Implementing a Maglev Load Balancer in the Linux Kernel

A Step Towards 5G

Submitted by
Jacob Hillman

Advisor: Professor Y. Richard Yang
Yale University
1 Abstract

This project provides an implementation for Google’s Maglev load balancer [2] in the Linux kernel space. The Maglev load balancer introduces consistent hashing and connection tracking to better handle unexpected faults and failures. Programming in the kernel space provides direct access to IP traffic as it arrives at a server, removing the need for costly communication with the user space. In the context of an improved 5G architecture, the load balancer is responsible for routing traffic to virtual containers containing VNF chains.

2 Introduction and Background

The Evolved Packet Core (EPC), the cellular core used for today’s current implementations of 4G/LTE, will soon be incapable of adapting to a rapidly increasing amount of data traffic. Modern technologies have led to more data intensive mobile applications, sharing of higher quality content and media, and a rapid increase in the number of IoT devices that funnel traffic through the EPC. Demand for a more efficient cellular core is just one driving force behind the push for the development of an improved 5G network architecture. To meet the requirements necessary to classify a network as 5G [4] [5], the EPC must be redesigned to be more scalable, modular, and to reduce latency. Redesigning the EPC leveraging network function virtualization (NFV) can greatly improve performance in all areas. This project focuses on the orchestration of virtual network functions (VNF’s) through the implementation of a crucial orchestration component: the load balancer. This paper will first contextualize the need for innovations towards a 5G architecture by outlining the current 4G architecture. It will then provide an overview of the Maglev load balancer and describe its implementation in the Linux kernel.

2.1 EPC and Current Network Architecture

The basic flow of network traffic is shown in figure 1. Traffic originates from a user entity (UE), which could be any device that connects to the internet, and follows to the Evolved Node B (eNodeB). The eNodeB is part of the Radio Access Network (RAN), the other main component of the architecture besides the EPC. Traffic from the eNodeB is forwarded to the EPC, and is then sent to its final destination (e.g. a server on the internet).
The EPC is composed of network functions, also referred to as middle-boxes. The composition of network functions within an EPC varies by instance of EPC, but there are three cellular-specific functions: Mobile Mobility Entity (MME), Serving Gateway (SGW), and Packet Gateway (PGW). MME handles signaling traffic from the eNodeB. When a new device connects to an eNodeB, its traffic is in the form of an attach request. The attach request is forwarded to the MME, which authenticates the user and provides mobility and session management. SGW and PGW process all data traffic once a connection has been established.

### 2.2 Network Function Virtualization

A Network Function (NF) is broadly defined as a functional building block within a network architecture. Each part of the EPC is an NF. In figure 1, MME, SGW, and PGW are all network functions. Another recognizable NF is Firewall. Network functions are connected to one another within the EPC, and traffic follows a specified path through a chain of NFs. Original implementations of network functions depended on dedicated proprietary hardware. Proprietary hardware is expensive to maintain and create, difficult to expand, and less practical to optimize. Network Function Virtualization (NFV) decouples network functionality from proprietary hardware by utilizing cloud-native architecture. Using function virtualization to create abstractions for easy implementation, cellular core architectures can be run on general-purpose servers. There are several benefits of using general-purpose servers, including flexibility, dynamicity, on-demand features, and easier network management.
2.3 5G

2.3.1 Goals

The development of a new 5G architecture is guided by a broad set of goals outlined by 5G white papers from various sources [4] [5]. Most descriptions of the ambitions of 5G include the following common characteristics:

1. <1 ms latency
2. >10 Gbps peak data
3. Up to 100x more devices than 4G
4. 10,000x more traffic

Overall, these goals contribute to an ultra reliable network. If achieved, these unprecedented network capabilities would unlock use cases such as remote robotic surgeries (due to ultra low latency) and self-driving cars (due to ultra high availability).

2.3.2 Network Slicing

Several novel frameworks have been proposed to achieve the goals of 5G, including PEPC [8], E2 [6], and SIMPLE [7]. These architectures all utilize network slicing and NFV in their solutions. Network slices utilize cloud-native architecture to group NFs, resources, and connection relationships [4]. The virtualization of NFs allows multiple custom configurations of function chains to exist on a server, with each chain of NFs on its own slice. This allows a network operator to configure slices for specific use cases. For example, an IoT device requires different VNFs than a cellular phone. Passing traffic from an IoT device directly to a chain that is specifically suited for this use case (one that likely has fewer VNFs) can decrease network load and increase availability.

The orchestrator has the responsibility of creating network slices and routing traffic to its corresponding slice. In this project, we implement a load balancer, which is the first step in directing traffic to the correct network slice.
Figure 2: Maglev Control Flow

3 Maglev

The core purpose of Google’s Maglev load balancer is not much different than a typical load balancer. However, in response to massive network traffic, Google optimized Maglev to increase its reliability and consistency. The control flow of the load balancer is shown in figure 2 and is described step-by-step below.

1. The packet enters the load balancer and a 5-tuple hash\(^1\) is calculated. The load balancer immediately tries to match this hash to a configured virtual IP (VIP). If no match is found, the packet is dropped.

2. The load balancer looks up the hash value in the connection tracking table (see 3.2). If no match is found in the connection tracking table, a backend destination is assigned to this hash using consistent hashing (see 3.1) and added to the connection tracking table.

3. The packet is sent to the destination after the back-end is selected.

3.1 Consistent Hashing

Consistent hashing evenly distributes load across destination back-end services by giving each destination the same number of entries in the hash lookup table. Therefore, each 5-tuple hash value has an equal likelihood of

\(^1\) A 5-tuple hash refers to the source IP, source port, destination IP, destination port, and IP protocol number of the packet
being assigned to any particular back-end. The Maglev specific goes into more detail about how this is achieved [2].

### 3.2 Connection Tracking Table

A back-end address is added to the connection tracking table when it is selected from the lookup table. This ensures all packets from the same source arrive at the same back-end server. This is important when the lookup table changes due to server issues such as overload. Even though the makeup of the lookup table may change, incoming traffic that has already been assigned a back-end address will continue to be sent there. This is critical for stateful protocols such as TCP.

### 4 Implementation

The Maglev load balancer is implemented as a Linux kernel module. The module is written in C and utilizes the Netfilter library to access the system’s network interface. The load balancer intercepts traffic in the kernel’s PRE-ROUTING stage, and assigns the packet’s new destination address based on the Maglev algorithm.

#### 4.1 The Kernel

The kernel is the foundation of every operating system. It’s the bridge between software and hardware, almost like an API for accessing computer resources such as a network interface controllers and disk drives. The kernel is the first part of an operating system that is initialized when a computer boots up. Changes can be made to the kernel through programs called modules, which can be added into the kernel either during or after initialization. In this project, the load balancer module is added into the kernel after initialization using the command insmod. [1]

#### 4.2 Development Environment

Programming Linux kernel modules (LKMs) is more risky and difficult than traditional userspace programming. Mistakes in the code cannot be easily
found using print statements (which is especially troublesome when program-
mming in C), and incorrect code that is loaded to the module can potentially
corrupt the operating system. Additionally, the computer device itself must
be restarted to remove an incorrectly coded module. Due to this, LKM
programmers typically develop in virtual environments. I used VirtualBox
to create a VM running Ubuntu 18.04 and Linux 4.18 and did most of my
programming in VSCode.

4.3 Netfilter
The kernel is responsible for handling network packets and is interrupted
when a packet arrives. There are several steps in the kernel protocol for
handling a packet. The Netfilter package gives access to each one of these
steps through the use of netfilter hooks. A hook is a way to give a callback
function to each one of these different stages. When a packet arrives at a
stage, is is passed to the callback function associated with that stage. In
the context of a load balancer, we are most interested in the very first stage,
called the PREROUTING stage, because we want to change the destination
information of the packet before it’s delivered to user space. This implemen-
tation uses a netfilter hook to change packet destination information as soon
as it arrives to the kernel. [3]

5 Future Work
This project is just one small step towards network orchestration. Network
orchestrators have to make more complex decisions on where to send traffic.
For example, the orchestrator must determine what type of network slice each
particular packet should go to, and load balance among slices that match that
requirement. In terms of this implementation of the Maglev load balancer
specifically, more immediate goals include support for multiple VIPs and
support for protocols other than UDP. Also, this implementation does not
include support for health checks, where the load balancer would be aware
of the status of each of the back-ends and act accordingly if a back-end is
overloaded or goes down.
6 Acknowledgements

I want to extend my gratitude to my advisor, Yang Richard Yang. Professor Yang has been an excellent mentor for me throughout my time at Yale and has provided me with an immense amount of knowledge and motivation. I would also like to acknowledge Sean Walker, an incredible programmer and friend who was a joy to collaborate with during this project.

References


[5] Nokia. 5g masterplan - five keys to create the new communications era. 2016.
