Design and Implementation of a Packet Sniffing Library

Olaniyi A. Ayeni, Elijah Oyekunle, Omoyele A. Odeniyi
School of Computing, Federal University of Technology, Akure, Ondo State. Nigeria

Abstract

This research presents a cross-platform, extensible packet sniffing library that provides an easy-to-use Application Programming Interface (API) for use in developing network applications. Our motivation for this research is the fact that there are numerous packet processing frameworks in use today, and providing support for these frameworks introduces additional complexity into the work of the application developer, especially when numerous operating systems have to be supported. Specific objective of this research is to design and implement an easy-to-use library that can help with integrating various packet processing frameworks based on common APIs into various applications to run on any operating system. This system is implemented using the Go programming language due to its native support for concurrency, memory safety, high-performance networking and multiprocessing features and runtime efficiency. It provides support for reading live from network devices, and also offline reading from files. It also provides an extensible API for getting statistics about the packet sniffing performance in real time. A major contribution to knowledge is to allow application developers to integrate various packet sniffing frameworks into their applications across various operating system platforms easily

1. Introduction

A packet sniffer (or packet analyzer) is a computer program or piece of computer hardware that can intercept and log traffic that passes over a digital network or part of a network. As data streams flow across the network, the sniffer captures each packet and, if needed, decodes the packet's raw data, showing the values of various fields in the packet, and analyzes its content according to the appropriate RFC or other specifications.

2. Research Objective

The specific objectives of this research are to:

a) design and implement a packet sniffing library in the Go programming language that is platform-independent, supporting Linux, Windows and FreeBSD systems.

b) to implement plugin architecture to easily add support for new packet processing backends.

c) to provide support for packet frameworks with zero copies, and also with multiple receive queues/workers for parallelizing actual packet processing

3. Related Work

This section examines some of the research that has been done around packet processing and sniffing. For each research, we will consider the motivations and objectives, the methodology used to conduct the research, the significance of each work to existing knowledge and finally, the limitation of each work. This will help to provide better understanding of the field upon which this research is based.

3.1.Enabling Packet Fan–Out In The Libpcap Library For Parallel Traffic Processing

Libpcap is a widely used network capture library implemented in C. However, despite the rise in multi-core systems and drastic increase in packet traffic for an average computer, libpcap does not natively fanout packet processing to make use of available CPU cores, and process packets in parallel. This paper aimed to extend libpcap to natively support fanout operations for both multi-threaded and multi-process applications.

Libpcap was extended with a single dedicated API to enable fan-out support. This API also accepts a specific fan-out mode which specifies how packet processing should be distributed to the various threads/processes. The performance of this new, extended pcap library was then evaluated using a simple multi-threaded application and it demonstrated accelerated packet capture rates.

This paper introduced a useful, native extension to the pcap interface to improve multi-processing and demonstrated its superiority to the default single-threaded implementation.

The modification of the libpcap interface was implemented in C, but could take advantage of the underlying AF_PACKET and TPACKET support for fanout, but it does not do this natively. Implementing a Go-native packet capture library will also take

 DOI: 10.20533/ijicr.2042.4655.2021.0135

advantage of this support and provide native fan out support out of the box.

3.2.Comparison of Memory Mapping Techniques For High-Speed Packet Processing

This paper sets out to survey available frameworks for high-speed packet processing. Specifically, three of them are considered: PF_RING ZC, Intel DPDK and netmap. The comprehensive overview of the architecture and API of these frameworks can provide insights into choosing the framework best fit for application requirements.

The common structure and techniques shared by these frameworks are presented. In addition, differences in implementation and architecture are closely examined. API and usage of these frameworks are demonstrated with code examples. The author then demonstrates the cost and performance of each framework by measuring their _behavior in different scenarios.

This paper presented an extremely valuable comparison among these various frameworks which provides a useful backbone for future researchers to compare other frameworks with other possible criteria. It also provides an overview of the tradeoffs made by these frameworks in order to achieve their performance objectives including ease-of-use, compatibility with existing software and API complexity.

Performance measurements presented focused on throughput for the investigated frameworks, but neglected other possibilities for comparison such as latency and energy consumption.

3.3. Netmap: A Novel Framework for Fast Packet I/O

Netmap was introduced to solve performance problems inherent in previous packet processing libraries. It achieves this by supporting zero-copy of packets, support of useful hardware features such as multiple receive queues (NIC RSS), amortizing system call overheads and memory pre-allocation to reduce per-packet dynamic memory allocation.

Netmap makes use of the existing, well-known software interfaces of Linux and BSD in its design and this helps reduce the barrier for programmers who are already familiar with these interfaces. netmap also does not require extensive modifications to available device drivers in order to be usable with it. The decision to use existing software interfaces helps to improve portability of existing software from existing Linux interfaces, and also supports unmodified libpcap clients through a compatibility library.

With thorough consideration for compatibility, netmap is able to realize fast and high-performance packet processing and I/O. This has enabled previously unavailable packet processing rates possible on commodity hardware, while maintaining reasonable tradeoffs.

Netmap sacrifices greater performance in order to use the existing well-known software interfaces of Linux and BSD. This can be examined in comparison to other techniques for high-speed packet processing such as PF_RING ZC and Intel DPDK which break with existing interfaces and require greater effort to port existing applications.

4. System design

This section describes the models and theory that the library is based on. It also describes the library from a high-level perspective, and how it fits into a networking application, the architecture of the library is covered and also an overall model for the library. We also describe the important features that the library provides to users of the library. Finally,we discuss how the library provides high performance and fast packet processing functionality and how the library enables users to monitor the performance of the packets processing (figure1 to figure3).

4.1. Library Description

This library is a platform-independent packet sniffing library with support for multiple backends and implementations. The initial development will provide support for Windows and Linux operating systems, and it will support the AF_PACKET framework on Linux, and npcap on Windows systems.

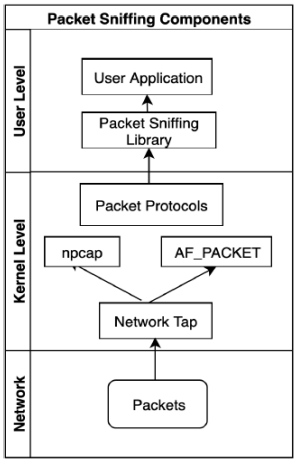

Figure 1. Packets sniffing components

The library will provide common interfaces and definitions that will ease the implementation of

 DOI: 10.20533/ijicr.2042.4655.2021.0135

various packet frameworks as needed apart from those provided with the initial development. The library will support sniffing over network interfaces as well as offline via packet files in supported formats. The library takes in packet data as an array of bytes and decodes it into a packet with a number of layers. Each layer corresponds to an actual OSI or TCP/IP definition of a layer (see Figure1). Four layers will be supported, and once a packet has been decoded, the layers of the packet can be requested from the packet.

The packet layers are:
1. packet.LinkLayer() - returns the TCP/IP layer 1 (OSI layer 2).
2. packet.NetworkLayer() - returns the TCP/IP layer 2 (OSI layer 3).
3. packet.TransportLayer() - returns the TCP/IP layer 3 (OSI layer 4).
4. packet.ApplicationLayer() - returns the TCP/IP layer 4 (OSI layer 7). This is the packet payload.

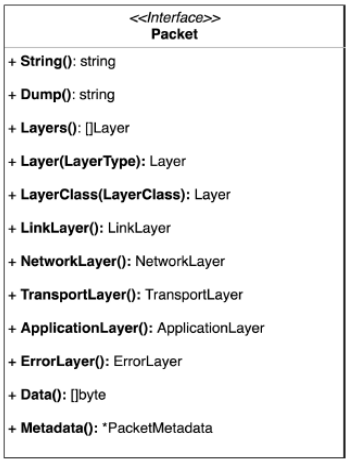

Figure 2. The packet interface

Packets coming in over the network are captured by user hardware, and they are passed on to the Kernel layer. In this layer, the packet frameworks are implemented such as npcap, AF_PACKET Berkeley Packet Filter (BPF), etc.

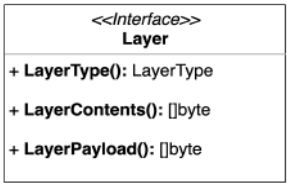

Figure 3. The layer's interface

The packet sniffing library connects to these packet frameworks and is able to decode them and extract useful information from them. The end user application will be responsible for interpreting the decoded packets via the packet sniffing library.

5. System Implementation

In this phase, we validate the packet sniffing library. We will show its usefulness in building network processing dashboards, specifically building a dashboard to monitor network usage and monitor bandwidth utilization in a computer system. We will present an overview of the dashboard that we are building. Here, we will show how the library can be used to bind to the network interface and start reading packets from the wire. Also, we show how we make API calls via the library to access the various layers in the packets, and finally we will present screenshots of the dashboard that show how everything ties together.

5.1. Network Dashboard Overview

As a proof of concept, we will build a real-time dashboard that presents several important features:
1. Display the size of total network data sent and received over all network interfaces
2. Display the total number of packets that have been sent and received and dropped.
3. Display a per-protocol breakdown of packets sent and received in the Link, Network and Transport layers. To do this, we built an HTTP backend, and provide packet sniffing capabilities using our packet sniffing library. The frontend application will be built using Angular 8 which will poll our HTTP server every second. The HTTP backend and our frontend will communicate via REpresentational State Transfer (REST) APIs. The main feature we will be examining is how we get access to the packets, while other details such as the REST APIs will be ignored.

5.2. Binding to Network Interfaces

Reading packets start by opening a packet datasource. To validate our packet filtering functionality, while opening a packet DataSource we will ignore port 22. A BPF filter to express that is written as "port not 22".

```
filter := "port not 22"
```

Next, we create a packet handle to bind to the "any" network device:

```
device := "any"
handle, err :=
afpacket.NewTPacket(afpacket.OptInterf
ace(device))
defer handle.Close()
handle.SetBPFFilter(filter)
```

Then, we create a packet data source from the handle that we have created:

 DOI: 10.20533/ijicr.2042.4655.2021.0135

```
Src:=gopacket.PacketDataSource(afpacke
tHandle)
decoder=gopacket.DecodersByLayerName[
"Ethernet" ]
source:=gopacket.NewPacketSource(src,
decoder)
```

At this point, we have successfully bound to the network device and created a packet source from it that we can use to sniff packets as they enter the computer system.

5.3. Aggregating Data from Packet Layers

First, we will create a global variable to hold the network summary:

```
totalTraffic := &trafficStats{
Bytes: 0 ,
Packets: 0 ,
}
```

Next, we will create our main control loop that process a packet at a time as they enter the system. We access the number of dropped packets by using the SocketStats() API provided by our library:

```
for packet := range source.Packets() {
_, afpacketStats, err :=
handle.SocketStats()
totalDropped = uint64
(afpacketStats.Drops())
packetLength := uint64
(packet.Metadata().Length)
totalTraffic.Bytes += packetLength
totalTraffic.Packets = uint64
(afpacketStats.Packets())
...
}
```

The part represented by "..." is how we represent the processing of each individual layer. There are three similar blocks of code processing each layer so we will only show for the Link layer, since the others are similar to this.

```
linkLayer := packet.LinkLayer()
if linkLayer != nil {
     linkLayerName := linkLayer.LayerType().String()

     if _, ok := layerMap[Link][linkLayerName]; !ok {
          layerMap[Link][linkLayerName] = &trafficStats{}
     }

     layerMap[Link][linkLayerName].Packets += 1
     layerMap[Link][linkLayerName].Bytes += packetLength
}
```

At this point, we have statistics for the Link, Network and Transport layers as well as the global network statistics.

5.4. A Look at the Dashboard

In this section, we will have a look at the resulting dashboard, after REST APIs have been created to power our frontend.

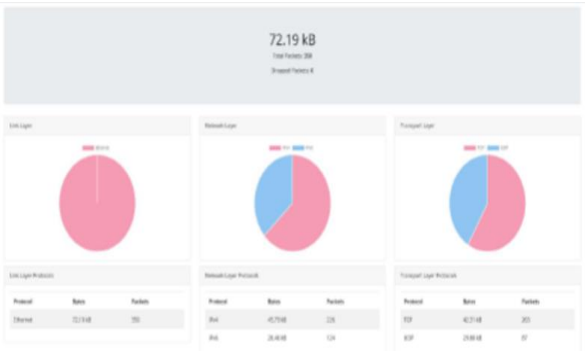

Figure 4. Network analysis dashboard.

Figure 4 which is the dashboard shows (i) the size of the total network data sent and received over all network interfaces, (ii)display the total number of packets that have been sent and received, and dropped and (iii) display a per-protocol breakdown of packets sent and received in the three layers(Link, Network and Transport layers).

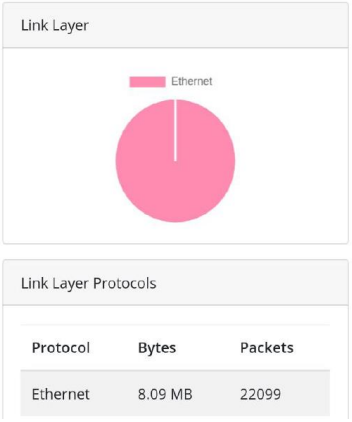

Figure 5. Protocols in the link layer
Ethernet is the protocol on Link layer

An important characteristic of a Data Link Layer is that datagram can be handled by different link layer protocols on different links in a path. For example, the datagram here is handled by Ethernet on the first link, PPP on the second link. The data link layer is concerned with local delivery of frames between devices on the same LAN (see Figure 5).

DOI: 10.20533/ijicr.2042.4655.2021.0135

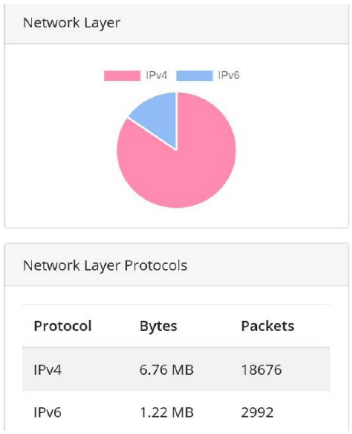

Figure 6. Protocols in the network layer

Here we are concerned with IPv4 and IPv6 protocols. here on network, the IPv4 transmit 6.76MB with 18676 packets while IPv6 transmit a total of 1.22MB with 2992 packet size (see Figure 6).

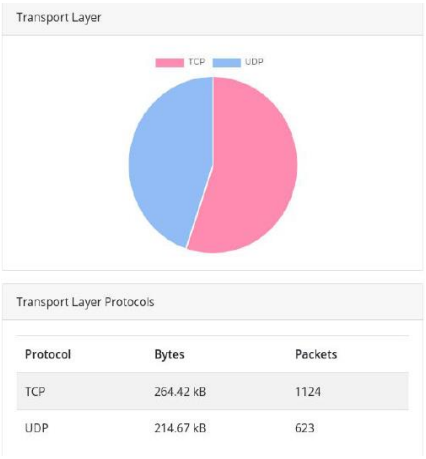

Figure 7. Protocols in the Transport layer

The transport layer is ruled by two protocols namely Transmission Control Protocol and User datagram protocol. TCP is responsible for connection-oriented delivery from source port to destination port address and here a total of 1124 packets were sent while for UDP, a total of 623 packets were sent within the same time frame (see Figure 7).

6. Recommendations

The library presented in this work is capable of providing essential packet sniffing functionality, but it can also be improved upon and extended to support more operating systems than currently supported. Currently, the library supports Windows and Linux operating systems so it will be great if more operating systems can be supported. The library can also be extended to support more packet frameworks than currently supported, such as pf_ring, DPDK, netmap. This will greatly increase the scope of use of the library as it can be deployed on more operating systems than supported at present.

7. Conclusion

We have designed an easy-to-use library to use for packet sniffing, which can be used in a variety of network applications. With this library, developers can easily read packets from network devices and files while providing some healthy abstractions. We also validated the library by building a network dashboard application for a network analysis use case. For programmers in the Go programming language, building network applications will now be faster because this library will natively handle all the low-level details for them. This will help increase the health, stability and functionality of applications built with the library. We have also demonstrated zero copy packets filtering with our network dashboard. With the main Packet interface that we have presented, other packet frameworks can be integrated in our library which acts as our plugin architecture.

9. References

[1] Rizzo, L., Deri, L., Cardigliano, A (2012). 10 Gbit/s line rate packet processing usingcommodity hardware: survey and new proposals, Retrieved from http://luca.ntop.org/10g.pdf. (Access Date: 15 Oct, 2019).

[2] Gallenmuller, S., Emmerich, P., Wohlfart, F., Raumer, D., & Carle, G. (2015). Comparison of frameworks for high-performance packet IO. 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).

[3] Bonelli, N., Giordano, S., & Procissi, G. (2017). Enabling packet fan-out in the libpcap library for parallel traffic processing. 2017 Network Traffic Measurement and Analysis Conference (TMA).

[4] García-Dorado, J. L., Mata, F., Ramos, J., Río, P. M. S. D., Moreno, V., & Aracil, J. (2013). High-Performance Network Traffic Processing Systems Using Commodity Hardware. Data Traffic Monitoring and Analysis Lecture Notes in Computer Science, 3–27.

[5] Xu, X., & Li, Z. (2009). High-Speed Packet Capture Mechanism Based on Zero-Copy in Linux. 2009 2nd International Conference on Biomedical Engineering and Informatics.

[6] Deri, L. (2004). Improving Passive Packet Capture: Beyond Device Polling. In Proceedings of the Fourth International System Administration and Network Engineering Conference.

 DOI: 10.20533/ijicr.2042.4655.2021.0135

[7] Deri, L. (2005). nCap: Wire-speed Packet Capture and Transmission. E2EMON '05 Proceedings of the End-to-End Monitoring Techniques and Services on 2005.

[8] Rizzo, L. (2012). Netmap: a novel framework for fast packet I/O. USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference.

[9] McCanne, S., Jacobson, V. (1992). The BSD Packet Filter: A New Architecture for User-level Packet Capture. USENIX'93 Proceedings of the USENIX Winter 1993.

[10] Risso, F., & Degioanni, L. (2001). An architecture for high performance network analysis. Proceedings. Sixth IEEE Symposium on Computers and Communications.

 DOI: 10.20533/ijicr.2042.4655.2021.0135