A tool for measuring available network bandwidth in the cloud

GANAPATHY RAMAN MADANAGOPAL

Master’s Degree Project
Stockholm, Sweden
September 24, 2015

TRITA-ICT-EX-2015:193
Abstract

Measurement of available path capacity with high accuracy over high-speed links deployed in cloud and transport backbones is vital for Internet Service Providers (ISP) and for various network applications to know how congested the different links are in a network and also to do traffic engineering efficiently. State-of-the-art active probing method BART (Bandwidth Available in Real-Time) measures the available path capacity by injecting the timestamped measurement packets into the network. Active probing measurement technique deeply relies on the ability to generate probe packets at required rate and time-stamped with high precision. While dealing with high-speed links (10+ Gbps) links, it is a challenging task for the measurement system to generate and time stamp the probe packets within few nano-seconds.

This thesis focus on finding a suitable architecture and an optimized algorithm to send and receive measurement packets over high-speed links. Five different architectures: Native UDP sockets, LibPcap, Loadable Kernel Modules, Netmap and Data Plane Development Kit (DPDK) are investigated for fast packets processing in Linux systems. Also, the optimized algorithm(s) that suits the specific architecture for sending and receiving measurement packets at various rates are designed. The performance metrics obtained from the measurement systems tested against various rates and burst size using different architectures are compared and the observations are discussed and analysed in detail to find the suitable architecture.
Sammanfattning

Mätning av tillgänglig kapacitet i datanät med hög noggrannhet över höghastighetslänkar i datacenter och transportnät är grundläggande för att ISP:er och olika nätapplikationer ska kunna veta hur överbelastade de olika länkarna är och för att kunna planera trafikflöden på ett effektivt sätt. Den senaste tekniken för aktiva mätningar, BART (Bandwidth Available in Real Time), mäter den tillgängliga kapaciteten hos en nätverksväg genom att sända tidsstämplade mätpaket genom nätet. Aktiv mätning är helt beroende av möjligheten att kunna generera mätpaket med önskad hastighet och att kunna tidsstämpla paketen med hög precision. Vid mätning över höghastighetslänkar (10+ Gbps) är den tekniska utmaningen att generera och tidsstämpla mätpaketen med några få nanosekunders noggrannhet.

Acknowledgements

I would like to express my gratitude to the people who were closely working with me during this master thesis, namely:

*Markus Hidell*, my examiner and academic supervisor at KTH, for accepting the master thesis and guiding me through the project. Your swift feedback and your support on various administrative issues helped me to complete the thesis.

*Azimeh Sefidcon*, my manger at Ericsson Research for selecting me and providing the opportunity to carry out my thesis in such prestigious organization.

*Christofer Flinta* and *Andreas Johnsson*, for providing the required background information to carry out the thesis and also helped me by providing necessary inputs and feedback on the thesis results, and on my report, which helped me to finish my thesis faster.

Special thanks to *Daniel Turull*, my supervisor at Ericsson Research, for the valuable guidance throughout the thesis, providing necessary test equipments, arranging my work space, providing necessary feedbacks and inputs, resolving technical and organizational issues, organizing meetings with people interested in the results of the project and helping in every aspect of this thesis.

Finally, I would like to thank all the people, especially my family and my friends who has helped, supported and encouraged me when required the most.

Sincerely,

Ganapathy Raman Madanagopal

September 24, 2015
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Sammanfattning</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Acronyms and Abbreviations</td>
<td>viii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Description</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Outline of the thesis</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Methodology</td>
<td>3</td>
</tr>
<tr>
<td>2 Background</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Bandwidth Available in Real-Time (BART)</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Two-Way Active Measurement Protocol (TWAMP)</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 TWAMP Architecture</td>
<td>5</td>
</tr>
<tr>
<td>2.2.2 TWAMP Light</td>
<td>6</td>
</tr>
<tr>
<td>2.2.3 Extension of TWAMP Protocol</td>
<td>7</td>
</tr>
<tr>
<td>3 Research Challenges</td>
<td>8</td>
</tr>
<tr>
<td>4 Existing technologies that can be considered for faster packet processing</td>
<td>11</td>
</tr>
<tr>
<td>4.1 LibPcap</td>
<td>12</td>
</tr>
<tr>
<td>4.2 Kernel Module</td>
<td>13</td>
</tr>
<tr>
<td>4.2.1 NetLink Sockets</td>
<td>13</td>
</tr>
<tr>
<td>4.2.2 Netfilter</td>
<td>14</td>
</tr>
<tr>
<td>4.3 Netmap</td>
<td>14</td>
</tr>
<tr>
<td>4.4 DPDK</td>
<td>16</td>
</tr>
<tr>
<td>5 Methodology and Implementation</td>
<td>18</td>
</tr>
<tr>
<td>5.1 Requirements</td>
<td>18</td>
</tr>
<tr>
<td>5.2 Packet sending algorithm</td>
<td>19</td>
</tr>
<tr>
<td>5.3 Implementation</td>
<td>20</td>
</tr>
<tr>
<td>5.3.1 Standard UDP - Pseudocode for sender and reflector</td>
<td>20</td>
</tr>
<tr>
<td>5.3.2 LibPcap - Pseudocode for sender and reflector</td>
<td>23</td>
</tr>
</tbody>
</table>
6 Results and Analysis
6.1 Comparing results from different architecture ................. 37
6.2 Evaluation of DPDK performance against different rates and burst size 44
6.3 Pros and Cons of the different techniques ......................... 51
   6.3.1 Using Standard UDP Socket .............................. 51
   6.3.2 Using LibPcap .......................................... 51
   6.3.3 Using Kernel Module .................................. 52
   6.3.4 Using Netmap ............................................ 52
   6.3.5 Using DPDK ............................................. 53
6.4 Timestamping and High resolution timers ......................... 54

7 Conclusions and Future work ........................................ 56
7.1 Conclusions .................................................... 56
7.2 Future Works .................................................... 57
7.3 Reflections on Environmental, Social, Ethical and Economical impact by the thesis ............................................. 58

References .............................................................. 63
List of Figures

1. TWAMP Light Architecture ........................................ 6
2. Netmap High Level Architecture .................................. 15
3. DPDK High Level Architecture .................................... 17
4. Packet sending algorithm .......................................... 19
5. TWAMP Sender module using UDP socket ....................... 21
6. TWAMP Reflector module using UDP socket ...................... 22
7. TWAMP Sender module using LibPcap socket .................... 24
8. TWAMP Reflector module using LibPcap socket .................. 25
9. TWAMP Sender module using Kernel Module ..................... 28
10. TWAMP Reflector module using Kernel Module ............... 29
11. TWAMP Sender module using Netmap Module .................... 31
12. TWAMP Reflector module using Netmap Module ............... 32
13. TWAMP Sender module using DPDK ............................. 34
14. TWAMP Reflector module using DPDK ......................... 35
15. Test setup topology .............................................. 36
16. Sending rates for different architectures at Sender ........... 38
17. Mean Absolute Percentage Error for different architectures at Sender 40
18. Standard Deviation for different architectures at Sender ........ 40
19. Receiving rates for different architectures at Receiver ........ 41
20. Scenario illustrating the problem leading to erroneous calculation in receiving rates due to packet buffering .................. 42
21. Mean Absolute Percentage Error for different architectures at Receiver 43
22. Standard Deviation for different architectures at Receiver .......... 43
23. DPDK performance against different rates and train size ........ 45
24. DPDK performance for the rate 1Gbps against different burst size . 45
25. DPDK performance for the rate 2 Gbps against different burst size . 46
26. DPDK performance for the rate 5 Gbps against different burst size . 46
27. DPDK performance for the rate 8 Gbps against different burst size . 47
28. DPDK performance for the rate 10 Gbps against different burst size . 47
29. Scenario illustrating percentage of error observed for different burst size 48
30. DPDK performance for the burst size 2 packets at different rates . 49
31. DPDK - Standard deviation for different rates and burst size on Sender 50
32. DPDK - Standard deviation for different rates and burst size on Receiver 50
33. TWAMP reflector packet format ................................. 65
34. Ericsson TWAMP extension sender packet format ............ 66
List of Tables

1  Time taken by sendto() system call .......................... 9
2  Results from different timestamping options .................. 55
3  Test device specifications ....................................... 64
List of Acronyms and Abbreviations

API  Application Programming Interface
BART  Bandwidth Available in Real Time
BPF  Berkeley Packet Filter
CPU  Central Processing Unit
DMA  Direct Memory Access
DPDK  Data Plane Development Kit
EAL  Environment Abstraction Layer
Gbps  Giga Bits Per Second
ICMP  Internet Control Message Protocol
IOCTL  Input/Output Control
IP  Internet Protocol
ISP  Internet Service Provider
KVM  Kernel-Based Virtual Machine
LKM  Loadable Kernel Modules
Mpps  Million Packets Per Second
MTU  Maximum Transmission Unit
NIC  Network Interface Controller
OWAMP  One-Way Active Measurement Protocol
PCAP  Packet Capture
PMD  Poll Mode Drivers
SNMP  Simple Network Management Protocol
TWAMP  Two-Way Active Measurement Protocol
UDP  User Datagram Protocol
1 Introduction

Cisco predicts the global Internet traffic will quadruple by 2015 and expected to reach 1 zetta bytes per year [1]. With the explosion of internet traffic, measurement of link capacity (a physical property of a link) and available bandwidth (the unused portion of the link capacity) with high accuracy over the network path is vital for Internet Service Providers (ISP) and for various network applications [2] to know how congested the different links are in a network and to do traffic engineering efficiently. According to [3], 70% of business are using or exploring cloud services today, which includes data centers migrated to cloud. Also [3] states that 45% of organizations cite bandwidth requirements as a barrier to cloud adoption. In order to function efficiently and achieve the required Quality-of-Service, accurate measurement of available path capacity has to be made.

Generally, the links deployed in the cloud and transport backbones are high-speed links with capacity 10 Gbps or more. A basic solution for measuring capacity in any kind of network are estimated using a file download or similar techniques. Also tools such as ICMP ‘Ping’ [4], Traceroute [5], UDP Echo [6] are popularly used. However, these tools were not intended to conduct the capacity measurements over high-speed links but were designed for simple troubleshooting of IP networks.

Generally the link capacity measurements are done using ‘Passive’ method by calculating the available link capacity at all the nodes in the path. Passive measurements systems attempts to measure the performance of the network without modifying the traffic or injecting measurement probe packets. Performance monitoring with Simple Network Management protocol (SNMP) [7] is good example for passive network monitoring. Sometimes, it is difficult for the network managers to gain access to intermediate nodes for performance measurements. In such cases ‘Active’ measurement is used by injecting measurement packets, so called probe packets into the network for measuring the available bandwidth.

State-of-the-art active probing methods are BART (Bandwidth Available in Real-Time) [8], Pathload [9], Pathchirp [10], Spruce [11], PTR [12] and TOPP [13] [14]. The basic principle is to inject probe packets into the network. These packets are time-stamped at sender and receiver. By analysing these time stamps, estimation of available bandwidth can be made. No prior knowledge of the underlying topology is needed make bandwidth estimations methods well-suited for end-to-end performance measurements. Above mentioned methods differs in flight pattern, i.e. how the probe packets are sent and in the estimation algorithm used. The overview of available bandwidth and tools can be found in this [15]

Many active probing methods for measuring available path capacity use short trains of packets. Active probing measurement deeply relies on the ability to generate probe packets at required rate and time-stamped with high precision. While dealing
with high-speed links (10+ Gbps) links, it is a challenging task for the host system to generate and time stamp the probe packets within few nano-seconds. Slight deviation from the required values can lead to erroneous results while measuring the end-to-end performance. Available path capacity measurements over high-speed links has always been a challenge.

1.1 Problem Description

The BART relies on the fact of generating measurements packets at a precise rate with accurate timestamps. But it is challenging for the BART applications built over the native Linux sockets to send and receive measurement packets at high-speed with accurate time-stampings over high-speed links with the capacity greater than 10Gbps where the inter-packet gap is in-order of few nano-seconds. So while measuring the performance of the monitored links, the BART measurement packets spends more time in the kernel-space than the flight time between the measurement devices (i.e. on the links). This results in erroneous calculation of available path capacity, round-trip time and other end-to-end path performance measurements. So it necessary to find a suitable architecture by which BART measurement packets can be generated and sent at required rate with accurate timestamping.

The thesis work is carried out with an proposition that revamping the architecture of the existing packet generation module with a new architecture to send and receive packets at higher rates has less or no correlation with the modules which calculates the network performance (like available bandwidth, round-trip time, latency).

1.2 Outline of the thesis

The thesis consist of 7 chapters. The chapter 1 is introduction, which briefly introduces the basic concepts of passive and active network performance measurement, the problem statement and the methodology adapted for the research. Chapter 2 presents the background necessary for the thesis. Chapter begins with the information about BART and the need for standard measurement protocol. Information about Two-way Active Measurement Protocol (TWAMP), the measurement protocol that will be used in this thesis. TWAMP’s architecture and extensions in the protocol. Chapter 3 presents the research challenge that will addressed by this thesis. In chapter 4, some related and existing works that can be considered for addressing the problem mentioned in this thesis are discussed. Chapter 5 explains the methodology adapted for the research and the high-level design of the various architectural approaches used for building the transmission and reflection modules on TWAMP sender and reflector. This chapter also introduces the tools and experimental setup used for measuring the performance of a network and for analysis. In Chapter 6, the experimental results obtained for different architecture
are discussed in detail to find out the suitable architecture which supports TWAMP protocol to send and reflect measurement probes over high-speed links. In Chapter 7, thesis results are given as conclusions and possible future works are discussed.

1.3 Methodology

This master’s thesis project incorporates ‘Quantitative’ research technique. The research is carried out with an ‘Experimental’ approach, which is generally used when a theoretical analysis is inadequate or infeasible. The experimental approach was chosen because there was no theoretical basis to meet the goal of the thesis. The research question and evaluation metrics in the form of requirements are identified, then an experimental study is conducted. Also ‘Iterative’ process was chosen so that results obtained can be refined incrementally. During the research initially the functional correctness of the solution was aimed. Later the solution is fine tuned to obtain better performance. Finally the obtained results are analysed and evaluated against the thesis requirements.
2 Background

This chapter provides essential background knowledge for understanding this thesis work. Section 2.1 briefly explains the concepts for BART and calculation of available bandwidth over end-to-end path with multiple links. Section 2.2 introduces the measurement protocol (TWAMP) which forms the base for this thesis, and its advantage over OWAMP are discussed. Following that, TWAMP architecture, TWAMP Light: a variation in TWAMP protocol, and extension to the TWAMP Light by Ericsson are discussed in sections 2.2.1, 2.2.2 and 2.2.3 respectively.

2.1 Bandwidth Available in Real-Time (BART)

BART [8] is used for estimating the end-to-end available path capacity over a network path. It estimates the bandwidth quasi-continuously, in real-time. It relies on self-induced congestion, and repeatedly samples the available bandwidth of the network path with sequences of probe packet pairs, sent at required rate. The ‘Available Bandwidth’ also referred as Available Path Capacity (APC), intuitively means the unused capacity of the link. However available path capacity is a dynamic metric that changes over time and depends on the utilization of link by the traffic at instance of time $t$.

According to RFC 5136 [16] section 2.3 and ITU-T Y.1540 [17] section 6.11, the capacity of the link, can be defined as the maximum number of IP-layer bits that can be transmitted from source $S$ and correctly received at the Destination $D$ over the link during the time interval $[t, \Delta t]$, divided by $\Delta t$. In an end-to-end path metric measurement might consist of more than one links. Given an end-to-end path $P$ with $N$ links, the path capacity simply becomes that of the link with the smallest capacity along that path.  

$$C(P,t,\Delta t) = \min_{l=1...N} C(l,t,\Delta t) \tag{1}$$

The average usage of a link $l$, $Used(l,t,\Delta t)$, is the actual number of IP-layer bits from the source, correctly received over link $l$ during the interval $[t, \Delta t]$, divided by $\Delta t$. The utilization of the overall IP-layer link capacity is expressed as

$$Util(l,t,\Delta t) = \frac{Used(l,t,\Delta t)}{C(l,t,\Delta t)} \tag{2}$$

The utilization represents the fraction of the capacity that is being used. The value lies between 0 (indicating nothing is used) and 1(indicating the link is fully saturated). Multiplying the utilization by 100 yields the percent utilization of the link.

\footnote{An alternate way to calculate APC: M.Sedighizad,B.Seyfe,K.Navaie,“MR-BART: Multi-Rate Available Bandwidth Estimation in Real-Time” URL: http://arxiv.org/pdf/0908.2402.pdf}
The available link capacity on a congested link can be obtained by multiplying the IP-layer link capacity with the complement of the IP-layer link utilization.

\[ \text{AvailCap}(l, t, \Delta t) = C(l, t, \Delta t) \times (1 - \text{Used}(l, t, \Delta t)) \] (3)

The over the entire path, the ‘Available Path Capacity’ (APC) simply becomes that of the link with the smallest available capacity along that path.

\[ \text{AvailCap}(P, t, \Delta t) = \min_{t=1, \ldots, N} \text{AvailCap}(l, t, \Delta t) \] (4)

BART uses active probing method to measure the available bandwidth. In order to send the measurement packets and to support interoperability among the devices in the network, a standardized protocol is required to monitor the performance of the enterprise networks effectively.

One-way Active Measurement Protocol (OWAMP) specified in RFC 4656 [18], developed by IETF’s IP performance metrics working group provides a common protocol for measuring one-way metrics between network devices. OWAMP can be used bi-directionally to measure one-way metrics in both directions between two network elements. But OWAMP does not accommodate round-trip or two-way measurements. So Two-way Active Measurement Protocol (TWAMP) as specified in RFC 5357 [19] has gained popularity over OWAMP has been implemented and used widely. TWAMP is the first standardized protocol of its kind as it collects two-way performance data in one measurement [20].

2.2 Two-Way Active Measurement Protocol (TWAMP)

TWAMP provides effective and a flexible method for measuring round-trip time and the complete IP performance of the underlying transport between any two devices in a network which supports the standard. TWAMP has been utilized for metrics measurements such as latency, packet loss, packet duplication and packet reorder to measure the quality of IP networks. TWAMP provides interoperability among devices which obviate the need for closed proprietary protocols for measuring the performance.

2.2.1 TWAMP Architecture

TWAMP is very similar to OWAMP, but with a change in the architecture, where session-receiver is replaced by a session-reflector. Unlike the session-receiver in the OWAMP, session-reflector in TWAMP does not collect any measurement data, but instead, on reception of any TWAMP measurement packet from the session-sender a response is sent back to the session-sender. The received packet contains the fields
added by the session-reflector in addition to the session-sender fields. Further, in this report ‘Session-Sender’ and ‘Session-Reflector’ will be referred as ‘Sender’ and ‘Reflector’ respectively for brevity.

Some important fields in the TWAMP packet which helps in the performance measurements are Sequence Number, Sender sending timestamp (T1), Reflector receiving time-stamp (T2), Receiver sending timestamp(T3). Also Sender receiving time-stamp (T4) (not in the TWAMP fields) can be recorded locally on Sender for measuring round-trip delays. For one-way delay measurements clocks needs to be synchronized at Sender and Reflector in both directions (T2-T1 and T4-T3). However, for Round-Trip Time (RTT) measurements it is not needed, since only T4 and T1 are used (RTT = T4-T1) and both the time stamps are generated at the sender side. Also, APC measurements don’t need time synchronization, since each time stamp (T1, T2, etc) is compared between packets and are therefore taken from the same clock i.e., ΔT1 = T1\text{current pkt} – T1\text{previous pkt}, etc.

In general, TWAMP defines two set of protocols: control protocol for setting up the TWAMP sessions and other for measuring the performance of the monitored links through transmission and reception of TWAMP measurement packets. A session protocol helps in initiating a TWAMP session between a sender and a reflector, where the information about addresses, ports and security mode is agreed upon during the start of the TWAMP session and another test protocol for transmission and reception of measurement packets.

2.2.2 TWAMP Light

![TWAMP Light Architecture](image)

Figure 1: TWAMP Light Architecture

TWAMP-Control protocol provides flexibility to set up monitoring sessions and exchange of probe packets. In certain scenarios, it possible to eliminate some of the entities which are in TWAMP architecture. ‘TWAMP Light’ an alternate architecture approach which does not require the TWAMP-Control protocol. TWAMP Light is
an implementation with simplified architecture to provide a network with light test points. TWAMP Light architecture can be realized in a two host scenario where a ‘Controller’, which comprises of Control-Client and Session-Sender and the other host called ‘Reflector’, which comprises Session-Reflector.

In this scenario only TWAMP-Test packets are exchanged \(^2\). The reflector is consider to be active all the time. On reception of TWAMP-Test packets from the sender, the reflector sends response test packet to the each received TWAMP-Test packet, like a ping-type reflector where response is immediately sent out for each packet.

### 2.2.3 Extension of TWAMP Protocol

An extension to the TWAMP protocol defined in RFC 6802 [21], Ericsson Two-Way Active Measurement Protocol (TWAMP) with Value-Added Octets. The notion of embedding meaningful fields in the padding octets has provided a viable method for carrying additional information within the TWAMP-Test protocol. Additional fields: *Last Sequence number In Train*, *Desired Reverse Packet Interval* are introduced in the Session-Sender TWAMP packet. The Last Sequence Number in train field instructs the reflector to wait till the last sequence number as mentioned in the packet arrives before sending the response packets. Here the response packets are sent out as a train, in contrast to the ping-type reflector. This method helps in detecting bottle neck in both directions. The Desired Reverse Packet Interval field can be set at sender packet, so that response packets are sent back at the desired rate to detect bottle necks on reverse direction too. Also this method provides some advantage in scenarios where the intermediate routers in the network has lower packet processing capability which affects the performance of the incoming traffic from sender by the reverse path response traffic from the reflector. In this thesis, research and experiments will be conducted using Ericsson’s extended TWAMP protocol.

Further, this thesis will investigates the challenges faced by BART technique using TWAMP over high-speed links. The research challenges faced by TWAMP when used over high-speed link for performance measurements are discussed in the following section.

\(^2\) For reader’s quick reference, TWAMP packet format described in RFC 5357 and RFC 6802 are presented in appendix section
3 Research Challenges

As mentioned earlier, the BART protocol relies on the fact of generating probe packet at a precise rate and with accurate timing can be challenging over high-speed networks [22]. TWAMP-Test packets are transported using UDP. Mostly, in the measuring device, session-sender and session-reflector applications running on user-space are built to communicate with the host network stack using the native UDP sockets using the `send()` or `sendto()` system calls. It is challenging for the TWAMP application built use native socket communication running over Linux distributions to generate packets at high-speed with accurate time-stampings over high-speed links with the capacity greater than 10Gbps. During the investigation, it was evident that TWAMP application running over native Linux host stack was able to generated and sent packets at the rate approximately equals to 2.8 Gbps. Luigi Rizzo et al., [23] found similar results using a case study to evaluate the time spent packet in processing at various layers of the network stack. Time taken at each level was measured by instrumenting `sendto()`, system call to force an early return from the system call at different depths.

The table 1 shows the time spent at various layers of the network stack when a test program loops around sendto() call bound on a UDP socket on a system running FreeBSD. The time taken by the packet from the application till it reaches the device drivers is approximately 4.7 micro seconds. Similar results are obtained for TWAMP test application running over Linux when tested with TWAMP (uses UDP as transport protocol) packet trains of size 1514 bytes (MTU size) and 64 bytes (minimum Ethernet frame size). On an average, the packets are send at rate 3Gbps.

This is due to the fact that, application using the native Linux network stack to send and receive the packets, has to pass the socket between the application on user space to the Transport layer on the kernel space using send/write system calls. An overview of packet control flow (and the associated data buffering) of the Linux networking kernel can be found in [25]. In nutshell, the data from the application is fragmented into segments and has to be copied from user space to sk_buff data space (kernel space, probably DMA-able space). In Linux, a standardized buffer format for handling the packets at kernel level is called sk_buff. The application data which is fragmented, are queued in the transport layer. Function `udp_transmit_skb()` creates udp header. It clones the sk_buff and passes the control to network layer where the function `ip_queue_xmit()` queues the datagram and then creates the IPv4 header and passes it to Link Layer. The main function of the kernel in link layer is to schedule the packets to be sent out. For scheduling Linux uses the queueing discipline (struct Qdisc) [24]. `dev_queue_xmit` function puts the sk_buff on the device queue and transmission commence by sending it to the NIC after the timer `netif_schedule()` expires. In addition, the context-switching between other active process, and interrupts for each packet sent out, adds overhead to the process from sending packet at higher rates.
<table>
<thead>
<tr>
<th>Layer</th>
<th>File</th>
<th>Function/description</th>
<th>time (ns)</th>
<th>delta (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>user program</td>
<td>sendto() system call</td>
<td>8</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>uipc_syscalls.c</td>
<td>sys_sendto</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td></td>
<td>uipc_syscalls.c</td>
<td>sendit</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>uipc_syscalls.c</td>
<td>kern_sendit</td>
<td>118</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>uipc_socket.c</td>
<td>sosend</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>uipc_socket.c</td>
<td>sosend_dgram</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>uipc_socket.c</td>
<td>sockbuf locking, mbuf allocation, copyin</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td></td>
<td>udp_usrreq.c</td>
<td>ip_output</td>
<td>330</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>udp_usrreq.c</td>
<td>ip_output route lookup, ip header setup</td>
<td>273</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>udp_usrreq.c</td>
<td>ip_output route lookup, ip header setup</td>
<td>273</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>udp_usrreq.c</td>
<td>ip_output route lookup, ip header setup</td>
<td>273</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>udp_usrreq.c</td>
<td>ip_output route lookup, ip header setup</td>
<td>273</td>
<td>57</td>
</tr>
<tr>
<td>IP</td>
<td>ip_output.c</td>
<td>ip_output route lookup, ip header setup</td>
<td>330</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>ip_output.c</td>
<td>ip_output route lookup, ip header setup</td>
<td>330</td>
<td>198</td>
</tr>
<tr>
<td>Ethernet</td>
<td>if_ethersubr.c</td>
<td>ether_output MAC header lookup and copy, loopback</td>
<td>528</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>if_ethersubr.c</td>
<td>ether_output MAC header lookup and copy, loopback</td>
<td>528</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>if_ethersubr.c</td>
<td>ether_output MAC header lookup and copy, loopback</td>
<td>528</td>
<td>162</td>
</tr>
<tr>
<td>Device Drivers</td>
<td>ixgbe.c</td>
<td>ixgbe_mq_start</td>
<td>698</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>ixgbe.c</td>
<td>ixgbe_mq_start_locked</td>
<td>720</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ixgbe.c</td>
<td>ixgbe_xmit</td>
<td>698</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>ixgbe.c</td>
<td>mbuf mangling, device programming</td>
<td>730</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ixgbe.c</td>
<td>mbuf mangling, device programming</td>
<td>730</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ixgbe.c</td>
<td>mbuf mangling, device programming</td>
<td>730</td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td>-</td>
<td>on wire</td>
<td>950</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>on wire</td>
<td>950</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The path and execution times for sendto() on a FreeBSD HEAD 64-bit. *Source:* [23]

It is evident from table 1, that several functions at various levels in the network stack consumes a significant share of the total execution time for the `sendto()` system call on a system running FreeBSD operating system. Any network I/O operation (raw socket or UDP or BPF writer) has to pass many expensive layers. And also it is difficult to avoid system call. Construction/copying of initial mbuf (a FreeBSD term similar to sk_buff in Linux), route and header setup at IP layer and MAC header setup are expensive i.e time consuming operations. And finally, it takes long time to translate into mbufs into NIC format. Also it is evident that more time is taken to process the packet at link-layer and device drivers level.

Similarly, on the receiving end the received packets are buffered before processed by kernel. Then packets are processed, examined and queued at each layer in network stack leads to similar overhead before reaching the application.
The results obtained can be improved by changing CPU mode to “Performance Mode” [26] which sets the CPU statically to the highest frequency, approximately to the scaling_max_freq. Also some optimizing the code, mainly by pre-allocating the buffers for TWAMP payloads to minimize the time utilized on memory related operations (allocation and copying) can improve the performance but not significantly in reaching the wire-speed (10Gbps).

It is necessary for the TWAMP application to generate measurement packets at the required rate. TWAMP measurement packets which are transported using UDP faces the above discussed challenges when the probe packets has to be sent at higher rates (at wire-speed). Also for accurate estimation of available path capacity, it is necessary that timestamps must be accurate with high precision. When dealing high-speed links, where the inter-packet gap is in-order of few nano-seconds, the timestamps on the TWAMP measurement packets becomes obsolete while reaching the NIC after passing through various layer on the network stack with multiple queues on the control flow path. So while measuring the performance of the monitored links, the TWAMP measurement packet spends more time in kernel-space than the flight time between the sender and reflector (i.e. on the links). This results in erroneous calculation of available path capacity, round-trip time and other end-to-end path performance measurements.

There is an ultimate necessary to device a method by which TWAMP measurement packets can be generated and sent at required rate (even at wire-speed) with timestamps of high precision to measure the performance of the high-speed links with high accuracy.
4 Existing technologies that can be considered for faster packet processing

It is clear that from previous section that TWAMP faces a strong challenge in generating measurement packets at higher rates with high accuracy time stamps over high-speed link. Optimizing the TWAMP application using the native UDP socket has not produced the expected results. Architectural revamp is required to solve this challenge. The current architecture using native UDP socket has to be replaced by a new architecture or method on both sender and receiver in-order to send and reflect packets at higher rates. It is to be noted that, the module which calculates the network performance (like available bandwidth, round-trip time, latency) has less or no correlation with the modules that generate and reflect packets on session-sender and reflector respectively. So it sufficient to revamp the sender and reflector module with new architecture and a suitable algorithm for packet transmission using the new architecture. The module which calculates the performance metrics can be reused.

Also it is necessary that the new architecture has to support or abide the following requirements for successful functioning of existing TWAMP system while porting it to the new architecture.

- The new architecture or the method should be able to generate, send and reflect packets even at wire-speed.
- The new architecture should be compatible with Linux.
- Less or no modification to the new architecture’s source code \(^3\) or to any drivers.
- But it should provide an opportunity to run the existing TWAMP-Client application (which instructs and tunes the session-sender to send measurement packets at different rates) over the new architecture.
- The architecture should be stable i.e., minimum deviation from the desired values and less fluctuations from the previous trail values.

In the following section some of the architectures/methods that are widely used (or that are existing and can be explored) to solve similar challenges will be introduced. Also some information regarding the steps required to modified the existing TWAMP application to suit the new architecture(s) will be discussed.

\(^3\) Since the the existing Ericsson Research TWAMP lab application is written using C language and working close to kernel, it will better if the source of the new architecture is written in C language for better compatibility.
4.1 LibPcap

PCAP (Packet Capture) [27] [28] consist of an API for sending and capturing network traffic. The pcap API is written in C language and offers an interface for capturing link-layer frames over a wide range of system architectures. LibPcap works in user-space and less OS dependent. But its library has to emulate the functionality of Berkeley Packet Filter [29] for filtering and buffering the packet at kernel level. The buffer at user-level is used to store packets coming from the kernel and, it prevents the user-level application from accessing kernel-managed memory. The filtering is performed when the packet is at network driver’s level. This is to avoid copying of non-conformant packets to the user space.

LibPcap provides a set of APIs that can be linked to the user’s application. On programmers perspective it hides the interaction between the application and kernel network stack. Using the provided API user can send and capture packets at Ethernet level. The libpcap file format is the main capture file format used in TcpDump [28], Wireshark [30], Snort [31], and many other networking tools are created based on libpcap libraries.

A simple example to send packets using LibPcap can be found here [32]. The packets has to be manually-crafted, i.e. the headers fields on various layers (UDP, IP and Ethernet) has to filled by the application before sending the packet. In order to send the packet network adapter for the particular interface has to opened using pcap_open. The hand-crafted packet can be sent out using function pcap_sendpacket() which takes the arguments, a buffer containing the data to send, the length of the buffer and the outgoing interface adapter. Since the buffer contains only the payload, it is responsibility of the user-application to craft the protocol headers correctly (including the checksum calculations). Similarly, to sniff the packets from the interface, [33] can be used for reference. pcap_lookupdev() function can be used to lookup for the interface to sniff the packets. API pcap_open_live() opens the device for sniffing and pcap_compile() and pcap_setfilter() can be used for filtering the traffic to be captured. Function pcap_next() or pcap_loop() can be used to capture the packets. The sniffed packets are stored in a buffer which points to the starting location of Ethernet frame from which various headers (IP, UDP, TWAMP) can be accessed by moving the pointers to corresponding length.

Since the pcap applications runs on user-space, the existing TWAMP application (only the session-sender and session-reflector logic) can be ported and modified to support sending and receiving of TWAMP measurement packets using libpcap library. Also modules which can create and fill the various headers (including calculation of checksums) is required in addition.
4.2 Kernel Module

Loadable Kernel Modules (LKM) [34] are pieces of code that can be loaded and unloaded into the kernel upon demand. They help to extend the functionality of the kernel without the need to reboot the system. Many well-known applications use kernel modules to bypass the native Linux networking stack. Pktgen [35], a packet generator is one such example, which allow to send pre-configured or manually-crafted packets over the network at faster rates. The tools comes almost with all Linux kernels. Pktgen interacts directly interacts with the device interface. In Linux, sk_buff [36] is a common data structure for network related queues and buffers in the kernel space. It contains all the control information required for the packet. The sk_buff elements are arranged in doubly-linked list fashion. Kernel stores the packet in the format specified in sk_buff for the packet to be sent or received packets. Also it consist of a head and a tail pointer to sk_buff elements along with pointers pointing to transport, network, MAC layers. The data space for sk_buff is allocated from kernel memory.

In TWAMP sender, manually-crafted TWAMP measurement packets can be formatted using sk_buff format and can be directly injected into the transmission queue associated for the particular interface. Function netdev_get_tq_queue() returns the transmission queue associated with the interface. Also function ndo_start_xmit() provides the opportunity to directly inject packets into the transmission queue. By porting the algorithm of session-sender and session-reflector to a kernel module can be improvise the rate at which packets are sent out since the entire native network stack and queues associated are by-passed.

4.2.1 NetLink Sockets

If ‘Loadable kernel module’ is considered as the final architecture to send TWAMP measurements, then a suitable method to communicate between kernel module and the TWAMP-client application running on the user-space has to be enabled for initiating the TWAMP session and to send back the results obtained from the reflected probe packets for performance measurements. There are many ways the Linux kernel interacts with user space programs such as file systems (Procfs, Sysfs), Ioclt methods, Systems calls, Mmap. More information about Kernel Space - User Space Interfaces can be found here [37]. Netlink sockets [38] [39], a datagram-oriented socket messaging system from user-space to kernel and vice-versa. Netlink sockets uses BSD socket infrastructure supporting the native socket(), bind(), sendmsg(), and recvmsg() and the common socket polling mechanism. It allows upto 32 buses in kernel-space and supports both Unicast (1:1) and Multicast (1:N) communication channels. Netlink socket programming basics for can be found here [40]. Since the ease and flexible nature of Netlink sockets, the TWAMP parameters such as number of trains, packets per train, sending rate can be passed via Netlink sockets from the TWAMP-client.
application on user-space to kernel module which will can acts as Session-Sender.

### 4.2.2 Netfilter

To receive the TWAMP measurement packets on reflector and sender, prominent Netfilter [41] hooks can be used. Netfilter is a framework for packet mangling. It intercept traffic from/to the driver and pass it to processing modules without additional data copies. Netfilter consist of series of hooks (NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN, NF_IP_FORWARD, NF_IP_FORWARD) in well-defined points in the protocol stack. It intercept traffic from/to the driver and pass it to processing modules without additional data copies. Multiple hooks can be registered on kernel to listen different protocols. On passing through Netfilter framework, the packets are examined and delivered to the corresponding module (if any registered) for further handling of packets (if required can be altered). After processing NF_DROP)discards the packet, NF_ACCEPT)allows the packet through to pass to next hook point, NF_STOLEN allows to ignore the packet and NF_QUEUE requests the netfilter to queue the packet for user-space. The rest of the packets for which the hooks are not registered by default passes through the native network stack. ‘IPTables’ [42], a program to configure Linux kernel firewall is implemented using Netfilter modules. TWAMP measurement packets can be hooked at the initial (NF_IP_PRE_ROUTING) stage on reflector and sender.

### 4.3 Netmap

Netmap [23] is a novel framework from Luigi Rizzo et.al from Universita di Pisa, aims to reduce the cost of moving traffic between hardware and the host network stack. It gives access for the applications running on the user-space to network packets on sending and receiving side. According to article [23], test application built over Netmap tested on 10Gbps (with single core processor running at 900MHz), sends and receive 14.88 Mpps (i.e. the full wire-speed on 10Gbps links). This is due to the three major improvement in packet processing method of Netmap

1. Dynamic memory allocations per-packet is removed by pre-allocating resources.
2. Overheads for the system calls are amortized over large batches.
3. Memory copies are avoided by sharing buffers and meta-data between user-space and kernel. But access to device registers and kernel memory areas are still restricted.
The NIC is partially disconnected from the host stack, when the application requests to put the interface in netmap mode. The disconnection in the data path is unaware to operating system and so it continues to use the interface during regular operations. The application can associate an interface in netmap mode by opening a special device /dev/netmap and issuing an ioctl() call. The circular queues of buffers (netmap rings) implemented in shared memory helps the program to exchange packets with NIC and with host stacks. The application uses poll() [43] mechanism to send and receive packets. Two non-blocking IOCTL calls (NIOCTXSYNC, NIOCRXSYNC) can be used to send and receive packets. But the netmap requires the application to fill the available buffers in the TX rings before issuing the ioctl(., NIOCTXSYNC) call to inform the OS to send the packets out. Similarly on receiving side, on issuing ioctl(.,NIOCRXSYNC) returns the number of packets immediately available in the RX ring with their lengths and payloads. Netmap also provides a provision for zero-copy packet forwarding mechanism between the interface. Also Netmap offers a wrapper API to support Libpcap compatibility.

The source code of the Netmap can be downloaded from [44]. In order for TWAMP application to use Netmap to have direct access over higher speed links, the original IXGBE module (i.e. ixgbe.ko) has to replaced with the IXGBE module obtained as the result of compiling the Netmap source code. Since the communication between the host stack and NIC is discontinued when Netmap is in operation, it is necessary to forward the the other traffic (apart from TWAMP measurement packets) from/to the host stack and NIC. Netamp enables this facility by providing additional pair of rings which can be to a Netmap file descriptor with a NIOPREG call. This abstraction can be provided by NIOCTXSYNC on one the rings buffers into mbufs and passes them to the host stack, where it appears like the packets being arrived from the physical interface. In contrary, packets coming from the host stack are queued on one of the Netmap rings and are passed to the Netmap client on the following NIOCRXSYNC calls.

Figure 2: Netmap High Level Architecture. Source: [23]
4.4 DPDK

Data Plane Development Kit (DPDK) [45], developed by Intel, is a set of libraries and NIC drivers for faster processing of packets in data plane applications. It provides a framework (written on C) for the building high speed packet networking applications over Intel x86 processors. DPDK is an opens source BSD licensed project which supports wide range of processors from Intel Atom processors [46] to Intel Xeon processors [47]. In near future, the support for other processor architecture (like IBM POWER8) are to be released. Also DPDK can support wide range of NIC’s such as CISCO enic, Emulex oce, Intel series, Mellanox mix4. The list of supported NIC’s are [48].

DPDK provides API to libraries, hardware accelerators and other operating system element through creation of Environment Abstraction Layer (EAL). Upon creation of EAL which is environment specific, user applications can be linked to the libraries. The following are the main components in the DPDK optimized NIC drivers, which helps in faster processing of packets at data plane.

• A queue manger for lockless queues.
• A buffer manger to pre-allocate fixed size buffers.
• A memory manger allocates pool of objects in memory. In-order to spread objects equally on all DRAM channels, a ring buffer is used to store free objects.
• Asynchronous Poll Mode drives (PMD) to eliminate the performance overhead of interrupt processing.
• Set of libraries (framework) which helps in packet processing.

Once the source code is compiled for the specific environment, the DPDK kernel modules (igb_uio.ko and rte_kni.ko) has to be loaded, which enables the functionalities to by-pass the native kernel stack. It is possible for the application developer associate the user application to specific processor cores, number of memory channel to be used and the interface from which the packets has to be processed. So atleast one core of the CPU has to dedicated for DPDK to process the packets. Also, native memory related system calls such alloc(), memcpy() are optimized for better performance results. Unlike Netmap (which takes control of interface only when a user application using Netmap API is in running state), DPDK takes full control of the interface once the interface is binded with it. The native host stack loses control over the interface i.e. the interface will not be shown on Linux commands such as ifconfig.

\footnote{DPDK is not a network stack and does not provide functionalities such as Layer-3 forwarding, IPSec, NAT etc.}
DPDK provides device level filtering of packets using 5tuple filters (Source IP, Destination IP, Protocol). The matched packets can be queued separately from the default queue. This will be help to separate TWAMP packets for processing.

Since DPDK does have its own native stack, a separate applications has to be built to handle ICMP, ARP or other packets. But DPDK provides control plane solution called Kernel NIC Interface (KNI) which allows the user application to redirect the traffic to the kernel networking stack. This is achieved using IOCTL call to create KNI virtual device in Linux kernel. More information can be found in the article [50] (section 8 in specific). The other traffic (apart from TWAMP measurement packets) are queued separately, can be rerouted to the kernel using the above mentioned mechanism. In recent, DPDK has gained much popularity mainly in the field of Software Defined Networks (SDN), Network Function Virtualization (NFV) for faster packet processing [51] [52].

MoonGen [53], a scriptable high-speed packet generator software under MIT license is built on DPDK. According MoonGen documentation it can saturate 10 GbE links with minimum sized packets using single CPU core. Multi-core scaling results in higher rates: upto 178.5 Mpps at 120 Gbps is proven to be achieved.
5 Methodology and Implementation

This thesis is carried out using ‘Quantitative’ method [54], where the base modules for generating, sending and reflecting TWAMP measurements packets on sender and receiver using the above discussed architectures (UDP socket, LibPcap, Kernel Module, Netmap and DPDK) over the network discussed in next section. The results obtained from the different architectures are compared against each other and the obtained results are analysed to select the suitable architecture based on certain parameters that are discussed in the section 6.

Section 5.1 enumerates the requirements of prototypes. Following that, in section 5.3 the logic/algorithms that are used to built the session-sender and session-reflector module based on different architecture are explained. Finally in section 5.4, the test scenario: network topology, specifications of test devices over which prototypes will be built and tested are explained.

5.1 Requirements

In order to comply with the requirements, following initial criteria must be satisfied by the architectures when the prototypes are built and tested using different architectures.

- Able to generate TWAMP packets at wire-speed with high accuracy and proper timestamps.
- Provision to control and send TWAMP packets at desired rates with high accuracy and proper timestamps.
- Stable and less fluctuating with the values over trails.
- Able to run and generate TWAMP measurement packets on Linux.
- Less/no modification to the source code.
- Provide opportunity to build application on of the architecture.
- Compatible with one or more hardware platforms and various Linux distributions.
- Work without modifying any device drivers manually.
5.2 Packet sending algorithm

Figure 4 shows the algorithm used for the sending packets using different architecture. The first diagram shows the method which commonly used for sending measurement packets at the required speed. The packets are sent out at the required rate using a constant inter-packet delay between the packets. In high-speed links it is difficult to generate and send packets with small inter-packet delays in order of few nano-seconds. In order to overcome the challenge, the proposed algorithm as show in the second diagram has been used in this thesis. Instead of creating a constant inter-packet delay, the initial packets are sent out as a burst, while the last packet is delayed and sent at a required rate. Using this method, it is easy to create a single large delay or wait-time when compared to multiple small inter-packet delay. Also another work around to send packets over high-speed links would be to insert garbage packets of size equals to inter-packet delay between the TWAMP measurement packets. On reception these garbage packets can be discarded from the calculations.
5.3 Implementation

In this section, the high-level design of the prototypes for session-sender and session-reflector using the native UDP socket, LibPcap socket, Kernel Module, Netmap and DPDK architecture are explained in details. The high-level design of modules are expressed using flowchart diagram for the readers to understand easily. The algorithm for session-client to calculate APC, RTT and other network performance measurements are out of the scope of this thesis. For brevity and modularity purpose, the session-client module is made function independently on the user-space irrespective of the under-lying architecture used for session-sender. Common API(s) are provided between session-client and session-sender to communicate the parameters required to initiate the TWAMP measurement sessions and to send back the results for performance measurements.

5.3.1 Standard UDP - Pseudocode for sender and reflector

This section describes the high-level design used for the session-sender and session-reflector module designed using native UDP socket. From the session-client module, the following parameters are passed to the session-sender module for creating the session.

- Reflector IP and port
- Number of trains
- Packets per train
- Rate at which the trains has to be sent

First, the UDP socket is created for communicating with host network stack. On struct sockaddr, the destination IP and port received from the session-client are used for sending out the TWAMP measurement packets. On the other hand, to receive the packets from reflector, the socket is with bind() with sender IP and port on which the sender is listening. The memory for ‘N’ packets are preallocated in the user-space. This is to avoid allocating memory dynamically on per-packet basis during the measurement time, which significantly helps in sending packets at higher rates. For the same reason, the TWAMP fields are filled in prior except the sending timestamp, which will be update before sending the packet for more accuracy. Also the total time taken to send ‘N’ packets at the specified rate is calculated. The first N-1 packets are sent out as a single burst.

Before sending out, the packets are timestamped with the time ‘T1’ in nano-second precision using clock_gettime(CLOCK_REALTIME, time) system call. The timestamp obtained using the above call is store on struct timespec object which provides the provision to store timestamp in nano-second accuracy. sendto() call is used to put the TWAMP payload into the UDP socket.
Figure 5: High level flow diagram of TWAMP Sender module using UDP socket
The last packet is delayed and sent out at the time $T$ to match the required rate. Spin-loop with nano-second precision is used for creating delays. Once the first train of packets is sent out, the memory is allocated for next ‘N’ packets and the above process repeats.

Similarly, on session-sender to receive the reflected packets, `recvfrom()` call over the socket delivers the TWAMP payload. On reception, the packets are timestamped $T4$ and the packets is copied to the pre-allocated buffers. Upon reception of the last packet in the train the results are sent to session-client for performance measurements. Once the measurement session is completed the socket can be closed using `close(sock_fd)`.

On the reflector, buffers are pre-allocated at user-space for storing the received TWAMP packet from sender. The received packets are timestamped ($T2$) and stored in the buffer until the last packet in the train arrives. Once the last packet arrives, the reflector headers are added to the packets and sent out. The packets are modified in the same buffer to speed up the process. Using the same logic specified in session-sender for sending out the TWAMP packets, the reflected packets are timestamp ($T3$) before sent out using `sendto()` call. Once all the packets in the train are sent out, the session-reflector is prepared to receive the next ‘N’ TWAMP packets and cycle repeats.
5.3.2 LibPcap - Pseudocode for sender and reflector

This section describes the high-level design used for the session-sender and session-reflector module designed using LibPcap socket. From the session-client module, the following parameters are passed to the session-sender module for creating the session.

- Reflector IP and port
- Number of trains
- Packets per train
- Rate at which the trains has to be sent

First step in the session-sender is to create a LibPcap socket which can communicate with the network stack. The device/interface is opened to LibPcap using `pcap_open_live(device)`, in-order to get the IP of the device `pcap_lookupnet()` is used and to receive only the reflected TWAMP packets a filter is set using `pcap_setfilter()`. Once the LibPcap socket is opened then the TWAMP measurement session can be initialed.

Similar to the UDP architecture, the memory for ‘N’ packets are preallocated in the user-space. Also, the TWAMP fields are filled in advance except the sending timestamp, which will be update before sending the packet for more accuracy. In addition, the UDP, IP and MAC headers and IP checksum has to be calculated and filled by the TWAMP application. The total time taken to send ‘N’ packets at the specified rate is calculated. The first N-1 packets are sent out as a single burst. Before sending out, the packets are timestamped with the time ‘T1’ in nano-second precision using `clock_gettime(CLOCK_REALTIME)` system call. `pcap_sendpacket()` call is used to put the TWAMP payload into the LibPcap socket as the Link Layer frame. The last packet is delayed and sent out at the time T to match the required rate. Spin-loop with nano-second precision are used for creating delays. Once the first train of packets is sent out, the memory is allocated for next ‘N’ packets and the above process repeats.

To receive the reflected packets on session-sender, `pcap_loop()` is called repeatedly over the socket to receive and deliver the TWAMP payload. On reception, the packets are timestamped T4 and the packets are copied to the pre-allocated buffers. Upon reception of the last packet in the train the results are sent to session-client for APC, RTT and other performance measurements. Once the measurement session is completed the socket can be closed using `pcap_close(sockfd)`.
Figure 7: High level flow diagram of TWAMP Sender module using LibPcap socket
On the reflector, buffers are pre-allocated at user-space for storing the received TWAMP packet from sender using `pcap_loop()` continuously. The received packets are timestamped (T2) are store in the buffer until the last packet in the train arrives. Once the last packet arrives, the reflector headers are added to the packets and sent out. The packets are modified in the same buffer to speed up the process. Using the same logic specified in session-sender for sending out the TWAMP packets, the reflected packets are timestamp (T3) before sent out using `pcap_sendpacket()` call. Once all the packets in the train are sent out, the session-reflector is prepared to receive the next 'N' TWAMP packets and cycle repeats.

Figure 8: High level flow diagram of TWAMP Reflector module using LibPcap socket
5.3.3 Kernel module - Pseudocode for sender and reflector

This section describes the high-level design used for the session-sender and session-reflector module designed using Loadable Kernel Modules. The design is quite different from other, because entire host network stack is by-passed. From the session-client module, the following parameters are passed to the session-sender module running as the kernel module.

- Reflector IP and port
- Number of trains
- Packets per train
- Rate at which the trains has to be sent

In order to establish this communication, Netlink socket are used. These sockets are bi-directional can help to send the session parameter down to kernel module and upon completion of TWAMP measurements, results can be sent back to session-client via Netlink sockets. From the session-client the Netlink sockets are created similar to native UDP sockets. Netlink socket is created using `socket(PF_NETLINK, NETLINK_PORT)`. The Netlink port is specified for communicating with the Netfilter receiver on kernel module. The the socket is `bind()` for receiving packet from kernel. The address family is set to AF_NETLINK. Also the session-client (user-space application) process id is filled in the Netlink header for helping the Kernel module to send back the results. Multiple user-space application can communicate with a single kernel module bi-directionally. In-order to facilitate this process-id is utilized. The payload of Netlink packet is filled with TWAMP session parameters to initiate the session. Finally `sendmsg(fd, &netlinkMsg)` is sent out to the kernel module to start the TWAMP measurement session.

On reception of the session parameters in kernel module, through pre-defined hooks listening on specific port, the TWAMP session parameters in payload of application payload extracted and memory for ‘N’ packets are preallocated in the kernel-space using `sk_buff`. Also the TWAMP fields are filled in advance except the sending timestamp, which will be update before sending the packet for more accuracy. In addition, the UDP, IP and MAC headers created and filled manually since the native host stack is by-passed completely. Also IP checksum has to be calculated manually.

The total time taken to send ‘N’ packets at the specified rate is calculated. After fetching the access to the transmission queue, first N-2 packets are sent using `xmit_more = 1` option and the N-1th packet is sent with `xmit_more` is set to zero. Setting `xmit_more = 1`, helps in mitigating the interrupts on per-packet basis. `xmit_more = 1` indicates the transmission queue that more packets will be arriving to the queue. The last packet is delayed and sent out when timer reaches the total time (which is calculated earlier). For smaller delays (<100µs) simple spin loop is used, else, high resolution timers to do nano-sleep are used using `hrtimer_activate()`.
The \((N-1)\)th packet is set with option \texttt{xmit\_more = 0} indicating the end of the packet burst. Before sending out, the packets are timestamped with the time ‘T1’ in nano-second precision using \texttt{ktime\_get()} call. Also it is to be noted that before injecting the TWAMP packet into transmission queue, packet has to be formatted by adding the required meta-data to adapt to the Linux system architecture.

Similarly, on session-sender, the incoming TWAMP packets are received by Netfilter hooks at PRE_ROUTING stage before the packet is processed by the native host stack. Each incoming packet has to be inspected by the application to confirm the packet is a TWAMP measurement packet or not. The TWAMP packets are timestamped T4 and the sk\_buff corresponding to the packet is stored without releasing the memory. Once after receiving ‘N’ packets, the timestamp T1, T2, T3 and T4 from the TWAMP packets are copied and sent back to the session-client for APC, RTT and other performance measurements through Netlink sockets. The sk\_buff are freed and memory for ‘N’ next train of TWAMP packets are allocated in the sk\_buff and the process continues. Finally, on reception of results for all the trains sent, the Netlink socket is closed.

In Session-Reflector the main difference in the architecture of is that the entire module is ported to kernel level since there is no need for taking the packet to user-space. The incoming packets are modified and sent out using the logic build inside the loadable kernel modules. Processing TWAMP packet at kernel level increase the performance of the reflector. The incoming TWAMP packets are received by Netfilter hooks at PRE_ROUTING stage before the packet is processed by the native host stack. Each incoming packet has to be inspected by the application to confirm the packet is a TWAMP measurement packet or not. The TWAMP packets are timestamped T2 and the sk\_buff corresponding to the packet is stored. After receiving ‘N’ packets, same sk\_buff’s reused for receiving the packet can be modified before sending out the response. The reflector header are added to the packets. In addition, UDP, MAC IP headers and IP checksum are calculated.

After fetching the access to the transmission queue, first \(N-2\) packets are sent using \texttt{xmit\_more = 1} option and the \(N-1\)th packet is sent out and the \texttt{xmit\_more} is set to zero. High-resolution timers are used to delay the last packet to sent out at required rate. Before sending the packets are timestamped are timestamp with T3 using \texttt{ktime\_get()} kernel function. Once the response packets are sent out, the session-reflector is set to receive the next ‘N’ packets from the next train.
Figure 9: High level flow diagram of TWAMP Sender module using Kernel Module and NetLink sockets
Figure 10: High level flow diagram of TWAMP Reflector module using Kernel Module and NetLink sockets
5.3.4 Netmap - Pseudocode for sender and reflector

This section describes the high-level design used for the session-sender and session-reflector module designed using Netmap architecture. Before establishing the TWAMP session, the loadable kernel modules to support Netmap architecture are loaded in prior. From session-client module, the parameters required to create the session: Reflector IP and port, number of trains, packets per train and the rate at which the trains has to be sent are passed to the session-sender module. First, the Netmap has to establish connection with interface and take control of the interface to send and receive TWAMP packets by disconnecting the interface attached to the native stack until the TWAMP measurement application is running. On closing the application running over Netamp, the interface is re-connected with the native host stack.

On Sender, in order to begin the TWAMP measurement sessions, the Netmap has to open the file descriptor to connect the required interface with the Netmap architecture, \( fd = \text{nm_open(netmap:interface)} \). The IOCTL call ioctl(fd, NI_CCREGIF) provides additional unbound buffers in same memory space. The pointer to the interface and the corresponding interface transmission ring are obtained using \( \text{NETMAP_IF()} \) and \( \text{NETMAP_TXRING()} \). Then, the memory for ‘N’ packets are preallocated in the user-space. Also the TWAMP fields are filled in prior to the measurements, except the sending timestamp, which will be update before sending the packet for more accuracy. In addition, the UDP, IP and MAC headers and IP checksum has to be calculated and filled by the TWAMP application. The total time taken to send ‘N’ packets at the specified rate is calculated. Once the TWAMP measurement packets are created, the Netmap starts to poll the interface using \( \text{poll()} \) system call continuously. Check if the transmission ring buffer is not empty, if not empty, the first N-1 packets are sent out as a single burst. Before sending out, the packets are timestamped with the time ‘T1’ in nano-second precision using \( \text{clock_gettime(CLOCK_REALTIME)} \) system call. The packets are copied into the transmission ring buffer which in turn sends out the packets.

The last packet is delayed and sent out at the time \( T \) to match the required rate. Spin-loop with with nano-second precision are used for creating delays. Once the first train of packets is sent out, the memory is allocated for next ‘N’ packets and the above process repeats. Also the reflected packets are received and timestamp (T4). The received packets are stored in the Netmap buffer on which the packets are received. After receiving N\textsuperscript{th} packet, the results (T1, T2, T3 and T4) are sent to the session-client for APC and other network performance measurements.

Similarly, on session-reflector to receive the reflected packets, the Netmap has to open the file descriptor to connect the required interface with the Netmap architecture, \( fd = \text{nm_open(netmap:interface)} \). The interface is set to polling mode and then the interface is polled continuously to send and receive packets. On arrival of TWAMP packets from sender, the packets are received using the function
nm_nextpkt(fd, pktHeader) and timestamped ‘T2’.

Figure 11: High level flow diagram of TWAMP Sender module using Netmap
Figure 12: High level flow diagram of TWAMP Reflector module using Netmap

After receiving ‘N’ packets the reflector header is added. Source and destination IP, MAC, UDP ports are swapped. The pointer to the interface and the corresponding interface transmission ring are obtained using `NETMAP_IF()` and `NETMAP_TXRING()`. First N-1 packet are timestamped (T3) and sent out. While the last packet is sent out after the time after the required delay. Then the reflector is prepared to receive ‘N’ of the next train of packets.
5.3.5 DPDK - Pseudocode for sender and reflector

This section describes the high-level design used for the session-sender and session-reflector module designed using DPDK architecture. Before establishing the TWAMP session, the loadable kernel modules to support DPDK architecture are loaded in prior. Also to the interface used for measurement is bounded to the DPDK. Once the interface is bound to the DPDK drivers, the interface is disconnected from the native kernel stack until the interface is re-attached to the native kernel stack. From the session-client module, the following parameters are passed to the session-sender module for creating the session.

- Reflector IP and port
- Number of trains
- Packets per train
- Rate at which the trains has to be sent
- Number of dedicated CPU cores and memory channel to be associated with TWAMP application

Before starting the TWAMP measurement session, the CPU core(s) and memory channel(s) has to be associated with the DPDK in order to function. The associated cores are completely utilized by DPDK until the TWAMP application stops.

The session-sender application starts by initializing the DPDK parameters i.e., binding the logical cores, associating memory channels. Then pool of static buffers used to send and receive the packets are allocated by `rte_mempool_create(NUM_MBUFS)`. The transmission and reception rings associated with the interface has to be configured using `rte_eth_dev_configure(port, rx_rings, tx_rings)`. `rte_eth_dev_start(port)` command starts polling the interface continuously. 5 tuple filters (Source IP, Destination IP, Source Port, Destination Port, Queue Id) are added to filter the TWAMP measurements packets from the other traffic. The queue id is added so that all the TWAMP packets ends up in the desired queue, while the other traffic is redirected to the default queue on reception. The callback functions are added to process the packets queued in the TWAMP and default queue separately.

In order to send the measurement packets, the memory for ‘N’ packets are requested and allocated from the pool of pre-allocated static buffers. The TWAMP fields are filled in advance except the sending timestamp, which will be update before sending the packet for more accuracy. In addition, the UDP, IP and MAC headers and IP checksum has to be calculated and filled by the TWAMP application. The total time taken to send ‘N’ packets at the specified rate is calculated. The first N-1 packets are sent out as a single burst. Before sending out `rte_eth_tx_burst(port, buf, N-1)`, the packets are timestamped with the time ‘T1’ using `clock_gettime(CLOCK_REALTIME)` system call. The last packet is delayed and sent out `rte_eth_tx_burst(port, buf[N], 1)` at the time $T$ to match the required rate. Spin-loop are used for creating delays.
Figure 13: High level flow diagram of TWAMP Sender module using DPDK
Figure 14: High level flow diagram of TWAMP Reflector module using DPDK
Once the ‘N’ packets are sent out, the memory allocated for the ‘N’ packets are made free and given back to the memory pool. The call back function registered previously receives the reflected packet. The received packets are timestamped (T4). The received packets are stored. After receiving N\textsuperscript{th} packet, the results (T1, T2, T3 and T4) are sent to the session-client for APC and other network performance measurements and the corresponding packet buffers are made free.

Similarly, on session-reflector, first the DPDK parameters are initialized i.e., binding the logical cores and associating memory channels. Then pool of static buffers used to send and receive the packets are allocated by \textit{rte_mempool_create(NUM\_MBUFS)}. The transmission and reception rings associated with the interface are configured and the interface is polled continuously. 5 tuple filters are added to filter the TWAMP measurements packets from the other traffic. Based on the filtering the TWAMP packets are redirected to a separate queue for processing, while the other traffic ends up in default queue. On reception of TWAMP packets, the packets are timestamped (T2) and the same memory buffer is used to edit and the send the response packet. After receiving ‘N’ packets, the reflector headers are added. First N-1 packet are timestamped (T3) and sent out using \textit{rte\_eth\_tx\_burst(port, buf[N], 1)}. While the last packet is sent out after the time after the required delay. Then the the reflector is prepared to receive ‘N’ of the next train of packets.

5.4 Test Setup

The experiments are tested over the network with single-hop connectivity, i.e., where TWAMP sender and reflector are connected to a switch over a high-speed links (of 10 Gbps). Two devices (one as Sender and other as Reflector) with same hardware specifications and operating system is used for testing. Test devices consist of 24 CPUs with Intel(R) Xeon(R) CPU X5660 multi-core processor [55] with base frequency of 2.8 GHz. But for the experiments one only one CPU has been utilized. Operating system used was Ubuntu 14.04 [56] with Linux Kernel version 3.18.4 [57]. The detailed information of the test devices can be found in table 3, in appendix section.
6 Results and Analysis

In this section, the results obtained from the five different architectures (Native UDP Socket, LibPcap, Kernel Module, Netmap and DPDK) for sending and reflecting TWAMP packets using the optimized algorithm mentioned for respective methods in the previous section, are presented. In Section 6.1, the results of the different architecture’s ability to send and receive TWAMP measurement packet at full wire-speed (10Gbps) are presented and compared. Also parameters such as Minimum rate, Maximum, Average, Standard deviation and Mean Absolute Error for both sender and reflector module using different architecture are presented. In section 6.2, a comparison graphs showing the results obtained for sender and reflector module in specific for DPDK when tested against different rates and train size (number of packets in the burst). Also the reasoning behind the results obtained are provided. In section 6.3 the pros and cons of the different architecture are discussed and finally, in section 6.4 information related to different timestamping methods available, and high resolution timers are discussed and the test results are presented.

6.1 Comparing results from different architecture

This section, presents and discusses the results from the experiment; ability to generate, send and receive TWAMP measurement packets at wire-speed.

Using the optimized algorithm(s) that suits the different architecture, the experiment aims to generate and send TWAMP measurement packets at wire-speed. Single train (burst size) of 50 packets. Each packet is of size 1514 bytes of (MTU size of the link) including Ethernet, IP and UDP Headers accounting to size 42 bytes. Rest 1472 bytes are accounted by TWAMP Header and padding. In order to obtain reliable results, the same experiment is repeated for 10 times for each architecture and the following values are recorded.

- Average (Mean) sending or receiving rate for the each architecture.
- Minimum (Min) sending or receiving rate obtained from the trails for the each architecture: Lowest Performance.
- Maximum (Max) sending or receiving rate obtained from the trails for the each architecture: Best Performance.

---

5 To make rate calculations easier, bits in the Ethernet header are also taken into consideration.
Figure 16: Mean, Min, Max rates for different architectures at Sender when 50 TWAMP measurements packets are sent at wire-speed (10 Gbps)

Figure 16 shows the Mean, Minimum and Maximum rate for different architecture at Sender. From the figure, it evident the existing architecture i.e., built over native UDP sockets with optimization can generate and TWAMP measurements packets approximately at 3 Gbps. While LibPcap and Kernel Modules are able send TWAMP measurement packets at 6 Gbps and 8.64 Gbps respectively, but quite far from reaching wire-speed. Netmap and DPDK were able to send almost at wire speed.

As discussed in earlier sections, sending rate for native UDP socket is low due to the time taken to build corresponding layers header on the network stack, multiple queues in kernel stack and at the interface level, and some minor delays contributed by to context switching and interrupt overheads. LibPcap, by manually building the headers (pre-built) and injecting packets at Ethernet level helps in increasing the sending rate almost twice (upto 6 Gbps). Using kernel modules, by bypassing the native kernel stack completely, injecting packets at the lowest point in the kernel into the device queues directly, by pre-allocated sk_buffs and utilizing the new xmit_more() (with reference to section 5.3.3) to reduce the interrupt overheads, helps in reaching sending rates upto 8.64 Gbps. Netmap and DPDK on the other hand with pre-allocated memory buffers for packets, mapping interface directly with the user-application and completely by-passing queues helps the architecture to send packets at full wire-speed.
Also it is necessary to verify that the architecture is stable i.e., not having too much fluctuations from the previous values. In such situations like where two systems: Netmap and DPDK has almost similar results, it necessary to analyse two more factors **accuracy** (how accurate the results obtained are from the expected value), second **stability** (fluctuations from the previous values or the distribution of values from the mean)

1. **Mean Absolute Percentage Error (M)** - is a measure for accuracy expressed in percentage, defined by formula

\[
M = \left( \frac{1}{N} \sum_{i=1}^{N} \left| \frac{E_i - X_i}{E_i} \right| \right) \times 100\%
\]

where \( E_i \) is the expected value of that iteration, \( X_i \) is the obtained value in that iteration. In nutshell, this Mean Absolute Percentage Error (M) will give the percentage of error/deviation of the mean value from the expected value.

2. **Standard deviation (\( \sigma \))** - is a measure that denote how spread out the numbers are. High standard deviations denotes higher fluctuations in the values obtained from the trails and vice-versa.

\[
\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}
\]

where \( \mu \) is the mean of the values and \( x_i \) is the value of the current iteration. Both Mean Absolute Percentage Error and Standard Deviation are non-negative values.

Figure 17 shows the Mean Absolute Error in Percentage. UDP has the highest percentage of error 69.8% while DPDK has the least error about 1.90% from the expected value (10 Gbps). And figure 18 shows the Standard Deviation of various architectures. The LibPcap has the most standard deviation of 720.41 Mbps from the mean while the Netmap has 24.13 Mbps and DPDK has the lest standard deviation of 1.79 Mbps which indicates the values are closely distributed near the mean and this indicates DPDK architecture is stable on Sender. On Sender, architecture using DPDK is able to send packets at wire-speed, has the lowest Mean Absolute Error which indicates values are near the expected values and also has the least Standard Deviation indicating its architecture has less fluctuation when compared with its adjacent values.
Figure 17: Mean Absolute Percentage Error for different architectures at Sender

Figure 18: Standard Deviation for different architectures at Sender
From the above results the following conclusion can be drawn. Even though Netmap and DPDK are able to send packets at wire-speed and has lower mean absolute percentage error, the fluctuations (Standard Deviation) comparatively higher than DPDK. So in-order to test the reflectors receiving performance, the architecture built using DPDK will be used at the sender to send the TWAMP measurement packets, while the performance of receiving module for different architecture are tested at the reflector.

Figure 19 shows that the average receiving rate of native UDP socket is 8.64 Gbps, and the kernel module is 8.68 Gbps which is same as its sending rate. DPDK has the best receiving rate, the packets are received and processed at wire-speed. LibPcap has the lowest receiving rate of 5.86 Gbps. The possible reason might be the fact the entire packet (Ethernet header till the payload) is copied to the user space and the headers are processed manually. The notable difference in the graph is the values of Netmap recording 121.75 Gbps over 10 Gbps link. This dubious value is the result of packets buffered before time-stamping. Buffering the packets disturbs the inter-packet timing. Closer the packets (less inter-packet timing) higher the rates.

Figure 19: Mean, Min, Max rates for different architectures at Receiver
Figure 20: Scenario illustrating the problem leading to erroneous calculation in receiving rates due to packet buffering

Figure 20 illustrates the reason for higher rates seen at the receiving side when using Netmap as the architecture. Netmap uses polling mechanism and the overhead for the system calls are amortized over large batches. The packets are buffered and send to TWAMP application running in user-space where the packets are timestamped. In figure 20, the first diagram shows the expected behaviour where the packet are timestamped as soon as they are received. In the same figure, the below diagram depicts the effect of buffering packets which happens with Netmap. Due to buffering the inter-packet gap is lost and most of the packets contains same timestamps. Thus the calculated time difference between the first and the last packet will be much smaller than the original time difference which leads to higher receiving rates during the calculations.

Figure 21 presents the mean absolute error percentage for different architectures at receiver. Netmap records the highest error percentage of 1117.83% which was the direct consequence of higher receiving rate due to the contraction of inter-packet timing resulted due to buffering phenomenon. While DPDK having the lowest error of 2.21%. Also figure 22 which shows the standard deviation at receiver indicates UDP and Netmap has the higher fluctuations in the receiving rates while kernel module and DPDK has the lowest fluctuations.

It is clear from the above discussion, even though Netmap and DPDK are able to send and receive packets at wire-speed, and with mean absolute percentage error and standard deviation are high for standard deviation. But on other hand DPDK has the best sending and receiving rates, lowest mean absolute percentage error and standard deviation, makes DPDK to have edge over other architectures. In order to
finalize DPDK as the architecture to send and receive TWAMP packets more detail study has to be made on DPDK by conducting experiments where the architecture’s performance is tested against different rates and burst size (packets per train).

Figure 21: Mean Absolute Percentage Error for different architectures at Receiver

Figure 22: Standard Deviation for different architectures at Receiver
6.2 Evaluation of DPDK performance against different rates and burst size

In this section, the performance of DPDK is tested against the different rates and packets per trains. The results are recorded on both sender and reflector. Figure 23 shows the graph specific to DPDK architecture. Graph is plotted for results from the combination of rates (1, 2, 5, 8, 10 Gbps) against the burst size of (10, 25, 60, 100 packets per train). This provides the opportunity to study how DPDK’s performance at various with rates and burst size.

In the figure 23, the results for sending rate (at Sender) are plotted with continuous lines, while the receiving rates (at reflector) for the corresponding sent rate are plotted with dashed lines. From the graph two distinct observation can be made.

- **Rates** - DPDK has better performance in terms of accuracy at lower rates (1 or 2 Gbps) when compared to higher rates (8 or 10 Gbps) on both sender and receiver. This is due to fact that, at lower rate the inter-packet timing is more. So at the sender and receiver, the device interface has enough time to process the packet. While, at the higher rates the inter-packet gap is less, which demands the system to process packets at faster rate, resulting in less accuracy.

- **Burst Size** - Longer the train size higher is the DPDK’s performance and vice-versa. The effect is prominent at receiver side, where shorter bursts has much deviation from the desired values. On detailed study revealed, even though DPDK uses continuous polling mechanism theoretically. In real the polling is non-continuous i.e., there is a finite amount of time (in few nano-seconds) between the two poll-in calls by TWAMP application on user-space which results in probability that few initial packets might be received in the interface buffer between the two poll() calls. This distorts the inter-packet timing resulting in higher rates on calculation. In DPDK architecture the polling mechanism is optimised in such a way the first packet received is processed, which wakes and speeds up the polling mechanism. On largest burst which are processed in multiple subsequent poll() calls, the effect of initial packets are amortized over the remaining chunk. Also at receiver DPDK architecture has been customized to receive packets in burst not less then 32. To achieve higher throughput, it assumes that the receiving max burst size (RX_MAX_BURST_SIZE) is equal to or greater than 32 per burst [58] section 2.1.1.3. Packets are processed in batches of maximum 32 (by default). The burst size can set to a higher number but less than 32 which returns an error.

For readability, the results from figure 23 is presented in magnified scale. Figures [24, 25, 26, 27 and 28] shows the graphs plotted against different burst size against the rates 1, 2, 5, 8 and 10 Gbps respectively.
Figure 23: DPDK performance against different rates and train size

Figure 24: DPDK performance for the rate 1Gbps against different burst size
Figure 25: DPDK performance for the rate 2 Gbps against different burst size

Figure 26: DPDK performance for the rate 5 Gbps against different burst size
Figure 27: DPDK performance for the rate 8 Gbps against different burst size

Figure 28: DPDK performance for the rate 10 Gbps against different burst size
From the figures [24, 25, 26, 27 and 28] it is evident that for the combination of smaller burst size at higher rates, DPDK performance tends to deteriorate in the magnitude of Mbps when compared to other architecture whose rates has error in-order of few Gbps. But for the same rates with large burst size the accuracy improves. Best results (in terms of accuracy) are seen at lower rates with larger burst size. This is due to the fact that the overhead (time to create the packets) are amortized in batch. Because, even thought the interface is continuously polled, practically there is a finite amount of time between the two consecutive poll calls. And there is a possibility that the packet might land in between the two polls.

Figure 29: Scenario illustrating percentage of error observed for different burst size

Figure 29 illustrates this scenario. On the axis, the numbered labels denotes the time at which the interface is being polled. For simplicity, the polling interval is assumed to be 1 second. In figure 29 burst size is considered to be 2 packets. The first packet is received at 0.5 second. Since the interface is not polled, the packet resides in the buffer until the interface is polled. At instance 1 (second), the packet has not completely received. So the first packet is received and processed at instance 2. Similarly the when the second packet is received, at instance 3 the interface is polled and the packet is processed. The total time taken by the interface to receive and process the 2 packets is 2 seconds. But the actual time is 2.5 seconds. The
time calculated is lower than the original time, leads to higher received rates. The percentage of error in the calculated results will be 20%.

In the same scenario, using larger burst size (of 4 packets), the total time to receive 4 packets is calculated as 5 seconds. While the original time is 5.5 seconds. The percentage of error in the calculated results will be 9.09%. Further on increasing burst size to 50 and 100 packets, the error percentage decreases to 0.66% and 0.33% receptively. Similarly on sender, the rates are little lower than the expected rate because the timestamped packets remains in interface buffer till the next polling, which results in larger time intervals and lower rates in the calculation.

Results from figure 30 supports the above argument for train size of 2 packets sent and received at different rate using DPDK architecture. Results obtained at sender and reflector are plotted in the graph. It is evident from the graph, train size of 2 packets has least performance when compared to results obtained using other train sizes. Highest degradation in performance can be found in reflector end at higher rates.

But using larger burst demands more memory to be allocated for packet buffers. Since TWAMP doesn’t pose any limitations on burst size, reasonable burst size varying from 60 to 100 can be used to achieve better performance results.

![Figure 30: DPDK performance for the burst size 2 packets at different rates](image-url)
The following figures 31 and 32 shows the fluctuations (standard deviation) for different rates and burst size on sender and receiver respectively. Obtained results are similar to the previous graphs where the smaller burst size on higher rate has more fluctuations (predominantly seen on receivers end), while larger burst size on lower rates has the least/negligible fluctuation (seen on sender result). Also it evident that, modules (on sender and receiver) using DPDK architecture has the least fluctuations in the results when compared to results obtained from the other architectures.

Figure 31: DPDK - Standard deviation for different rates and burst size on Sender

Figure 32: DPDK - Standard deviation for different rates and burst size on Receiver
6.3 Pros and Cons of the different techniques

After analysing results from various graph, this section aims to highlight pros and cons of various architecture for the readers. The pros and cons for each architecture are described on the outlook to suit TWAMP, not on the architecture itself.

6.3.1 Using Standard UDP Socket

Pros:

• Simple to implement and TWAMP application can run on user-space.
• All the TCP/IP layer headers are created and filled by the native network stack.
• Uses native host stack for communicating with interface.
• Stable, widely used and compatible over most of the operating systems.

Cons:

• Queues over multiple TCP/IP layers and on the interface level slows down the packet processing.
• Maximum speed at which TWAMP measurement packets were sent: 3Gbps
• Also on the reflector side (receiving end), the standard deviation (fluctuation) is very high.

6.3.2 Using LibPcap

Pros:

• In terms of packet processing speed at sender, LibPcap is comparatively better than UDP in sending out TWAMP packets.
• By-passing certain queues in the network stack helps in improving the packet processing speed.

Cons:

• Maximum speed of 6 Gbps can be reached. Still far from reaching the wire-speed.
• On sender and reflector, packet headers has to processed manually.
• On Sender, the standard deviation (fluctuation) is very high.
• Since the headers are manually processed, TCP/IP headers, checksum and other details has to taken care by the TWAMP application.
• On average packets are received and processed only at rate of 6 Gbps on receiving end.
6.3.3 Using Kernel Module

Pros:

- By-passing the host network stack completely helps to reach sending rate till 8.5 Gbps.
- Full control over the packet is available (including the packet meta-data).
- Flexibility in packet processing at reception. TWAMP packet that matches to the 4 tuple Src IP, Dst IP, Src Port, Dst Port can be processed, while the rest of the traffic can be dropped or given back to host stack to processing the packet.
- At reflector the packet in SK_BUFF can be reused to transmit the TWAMP reflector packet adding the TWAMP reflector header and swapping the TCP/IP headers. This save time and memory in re-allocating the memory for packets at reflector.

Cons:

- Netfilter hook are unable to process packets faster. Each packet has to be opened and inspected to match the 4 tuple filter by the TWAMP application to check whether the received packet is TWAMP packet or not.
- Packet headers and checksum has to computed manually.
- Also the results from the measurement packets has to be sent back to the session-client application running on user-space which requires a additional communication mechanisms (such as Netlink sockets).
- Since the operations are done at kernel level, the application is expected to catch and handle the exceptions properly. Chances are high that memory related issues such as segmentation fault or null pointers access can lead to core-dump or kernel panic resulting in host machine freeze.

6.3.4 Using Netmap

Pros:

- With Netmap, TWAMP application is able send and receive packets almost at wire speed (10Gbps).
- Packets are processed by the native stack until a Netmap application is started over the particular interface.
- Possibility of re-injecting the other traffic back to host stack by re-routing them through a separate queue that leads to host stack.
- Application program can be written in user space and API of Netmap can be used to send or receive the packets.
Cons:

- On the reflector side (receiving end), the standard deviation (fluctuation) is very high.
- Also packets accumulates in the buffer distorts the inter-packet time gap which in turn leads to dubious receiving rates.
- Because of continuous polling, CPU utilization is very high all the time.
- To avoid this, TWAMP process can remain idle state. But in order to achieve this a special process or a control protocol is required to wake the TWAMP process into running state during performance measurement.

6.3.5 Using DPDK

Pros:

- With DPDK TWAMP application is able send and receive packets almost at wire speed (10Gbps).
- DPDK has the lowest fluctuation on both sender and receiver.
- TWAMP packets are sent with higher accuracy at the expected rate i.e. lowest mean absolute percentage error.
- Hardware level 5 tuple filter can be enabled to separate TWAMP traffic from the rest.
- Possibility of re-injecting other traffic back to host stack by re-routing them through a separate queue that leads to host stack (rte_kni module is required to achieve this process).
- TWAMP application program can be written in user space and API of DPDK can be used to send or receive the packets

Cons:

- Once the interface is bound to DPDK, there is no mechanism to handle the other ICMP or TCP packets. Separate applications are required to handle the packets
- Because of continuous polling, CPU utilization is very high all the time. CPU can be set to be in idle state. Special control mechanisms can be used to wake the CPU during performance measurement.
- Also in the receiver buffer is optimized and cannot be less than 32 packets.
- DPDK Support is available only for certain NICs. The supported NICS can be in this list [48].
6.4 Timestamping and High resolution timers

This section is included as reference to the work related to timestamping and high-resolution timers that are involved in this thesis. Most of the time this topic emerges into the discussion when the work revolves around time-sensitive applications. Some inputs are provided for quick reference in future. Different Software timestamps APIs available user and in kernel space and user-space. *struct timespec* [59] contains provision to record time in nano-seconds when compared to *struct timeval* where only time upto micro-seconds can be recorded. In kernel modules getnstimeofday() and ktime_get() APIs are provide current time in nano-second resolutions. Generally spin loop are used for creating delays. For delays less than $1\mu$s high resolution timers can be used hrtimer_expires_remaining() can be used for getting higher accuracy. In user-space clock_gettime() with CLOCK_REALTIME provides the better accuracy with nano-second precision. But in user-space APIs for higher-resolution timers are not readily available.

The other topic which is comes into picture when dealing with timestamps is, Hardware timestamps. Currently, most of the NIC’s supports provision to timestamp the in packets at NIC level. More information regarding different types of software and hardware timestamps can be found here [60]. A program that demonstrates how the various time stamping features in the Linux kernel can be used as quick reference to analyse different time stamping techniques can be found here [61], which also provides information regarding NICs that supports software and/or hardware timestamps. The timestamps for packets are enabled using the following macros defined

- **SO_TIMESTAMP** - Generate timestamps for incoming packets using system time via recvmsg() in a control message and provides $\mu$sec resolution using struct timeval.
- **SO_TIMESTAMPNS** - Same as SO_TIMESTAMP. But provides nano second resolution using struct timespec.
- **SO_TIMESTAMPING** - Support timestamps generation on reception, transmission or both. Also it supports timestamps from multiple time sources such as
  - SOF_TIMESTAMPING_TX_HARDWARE,
  - SOF_TIMESTAMPING_RX_HARDWARE,
  - SOF_TIMESTAMPING_TXSOFTWARE,
  - SOF_TIMESTAMPING_RXSOFTWARE,
  - SOF_TIMESTAMPING_TX_SCHED,
  - SOF_TIMESTAMPING_TX_ACK

including software and hardware for stream sockets. The results from the experiments indicated that hardware timestamps did not provide much difference to accuracy in comparison with software timestamps. Also hardware timestamps added overhead to the the process in fetching the timestamps from the hardware through system calls.
This can be seen from the below results.

A simple research experiment was conducted at the receiver, where the incoming TWAMP packets are timestamped enabling different options mentioned below. The results obtained for 50 packets send at 6 Gbps using different timestamping options are shown in table 2.

<table>
<thead>
<tr>
<th>Timestamping Option</th>
<th>Time Taken $\Delta t$ (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SOF_TIMESTAMPING_RX_HARDWARE$</td>
<td>114262</td>
</tr>
<tr>
<td>Hardware time stamping of incoming packets</td>
<td></td>
</tr>
<tr>
<td>$SOF_TIMESTAMPING_RX_SOFTWARE$</td>
<td>108565</td>
</tr>
<tr>
<td>Software fallback for incoming packets</td>
<td></td>
</tr>
<tr>
<td>$SOF_TIMESTAMPING_SYS_HARDWARE$</td>
<td>112961</td>
</tr>
<tr>
<td>Request reporting of transformed HW time stamps</td>
<td></td>
</tr>
<tr>
<td>$SOF_TIMESTAMPING_RAW_HARDWARE$</td>
<td>111471</td>
</tr>
<tr>
<td>Request reporting of raw HW time stamps</td>
<td></td>
</tr>
<tr>
<td>$SIOCGSTAMPNS$</td>
<td>111239</td>
</tr>
<tr>
<td>Check last socket time stamp</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Results from different timestamping options

From the results it clear that the difference between hardware timestamps and software timestamps are not much significant. The difference are in magnitude of few nano-seconds. Also, the system calls in fetching timestamps from the NIC adds overhead to the process.
7 Conclusions and Future work

In this chapter, the conclusions of this thesis are summarized and some possible future works are suggested. Also some reflections on economic, social, and ethical issues associated with this thesis project are discussed.

7.1 Conclusions

In general, the thesis contributes on finding a suitable tool/architecture and a suitable algorithm to measure the network performance over high-speed links using Two-Way Active Measurement Protocol. Most of the measurement systems, build using native UDP socket over Linux network stack, faces a strife challenge when the measurement packets has to be sent over the high-speed (10+ Gbps) links due to the multiples queues in the various TCP/IP layers and the latency in packet processing. Optimising the measurement systems using native UDP sockets did not provide significant improvement in the sending rates (only upto 3 Gbps). The measurement system built using LibPcap architecture helps in sending and receiving TWAMP measurement packets upto 6 Gbps and far from reaching the wire-speed.

The active measurement system build using Kernel Module, by-passing the entire kernel stack improves the maximum sending rate upto 8.5 Gbps with utilizing the opportunity to mitigate per packet interrupts and the Netfilter hooks to receive the packets shows the lest fluctuation in comparison with other methods. But on the other hand, the TWAMP application has to provide mechanism to create and fill the different TCP/IP layer header and other operation which are handled by the kernel stack. The architecture using Netmap which aims to reduce the cost of moving traffic between hardware and the host network stack for faster packet processing, helps the measurement system to reach wire-speed. But on the receiving end, Netmap shows more fluctuation in the values in the iteration and also has the highest mean percentage error which is the result of buffering the received packets.

On the other hand using DPDK where the interface drivers are mapped to the user-space applications and with its own set of libraries to process the packet in efficient methods, helps the measurement systems to send and receive TWAMP packets at full wire-speed. In comparison with the other architecture DPDK has the lowest mean percentage error and lowest standard deviation when measuring networking performance over high-speed links.
Another interesting phenomenon is the results from the experiments for different train size (or burst size) vs various sending and receiving rates using DPDK shows the least accuracy or more mean error percentage when the burst size is lower. However, the higher accuracy is seen with larger burst size. Better results are obtained for burst size greater than 50. Since TWAMP does not impose any restriction on burst size, larger burst size can be used during measurement for higher accuracy. Also the mean percentage error for DPDK over different sending and receiving rates is comparatively lower then other techniques where the packets are sent and received with higher accuracy at lower speed when compared to higher rates. Considering the above mentioned reasons, the measurement module built using DPDK along with the optimized algorithm, suits the requirements for measuring the network performance, especially over the high-speed links.

7.2 Future Works

In this master thesis project, the suitable architecture and optimized algorithm was designed and implemented using DPDK architecture to send TWAMP measurement packets at higher rates. The next step would be find an appropriate method to re-inject the other traffic into the native stack. As mentioned earlier, KNI control plane allows the user-space applications to exchange packets with kernel networking stack. Currently, the algorithm separates the TWAMP measurement packets from the rest of the traffic with the help of five-tuple filters. The TWAMP measurement packets ends in a different queues while the other traffic lands in default queue. This can be utilized to redirect the traffic to host stack. Simplest way is to redirect the packet to another interface that is still in contact with host stack. Or an internal communication channel has to be formed to redirect the packets to the host kernel stack.

Another major work that has significantly wider scope for research is, when the TWAMP applications are deployed in a virtualized environment like cloud using DPDK architecture. The TWAMP application running inside the virtual machines can be used to analyse and differentiate the performance of virtual devices/KVM and with the performance of the high-speed links, e.g. the time difference between timestamps T2 and T3 can provide the time spent in processing the TWAMP packet in the refector, which can help to calculate the performance of virtualized devices. Some recent work have been proposed to integrate OpenStack over DPDK as base architecture. Therefore, it is worth to study in detail the benefits of porting TWAMP applications over virtualized environment.
7.3 Reflections on Environmental, Social, Ethical and Economical impact by the thesis

This master’s thesis project facilitates the opportunity to identity the suitable architecture that helps in available path capacity measurements over the high-speed links with high accuracy. Also the capacity measurements made using the proposed architecture and the optimized algorithm assist in detecting the congested links in the network by which the network operator can fine tune the network to provide better quality of service. This in turn increases the client’s satisfaction with the services provided by the network operator, which is a beneficial social effect of this thesis.

In General, in order to generate and send TWAMP measurement packets at higher rates, powerful test devices with high-end hardware configuration and customized network stack are used for generating packets at high speed. But by using the proposed architecture and the optimized algorithm, TWAMP measurement packets can be generated at higher rates on the standard test equipments. This thesis, being a less expensive alternative makes an economic contribution to the network operator’s business. The time apart from the capacity measurement, the test equipment can be switched back to use the native host stack by which the device can be reused for other applications. This provides economical savings by annihilating the cost for dedicated high-end measurement devices.

Due to business and ethical reasons, the TWAMP algorithm used in the existing systems and results of the rate at which the packets are sent using some vendor specific equipments are not disclosed.

By using TWAMP light version, necessity of exchanging the control packets such as transmitting keep alive packets frequently is eliminated. This in turn reduced the power consumption of the devices and links. Also using the standard test devices in ‘on-demand’ mode helps in lower power consumption when compared to devices with dedicated hardware of high-end specifications. These factors helps the thesis to have a positive environmental impact by reducing the power consumption. However, arguments can be raised regarding the continuous polling of interface by DPDK architecture can increase the power consumption by the test device due to increase in CPU load. Generally the TWAMP test sessions last for few minutes. Apart from the performance measurement time, the device can be set to sleep mode until required. This argument intuitively indicates the power consumed during the capacity measurements is lower than normal TWAMP protocol where the sender and reflector devices need to be running all the time in-order to maintain the TWAMP control session.
References


[16] Defining Network Capacity

[17] Internet protocol data communication service – IP packet transfer and availability performance parameters
https://www.itu.int/rec/T-REC-Y.1540

[18] A One-way Active Measurement Protocol (OWAMP)


http://lartc.org/howto/lartc.qdisc.html
[25] Packet control flow through Linux kernel
http://www.linuxfoundation.org/collaborate/workgroups/networking/kernel_flow

[26] CPU frequency and voltage scaling code in the Linux(TM) kernel
https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[27] Programming with LibPcap - Sniffing the network from our own application

[28] TcpDump
http://www.tcpdump.org/

[29] Linux Socket Filtering aka Berkeley Packet Filter (BPF)
https://www.kernel.org/doc/Documentation/networking/filter.txt

https://wiki.wireshark.org/

[31] Snort
https://www.snort.org/

[32] Sending Packets using LibPcap
https://www.winpcap.org/docs/docs_40_2/html/group__wpcap__tut8.html

[33] Receiving Packets using LibPcap
http://www.tcpdump.org/pcap.html

[34] Introduction to Linux Loadable Kernel Modules
http://www.tldp.org/HOWTO/Module-HOWTO/x73.html

[35] Pktgen - Linux Foundation
http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen

[36] SK_BUFF - A common data structure for network-related queues and buffers in the kernel
http://www.linuxfoundation.org/collaborate/workgroups/networking/sk_buff

[37] Kernel Space - User Space Interfaces
http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html#ss4.1

[38] Communicating between the kernel and user-space in Linux using Netlink sockets
[39] Brief introduction to Generic Netlink - Linux Foundation
   http://www.linuxfoundation.org/collaborate/workgroups/networking/
   generic_netlink_howto

[40] How to use Netlink Sockets
   http://binwaheed.blogspot.se/2010/08/after-reading-kernel-source-
   i-finally.html

[41] Netfilter
   http://netfilter.org/

[42] IPTables
   http://linux.die.net/man/8/iptables

[43] Poll() in Linux
   http://linux.die.net/man/2/poll

[44] Netmap - Source code
   https://code.google.com/p/netmap/


[46] Intel Atom Processor
   http://www.intel.com/content/www/us/en/processors/atom/atom-
   processor.html

[47] Intel Xeon Processor E7 Family
   http://www.intel.in/content/www/in/en/processors/xeon/xeon-
   processor-e7-family.html

[48] DPDK Supported NICs
   http://dpdk.org/doc/nics

[49] Roving Reporter: Four Telecom Workloads, One Architecture - DPDK
   blog/2012/01/03/roving-reporter-four-telecom-workloads-one-
   architecture

[50] DPDK Sample Applications User Guide
   guides/intel-dpdk-sample-applications-user-guide.pdf

[51] SDN, NFV, DPDK and the Open Networking Platform - FAQ
   https://software.intel.com/en-us/articles/sdn-nfv

[52] High Performace NFV Infrastructure (NFVI): DPDK Host applications with
    Neutron/OpenStack and VNF Acceleration
   http://events.linuxfoundation.org/sites/events/files/slides/
   Openstack-v4_0.pdf
[53] MoonGen: A Scriptable High-Speed Packet Generator  
http://arxiv.org/abs/1410.3322


[55] Intel Xeon Processor X5660 Specifications  

[56] Ubuntu Manual - Getting Started with Ubuntu 14.04  

[57] Linux Kernel 3.18.4 Stable Released, Install In Ubuntu and Linux Mint  

[58] DPDK - IXGBE Drive  

[59] clock_gettime(3) - Linux man page  
http://linux.die.net/man/3/clock_gettime

[60] Linux Timestamping  
https://www.kernel.org/doc/Documentation/networking/timestamping.txt

[61] The Linux PTP Project  
http://linuxptp.sourceforge.net/
## Appendix

*Table 3 shows the specification of test equipments used for the experiments.*

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU op-mode(s)</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s)</td>
<td>24</td>
</tr>
<tr>
<td>On-line CPU(s) list</td>
<td>0-23</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>6</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>2</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>2</td>
</tr>
<tr>
<td>Vendor ID</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CPU family</td>
<td>6</td>
</tr>
<tr>
<td>Model</td>
<td>44</td>
</tr>
<tr>
<td>Stepping</td>
<td>2</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>1600.000</td>
</tr>
<tr>
<td>BogoMIPS</td>
<td>5597.85</td>
</tr>
<tr>
<td>Virtualization</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32k</td>
</tr>
<tr>
<td>L1i cache</td>
<td>32k</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256k</td>
</tr>
<tr>
<td>L3 cache</td>
<td>12288k</td>
</tr>
<tr>
<td>NUMA node0 CPU(s)</td>
<td>0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22</td>
</tr>
<tr>
<td>NUMA node1 CPU(s)</td>
<td>1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23</td>
</tr>
</tbody>
</table>

*Table 3: Test device Specifications*
For readers quick reference:

Figure 33 shows the TWAMP packet format at the reflector end, which includes the sender and reflector headers together.

Figure 33: TWAMP reflector packet format. Source [19]
Figure 34: Ericsson TWAMP extension sender packet format [21]

Figure 34 shows the Ericsson TWAMP extension packet format at the sender end. Additional fields such as version number, last sequence number in the train, desired reverse packet interval are included in the extension.