Extending Tcpdump To Support The Homa Transport Layer Protocol

Junhan Chen

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2023

Abstract

This document presents the work of the Extending Tcpdump To Support Homa Transport Layer Protocol project. As a recently proposed transport layer protocol that is intended for data centers, Homa has attracted massive attention due to its excellent performance. This project aims to extend Tcpdump (a popular command-line packet dissector) to be able to present information about Homa traffic to users. To achieve this goal, a study was first conducted to understand Tcpdump and Homa packets in depth. Based on the results of the study, a Homa dissector was implemented as an extension module to Tcpdump, which is modularised and memory-safe. Several tests and analyses were performed to evaluate the quality of the implementation. The result of the evaluation shows that the Tcpdump Homa dissector is generally reliable, and maintainable. However, there are still limitations in the Tcpdump Homa dissector, including imperfect exception detection and handling mechanisms and missing links to application layer dissectors. In addition to fixing these limitations, the possible future work for this project includes writing and automating more tests to be able to contribute the Homa dissector to Tcpdump.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Junhan Chen*)

Acknowledgements

Many thanks to my project supervisor Dr. Michio Honda, who provided significant support to this project. Also thanks to the CouldLab, which provides computation resources to perform tests.

Table of Contents

1 Introduction **1**

1.1 Tcpudump 1

1.2 Homa 1

1.3 Motivation and Objective 2

1.4 Document Structure 2

2 Background **3**

2.1 Network Packet and Dissection 3

2.2 Network Interface and Switch 3

2.3 Communication Pattern of DCN 4

2.4 Homa 5

2.4.1 RPC Oriented 5

2.4.2 Shortest Remaining Process Time and Priority 5

2.4.3 Overcommitment 6

2.4.4 Reliable Transmission 7

2.5 Related Work 7

3 Conceptual Study **8**

3.1 Working Principle of Tcpdump 8

3.1.1 Process of Packet Capture 9

3.1.2 Process of Packet Dissection 10

3.2 Homa Packet Format 12

3.2.1 Common Header 12

3.2.2 DATA Packet 13

3.2.3 GRANT, RESEND, CUTOFFS and ACK Packet 15

3.2.4 UNKNOWN, BUSY, FREEZE, NEED_ACK and BOGUS packet 17

4 Design and Implementation **18**
4.1 Design . . . 18
4.2 Implementation . . . 19
4.2.1 Memory Access and Binary Data Decoding . . . 19
4.2.2 Exceptions Detection and Handling . . . 22
4.2.3 Packet Priority . . . 22
4.2.4 User Option and Output Content . . . 23
4.2.5 Integration . . . 24

5 Evaluation **25**
5.1 Functionality and Reliability . . . 25
5.1.1 Static Test . . . 26
5.1.2 Illustration of Output . . . 27
5.1.3 Environment Configuration and Homa Application . . . 28
5.1.4 Simulated Scene Test 1 . . . 28
5.1.5 Simulated Scene Test 2 . . . 30
5.1.6 Simulated Scene Test 3 . . . 31
5.2 Maintainability and Usability . . . 32
5.2.1 Maintainability . . . 32
5.2.2 Usability . . . 33

6 Conclusion **34**
6.1 Limitations . . . 34
6.1.1 Limitations On Implementation . . . 34
6.1.2 Limitation On Test . . . 34
6.2 Future Work . . . 35
6.2.1 Link With Application Dissectors . . . 35
6.2.2 Tests That Are Closer To Usage Scene . . . 35
6.2.3 Contribute To Tcpdump . . . 35
6.2.4 Overall Conclusion . . . 36

Bibliography **37**
.1 Appendix 1 Code to generate Homa traffic in the input of static test . . 38

Chapter 1

Introduction

1.1 Tcpudump

Tcpdump [7] is an open-source command-line packet dissector (also known as an analyzer or sniffer), that is able to capture packets flowing over specific network interface(s) and dissect the packet to provide traffic information to users. Today, Tcpdump is a popular tool used by various computer science communities (e.g. network administrators, security experts) for various purposes, including traffic monitoring, troubleshooting and intrusion detection [15]. Meanwhile, it has now been adopted as a native tool by various UNIX-like Operating Systems (OS) such as Linux, MacOS and FreeBSD.

1.2 Homa

To meet the rapidly growing demand for computing resources, the last decade has seen the emergence of data centers [17]. Data centers typically host large scale networked computing nodes to provide Internet or cloud services in a distributed manner. Thus, the performance of the Data Center Network (DCN) becomes a non-negligible part of then overall data center performance. However, due to its unique characteristics, traditional networking protocols do not fit well in the data center environment, resulting in inefficient use of DCN infrastructure resources. The actual performance of DCN is lower than the potential hardware can deliver [13]. To achieve the expected performance, data centers has to invest more capital in network hardware [11]. In addition to the performance and cost issue, the mismatch between traditional protocols and DCN can also make data centers less scalable, as the additional communication overhead

introduced by adding more computing nodes can exceed the computation power brought up by nodes. It has been shown that the network has become the major bottleneck of data center scalability [9]. The research community has started to design new protocols suitable for DCN. Homa [13] is proposed by one of these studies, which is a new transport layer protocol that aims to improve overall packet latency in DCN. It is hugely different in many aspects from the heavily used transport layer protocol today - TCP. Homa provides remarkably lower latency than TCP in the data center context, in extreme cases, the 99th percentile tail latency of Homa packets can be on the order of microseconds [13]. Nowadays, Homa has been implemented as a kernel module of Linux [14] with nearly complete functionality for use in production environments.

1.3 Motivation and Objective

Although Homa is still in the experimental phase, due to its powerful performance, it can be expected that Homa will be used in data center environments in the future. To support the use of Homa, the objective of this project is to extend Tcpdump to be able to capture and dissect Homa packets. To be more specific, the extended Tcpdump should be able to identify Homa as a transport layer protocol, decode the raw data and print out useful information to users.

1.4 Document Structure

This document presents the work that has been done to fulfil the objective, the result achieved, its evaluation, limitations and future work. The second chapter illustrates the background knowledge and related work. The third chapter describes the conceptual study to gain insight into Tcpdump and Homa. The fourth chapter gives the design and implementation of the Homa dissector. The result is presented, tested and evaluated in the fifth chapter. Finally, the sixth chapter gives a brief conclusion on the limitations and future work.

Chapter 2

Background

2.1 Network Packet and Dissection

Packets are the basic unit of communication for computer networks. Conceptually, a packet is data that is encapsulated by a set of protocol headers. These headers contain metadata to enable communication between protocols running on the sender and receiver ends. According to the OSI Reference Model [10], a network packet typically begins with a data link layer protocol header (e.g. Ethernet, Wi-Fi), followed by a network layer protocol header (e.g. IPV4, IPv6) and a transport layer protocol header (e.g. TCP, UDP), and may end with an application layer protocol header and data payload. In addition to packets that transmit data, there are also control packets that are used to configure network systems. Physically, a packet is just a bitstream flowing down the data link, or binary data stored in memory, which itself makes no sense without a well-defined protocol format. Although originally designed for communication between protocols, the metadata of packet headers can also be used to gain insight into network traffic patterns. For example, by looking at the fields within IPv4 headers, one can generally model the ongoing communications between nodes on a given network. Thus, packet dissection mainly focuses on the data within headers of different layers, as the data payload contains almost no information about traffic.

2.2 Network Interface and Switch

Network interfaces are software (e.g. loopback interface) or hardware (e.g. Network Interface Card (NIC)) that enable devices to send and receive network packets. They use specific physical and data link layer standards to communicate with other nodes within

the network. A NIC also comes with its driver, which is a software component that controls the behaviour of the NIC. There are a variety of types of NICs manufactured by different vendors (e.g. Intel, Cisco), and a particular type of NIC may support one or multiple data link layer protocols. When network interfaces are mentioned in this document, it always refers to the hardware - the NIC.

A switch is a device within a computer network that performs packet switching. A switch typically contains a number of network interfaces connected to computers or other network devices. When packets are received from an ingress interface, a switch evaluates the header fields of packets and forwards them to a particular egress interface. If too many packets are forwarded to an egress interface for it to send them all at once (due to bandwidth limitation), these packets could be buffered in a queue to wait for the interface to become free. The most common queue for an egress interface is the First In First Out (FIFO) queue, where packets that arrive early are transmitted early. Unlike the FIFO queue, the priority queue of network switches consists of multiple queues with different priority levels (usually 8 or 16). Packets are placed in different queues based on their priorities, which can be specified by fields of different layer headers (e.g. Differentiated Services field of IPv4 header, Priority Code Point field of IEEE 802.1Q [6] header). Queues with higher priority levels are always transmitted before lower ones. In this way, network switches provide a mechanism for classifying network traffic and providing services of different quality to different classes.

2.3 Communication Pattern of DCN

The overall communication pattern of the DCN differs from the general network, which has affected the design of Homa. Specifically, the communication of applications within data centers is dominated by a high volume of small messages (typically hundreds of bytes) [13]. This is partially due to an inter-process communication method commonly used in DCNs called Remote Procedure Call [16] (RPC). An RPC can be considered as a short session between client and server applications. The process of RPC is fixed, which consists of sending a request message from a client to a server, processing the request at the server and sending back a response message from the server to the client. Once the response is received by the client, the RPC terminates. Due to the brevity of PRC, an application can hold many RPCs simultaneously, with each RPC transmitting a small amount of data. Given the widespread use of RPC, the service provided by Homa to upper layer applications is also based on the RPC paradigm.

2.4 Homa

Homa is a relatively new transport layer protocol that is still under development. However, the basic functionalities and core features are mature enough as of the Homa/Linux implementation [12]. In this section, some of the outstanding features and design concepts are presented to have a better understanding of how Homa achieves its performance. When Homa packet is mentioned, it refers to a transport layer segment with a Homa protocol header followed by the data payload.

2.4.1 RPC Oriented

Homa is RPC oriented, it maintains the state of each ongoing RPC. Each RPC can be identified by an RPC id, which is specified in the headers of Homa packets. For each ongoing RPC, the Homa protocol running at the client and server end can be recognised as sender or receiver according to its position of sending and receiving message. That is, when sending a request message, the client side can be recognised as the sender of the message, while the server side can be recognised as the receiver of the message. When receiving a response from the server, the roles are reversed. Since the states of a particular RPC are cleaned instantly after receiving the response, it reduces the memory overhead of Homa, and allows Homa Protocol to maintain states of more RPCs at a given time. Meanwhile, as the entire message is sent from the application to Homa at once before transmission, the size of the message can be known in advance by both sender and receiver, allowing Homa to schedule packets based on message size.

2.4.2 Shortest Remaining Process Time and Priority

Homa attempts to approximate the Shortest Remaining Process Time (SRPT) mechanism, which favours messages with fewer remaining bytes to be transmitted network-wide. Two major mechanisms are implemented to approximate the SRPT. The first mechanism is the receiver-driven transmission authorization and priority allocation. When an RPC is initialised, the client can send the first part of the request message without permission (called `unscheduled bytes`). However, the following bytes (called `scheduled bytes`) may only be sent if a `GRANT` Homa packet sent by the receiver is received by the sender. The `GRANT` packet specifies the chunk of data that is authorised to be transmitted (which could be divided into several `DATA` Homa packets), along with the priority that the sender should use for these `DATA` packets. The priorities are then

expected to be used in the priority queue of network switches to prioritize the transmission of high-priority packets. In practice, there may be multiple senders wishing to send messages to the same receiver simultaneously. Through this mechanism, the receiver is able to control the latency of `DATA` packets of all incoming messages to some extent, so that packets from messages with fewer remaining bytes are allocated with higher priority, prioritized by network switches during transmission and are expected to have a shorter latency. In addition to the receiver side, the sender side also implements a mechanism to approximate SRPT. Similarly, it is possible for the sender to have multiple messages that are able to be transmitted at a given time (either granted or `unscheduled bytes`). In this case, the sender will always send the message with the fewest remaining bytes first (which is also likely to have a higher priority).

All types of Homa packets should have a priority specified, in addition to `scheduled DATA` packets, this also includes `unscheduled DATA` packets and control packets. Homa defines a total of 8 priority levels, represented by numbers from 0 to 7, with the higher the number the more important. All types of control packets should have the highest priority. The assignment of priority to `Unscheduled Data` packets is also controlled by the receiver, this is done by advertisements to all the receiver's peers in advance, the advertisement (i.e. `CUTOFFS` packet) suggests how to assign priority to `unscheduled DATA` packets that sent to the source of the advertisement based on message length. Normally, different sets of priorities are used for `scheduled` and `unscheduled DATA` packets. The whole sequence of priority levels is divided into two parts by each receiver, with the higher priority levels part used for `unscheduled DATA` packets and another part used for `scheduled DATA` packets. A receiver makes the division based on observed traffic and tries to approximate the ratio of `unscheduled` priorities to `scheduled` priorities to the ratio of `unscheduled` traffic to `scheduled` traffic.

2.4.3 Overcommitment

As described in the previous subsection, the receiver may have multiple incoming messages waiting to be granted simultaneously. A further question might then be, how many messages a receiver shall grant at any given time. To avoid downlink (the data link from the Top of the Rack switch to the receiver network interface) from being idle, it is reasonable for the receiver to grant more messages at a given time, as the sender might not wish to transmit the granted message instantly. However, there is a trade-off between bandwidth utilization rate and buffer occupancy rate, as granting too many

messages at one time could result in the switch buffer being occupied by packets to a single receiver, which may prevent other nodes from using the buffer. Meanwhile, the receiver could be overwhelmed if all granted messages were sent instantly. Thus, Homa balances the trade-off using a configurable parameter called `max_incoming`, which defines the maximum number of bytes that are allowed (either granted or `unscheduled bytes`) to be or are being transmitted but not yet received. Meanwhile, Homa maintains a run-time estimate of the actual incoming bytes, so that new grants can only be issued if the actual incoming bytes are less than `max_incoming`. In practice, the receiver is normally allowed to grant 8 messages at a time [12].

2.4.4 Reliable Transmission

Homa provides reliable transmission through a receiver-driven retransmission mechanism. For each granted chunk of data, Homa maintains a timer, when the timer expires but the receiver has not received any packet of that chunk of data, it will issue a `RESEND` packet specifying the sequence of bytes within that message that needs to be retransmitted. If several `RESEND` packets are sent without response, the receiver will assume that the peer has crashed and terminate the RPC.

2.5 Related Work

A similar work to this project is the Wireshark Homa dissector [2]. Wireshark [8] is another popular packet dissector whose functions are generally the same as Tcpdump. The major difference between Wireshark and Tcpdump is the user interface. Tcpdump requires users to input options on the command line, and display packet information on the terminal. While Wireshark provides a more usable Graphic User Interface (GUI) that allows users to perform customized packet capture and dissection operations by simply clicking on different buttons, and viewing the output from the GUI. This related work implements a Homa dissector as a plugin to Wireshark, with limited functionality. For example, when dissecting a `DATA` packet (a type of Homa packet that carries data, the details of which are described in `section 3.2`), the Wireshark Homa dissector does not identify its payload as request or response data. In contrast, the Tcpdump Homa dissector implemented in this project indicates this by identifying the packet sender and receiver as client or server. The details of how the Tcpdump Homa dissector achieves this are described in `section 3.2.1` .

Chapter 3

Conceptual Study

To achieve the objective of this project, it is essential to (1) understand how Tcpdump works and (2) understand the format of Homa packets. However, although both Tcdump [4] and Homa/Linux [12] are open source, there are no documents describing the above conceptual knowledge. Therefore, the study was conducted to have a better understanding of Tcpdump and Homa. This chapter presents the results of the study.

3.1 Working Principle of Tcpdump

Tcpdump is written in C language, which depends on various C libraries from the Standard C Library to OS libraries and third-party libraries. The architecture of Tcpdump is complex. It contains multiple modules (e.g. input module for user input processing, utility modules for I/O and memory access) and multiple I/O routines. In terms of input, Tcpdump can capture raw packets from a give interface in real time or from a saved `pcap` file. The output of Tcpdump can be directed to the Standard Output Stream (which is terminal by default) or to a given `pcap` file. As the content written to the `pcap` file is binary without the need for dissection, this project focuses on the output to the terminal, which is generally human-readable information for each input packet. Despite the complexity of Tcpdump, the core modules can be abstracted to packet capture and dissection. This section describes the two modules in detail, providing the basis for the design work.

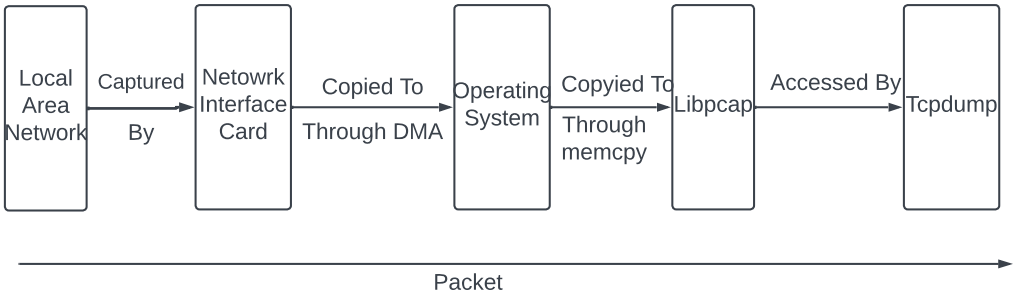

Figure 3.1: Packet Capturing

3.1.1 Process of Packet Capture

The packet capture presented in this subsection refers to capturing packets from network interfaces in real time, as reading from a saved `pcap` file is straightforward. As a process running in user space, since Tcpdump doesn't have the privilege to capture packets directly from network interfaces, it has to ask the OS through a system call, which in turn asks the network interface to capture packets. However, the involvement of OS introduces another issue, as different OSs may provide different Application Programming Interfaces (API) for system calls, it increases the difficulty of being cross-platform for Tcpdump. To solve this problem, an indirect layer is added between Tcpdump and the OS - the Packet Capture Library (Libpcap) [7], which is a C programming language library that provides cross-platform user-level packet capture interfaces. Till now, the overall picture of packet capture for Tcpdump is clear. As shown in `Figure 3.1`, as a packet traverses the network, it is captured and buffered in the NIC. After some simple computation like an integrity check, the NIC copies the packet into the kernel memory through Direct Memory Access (DMA), and then informs the OS. Normally, the packet in kernel memory is then processed by the OS protocol stack, but as for Tcpdump, the raw packet is copied to the buffer in user space allocated by Libpcap through `memcpy`, which can be accessed by Tcpdump.

Although a NIC may support multiple data link layer protocols, the packets captured by a live capture opened by Libpcap on a particular interface will support only one link layer protocol. In other words, each time Tcpdump is launched on a particular interface, the link layer protocol for all raw packets it receives will be the same. Meanwhile, the link layer protocol of the packets provided by Libpcap will not necessarily be the same as the link layer protocol on the network, as the NIC or OS may change the real hardware protocol. An example is the Linux Cooked Mode, a packet-capturing mode

provided by Linux and utilised by Libpcap, which allows applications to capture packets from the `any` interface, but with the real link layer protocol header replaced with a fake `sll` or `sll2` header.

3.1.2 Process of Packet Dissection

Once a raw packet is received, the rest of the packet dissection is done entirely within Tcpdump. In general, packet dissection means extracting and printing information from raw packets in a header-to-header fashion based on the protocol-specific header formats. Though in theory all network packets should follow the OSI reference model, in practice, the packet structure can be more complex and diverse than expected. To give a few examples, an IPv6 packet tunneling through an IPv4 netowrk might have two network layer headers (an IPv6 header encapsulated within an IPv4 header). A transport layer segment (a transport layer protocol header followed by the payload) may be fragmented by the IPv4 protocol. Tcpdump adopts a successive and modularized approach to make packet dissection universal and robust. Each module (called dissector) receives input from a lower layer dissector and is responsible for dissecting headers of a specific protocol.

`Figure 3.2` shows the overall architecture of the Tcpdump dissector and an example dissection routine of a packet. As mentioned in the previous subsection, it is guaranteed that all packets accepted from a Libpcap live capture are encapsulated by the same type of link layer header. which is indicated by a numeric value (the full table of values and corresponding link layer header types can be found here [1]). Therefore, the dissection routine is always started with a link layer dissector picked using the link type value provided by Libpcap. The selected dissector will process the raw header data according to the defined protocol format and print information to users. Upon completion, the current dissector will invoke a next one based on some specific fields within the current header (e.g. the `ethertype` field of Ethernet indicates the protocol type for the next header), passing the payload of the current protocol packet as an argument to the next dissector. Ideally, this process is repeated at each layer of the OSI model until the application layer protocol is dissected. However, in practice a variety of situations may arise.

(1) As in the IPv6 tunnelling example, a packet might contain multiple headers of the same layer. Therefore, it is possible for a dissector to invoke a peer dissector in the same layer. (2) Unsupported or unknown protocols may be encountered. If this

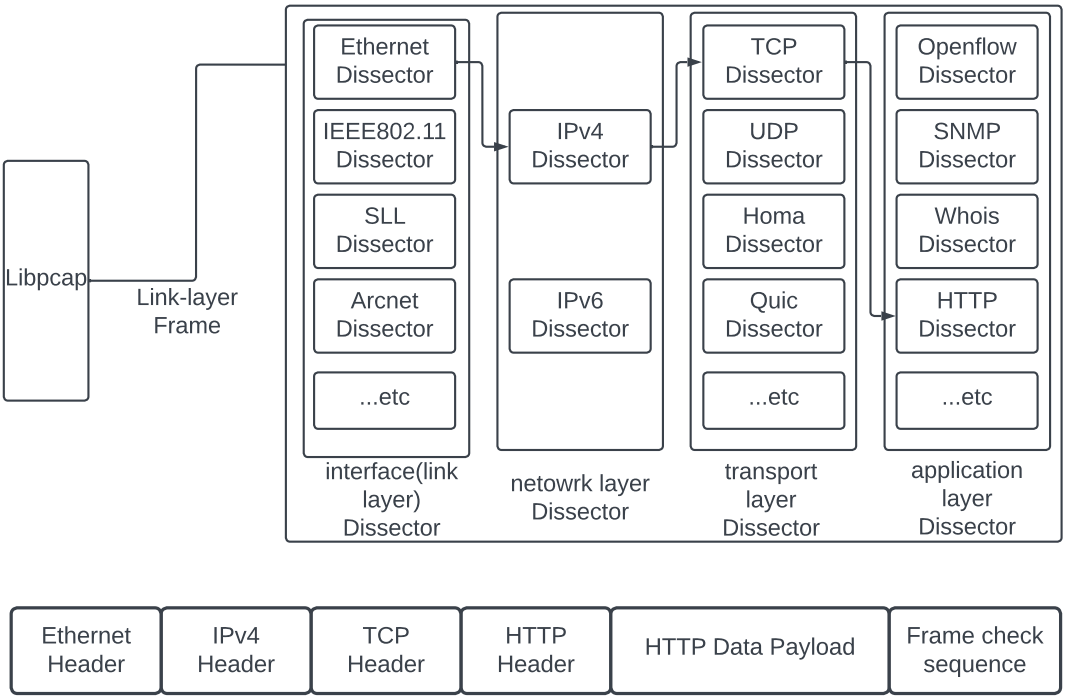

Figure 3.2: Tcpdump Dissector With An Example Routine

occurs, the dissector will terminate instantly. (3) The raw packet provided might be invalid or incomplete. The former means that the header(s) of the packets do not follow the protocol definition, which can be caused by sender-side error or disturbance within transmissions. All dissectors in Tcpdump should perform a validity check on the input packet header. However, it is not always possible to detect all invalid packets. In some cases, invalid packets can be detected (e.g. checksum, the value of the length field is less than the size of the raw packet) with the dissector routine terminating immediately and invalid information being printed to users. In other cases, the bad data is dissected and presented to users.

The latter means that the header(s) of packets are partial or not present. This can be caused by truncation (only part of the packet is captured by network interface) or fragmentation (the packet of the upper layer is fragmented by the lower layer, such that the lower layer packet only carries part of the upper layer packet). A packet can be truncated at any point from the link layer header to the application layer payload. Fragmentation is a bit different. Fragmentation is caused by the individual behaviour of a specific protocol. And it is certain that only the upper layers of the protocol that perform fragmentation are affected. Thus, in Tcpdump the corresponding dissectors of protocols that support fragmentation (e.g. IPv4 dissector) are responsible for detecting

and handling fragmentation. In general, if fragmentation is detected by the current dissector, it will only invoke the next dissector if the payload of the current packet is the first fragment of the sequence of fragments, as the first fragment is most likely to contain a complete header of next level. In other cases, the dissector will terminate the dissection routine. As a result, a normal dissector in Tcpdump does not need to worry about fragmentation, as it is guaranteed that even though packets of protocol corresponding to that dissector are fragmented, only the first fragments will be passed to that dissector. However, all dissectors in Tcpdump still need to implement mechanisms to check the completeness of their input packet header. If the incomplete headers are detected, the dissector routine will also terminate immediately with an incomplete notification printed to users.

3.2 Homa Packet Format

Homa has ten types of packets [12] in total. Each type of packet can have a unique format and is used for different purposes. In summary, they are (1) `DATA` for transmitting messages, (2) `GRANT` for authorising the transmission of a sequence of data, (3) `RESEND` for requiring the retransmission of a sequence of data, (4) `UNKNOWN` to report receipt a packet with an unknown RPC id, (5) `BUSY` in response to `RESEND` to indicate that retransmission will be delayed, (6) `CUTOFFS` for advertising priorities to be used for `unscheduled bytes`, (7) `ACK` to acknowledge receipt of a response message, (8) `NEED_ACK` to require an `ACK` for a specific RPC, (9) `FREEZE` and (10) `BOGUS` for debugging purposes.

3.2.1 Common Header

The common header is the first chunk of data that is common to all types of Homa packets. It is designed to be compatible with the TCP header in order to utilise TCP Segmentation Offload service provided by NICs. The format of the common header is shown at `Figure 3.3`, with each line representing a chunk of 32-bit data. The length of the common header is constant - 28 bytes. It can be seen that to be compatible with TCP header, a lot of fields are wasted. The offset and checksum fields are useless too, although the offset field is designed to indicate the length of `DATA` header using the higher order four bits (in units of 4-byte words), the length of `DATA` header is constant. The first two fields hold the source and destination port which are used to identify

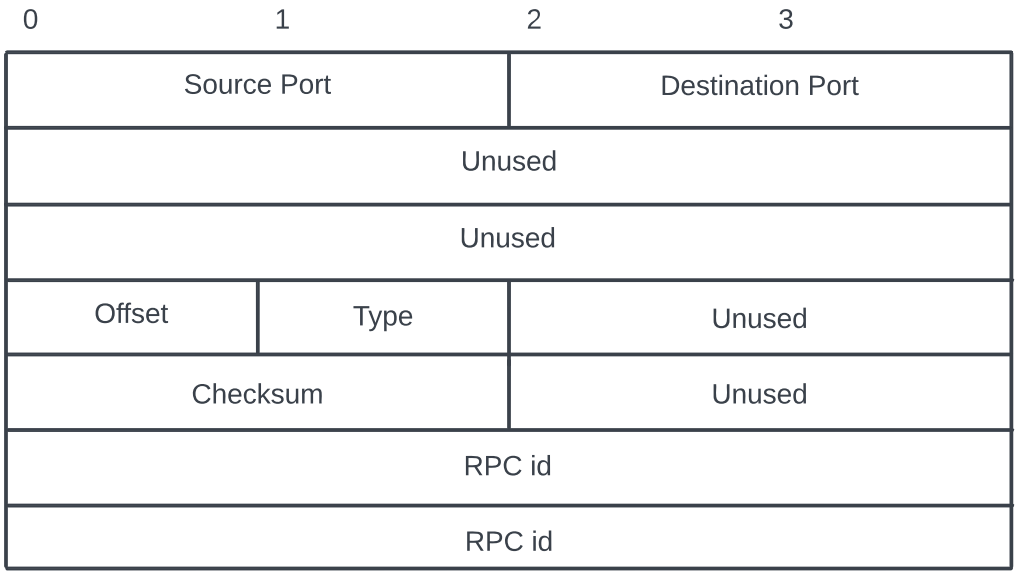

Figure 3.3: Common Heder Format

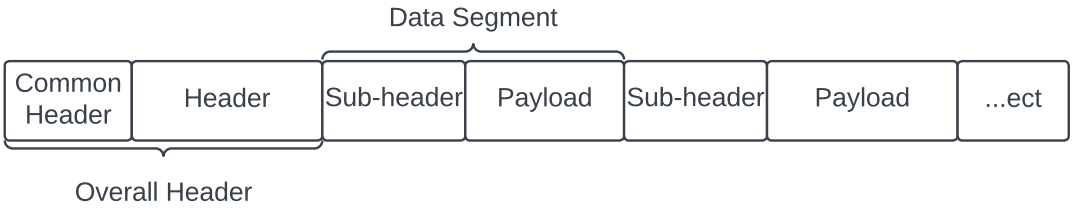

Figure 3.4: Data Packet Format

applications. The type field holds that type of current packet. The RPC id field holds the identifier of the RPC used at the packet sender side. Homa adopts an asymmetric RPC id mechanism, the client and server end will use different RPC ids to refer to the same RPC. Based on this mechanism, the role of the packet sender can be distinguished based on the RPCid field. If the lowest bit of RPC id is set, then the packet sender is the server and vice versa.

3.2.2 DATA Packet

`DATA` packet is sent by the message sender to transmit a chunk of data from the message, it contains an overall header followed by a variable number of segments, each segment containing a sub-header and a chunk of data. `Figure 3.4` illustrates the format of the `DATA` packet. The overall header consists of a common header and a `DATA` header.

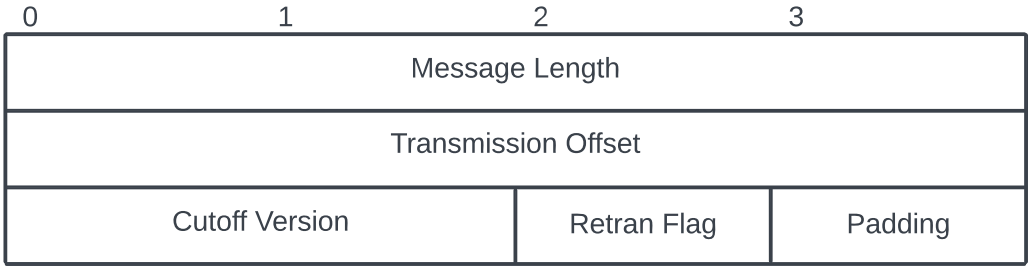

Figure 3.5: Data Header Format

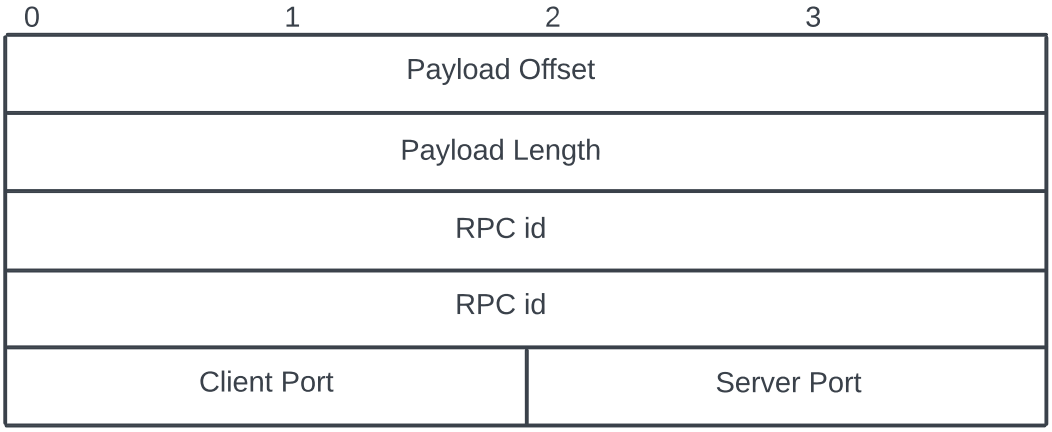

Figure 3.6: Data Segment Sub-header Format

`Figure 3.5` presents the format of `DATA` header. The message length field holds the total length of the message to be transmitted in bytes. The transmission offset field provides an offset of a message, such that all data before the offset is expected to be transmitted. These two fields together provide information to the message receiver to keep track of unreceived bytes of the message, so that the receiver can grant and allocate priority based on SRPT. The Cutoff Version field tells receiver the most recent version of `unscheduled bytes` priority advertisement (i.e. the `CUTOFFS` packet) received from that receiver. The Retran Flag field suggests if the `DATA` packet is being sent in response to a `RESEND`. The length of the overall header is also constant - 40 bytes. The subheader of each segment holds metadata about the payload it carries. `Figure 3.6` shows the format of the segment subheader. The first two fields hold the offset of the first byte of the payload within message and the length of the payload in bytes respectively. This can help the message receiver to position the segment payload within the message. The following fields, which together identify an RPC, form an acknowledgement of

Figure 3.7: Grant Packet Header Format

receipt of a response. This acknowledgement takes effect only when the RPC id field is greater than zero. Thus, as a side effect, the segment sub-header is able to acknowledge another RPC. However, the acknowledgement fields of the segment subheader are not optional, which means these fields are used even though no acknowledgement is carried by the segment subheader. The length of segment sub-header is also constant - 20 bytes. Meanwhile, although the number of segments carried by a `DATA` packet is variable (at least one), there is no field within the headers that explicitly suggests this.

3.2.3 GRANT, RESEND, CUTOFFS and ACK Packet

`GRANT` packet is sent by message receiver to sender to require the transmission of chunks of the data. `GRANT` packet contains only a header without a payload. Actually, all control packets (packets that exclude `DATA`) carry no data payload. The structure of `GRANT` header is straightforward, as shown in `Figure 3.7`, excluding the common header, the first field is a 32-bit integral offset indicating that the sender should send all data up that offset. The second field is an 8-bit integer, which specifies the priority the sender should use for new `DATA` packets. The length of `GRANT` header is constant - 33 bytes.

`RESEND` packet is sent by the receiver if a timeout occurs but the expected data has not been received. `Figure 3.8` shows the format of `RESEND` header, the first two fields - the offset and length of retransmitted data collaboratively locate the chunk of retransmitted data within the message. The third field specifies the priority used for retransmission packets. The length of `RESEND` header is constant - 37 bytes.

`CUTOFF` packet is sent from the sender to instruct the recipient on the assignment of priority if the recipient with to send `unscheduled DATA` packets to the sender. In general, the assignment of priority is based on message length, with higher priority assigned to shorter messages. `Figure 3.9` shows format of `CUTOFF` packet header (exclude the common header), The first field is an array of 8 elements, the indexes of elements represent the priority levels, and the value of each element suggests the

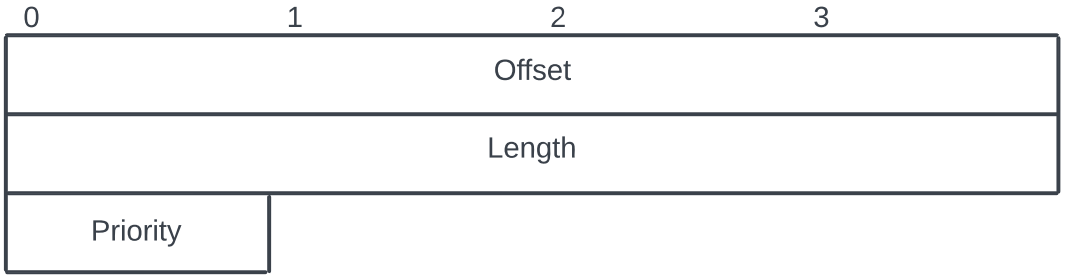

Figure 3.8: Resend Packet Header Format

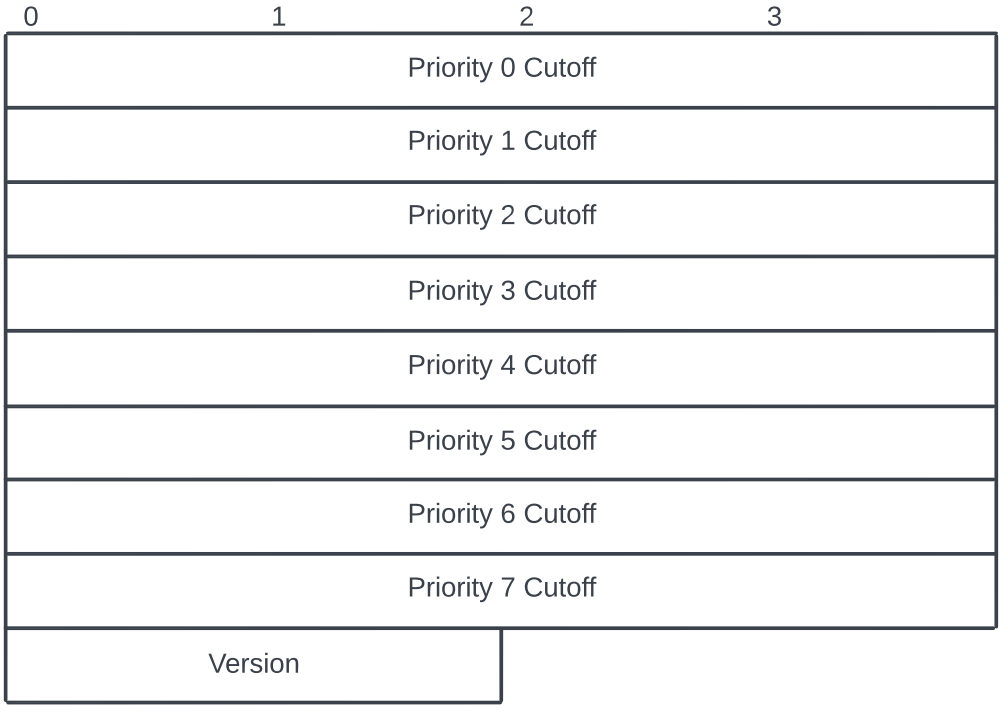

Figure 3.9: Cutoff Packet Header Format

maximum message size (in bytes) that can use the corresponding priority level. The lower bound of a priority level (minimum message size that can use that priority level) can be calculated based on the upper bound of next higher priority level. The second field holds the version of received `CUTOFFS` packet, which is unique among versions of all `CUTOFF` packets from specific sender to specific receiver, as a Homa node may update its allocation of `unscheduled` priorities from time to time. The length of `CUTOFF` header is constant - 62bytes.

`ACK` packet is sent from client to server to acknowledge the receipt of response message for a set of RPCs, such that the server may clean the corresponding states. The `ACK` header contains only one field (exclude common header) - a 16-bits number holds the number of RPCs being acknowledged. The header is then followed by that number of 12-byte structures (same as the acknowledgement structure at data segment header) that identify completed RPCs.

3.2.4 UNKNOWN, BUSY, FREEZE, NEED_ACK and BOGUS packet

In addition to `BOGUS` packet whose format is not specified in any document, the format of the remaining packets is identical. These types of packets only contain a common header, with the only difference being the type field within the common header. So far, the format of all Homa packet types become clear. Overall, it can be seen that the format design of Homa packets is redundant. A lot of fields within the common header are wasted in order to be compatible with the TCP header. Meanwhile, the acknowledgement structure within the segment header could be optional.

Chapter 4

Design and Implementation

This chapter describes the design and implementation of a Tcpdump extension module for dissecting Homa packets- the Homa dissector. The design section gives an overall architecture, while more details are presented in the implementation section.

4.1 Design

The overall function of the Homa dissector is, given a captured Homa packet, it should dissect the packet and print out traffic information to users. Modularization and encapsulation are the key principles used in the design of the Homa dissector. Since the formats of different types of Homa packets are distinct, the whole dissector is modularized into multiple modules, each module being responsible for the dissection of a single Homa packet type. In the C Programming Language, modules are represented in the form of functions. Meanwhile, all the packet dissection modules share a common header dissector as all types of Homa packets have the same format of the common header. Modularization can improve maintainability and readability since the modification of a single module does not affect other modules. In addition to the interface function, which is designed to be exposed to the outside, other modules within the Homa dissector are encapsulated to be used only internally. Encapsulation can improve the usability of the interface and overall security. `Figure 4.1` illustrates the architecture of the Homa dissector. `homa_printer` is the interface function provided to outside, it accepts a raw Homa packet as input, prints out packet information to users and returns.

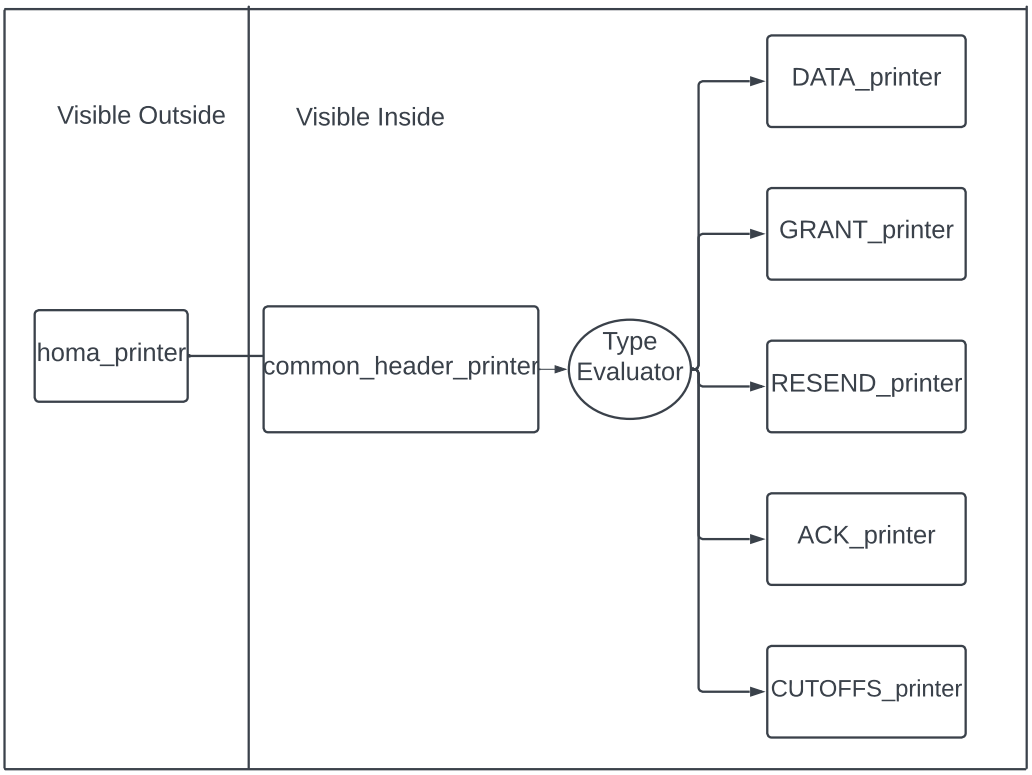

Figure 4.1: Homa Dissector

4.2 Implementation

The actual implementation of the Homa dissector has to take into account many more details than just the above architecture. These details are related to the memory safety, reliability, maintainability and usability of Homa dissector. This section is going to talk about the implementation details of the Homa dissector, including memory access, data decoding, exception detection, exception handling, packet priority, user input and integration.

4.2.1 Memory Access and Binary Data Decoding

As mentioned in the previous chapter, packets are essentially binary data. Therefore, in order to gain packet information, it is essential to be able to decode binary data.

However, an issue that needs to be addressed before decoding is the unaligned memory access. Memory Access Granularity is the minimum size of data a processor can read or write from memory at each time. The value of memory access granularity depends on the computer architecture, it can be four or eight bytes. Due to the existence

of memory access granularity, data stored in memory need to follow a special rule called Memory Natural Alignment, which requires that the start address of data to be a multiple of some particular number (e.g. four or eight, depending on the memory access granularity). Addresses within the memory that are a multiple of the particular number are called the Natural Boundary. By aligning to the Natural Boundaries, the operation of the memory access can be efficient. For example, if a four-byte integer is stored at the memory address `0x03` with a memory access granularity of four, to read the integer the processor needs to perform two memory access operations, that are reading from `0x00` to `0x03` and reading from `0x04` to `0x07`. By contrast, if this integer is stored at `0x04`, then only one memory access operation is required. However, raw packets received from network interfaces are typically compact, which means there is no gap between data. This will inevitably result in some unaligned data, as it is not always possible to satisfy compactness and alignment simultaneously. Therefore, the decoding of packet binary data will inevitably need to access to unaligned data. The operation of accessing unaligned data is defined as Undefined Behavior in the C11 standard, which means that the actual behaviour of this operation is unforeseen and may depend on computer architecture, compiler and OS. On some types of architecture (e.g. X86, arm) this operation is supported. However, the compiler might optimize the operation of unaligned access, resulting in uncontrolled behaviour in architectures that do not support it. To avoid uncontrolled behaviour on architectures that do not support unaligned access, Tcpdump wraps C memory access functions (e.g. `memcpy`, `memcmp()`) in outer functions to avoid compiler optimization.

Another issue that needs to be solved in order to decode binary data is the Byte Order. Packets always contain multi-byte data that consists of more than one byte. Byte Order defines how multi-byte data is laid out in memory. The Big-Endian means that the most significant byte (the leftmost byte) of multi-byte data is stored at the lower address of memory, while the least significant byte (the rightmost byte) is stored at the higher address. The byte order of raw packets received from network interfaces is called the Network Order, which is typically Big-Endian. By contrast, Little-Endian places the least significant byte of multi-byte data in the lower address, and the most significant byte in the higher address. `Figure 4.2` shows an example of Big-Endian and Little-Endian. Host Order is the byte order in the current computer architecture or OS, it can be Big-Endian or Little-Endian. Therefore, if the host order conflicts with the network order, the decoding of multi-byte data could be problematic. To avoid this issue, a conversion function (e.g. `ntohs()` in C) is used when decoding multi-byte data

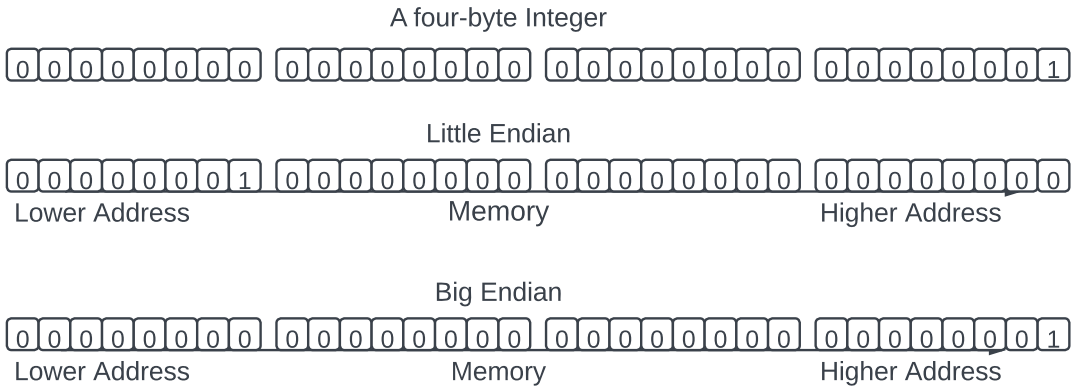

Figure 4.2: An example of Byte Order

from packets to convert data in network order to host order.

With the two major issues solved, the decoding of binary data can be done in a more secure way. The definition of various Homa packet header formats is the core to decoding binary data. For example, the definition of the Homa common header format suggests that the binary data of Homa packets start with a 16-bit integer representing the source port. By strictly following the Homa packet header formats studied in the previous chapter, the binary data of Homa packets can be conceptually divided into multiple fields and payload, with each field has its size, data type and semantics. Therefore, one way to decode binary data is to manipulate pointers in the C Programming language. The type of pointer suggests how the pointed data should be handled and decoded. By maintaining a pointer that jumps across fields within packet binary data, the value of each field can be decoded by converting the type of the pointer to the corresponding type of that field. However, this method can be error-prone and not memory safe. As there is no boundary check for accessing memory in C, it is easy to jump out of the memory of packets to access some invalid data. An alternative way that is safer in terms of memory is using compact `struct`. A `struct` in C consists of multiple members of data. Typically the members of a `struct` are laid out continuously in the memory with possible gaps to satisfy the alignment requirement in the memory. Compact `struct` forcibly removes the possible gaps between members. By defining a compact `struct` that represents each field of a specific Homa packet header using a member, the chunk of binary data treated as a `struct` can be safely decoded through accessing each member of that `struct`. In the implementation of the Homa dissector, the method of compact `struct` is adopted to decode binary data. Each `struct` is defined according to the corresponding format of the Homa packet header.

4.2.2 Exceptions Detection and Handling

As mentioned in the packet dissection section of Chapter 3, the Homa dissector should implement mechanisms to detect and handle invalid and incomplete Homa headers. Different handling strategies are used for the two exceptions. If a header is detected to be invalid, there is little can be done, since the invalidity of a header might mean that the whole packet is corrupted during transmission or generation. Thus, if the Homa header is detected to be invalid (through checking the validity of each field, e.g. the type filed of common header should be in the range 10 to 19), the Homa dissector will terminate instantly with an invalid notification provided to users. In addition to the field check, the checksum can also be performed to check the integrity of packets. However, the checksum is not implemented in the Homa dissector because (1) Homa packets do not hold a checksum field and (2) the integrity of the Homa packet might be checked in a lower layer dissector, for example in the Ethernet dissector. By contrast, the best-effort strategy is used to handle incomplete Homa headers, i.e. the Homa dissector makes the best effort to dissect as much information as possible. As the sizes of all types of headers in Homa are constant, the incompleteness of headers can be detected by comparing the actual captured size with the defined size of the header. However, an incomplete header does not always mean that no information can be extracted. If a Homa header is detected to be incomplete, the Homa dissector will still try to print information on the common header part and then terminate.

4.2.3 Packet Priority

Homa uses priorities to favour the transmission of messages with fewer remaining bytes. The priority of Homa packets is a special field, although it is specified by Homa when sending packets, the actual user of this field is the network switches instead of the Homa at the receiving end. Normally, a network switch can implement at most up to the network layer, which means that it can process at most the network layer header of a packet and cannot access the transport layer header. To enable network switches to access the priority field, it is placed in the network layer header. To be specific, Homa uses the high-order three bits of the Differentiated Services field in the IPv4 header or the high-order four bits of the Traffic Class field in the IPv6 header to hold the priority level. Therefore, the Homa dissector also needs to access the binary data of the network layer header, which is passed by the IPv4 or IPv6 dissector.

```
16:50:53.703156 ?     In  IP 192.168.11.142 > 192.168.11.6: Homa, Priority 5,  32774 > 2000 DATA, Client > Server RPCid 34
16:50:53.703156 ?     In  IP 192.168.11.142 > 192.168.11.6: Homa, Priority 5,  32774 > 2000 DATA, Client > Server RPCid 34
16:50:53.703216 ?     Out IP 192.168.11.6 > 192.168.11.142: Homa, Priority 5,  2000 > 32774 DATA, Server > Client RPCid 35
16:50:53.703292 ?     In  IP 192.168.11.142 > 192.168.11.6: Homa, Priority 5,  32774 > 2000 DATA, Client > Server RPCid 36
16:50:53.703292 ?     In  IP 192.168.11.142 > 192.168.11.6: Homa, Priority 5,  32774 > 2000 DATA, Client > Server RPCid 36
16:50:53.703323 ?     Out IP 192.168.11.6 > 192.168.11.142: Homa, Priority 5,  2000 > 32774 DATA, Server > Client RPCid 37
16:50:53.706353 ?     Out IP 192.168.11.6 > 192.168.11.142: Homa, Priority 7,  2000 > 32774 NEED ACK, Server > Client RPCid 37
```

Figure 4.3: Default Output

```
16:50:53.703156 ?     In  IP (tos 0xa0, ttl 64, id 0, offset 0, flags [DF], proto Homa (253), length 1500)
    192.168.11.142 > 192.168.11.6: Homa, 32774 > 2000 DATA, Client > Server RPCid 34 (Message Length 2840, 2840 bytes is sent, cutoff version 1)
16:50:53.703156 ?     In  IP (tos 0xa0, ttl 64, id 1, offset 0, flags [DF], proto Homa (253), length 1500)
    192.168.11.142 > 192.168.11.6: Homa, 32774 > 2000 DATA, Client > Server RPCid 34 (Message Length 2840, 2840 bytes is sent, cutoff version 1)
16:50:53.703216 ?     Out IP (tos 0xa0, ttl 64, id 0, offset 0, flags [DF], proto Homa (253), length 2940)
    192.168.11.6 > 192.168.11.142: Homa, 2000 > 32774 DATA, Server > Client RPCid 35 (Message Length 2840, 2840 bytes is sent, cutoff version 1)
16:50:53.703292 ?     In  IP (tos 0xa0, ttl 64, id 0, offset 0, flags [DF], proto Homa (253), length 1500)
    192.168.11.142 > 192.168.11.6: Homa, 32774 > 2000 DATA, Client > Server RPCid 36 (Message Length 2840, 2840 bytes is sent, cutoff version 1)
16:50:53.703292 ?     In  IP (tos 0xa0, ttl 64, id 1, offset 0, flags [DF], proto Homa (253), length 1500)
    192.168.11.142 > 192.168.11.6: Homa, 32774 > 2000 DATA, Client > Server RPCid 36 (Message Length 2840, 2840 bytes is sent, cutoff version 1)
16:50:53.703323 ?     Out IP (tos 0xa0, ttl 64, id 0, offset 0, flags [DF], proto Homa (253), length 2940)
    192.168.11.6 > 192.168.11.142: Homa, 2000 > 32774 DATA, Server > Client RPCid 37 (Message Length 2840, 2840 bytes is sent, cutoff version 1)
16:50:53.706353 ?     Out IP (tos 0xe0, ttl 64, id 0, offset 0, flags [DF], proto Homa (253), length 48)
    192.168.11.6 > 192.168.11.142: Homa, 2000 > 32774 NEED ACK, Server > Client RPCid 37
```

Figure 4.4: More Information

4.2.4 User Option and Output Content

Tcpdump provides a variety of options for users to customize the capture and dissection of packets. Options that are related to dissectors are almost all about the volume of information printed to users. These options are listed in `Table 4.1` in order of increasing amount of information. The implementation of the Homa dissector also

Option	Description
`-q`	Print less information
No option	Default volume of information
`-v`	Print more information
`-vv`	Print even more information

Table 4.1: Options Related to Dissector

supports these options. By default, the Homa dissector only prints out information in the common header, that is: priority of the Homa packet, the source and destination port, the type of Homa packet, the RPC id and an identification of client and server for the two communicating machines. `Figure 4.3` shows an example of the default output. If the `-q` option is set, the RPC id and client and server identification part are omitted from the default output. If the `-v` option is set, all information in the Homa packet header will be printed, this only works for those types of Homa packets whose header contains more than just the common header (i.e.`DATA`, `RESEND`, `CUTOFFS`, `ACK`, `GRANT` packets). `Figure 4.4` shows an example of output with `-v` option set. The information in the Homa header addition to the common header is enclosed in '()'. If the `-vv` option is

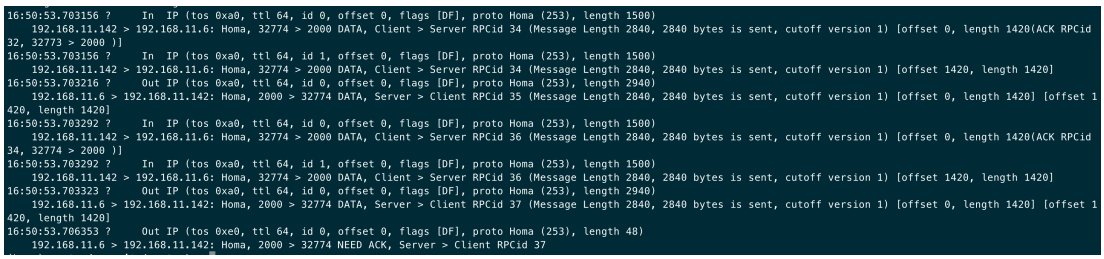

Figure 4.5: Even More Information

set, all available information will be printed. This only works for `DATA` packets, as only `DATA` packets contain traffic information in non-header places. `Figure 4.5` shows an example output with the `-vv` option set. The additional information beyond the header is enclosed in '[]'.

4.2.5 Integration

This subsection describes how the Homa dissector is integrated into Tcpdump, such that when a packet that uses Homa as its transport layer protocol is captured, the Homa dissector can be invoked correctly by a network layer dissector. Currently, the Homa dissector is linked to IPv4 and IPv6 dissectors. Both IPv4 and IPv6 headers have an 8-bit field (Protocol in the IPv4 header and Next Header in the IPv6 header) that holds the type of protocol for the next header. The list that maps numeri values in this field to specific protocols is maintained by the Internet Assigned Numbers Authority (INAN) [5]. The IPv4 and IPv6 dissectors in Tcpdump refer to this list to invoke the next dissector based on the given value in the next-protocol field. Normally, a deployed protocol needs to register with INAN to get its unique number, but INAN also defines numbers 253 and 254 to be used by experimental protocols. Currently, as Homa is still in the experimental phase, it is not formally registered with INAN, but is temporarily using the number 253.

Chapter 5

Evaluation

This chapter evaluates the result of this project - the Homa dissector in several aspects. First, several tests are performed to evaluate the reliability and functionality of the Homa dissector. Functionality evaluation includes functional integrity evaluation (i.e. whether Homa dissector provides all expected functions) and functional correctness evaluation (i.e. whether Homa dissector is able to dissect all types of packets correctly). While the reliability evaluation evaluates whether Homa dissector sustains in a large volume of traffic and exceptions. Then a critical analysis is presented to evaluate the maintainability and usability of Homa Dissector.

5.1 Functionality and Reliability

There are four sets of tests in total. The first set takes input from a static `pcap` file that contains raw Homa packets and tests the functional integrity and basic functional correctness of the Homa dissector. With the output of the static test, a subsection is also inserted to illustrate the meaning of general Homa dissector output. The other three sets take input from network interfaces that capture real Homa traffic in simulated Homa usage scenarios, testing the functionality and reliability of the Homa dissector. To be closer to the real usage scenario, the simulated scene tests are carried out in two physically separated nodes provided by CloudLab [3], a dedicated subsection is presented to illustrate the environment configuration and how simulations are done.

5.1.1 Static Test

The static test is designed to test the integrity and basic correctness of functions provided by the Homa dissector.

Input. The input of Tcpdump for this test is a `pcap` file (`homa_traffic_all_types.pcap`, which is provided by the project supervisor Dr. Honda) that contains all types of Homa packets. The Homa traffic in this file are not real, they are generated manually as it is difficult to collect all types of Homa packets in real traffic. The code used to generate Homa traffic in this `pcap` file is attached at `appendix 1`. The generation code specifies the value of some fields within Homa packets.

Tcpudmp Command. The exact Tcpdump command used in this test is `tcpdump -r homa_traffic_all_types.pcap -vv`.

Output. The output of Tcpdump for this test is:

```
(1) 14:32:58.864983 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000 DATA,
     Client > Server RPCid 4 (Retransmission, Message Length 1000, 1000
    bytes is sent, cutoff version 0) [offset 0, length 1000(ACK RPCid 2,
    32768 > 2000 )]

(2) 14:32:58.864983 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000
    FREEZE, Client > Server RPCid 4

(3) 14:32:58.864983 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000 BUSY,
     Client > Server RPCid 4

(4) 14:32:58.864983 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000 NEED
     ACK, Client > Server RPCid 4

(5) 14:32:58.865091 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000
    RESEND, Client > Server RPCid 4 (offset 101, length 202, priority 7)

(6) 14:32:58.865091 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000 ACK,
     Client > Server RPCid 4 [RPCid 15679845049023743747, 33727 > 65535]

(7) 14:32:58.865091 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000
    CUTOFFS, Client > Server RPCid 4 (cutoff version 1, [priority 0:cutoff
     1001][priority 1:cutoff -1][priority 2:cutoff 0][priority 3:cutoff
    819122368][priority 4:cutoff -1][priority 5:cutoff 809125635][priority
     6:cutoff -2084569089][priority 7:cutoff 1491306162])

(8) 14:32:58.865091 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000
    GRANT, Client > Server RPCid 4 (offset 303, priority 71)

(9) 14:32:58.865091 192.168.11.11 > 192.168.11.10: Homa 32769 > 2000 DATA,
     Client > Server RPCid 4 (Message Length 1000, 1000 bytes is sent,
    cutoff version 0) [offset 0, length 1000(ACK RPCid 2, 32768 > 2000 )]
```

Each chunk represents the dissection of a particular type of Homa packet, which

starts with a timestamp, followed by the source and destination IPv4 addresses and the dissection of the Homa packet. The link layer and network layer portion of the actual output are removed for brevity. As shown in the output, the Homa dissector is able to dissect all types of Homa packets (except the `BOGUS` packet whose format is not given). Meanwhile, the values of the output are checked against the values specified in the generation code. The result of the check shows that Homa dissector is able to dissect all types of Homa packets correctly to some extent. Therefore, through this test, the functional integrity and basic functional correctness of the Homa dissector are confirmed.

5.1.2 Illustration of Output

The first chunk is a dissection of `DATA` packet, in addition to the common header part whose fields are illustrated in the **User Option and Ouptut Content** section, the content wrapped in `()` contains **(1)** a flag indicating that this `DATA` packet is sent as a response to a `RESEND` packet, (2) the total length of the message to which the data belongs, **(3)** the offset (in bytes) within the message that the receiver can expect to receive, **(4)** the latest version of `CUTOFF` packet received from the receiver. The content wrapped in `[]` is the sub-header of a data segment, which contains **(1)** the offset within the message of data carried by this segment, **(2)** the size of the data, **(3)** the acknowledgement of completion of particular RPC, which can be apart from the RPC of the current packet and is optional.

The second, third and fourth chunks are dissections of `FREEZE`, `BUSY` and `NEED ACK` packets respectively. These types of packets contain only the common header.

The fifth chunk is the dissection of a `RESEND` packet, in addition to the common header part, the content wrapped in `()` contains **(1)** offset within the message of data that needs to be retransmitted, **(2)** the length of retransmission data and **(3)** the priority to be used by the retransmission data. The sixth chunk is the dissection of a `ACK` packet, which contains an acknowledgement of completion of a particular RPC, wrapped in `[]`. A three-tuple - RPCid, source and destination port number is used to uniquely identify an RPC network-wide. The seventh chunk is the dissection of a `CUTOFF` packet, the content wrapped in `()` contains **(1)** the version of current `CUTOFF packet`, **(2)** the maximum message size that can use each priority level. The eighth chunk is the dissection of a `GRANT` packet, the content wrapped in `()` contains **(1)** an offset within the message that all data up to that offset is granted to be transmitted, **(2)** the priority

should be used by new DATA packets.

5.1.3 Environment Configuration and Homa Application

Software	Version
Homa Linux Version	6.1.38
Linux Kernel Version	6.1.38
Libpcap Version	1.11.0-PRE-GIT
Tcpdump version	5.0.0-PRE-GIT

Table 5.1: Environment Configuration

Table 5.1 shows the version of required software in simulated scene tests. The Homa transport protocol is built from Homa\Linux [12], inserted as a Linux kernel module. The Tcpdump version is the base version on which the Homa dissector is implemented.

To simulate usage of Homa and to generate real Homa traffic, two groups of applications are adopted, each group consisting of a client and a server that communicate with each other using Homa. The client end is responsible for sending request messages to and receiving response messages from server, while the server end is responsible for receiving requests, generating and sending responses. The first group of Homa Applications is implemented within the project, which performs a total of five RPCs, when all RPCs are finished, the client end will output the information of each RPC (including RPCid, request and response message). The second group adopts the cp_node program provided by Homa\Linux [12], which performs RPCs communication continuously until it is interrupted.

5.1.4 Simulated Scene Test 1

The static test in the previous section confirms that Homa dissector is able to dissect manually generated Homa packets correctly. This test further evaluates the functional correctness of Homa dissector against real Homa traffic. This test adopts the first group of Homa applications to generate real Homa traffic. Each application (server or client) runs on a different node, the extended Tcpudmp is launched on both nodes to capture packets from all available network interfaces.

Input. The input of Tcpdump for this test is the real Homa traffic generated by client and server.

Tcpdump Command. The exact Tcpdump command used in this test is `tcpdump -i any -vv`

Output. The output of Tcpdummp is:

```
(1) 12:48:19.827125 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
     3000 > 9999 DATA, Client > Server RPCid 5404864 (Message Length 10,
    10 bytes is sent, cutoff version 1) [offset 0, length 10(ACK RPCid
    5404862, 3000 > 9999 )]

(2) 12:48:19.827138 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
     3000 > 9999 DATA, Client > Server RPCid 5404866 (Message Length 10,
    10 bytes is sent, cutoff version 1) [offset 0, length 10(ACK RPCid
    5404860, 3000 > 9999 )]

(3) 12:48:19.827144 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
     3000 > 9999 DATA, Client > Server RPCid 5404868 (Message Length 10,
    10 bytes is sent, cutoff version 1) [offset 0, length 10(ACK RPCid
    5404858, 3000 > 9999 )]

(4) 12:48:19.827150 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
     3000 > 9999 DATA, Client > Server RPCid 5404870 (Message Length 10,
    10 bytes is sent, cutoff version 1) [offset 0, length 10(ACK RPCid
    5404856, 3000 > 9999 )]

(5) 12:48:19.827156 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
     3000 > 9999 DATA, Client > Server RPCid 5404872 (Message Length 10,
    10 bytes is sent, cutoff version 1) [offset 0, length 10(ACK RPCid
    5404854, 3000 > 9999 )]

(6) 12:48:19.828156 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
     9999 > 3000 DATA, Server > Client RPCid 5404865 (Message Length 19,
    19 bytes is sent, cutoff version 1) [offset 0, length 19]

(7) 12:48:19.828185 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
     9999 > 3000 DATA, Server > Client RPCid 5404867 (Message Length 19,
    19 bytes is sent, cutoff version 1) [offset 0, length 19]

(8) 12:48:19.828203 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
     9999 > 3000 DATA, Server > Client RPCid 5404869 (Message Length 19,
    19 bytes is sent, cutoff version 1) [offset 0, length 19]

(9) 12:48:19.828223 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
     9999 > 3000 DATA, Server > Client RPCid 5404871 (Message Length 19,
    19 bytes is sent, cutoff version 1) [offset 0, length 19]

(10) 12:48:19.828240 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us:
    Homa 9999 > 3000 DATA, Server > Client RPCid 5404873 (Message Length
    19, 19 bytes is sent, cutoff version 1) [offset 0, length 19]
```

The link layer and network layer parts of dissections are also removed. The first five dissections represent the five requests sent from the client to the server. The remainder represents the five responses sent back to the client. The output of the client end is:

```
rpcid: 5404864, request: Request:5, response: Request:5 Response
rpcid: 5404866, request: Request:4, response: Request:4 Response
rpcid: 5404868, request: Request:3, response: Request:3 Response
rpcid: 5404870, request: Request:2, response: Request:2 Response
rpcid: 5404872, request: Request:1, response: Request:1 Response
```

By checking each entry of the dissection against the corresponding RPC of the client-side output, it can be shown that the Homa dissector is able to dissect real Homa traffic correctly. (Note that although the length of the request message and response message shown in client output are 9 bytes and 18 bytes respectively, the size of actual messages is one byte larger to accommodate a `'\0'` character). Therefore, this test further confirms the correctness of the Homa dissector functions.

5.1.5 Simulated Scene Test 2

This test evaluates the reliability of the Homa dissector against truncation. The setup of this test is the same as the Simulated Scene Test 1 except that the packets received by Tcpdump are truncated intentionally.

Input. The input of Tcpdump for this test is real truncated Homa traffic generated by client and server.

Tcpdump Command. The exact Tcpdump command used in this test is `tcpdump -i any -s 90 -vv` , where the `-s` option suggests that only the first 90 bytes of each packet are captured. As the link layer header of packets captured from `any` interface is 16 bytes long, meanwhile there are no optional fields in the IPv4 header, this value can incorporate up to the common header of a `DATA` packet.

Output. The output of Tcpdump is:

```
(1)14:19:37.320643 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
    3000 > 9999 DATA, Client > Server RPCid 5404904 (Message Length 10, 10
     bytes is sent, cutoff version 1) [|homa]

(2)14:19:37.320656 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
    3000 > 9999 DATA, Client > Server RPCid 5404906 (Message Length 10, 10
     bytes is sent, cutoff version 1) [|homa]

(3)14:19:37.320662 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
    3000 > 9999 DATA, Client > Server RPCid 5404908 (Message Length 10, 10
     bytes is sent, cutoff version 1) [|homa]
```

```
(4)14:19:37.320669 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
    3000 > 9999 DATA, Client > Server RPCid 5404910 (Message Length 10, 10
     bytes is sent, cutoff version 1) [|homa]

(5)14:19:37.320677 hp012.utah.cloudlab.us > hp040.utah.cloudlab.us: Homa
    3000 > 9999 DATA, Client > Server RPCid 5404912 (Message Length 10, 10
     bytes is sent, cutoff version 1) [|homa]

(6)14:19:37.320564 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
    9999 > 3000 DATA, Server > Client RPCid 5404905 (Message Length 19, 19
     bytes is sent, cutoff version 1) [|homa]

(7)14:19:37.320587 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
    9999 > 3000 DATA, Server > Client RPCid 5404907 (Message Length 19, 19
     bytes is sent, cutoff version 1) [|homa]

(8)14:19:37.320600 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
    9999 > 3000 DATA, Server > Client RPCid 5404909 (Message Length 19, 19
     bytes is sent, cutoff version 1) [|homa]

(9)14:19:37.320612 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
    9999 > 3000 DATA, Server > Client RPCid 5404911 (Message Length 19, 19
     bytes is sent, cutoff version 1) [|homa]

(10)14:19:37.320625 hp040.utah.cloudlab.us > hp012.utah.cloudlab.us: Homa
     9999 > 3000 DATA, Server > Client RPCid 5404913 (Message Length 19,
    19 bytes is sent, cutoff version 1) [|homa]
```

The output of client end is:

```
rpcid: 5404904, request: Request:5, response: Request:5 Response
rpcid: 5404906, request: Request:4, response: Request:4 Response
rpcid: 5404908, request: Request:3, response: Request:3 Response
rpcid: 5404910, request: Request:2, response: Request:2 Response
rpcid: 5404912, request: Request:1, response: Request:1 Response
```

As shown in the output of Tcpdump, the Homa dissector prints all information up to the common header and terminates with an indication of truncation (the `[|homa]`). Therefore, this test confirms that the Homa dissector is reliable in the face of truncation.

5.1.6 Simulated Scene Test 3

This test evaluates the reliability of the Homa dissector against a large amount of Homa traffic. The setup is the same as Simulated Scene Test 1 except that the second group of Homa applications is adopted. The client and server ends are terminated approximately 30 seconds after being launched.

Input. The input of Tcpdump for this test is a large amount of real Homa traffic.

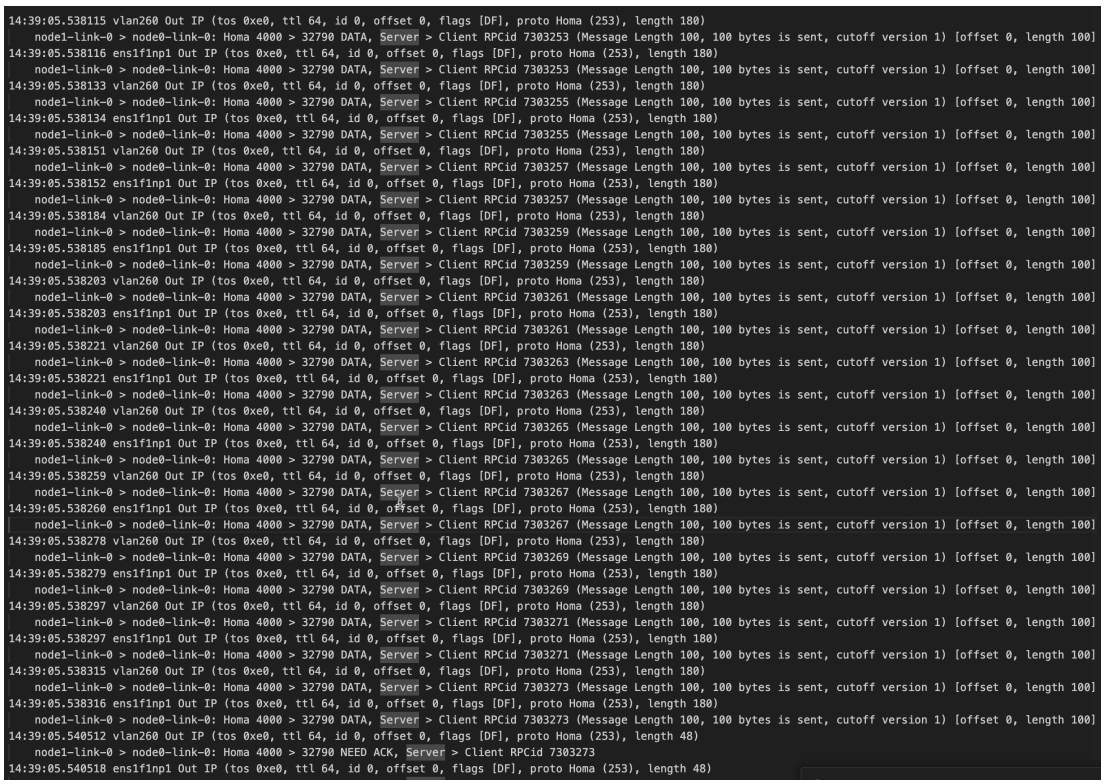

Figure 5.1: A fragment of output

Tcpdump Command. The exact Tcpdump command used in this test is `tcpdump -i any -vv`.

Output. The amount of data output in this test is extremely large, containing more than fifty thousand entries of dissections. Figure 5.1 shows a fragment of output. Through this test, it can be confirmed that Homa dissector is able to sustain massive data.

5.2 Maintainability and Usability

5.2.1 Maintainability

As described in section 4.1, the architecture of the Homa dissector is designed to be modularized, with each module responsible for the dissection of a certain type of Homa packet. Therefore, the modification of a specific type of Homa packet would only affect the corresponding module. Meanwhile, if new types of packets are added to Homa, the Homa dissector only needs to implement new modules without concerning any existing codes. So, it can be concluded the Homa dissector is generally maintainable.

5.2.2 Usability

Usability can be further divided into user usability and programmer usability. In terms of user, the usability of the Homa dissector is inherently the usability of Tcpdump, which is not very usable as a command line tool. Users need to understand and memorize the meaning of each option in order to fulfil their objectives. However, once familiar with these options, input in the command line can be more efficient than clicking through the GUI. In terms of programmers, the interface provided by the Homa dissector is usable. It abstracts the necessary information to programmers and hides unnecessary implementation details in encapsulation.

Chapter 6

Conclusion

6.1 Limitations

6.1.1 Limitations On Implementation

The implementation of the Homa dissector has several limitations. First, the truncation handling mechanism can still be improved. To be specific, the granularity of the truncation handling can be finer. When a truncation is detected, instead of making the best effort to parse the available data chunk by chunk (e.g. if a `DATA` header is truncated, check if the common header part is complete), it can be finer to parse the available data field by field. For example, if the common header of a Homa packet is truncated, the Homa dissector could make the best effort to parse those complete fields within the common header. Second, the function of detecting invalid packets is imperfect, the current implementation is only able to detect invalid fields whose values are out of the predefined range. Some other invalidity, for example, the values of fields are inconsistent with the actual situation, can be detected with a more perfect invalidity detection mechanism. Third, although Homa fragments data from application layer protocols, the current Homa dissector does not implement any fragmentation handling functions or links to existing application layer dissectors.

6.1.2 Limitation On Test

The tests performed on the Homa dissector are limited. First, since Homa applications cannot directly control the type of packets sent by Homa protocol, it is difficult to generate all types of Homa packets in real traffic. As a result, the real Homa traffic generated in several tests used to test several aspects of the Homa dissector contains

only partial types of Homa packets. Second, due to the same reason of not being able to control the Homa kernel module to generate real invalid packets, the reliability of the Homa dissector against invalid packets is not tested.

6.2 Future Work

6.2.1 Link With Application Dissectors

It may take some work to link the Homa dissector with all existing application dissectors. This is because (1), Homa fragments the message from the upper layer, so the Homa dissector needs to implement a mechanism to identify the first fragment of each message, as the remaining segments must not contain complete header data. (2) All types of Homa packets contain no field to indicate the type of protocol for the next layer, such that the Homa dissector might need to evaluate part of the data of the next layer to determine the type of protocol.

6.2.2 Tests That Are Closer To Usage Scene

Since Homa's usage scenarios are mostly in the data center, which can be much more complex than a single client and server. Thus, to be closer to the usage scene of Homa, larger-scale tests can be performed in the future with a dozen applications running on a cluster of nodes, each application can serve as a client and server simultaneously.

6.2.3 Contribute To Tcpdump

The implementation of the Homa dissector has met most of the requirements to contribute to the repository of Tcpdump [4], except for testing. For any extension, Tcpdump requires incorporating several tests that read from `pcap` files and compare the output of the extension with the correct output (which can be generated manually or by the extension itself). The static test performed in this project is similar to these required tests, except that the static test compares the output with the correct output manually. Therefore, to be able to contribute to Tcpdump, in the future, more `pcap` files need to be collected to create more tests, meanwhile, the test should be automated by using the CTest framework.

6.2.4 Overall Conclusion

To conclude, this project implemented a Homa dissector, which is an extension module of Tcpdump that is able to decode all types of raw Homa packets and print information to users. To implement the Homa dissector, a conceptual study was performed in early work to gain a deep insight into the working principle of Tcpdump and the formats for Homa packets. The Homa dissector was implemented based on a modularized design, meanwhile a lot of details were covered to make it safer and more reliable. Many tests including a static test that read input from `pcap` files and simulated scene tests that capture real Homa traffic were performed to evaluate the functionality and reliability of the Homa dissector. A critical analysis was presented to evaluate the maintainability and usability of the Homa dissector. It can be confirmed that the Homa dissector is generally reliable, maintainable and usable for programmers. Moreover, it can also be confirmed that Homa has provided all the required functions correctly. However, there still exist limitations, the current Homa dissector is not linked with other application dissectors, meanwhile, the exception detection and handling mechanisms within the Homa dissector are imperfect. Several future works were brought up to solve some of those imperfections and to further improve the quality of this project, including performing more tests in the context of data centers and automating static tests.

Bibliography

[1] All Link Types Supported by Libpcap. Available at https://www.tcpdump.org/linktypes.html.

[2] An Implementation of Homda Dissector in Wireshark. Available at https://github.com/PlatformLab/HomaModule/pull/43//.

[3] CloudLab. Available at https://www.cloudlab.us//.

[4] Code Repository of Tcpdump.
Available at https://github.com/the-tcpdump-group/tcpdump.

[5] IANA Lists of Protocol Numbers. Available at https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml/.

[6] IEEE 802.1Q Standard for Virtual Local Area Network. Available at https://standards.ieee.org/ieee/802.1Q/6844//.

[7] TCPDUMP&LIBPCAP. Available at https://www.tcpdump.org/.

[8] Wireshark Packet Dissector. Available at https://github.com/wireshark/wireshark.

[9] Kashif Bilal, Saif Ur Rehman Malik, Osman Khalid, Abdul Hameed, Enrique Alvarez, Vidura Wijaysekara, Rizwana Irfan, Sarjan Shrestha, Debjyoti Dwivedy, Mazhar Ali, Usman Shahid Khan, Assad Abbas, Nauman Jalil, and Samee U. Khan. A taxonomy and survey on green data center networks. *Future Generation Computer Systems*, 36:189–208, 2014. Special Section: Intelligent Big Data Processing Special Section: Behavior Data Security Issues in Network Information Propagation Special Section: Energy-efficiency in Large Distributed Computing Architectures Special Section: eScience Infrastructure and Applications.

[10] J.D. Day and H. Zimmermann. The osi reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[11] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, dec 2009.

[12] John Ousterhout. Homa/Linux Kernel Module. Available at https://github.com/PlatformLab/HomaModule/tree/main.

[13] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[14] John Ousterhout. A linux kernel implementation of the homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 99–115, 2021.

[15] Mohammed Abdul Qadeer, Arshad Iqbal, Mohammad Zahid, and Misbahur Rahman Siddiqui. Network traffic analysis and intrusion detection using packet sniffer. In *2010 Second International Conference on Communication Software and Networks*, pages 313–317, 2010.

[16] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. 24(3):68–79, jul 1990.

[17] Wenfeng Xia, Peng Zhao, Yonggang Wen, and Haiyong Xie. A survey on data center networking (dcn): Infrastructure and operations. *IEEE Communications Surveys Tutorials*, 19(1):640–656, 2017.

.1 Appendix 1 Code to generate Homa traffic in the input of static test

```
void homa_xmit_data(struct homa_rpc *rpc, bool force)

    homa_rpc_unlock(rpc);
    skb_get(skb);

    homa_xmit_unknown(skb, rpc->hsk);

    struct freeze_header freeze;
```

```
homa_xmit_control(FREEZE, &freeze, sizeof(freeze), rpc);

struct busy_header busy;
homa_xmit_control(BUSY, &busy, sizeof(busy), rpc);

struct need_ack_header h;
homa_xmit_control(NEED_ACK, &h, sizeof(h), rpc);

struct resend_header resend;
resend.priority = 7;
resend.offset = htonl(101);
resend.length = htonl(202);
homa_xmit_control(RESEND, &resend, sizeof(resend), rpc);

struct ack_header ack;
ack.num_acks = htons(1);
homa_xmit_control(ACK, &ack, sizeof(ack), rpc);

struct cutoffs_header h2;
h2.unsched_cutoffs[0] = htonl(1001);
h2.cutoff_version = htons(1);
homa_xmit_control(CUTOFFS, &h2, sizeof(h2), rpc);

struct grant_header grant;
grant.offset = 7;
grant.offset = htonl(303);
homa_xmit_control(GRANT, &grant, sizeof(grant), rpc);

__homa_xmit_data(skb, rpc, priority);
force = false;
homa_rpc_lock(rpc);
```