CS60038: Assignment 2

Building a load balancer using eBPF and XDP.

(Submission Deadline: November 05, 2023 EOD)

1. You need to submit the assignment through CSE Moodle.
2. Only one member from each group should submit the assignment solution as a single ZIP file.
3. Mention the name and roll numbers of the group members in your submission.
4. Please document your implementation properly. You should include a README file explaining how to compile and run your code, as well as the test cases that you have tested.

Overview

In this assignment you will be building a load balancer which will control the traffic for multiple backends using eBPF and XDP. You will be using docker to maintain different virtual servers and communicate between them. Before starting the assignment it is important to familiarize yourself with eBPF, XDP and Docker.

eBPF
In the previous assignment you worked with LKMs where you wrote kernel space code. eBPF provides you with similar functionality, you can write custom code in the kernel space with the added advantage that you do not have to work with procfs like you did in the previous assignment. For instance, I can write eBPF code which constantly prints "Hello World" except that it will be run in the kernel space :)

- https://www.tigera.io/learn/guides/ebpf/
- https://medium.com/@aos2022rnn/ebpf-fantastic-network-i-o-speeds-and-where-to-find-them-1d83e2fd6b2f

XDP
A load balancer is used to manage packet traffic across multiple identical backends, each of which can process the packet individually. Suppose you have three backends A,B,C each of which can process a packet P. However A and B already have a lot of packets queued up which they are yet to process, so backend C is most suitable for handling this

packet P. It is a load balancer's job to decide which packet should be sent to which server. In case of web applications, you as a client actually do not send a request directly to a backend, you actually send it to the load balancer, which knows the addresses of all the backends and decides what to do with a packet. This is possible when a load balancer already exists as a part of the interface and the clients know the address of the load balancer.
However , the above rerouting can be slow. Hence comes **XDP**, which can perform the load balancing on the kernel level and make your packet processing pipeline faster.

- https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/
- https://www.datadoghq.com/blog/xdp-intro/

Docker and Docker networking

Instead of using virtual machines, you will be using docker containers which will replicate your backend servers, the client which sends the requests and your load balancer. You can communicate between containers using the Docker networking interface which allows you to specify the host and ports for the individual containers.

- https://www.ibm.com/topics/docker
- https://www.simplilearn.com/tutorials/docker-tutorial/docker-networking
- https://docs.docker.com/network/

Part A: Getting yourself familiarized

[20 Marks]

Objective: In this part, you will learn to create a packet dropper using eBPF and XDP. You will build a server application that accepts incoming connections and prints TCP packets as they arrive. A BPF program running on the server will filter TCP packets arriving on the machine for the port on which the server is running. The packets will contain a single byte of data, and the packet should be dropped if the TCP sequence number is even; otherwise, it should be sent to the server. You will also add print statements on the client and server sides to monitor data transmission and utilize trace print statements in the BPF program to understand its behavior.

Server Application

1. Set up a Docker container for development with eBPF.
2. Create a server application that binds to a specific port and accepts incoming TCP connections.
3. Implement the server to print received TCP packets data.

eBPF Program

1. Create an eBPF program in C that will be attached to the XDP hook on the server's network interface.
2. Implement the eBPF program to filter incoming TCP packets for only the port on which the server is running, the other packets should be allowed to directly pass through.
3. Modify the eBPF program to drop packets if the TCP sequence number is even and forward others.
4. Add trace print statements to the eBPF program to log its behavior.

Client Application

1. Setup a Docker container for the client application.
2. Implement the client to send TCP packets with a single byte of data to the server. Put some delay between consecutive sends so that the sequence number gets incremented in units of 1 and alternate packets get dropped.
3. Add print statements to the client to log the data being sent.

Grading Criteria

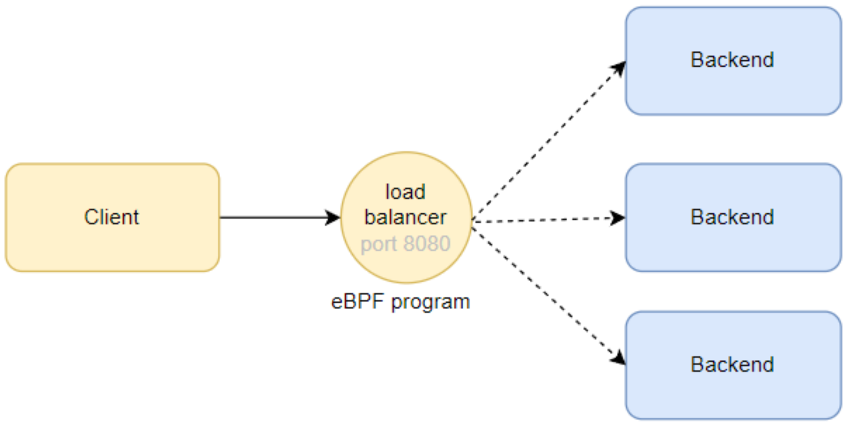
Your assignment will be evaluated based on the following criteria:

1. Correct implementation of the server, eBPF program, and client.
2. Proper handling of TCP packets.
3. Adequate use of print statements for monitoring and debugging.

Part B: Building a load balancer using XDP

[80 marks]

Objective: In this part, you will create a dynamic load balancing server which will distribute traffic between 3 workers based on the number of free working threads available on each worker server. A BPF program running on the load balancer will receive incoming packets from the client and send them to one of the 3 worker servers which has a free worker thread available. If more than one worker server has a free thread then you can decide the policy to distribute the workload in such cases. The workers communicate to the load balancer once the work is completed so it can know that a thread is free for work on this worker. In case none of the threads on any worker are free, the load balancer should have a functionality to queue the packet for sending later on when a thread becomes free.



Server Applications
There will be three identical backend servers in your system. Each server will have five worker threads which can process the packets. On receiving a packet, the thread processing it sleeps for the time in seconds sent in the data of the packet (this sleep duration is used to mimic processing of a packet). *After the sleep is over, it indicates to the load balancer that a thread is free* (more on this below).

Load Balancer Server
This will run our eBPF code which will intercept all packets sent to this server. If the packet is sent to a specific port (say 8080) it means it was meant to be sent to one of our backend servers and we will decide which server to send it to, otherwise we let the packet pass as it is.
For deciding the backend server we work as follows:-

1. Our eBPF program maintains **free_threads** for each backend , which will be initially five for each server as all threads are available.
2. When it receives a packet that requires to be forwarded to one of our backend servers, it sends it to any server which has an available thread and updates the free slots accordingly.
3. If no server has a free thread, then it adds the packet to a packet **queue** which stores delayed packets.
4. When it receives information from any of the backends indicating that a thread is free, then it first sends a packet from the queue if present, otherwise it updates the free thread count for that backend.

Client Application
The client application sends packets with some delay time in the data (choose a range of small values for this) to the designated port of the load balancer.

Inter-Container Communications
Here we highlight the different packets sent across the containers

Client -> Load Balancer : The client sends packets to the load balancer which it wants any of the backends to process. These packets need to be sent to a predefined port which will indicate to the load balancer that these packets need to be sent to one of the backends, all other packets are simply passed normally through the load balancer to their destination.
Server -> Load Balancer : Once a thread has finished processing a request, the server sends a packet to the load balancer to indicate there is a free thread. The data in this is irrelevant, simply the receiving of a packet by a server will indicate to the load balancer that the server has a free thread.
Load Balancer -> Server : The load balancer will send the packet to the free server chosen as discussed earlier. The packet headers and their checksums will be changed accordingly by the load balancer.

Part C: Testing

[50 marks]

For Part A the testing is quite simple. You need to show the following:

1. Print the data byte being sent in each packet on the client side (sent at a substantial delay so that it is easy to observe).
2. In the eBPF print statements show each packet being received, the sequence number and whether it will be dropped or not.
3. Print the data byte being received on the server side.

For Part B you need to show the following:

1. Print the data byte (sleep time requested) on the client side (sent at a substantial delay so that it is easy to observe).
2. In the eBPF print statements show each packet being received and to which server it is being forwarded or if it is being queued. Also print an appropriate message when a worker sends a packet indicating a thread has been freed and the number of free threads available at this moment. There should also be a message when a packet which had previously been queued is sent out.
3. On each of the servers print each received packet and after each packet has been processed (sleep complete) print the total amount of time slept till now.

For Part B conduct the testing with high sleep times and low sleep times. The wait queue feature will only be triggered when the sleep times are substantially high and all the threads are busy simultaneously.