of our program are functioning properly. The program is able to communicate properly with the command line using grpc and with the XDP program using perf events and BPF maps. All operations with these technologies are functioning properly. The XDP program is able to intercept packets properly, ignoring the ones not destined for our application, and parse the information correctly.

Future Work

- Adding additional protocols
    - Traceroute
    - Network layout information/routing protocol
- Control remotely via api instead of cmdline
    - Allows for more automation in larger environments
    - Could implement notification capability that could interface with other software.

Appendix

I. Common Terminology

- User-space
  - The non-privileged “space” where a typical program would run. This is typically where common applications such as FireFox run.
- Kernel-space
  - The privileged “space” in which operating system code executes. This code can do essentially anything to the operating system.
- OAM → Operations, Administration and Maintenance
  - Protocols for controlling link-layer devices or devices that handle the transmission of packets from A to B.
- eBPF → Extended Berkeley Packet Filter
  - Interface to data link layers. A small kernel subsystem that can execute code on each innovation of some kernel function. In our case, each time a packet arrives.
- XDP → eXpress Data Path
  - High performance data path within linux kernel using eBPFs. Allows one to execute code for each incoming packet before the kernel can.
- BFD → Bi-directional Forwarding Detection
  - Protocol to detect faults between forwarding engines. Can detect if a route between two servers has failed.
- CLI → Command Line Interface
  - A program that runs in a terminal and can be controlled through flags and command-line arguments or a config file.
- API → Application Programming Interface
  - Describes the methods, inputs, and outputs for interacting with a program or entity. Generally used in web applications for describing how one communicates with a backend server.
- gRPC → Google Remote Procedure Call
  - RPC framework that allows function calls and sending commands to remote server code.
- PerfEvents → eBPF Event Queue
  - PerfEvents are a method that eBPFs can use to communicate with the outside world (user-space). Events are appended to the end of a queue and the oldest event is read first (FIFO ordering).

II. Instructions for use

1. First steps are to clone the manager repository at:
   a. https://github.com/open-oam/manager_program

2. The repository contains three binaries:
    a. `client` for interacting with the server
    b. `server` that loads the eBPF and controls everything
    c. `xdp.elf` which is the primary eBPF program
3. Run `server -help` to view all of the flags for running the server. Now run the server binary with the proper flags on two different devices that have a route between them.
4. Next, on one server execute the client binary with the IP address of the other server:
    a. `./client -create -remote <other_ip_addr>`
    b. Write down the local discriminator of the session
5. You should now see on both servers, log messages describing that an async BFD session was created.
6. To switch to demand mode, run the following:
    a. `./client -change-mode DEMAND -disc <local_disc>`
7. To switch back to async mode:
    a. `./client -change-mode ASYNC -disc <local_disc>`
8. To stream state changes of a session:
    a. `./client -stream -disc <local_disc>`