TOWARDS PRECISE NETWORK MEASUREMENTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ki Suh Lee

January 2017

© 2017 Ki Suh Lee

ALL RIGHTS RESERVED

TOWARDS PRECISE NETWORK MEASUREMENTS

Ki Suh Lee, Ph.D.

Cornell University 2017

This dissertation investigates the question: How do we precisely access and control time in a network of computer systems? Time is fundamental for network measurements. It is fundamental in measuring one-way delay and round trip times, which are important for network research, monitoring, and applications. Further, measuring such metrics requires precise timestamps, control of time gaps between messages and synchronized clocks. However, as the speed of computer networks increase and processing delays of network devices decrease, it is challenging to perform network measurements precisely.

The key approach that this dissertation explores to controlling time and achieving precise network measurements is to use the physical layer of the network stack. It allows the exploitation of two observations: First, when two physical layers are connected via a cable, each physical layer always generates either data or special characters to maintain the link connectivity. Second, such continuous generation allows two physical layers to be synchronized for clock and bit recovery. As a result, the precision of timestamping can be improved by counting the number of special characters between messages in the physical layer. Further, the precision of pacing can be improved by controlling the number of special characters between messages in the physical layer. Moreover, the precision of synchronized clocks can be improved by running a protocol inside the physical layer by extending bit-level synchronization.

Subsequently, we make three contributions embodied in the design, imple-

mentation and evaluation of our approaches. First, we present how to improve the precision of timestamping and pacing via access to the physical layer at sub-nanosecond scale. SoNIC implements the physical layer in software and allows users to control and access every bit in the physical layer. Second, we demonstrate that precise timestamping and pacing can improve the performance of network applications with two examples: Covert timing channels and available bandwidth estimation. A covert timing channel, Chupja, is high-bandwidth and robust and can deliver hidden messages while avoiding detection. An available bandwidth estimation algorithm, MinProbe, can accurately estimate the available bandwidth in a high-speed network. Finally, we present how to improve the precision of synchronized clocks via access to the physical layer. DTP, Datacenter Time Protocol, extends the physical layer's link-level synchronization and implements a peer-to-peer clock synchronization protocol with bounded nanosecond precision. Together, these systems and approaches represent important steps towards precise network measurements.

BIOGRAPHICAL SKETCH

Ki Suh Lee earned his Bachelor of Science degree in Computer Science and Engineering from Seoul National University in Korea in 2007, and his Master of Science degree in Computer Science from Columbia University in 2009. Then, he joined the doctoral degree program in Computer Science in Cornell University, where he worked with Hakim Weatherspoon to build many systems using field-programmable gate array boards. His research interests broadly cover networking, operating systems, and computer architectures.

To Kayoung, Ethan, and my parents.

ACKNOWLEDGEMENTS

I would like to thank the faculty members at Cornell. I would like to thank my advisor, Hakim Weatherspoon, who showed and taught me how to become a researcher in computer systems. He always guided me to build a system and at the same time not to forget the fundamental research questions that I was trying to answer by building the system. I would not forget the time we spent together to write papers till late nights. Kevin Tang showed me how to do research based on math and theory in computer networks. Further I am sincerely grateful to Fred Schneider and Nate Foster who gracefully accepted to be a member of my special committee and provided me invaluable advice to finish my dissertation.

I would also like to thank my research collaborators. What could I have done without Han Wang. From the earlier years of my study Han was always by my side and helped me in every aspect of my research. I thank Tudor who guided me to become a Linux kernel programmer in my earlier years. I thank Chiun Lin, Erluo Li and Vishal Shrivastav for their collaborations on many projects. I also thank many undergraduate students who helped my research: Ethan Kao, Jason Zhang, and Favian Contreras. I would also like to thank my colleagues in the Computer Science department at Cornell and members of Systems Lab and Networking Lab: Ji-Yong Shin, Zhiyuan Teo, Zhiming Shen, Qin Jia, Praveen Kumar, Robert Escriva, Hussam Abu-Libdeh, Dan Williams, Qi Huang, Joonsuk (Jon) Park, and Moontae Lee. I also thank Ed Kiefer and Eric Cronise from CIT who helped me set up network testbeds.

Outside the Systems Lab and Networking Lab, I had a pleasure to serve God in a praise team where I played acoustic / bass guitar, mixed the sounds, and served as a Bible study leader. I thank all members of the team and am especially grateful to Jayhun Lee and Hyunjung Kim for so many reasons. I

would not forget enjoyable moments we had together as a team.

I cannot thank enough for my family for their support. I thank my lovely wife Kayoung who was at my side always and happily endured long seven years of life in Ithaca. Her endless love and support made me who I am now. Further I had a great joy of having a son, Ethan, during my study. He made me happy and laugh. I thank my parents for their endless support and prayer.

Thank God for everything.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vii
List of Tables x
List of Figures xi

1 Introduction **1**
1.1 Background 3
1.1.1 Terminology 3
1.1.2 Example: Estimating Available Bandwidth 4
1.2 Challenges for Precisely Accessing Time 7
1.2.1 Lack of Precise Pacing 7
1.2.2 Lack of Precise Timestamping 8
1.2.3 Lack of Precise Clock Synchronization 10
1.3 Contributions 12
1.4 Organization 13

2 Scope and Methodology **15**
2.1 Scope 15
2.1.1 IEEE 802.3 standard 16
2.1.2 Precise Timestamping and Pacing Need Access to PHY . . 20
2.1.3 Precise Clock Synchronization Needs Access to PHY . . . 23
2.2 Methodology: Experimental Environments 29
2.2.1 Hardware 30
2.2.2 Network Topologies 32
2.3 Summary 37

3 Towards Precise Timestamping and Pacing: SoNIC **38**
3.1 Design 40
3.1.1 Access to the PHY in software 40
3.1.2 Realtime Capability 41
3.1.3 Scalability and Efficiency 43
3.1.4 Precision 44
3.1.5 User Interface 45
3.1.6 Discussion 47
3.2 Implementation 48
3.2.1 Software Optimizations 48
3.2.2 Hardware Optimizations 53
3.3 Evaluation 56
3.3.1 Packet Generator (Packet Pacing) 58
3.3.2 Packet Capturer (Packet Timestamping) 60

3.3.3 Profiler . . . 62
3.4 Application 1: Covert Timing Channel . . . 63
3.4.1 Design . . . 66
3.4.2 Evaluation . . . 71
3.4.3 Summary . . . 89
3.5 Application 2: Estimating Available Bandwidth . . . 89
3.5.1 Design . . . 91
3.5.2 Evaluation . . . 92
3.5.3 Summary . . . 95
3.6 Summary . . . 96

4 Towards Precise Clock Synchronization: DTP 97
4.1 Design . . . 99
4.1.1 Assumptions . . . 99
4.1.2 Protocol . . . 100
4.1.3 Analysis . . . 105
4.2 Implementation . . . 108
4.2.1 DTP-enabled PHY . . . 108
4.2.2 DTP-enabled network device . . . 110
4.2.3 Protocol messages . . . 111
4.2.4 DTP Software Clock . . . 112
4.3 Evaluation . . . 114
4.3.1 Methodology . . . 115
4.3.2 Results . . . 116
4.4 Discussion . . . 120
4.4.1 External Synchronization . . . 120
4.4.2 Incremental Deployment . . . 121
4.4.3 Following The Fastest Clock . . . 122
4.4.4 What about 1G, 40G or 100G? . . . 123
4.5 Summary . . . 124

5 Related Work 125
5.1 Programmable Network Hardware . . . 125
5.2 Hardware Timestamping . . . 126
5.3 Software Defined Radio . . . 127
5.4 Software Router . . . 127
5.5 Clock Synchronization . . . 128

6 Future Work 136
6.1 Synchronous DTP . . . 136
6.2 Packet Scheduler . . . 137
6.3 SoNIC, DTP, and P4 . . . 138

7 Conclusion 140

A PHY Tutorial 142
A.1 Physical layer: Encoder and Scrambler . . . 142
A.1.1 Introduction . . . 142
A.1.2 Encoding . . . 143
A.1.3 Scrambler . . . 145
A.1.4 Task . . . 147
A.2 Physical Layer: Decoder and Descrambler . . . 148
A.2.1 Introduction . . . 148
A.2.2 Task . . . 149
A.3 Pseudo C code for 64/66b Encoder and Decoder . . . 151

B Fast CRC algorithm 157
B.1 Psuedo assembly code for Fast CRC algorirthm . . . 158

C Optimizing Scrambler 162

D Glossary of Terms 165

Bibliography 171

LIST OF TABLES

2.1 List of servers used for development and experiments 30
2.2 List of FPGA boards and NICs used for development and experiments 31
2.3 List of evaluated network switches. "SF" is store-and-forward and "CT" is cut-through. 31

3.1 DMA throughput. The numbers are average over eight runs. The delta in measurements was within 1% or less. 55
3.2 IPD and IPG of homogeneous packet streams. 71
3.3 Evaluated ϵ values in the number of /I/s and their corresponding time values in nanosecond. 71
3.4 ϵ and associated BER with (1518B, 1Gbps) 80
3.5 Characteristics of CAIDA traces, and measured I_{90} and BER . . . 82
3.6 I_{90} values in nanosecond with cross traffic. 83
3.7 Relation between ϵ, I_{90}, and $\max(I^{+}_{90}, I^{-}_{90})$ over different networks with (1518B, 1Gbps). Values are in nanosecond. 84
3.8 Parameter setting for existing algorithms. G is the gap between packet trains. R is the rate of probe. N is the number of probe packets in each sub-train. D is the gap between each sub-train. . 92

4.1 Specifications of the PHY at different speeds 123

5.1 Comparison between NTP, PTP, GPS, and DTP 128

LIST OF FIGURES

1.1 Collected interpacket delays between two servers. 6
1.2 PTP precision 11

2.1 IEEE 802.3 10 Gigabit Ethernet Network stack. 17
2.2 IEEE 802.3 64b/66b block format 19
2.3 Common approach to measure offset and RTT. 24
2.4 Clock domains of two peers. The same color represents the same clock domain. 27
2.5 FPGA development boards used for our research. 30
2.6 Simple evaluation setup 33
2.7 A small network. Thick solid lines are 10G connections while dotted lines are 1G connections. 34
2.8 Our path on the National Lambda Rail 35
2.9 Evaluation Setup for DTP 36

3.1 Example usages of SoNIC 43
3.2 Example C code of SoNIC Packet Generator and Capturer 45
3.3 SoNIC architecture 47
3.4 Throughput of packing 51
3.5 Throughput of different CRC algorithms 51
3.6 Throughput of packet generator and capturer 57
3.7 Comparison of packet generation at 9 Gbps 58
3.8 Comparison of timestamping 60
3.9 IPDs of Cisco 4948 and IBM G8264. 1518B packets at 9 Gbps . . . 61
3.10 Chupja encoding and decoding. 68
3.11 Maximum capacity of PHY timing channel 69
3.12 BER of Chupja over a small network and NLR. X-Y-Z means that workload of cross traffic is X (H-heavy, M-medium, or L-light), and the size of packet and data rate of overt channel is Y (B-big=1518B or S-small=64B) and Z (1, 3, or 6G). 73
3.13 Comparison of IPDs after switches process a homogeneous packet stream (1518B, 1Gbps) 76
3.14 I_{90} comparison between various switches 77
3.15 Comparison of homogeneous streams and covert channel streams of (1518B, 1Gbps) 78
3.16 BER over multiple hops of SW1 with various ϵ values with (1518B, 1Gbps) 81
3.17 Packet size distributions of CAIDA traces 81
3.18 Comparison of various timestamping methods. Each line is a covert channel stream of (1518B, 1Gbps) with a different ϵ value. 85
3.19 Kernel timestamping with (1518B, 1Gbps). 87
3.20 Hardware timestamping with (64B, 1Gbps) 88

3.21 Generalized probe train model 91
3.22 Bandwidth estimation of CAIDA trace, the figure on the top is the raw data trace, the figure on the bottom is the moving average data. 93
3.23 Measurement result in NLR. 94

4.1 Low layers of a 10 GbE network stack. Grayed rectangles are DTP sublayers, and the circle represents a synchronization FIFO. 109
4.2 DTP enabled four-port device. 110
4.3 Precision of DTP and PTP. A *tick* is 6.4 nanoseconds. 117
4.4 Precision of DTP daemon. 118

A.1 10G Network Stack 143
A.2 66b block format 145
A.3 Example 145
A.4 Scrambler 146
A.5 Descrambler 149

CHAPTER 1

INTRODUCTION

Time is "a finite extent or stretch of continued existence, as the interval separating two successive events or actions, or the period during which an action, condition, or state continues" [35]. In other words, we use the term, *time*, to specify the moment or duration of a particular event. We also use time to serialize the ordering of events. Time is inseparable from our daily lives and has been an interesting research topic in many different areas including physics and philosophy. For example, in physics, Einstein defined *time* as what a *clock* at a particular location *reads* to explore the concept of relativity [56]. When one wants to compare the time of two events from two different locations, the clocks from two locations must be *synchronized* to provide a common time. There are many commonly used *standard times* including Coordinated Universal Time (UTC).

In computer systems, *system time* is used to specify a moment of creation, update, or removal of an object, to measure the duration of an event, and to order a series of events. System time is measured by a *system clock*, which normally keeps track of the number of seconds since the Epoch (00:00:00 UTC, January 1, 1970) [33]. The synchronization between a system clock and the reference time (UTC) can be achieved via a time protocol such as Network Time Protocol (NTP) [98] or Precise Time Protocol (PTP) [18]. The time difference between a system clock and a reference time can vary from tens of nanoseconds to any number of seconds.

Time is also essential for many network applications and network management. For example, many network applications use *timestamps* of messages, i.e. marking the time at which the message is sent or received. A timestamp can be

generated by *reading* the system clock or the hardware clock of the network interface card (NIC) that sends or receives the particular message. Then, a process can measure a round trip time (RTT) to another process in a network by comparing two timestamps: One is generated before sending a request message and the other is generated upon receiving a response message. Similarly, a process can measure a one-way delay (OWD) to another process by sending a probe message with its current time. Then, the receiving process can compute the OWD by subtracting the received time from its own time. Note that measuring OWD requires the clocks of two processes to be synchronized.

In this dissertation, we focus on three fundamental research problems that are important for network measurements with respect to time: Timestamping messages, controlling timing of messages (*pacing*), and clock synchronization. Precisely timestamping and pacing messages and synchronizing clocks in a network is challenging, especially as the speed of the network continues to increase while at the same time processing delay of network devices continues to decrease over time. For instance, a packet can arrive every 64 nanoseconds in 10 Gigabit Ethernet (GbE), and even more frequently in faster networks such as in 40 or 100 GbE. On the other hand, it takes nearly a microsecond for a network switch to process a message. As a result, it is necessary to have a capability of timestamping and pacing messages at nanosecond scale and to synchronize clocks at similar granularity. Therefore, we seek to investigate the following research question in this dissertation: *How do we precisely access and control time in a network of computer systems?*

In the course of exploring the above research question, we devised an augmented physical layer of the network protocol stack that provides timing infor-

mation for network measurements. The augmented physical layer allows users to control and access every single bit over the medium for precise timestamping and pacing. Further, the augmented physical layers in a network provide tightly synchronized clocks with bounded precision. In this dissertation, we investigate how the physical layer can provide precise time and how it improves the performance of network applications.

1.1 Background

1.1.1 Terminology

A *clock* c of a process p[1] is a function that returns a local clock counter given a real time t, i.e. $c_p(t)$ = local clock counter. Note that a clock is a discrete function that returns an integer, which we call clock *counter* throughout the dissertation. A clock changes its counter at every clock *cycle* (or *tick*). If clocks c_i for all i are synchronized, they will satisfy

$$\forall i, j, t \;\; |c_i(t) - c_j(t)| \leq \epsilon \tag{1.1}$$

where ϵ is the level of *precision* to which clocks are synchronized [116].

Each clock is usually driven by a quartz oscillator, which oscillates at a given frequency. *Oscillators* with the same nominal frequency may run at different rates due to frequency variations caused by external factors such as temperature. As a result, clocks that have been previously synchronized will have clock counters that differ more and more as time progresses. The difference

[1] We will use the term *process* to denote not only a program that is being executed on a processor but also a program running inside a network interface card.

between two clock counters is called the *offset* or *skew*, which tends to increase over time, if not resynchronized. Therefore, clock synchronization protocols periodically adjust offsets between clocks (offset synchronization) and/or frequencies of clocks (frequency synchronization) so that they remain close to each other [116].

If a process attempts to synchronize its clock to true time by accessing an external clock source such as an atomic clock or a satellite, it is called *external synchronization*. If a process attempts to synchronize with another (peer) process with or without regard to true time, it is called *internal synchronization*. Thus, externally synchronized clocks are also internally synchronized, but not vice versa [50]. In many cases, monotonically increasing and internally synchronized clocks are sufficient. For example, measuring one-way delay and processing time or ordering global events do not need true time.

1.1.2 Example: Estimating Available Bandwidth

In order to illustrate the challenges of precisely accessing and controlling time, we will use an example of estimating available bandwidth, the maximum data rate that a process can send to another process without going over the network capacity between the two. Many algorithms for estimating available bandwidth use OWD [68, 111, 120].

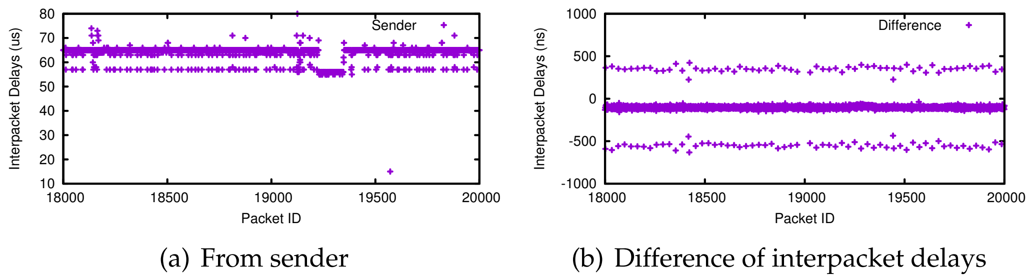
Suppose a process p wants to estimate available bandwidth between itself and a process q. The process p uses a train of probe messages $(m_1, \cdots, m_n)$ to estimate available bandwidth. The process p controls the time gap between messages to be a pre-computed interval: Let d_i^p be the time gap between m_i

and m_{i+1}. Each message m_i carries the timestamp t_i^p which is the time when the message was sent by p. The process q then timestamps incoming messages (t_i^q) and computes OWD of each message ($=t_i^q - t_i^p$). Available bandwidth estimation algorithms then examine and detect any changes in measured OWDs, which are good indicators of buffering, queuing or congestion in the network. Then the algorithms infer the available bandwidth.

There are a few potential places which can contribute to inaccuracy in the estimated available bandwidth. *Accuracy* means the closeness of an estimated bandwidth to an actual bandwidth available. First, the timestamp t_i^p of the probe messages could be imprecise mainly due to the time gap between the moment p timestamps a message[2] and the moment the message is transmitted over the wire. The timestamp of a message is *precise* when the timestamp is close to the actual time when the message was sent or received by hardware. Similarly the timestamp t_i^q could also be imprecise. Second, the time gap d_i^p might not be the actual time gap between m_i and m_{i+1}. The sender normally controls the time gap by sleeping for d_i^p seconds after sending m_i. However, sleeping, waking up and sending a message can introduce random delays, and thus can make the actual gap $d_i'^p$ to be larger than d_i^p. Pacing, or controlling time gap, is *precise* when an intended time gap between two packets d_i^p is close to the actual gap $d_i'^p$ between them after they are transmitted by hardware. Lastly, there is an uncertainty in the degree of synchronization of the clocks of p and q, which could vary from nanoseconds to any number of seconds. We explore each contributor in the following section.

Note that OWD is often estimated to be RTT / 2 as synchronizing clocks is

[2]We will assume that reading a clock and embedding time to a message can happen almost at the same time or with very small time gap for simplicity.



(a) From sender

(b) Difference of interpacket delays

Figure 1.1: Collected interpacket delays between two servers.

a difficult problem. However, such estimation is prone to errors mainly due to path asymmetry: The path that a request message takes might be different from the path that a response message takes and the time that a request message takes might also be different from the time that a response message takes.

Also note that time gaps between messages could be very small when the rate of probe messages is high. For example, time gaps of probe messages at 9 Gigabits per second (Gbps) are smaller than 1.5 microsecond in 10 Gigabit networks. As a result, accurate available bandwidth estimation requires at least microsecond level of precision of pacing and timestamping messages and synchronized clocks in a 10 Gigabit network. Without fine-grained control of time gaps and precise timestamping, the accuracy of available bandwidth estimation algorithms could dramatically decrease [125].

1.2 Challenges for Precisely Accessing Time

1.2.1 Lack of Precise Pacing

Pacing messages is to insert arbitrary delays between messages, i.e. pacing controls time gap between packets. Pacing is often used to improve the performance of network protocols [126, 135]. There are mainly three places where time gaps can be controlled: Userspace, Kernel-space, and hardware (NIC). Userspace applications can control the time gap between two packets by waiting or sleeping for a certain amount of time after sending the first packet. However, controlling the time gap with fine granularity at microseconds or nanoseconds scale is challenging because the scheduler of the system can introduce random delays into the time gap when the process waits or sleeps. In kernel space, controlling time gaps at microsecond granularity is possible with a high-resolution timer [121]. Pacing in kernel space is often limited to support transport protocols [121, 110], and other network applications not implemented in the kernel cannot normally use the high-resolution timer to control time gaps. Commodity network interface cards normally do not provide an interface for controlling time gaps, and hardware that supports pacing only supports Transmission Control Protocol [129].

In order to demonstrate how hard it is to precisely pace messages, we conducted a simple experiment. We used two servers each with a NIC with hardware timestamping (Discussed below in Section 1.2.2) capability on both transmit and receive paths. The NICs were identical and directly connected via a three-meter cable. The sender periodically sent a message and waited for one microsecond before sending the next. The sender recorded timestamps of out-

going messages reported by the hardware. Then, we computed time gaps between two successive messages and plotted them in Figure 1.1(a). Figure 1.1(a) shows that the actual time gap between messages is around 65 microseconds which is much larger than the intended time gap, one microsecond. This simple experiment demonstrates the challenge of precisely controlling time gaps. As a result, we investigate the following research question in this thesis: *What is required to precisely pace messages?*

In this dissertation, we explore an alternative way of packet pacing. Instead of waiting / sleeping, or relying on a high-resolution timer, we attempt to directly control the number of bits between messages in the physical layer. We mainly exploit the fact that Ethernet uses one bit per *symbol* over the medium for data communication. For example, it takes about 100 picoseconds to deliver one symbol over the wire in 10 Gigabit Ethernet (GbE). As a result, controlling bits in the physical layer allows precise pacing on the transmit path, which is not readily available in commodity NICs. We present SoNIC in Chapter 3 which achieves the precise pacing by implementing the physical layer of a network stack in software.

1.2.2 Lack of Precise Timestamping

Messages can also be timestamped at three places: Kernel-space, userspace, and hardware. On the receive (RX) path, kernel records arrival times of packets upon receiving them from hardware. Since so many factors are involved during the delivery of a packet to kernel space, such as DMA transaction, interrupt routines, and scheduler, kernel timestamping can be imprecise in a high-speed

network. As a result, userspace timestamping will also be imprecise because of delays added due to interaction between kernel and userspace.

Modern commodity NICs provide *hardware timestamping*. A NIC uses an additional quartz oscillator inside the hardware to timestamp incoming and outgoing packets at a very early (late) stage for the receive (transmit) path to achieve better precision normally between the Media Access Control (MAC) layer and the physical layer. The oscillator can be synchronized to an external clock or be free-running. As a result, the precision of timestamping can be significantly improved.

Unfortunately, hardware timestamping is still not precise. With the same experiment setup discussed in Section 1.2.1, we also captured timestamps of messages from the receiver and computed interpacket delays. Then, we compared computed delays from the sender and the receiver. Figure 1.1(b) illustrates the difference between corresponding interpacket delays in nanoseconds. As there is nothing in the middle between two identical NICs except a cable, we would expect that corresponding interpacket delays from the sender and the receiver would match or close to each other assuming that timestamps are precise. However, Figure 1.1(b) shows that the interpacket delays from the sender and the receiver could differ by four or five hundred nanoseconds for the most of time and at worst case could differ by ten microseconds (which is not shown in the graph).

Therefore, we investigate the following research question: *What is required to improve the precision of timestamping?*

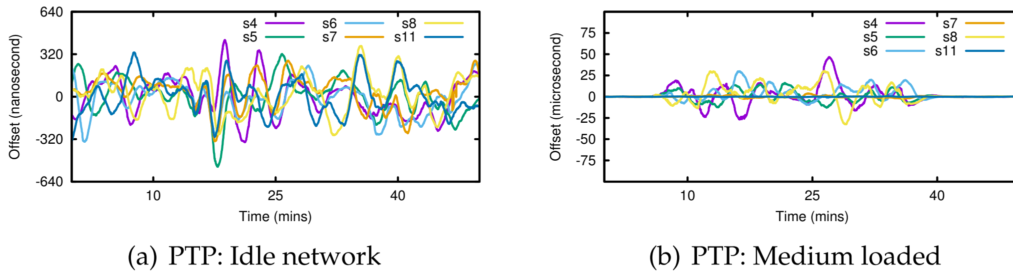
In this dissertation, we explore an alternative way of hardware timestamp-

ing. Instead of timestamping messages between the Media Access Control (MAC) layer and the physical layer (PHY), we attempt to timestamp messages inside the PHY. In particular, rather than using an extra clock to timestamp messages outside the physical layer, we use the clock that generates bits over the wire for timestamping. By doing so, we can use the number of bits between any two messages inside the physical layer as a base for precise timestamping. As noted in Section 1.2.1, it takes about 100 picoseconds to deliver one symbol over the wire in 10 GbE. As a result, the precision of timestamping can significantly improve. We discuss in Chapter 3 how SoNIC which implements the physical layer in software can also improve the precision of timestamping.

1.2.3 Lack of Precise Clock Synchronization

Synchronizing clocks is a challenging problem. It is challenging due to the problem of measuring round trip times (RTT) accurately. RTT is used by many clock synchronization protocols to compute clock *offset* (*skew*), the time difference between two system clocks. RTTs are prone to variation due to characteristics of packet switching networks: Network jitter, packet buffering and scheduling, asymmetric paths, and network stack overhead. As a result, any protocol that relies on RTTs must carefully handle measurement errors.

There are many time synchronization protocols that provide different levels of precision including NTP [98] and PTP [18]. NTP normally provides millisecond to microsecond precision in a Local Area Network (LAN), and PTP provides microsecond to tens of nanosecond precision in a LAN when properly configured. PTP employs many techniques to remove uncertainties in mea-



(a) PTP: Idle network (b) PTP: Medium loaded

Figure 1.2: PTP precision

sured RTTs. For example, hardware timestamping is commonly used and PTP-enabled switches are deployed to minimize network jitter.

Nonetheless, it is not possible to completely remove the network jitter and non-deterministic delays. Figure 1.2 illustrates the problem. Figure 1.2(a) shows the performance of PTP when the network is idle, and Figure 1.2(b) shows the performance of PTP when the network is loaded with network traffic. The detailed test environment is described in Section 2.2. The takeaway is that the performance of PTP can be as good as hundreds of nanoseconds, and can degrade to tens of microseconds depending on network conditions. Therefore, we investigate the following research question: *How can we minimize non-deterministic delays from the network to improve the precision of time synchronization protocols?*

In this dissertation, we developed a new time synchronization protocol called DTP, Datacenter Time Protocol (DTP), that can provide tens of nanosecond precision in a datacenter environment. The protocol runs inside the physical layer in order to eliminate many non-deterministic delay errors from the network. In particular, by running the protocol in the physical layer in a peer-to-peer fashion, it can minimize the network jitter and easily scale up to a larger network. Chapter 4 discusses the design and evaluation of the system.

1.3 Contributions

Traditionally, the physical layer of a network stack in a wired network has been considered as a black box and excluded from network research opportunities. However, we demonstrate that an augmented physical layer with timing information can improve many fundamental capabilities that are important for network measurements including packet pacing, packet capturing, and clock synchronization.

Our research is based on a fact that the physical layer of a network device is already synchronized with the physical layer of its directly connected device. Further, they send a continuous stream of symbols to each other in order to maintain the link between them. Our systems extend and utilize such symbol-level synchronization to achieve fine-grained timestamping, pacing and highly precise clock synchronization. In particular, we present the following three contributions in this dissertation.

Precise timestamping and pacing Accessing individual symbols on the wire requires complete access to the very bottom of the network protocol stack, the physical layer where symbols are transmitted over or received from the wire. Unfortunately, accessing the inside of the physical layer is not easy: The physical layer is often implemented as a physical chip with a limited or no interface. As a result, complete access to the physical layer requires a different approach: We demonstrate a new approach via SoNIC, a Software-defined Network Interface Card, where the entire physical layer is accessible from software in a way that a user application can control and capture the timing of physical layer bits and achieve bit level precision.

Covert timing channel and Estimating available bandwidth In order to demonstrate how precise timestamping and pacing offered by SoNIC can improve network applications, we implemented a covert timing channel, *Chupja*, and an available bandwidth estimation algorithm, *MinProbe*. Chupja is created by modulating timings of messages precisely and can deliver hundreds of thousands of bits covertly without being detected. MinProbe generates a train of probe messages with pre-defined intervals, measures received intervals, and can accurately estimate the available bandwidth.

Precise clock synchronization Achieving high precision in a clock synchronization protocol is not easy due to uncertainties from the network stack and network itself. We present a clock synchronization protocol that runs in the physical layer. DTP, Datacenter Time Protocol, is a decentralized protocol that eliminates many non-deterministic elements from the network. As a result, it provides tens of nanosecond bounded precision in a datacenter network. We demonstrate that DTP advances the state-of-the-art protocol.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 describes the scope of the problem and methodology including the details of how the physical layer works. We present SoNIC which provides precise timestamping and pacing in Chapter 3. Chapter 3 also covers two network applications that benefit from precise timestamping and pacing: Chupja in Section 3.4 and MinProbe in Section 3.5. Then, we describe DTP that provides precise clock synchronization in Chapter 4. We survey related work in Chapter 5, which is followed by

future work in Chapter 6 and conclusion in Chapter 7.

CHAPTER 2

SCOPE AND METHODOLOGY

In this chapter, we discuss the scope of the problem and the methodology we used for investigation. Our research focuses on how to improve the precision of timestamping, pacing and clock synchronization for network measurements. We use the physical layer to advance the state-of-the-art systems. As a result, we describe how the physical layer of 10 GbE performs first and then describe the scope and methodology.

2.1 Scope

The physical layer of the network stack can provide timing information necessary for precise timestamping, pacing and clock synchronization. In particular, when two physical layers are connected via a cable (we call them *peers*), each physical layer always generates a continuous bitstream that consists of either special characters or Ethernet frames to maintain the link connectivity. Further, each physical layer recovers the clock from the received bitstream generated by the peer's physical layer. In other words, two physical layers are already synchronized for reliable and robust communication. As a result, if we can control and access every single bit in the physical layer, we can achieve precise timestamping and pacing. Similarly, if we can extend the synchronization between two physical layers, we can achieve precise clock synchronization. Therefore, understanding how the physical layer operates is important for addressing research questions we investigate in this dissertation.

In this section, we first discuss how the physical layer operates in Section 2.1.1. We mainly focus on 10 Gigabit Ethernet (GbE) which our systems are based on. Other standards such as 40 and 100 GbE share many details and same techniques we use in 10 GbE can also be applied to them. In Section 2.1.2, we discuss how accessing the physical layer allows precise timestamping and pacing, and what are the challenges to do so in software in realtime. In Section 2.1.3, we describe why it is hard to precisely synchronize clocks and why implementing a protocol in the physical layer can improve the precision.

2.1.1 IEEE 802.3 standard

According to the IEEE 802.3 standard [19], the physical layer (PHY) of 10 GbE consists of three sublayers: the Physical Coding Sublayer (PCS), the Physical Medium Attachment (PMA) sublayer, and the Physical Medium Dependent (PMD) sublayer (See Figure 2.1). The PMD sublayer is responsible for transmitting the outgoing symbolstream over the physical medium and receiving the incoming symbolstream from the medium. The PMA sublayer is responsible for clock recovery and (de-)serializing the bitstream. The PCS performs the blocksync and gearbox (we call this PCS1), scramble/descramble (PCS2), and encode/decode (PCS3) operations on every Ethernet frame. The IEEE 802.3 Clause 49 explains the PCS sublayer in further detail, but we will summarize below.

When Ethernet frames are passed from the data link layer to the PHY, they are reformatted before being sent across the physical medium. On the transmit (TX) path, the PCS performs 64b/66b encoding and encodes every 64-bit

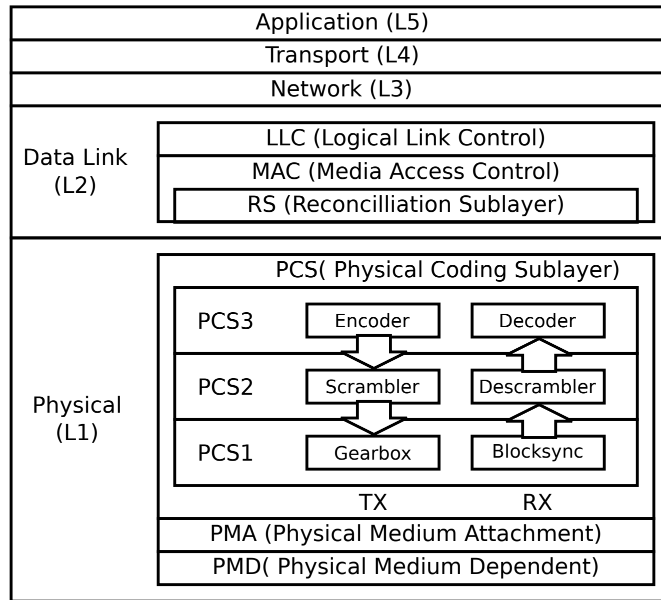


Figure 2.1: IEEE 802.3 10 Gigabit Ethernet Network stack.

of an Ethernet frame into a 66-bit *block* (PCS3), which consists of a two-bit *synchronization header* (syncheader) and a 64-bit *payload*. As a result, a 10 GbE link actually operates at 10.3125 Gbaud ($10G \times \frac{66}{64}$). Syncheaders are used for block synchronization by the remote RX path.

There are two types of *blocks*: Data blocks and control blocks. Both are 66 bits in length. The data block (/D/) is shown in the first row of Figure 2.2 and represents data characters from Ethernet frames. All other blocks in Figure 2.2 are control blocks, which contain a combination of data and control characters. Each rectangle labeled by D_i represents a 8-bit data character, and C_i represents a 7-bit control character. An idle character (/I/) is a control character that is used to fill the gap between two Ethernet frames. Given an Ethernet frame, the PCS first encodes a *Start* control block (S_0 and S_4 in Figure 2.2), followed by multiple data blocks. At the end of the frame, the PCS encodes a *Terminate* control block (T_0 to T_7) to indicate the end of the Ethernet frame. Note that one

of Start control blocks (S_4) and most of Terminate control blocks (T_0 to T_7) have one to seven control characters. These control characters are normally filled with idle characters (zeros).

There is a 66-bit Control block (`/E/`), which encodes eight 7-bit idle characters (`/I/`). As the standard requires at least twelve `/I/`s in an interpacket gap, it is *guaranteed* to have at least one `/E/` block preceding any Ethernet frame.[1] Moreover, when there is no Ethernet frame, there are always `/E/` blocks: 10 GbE is always sending at 10 Gbps and sends `/E/` blocks continuously if there are no Ethernet frames to send.

The PCS scrambles each encoded 66-bit block (PCS2) to maintain Direct Current (DC) balance[2] and adapts the 66-bit width of the block to the 16-bit width of the PMA interface (PCS1; the gearbox converts the bit width from 66- to 16-bit width.) before passing it down the network stack. The entire 66-bit block is transmitted as a continuous stream of *symbols* which a 10 GbE network transmits over a physical medium (PMA & PMD). On the receive (RX) path, the PCS performs block synchronization based on two-bit syncheaders (PCS1) and descrambles each 66-bit block (PCS2) before decoding it (PCS3).

Above the PHY is the Media Access Control (MAC) sublayer and Reconciliation Sublayer (RS). The 10 GbE MAC operates in full duplex mode; it does not handle collisions. Consequently, it only performs data encapsulation/decapsulation and media access management. Data encapsulation includes framing as well as error detection. A Cyclic Redundancy Check (CRC) is used to detect bit corruption. Media access management inserts at least 96 bits (twelve `/I/`

[1] Full-duplex Ethernet standards such as 1, 10, 40, 100 GbE send at least twelve `/I/`s (at least one `/E/`) between every Ethernet frame.

[2] DC balance ensures a mix of 1's and 0's is sent.

		sync	Block Payload								
		0 1	2 Bit Position 65								
1	/D/	0 1	D0	D1	D2	D3	D4	D5	D6	D7	
			BlockType								
2	/E/	1 0	0x1e	C0	C1	C2	C3	C4	C5	C6	C7
3	/S0/	1 0	0x78	D1	D2	D3	D4	D5	D6	D7	
4	/S4/	1 0	0x33	C0	C1	C2	C3		D5	D6	D7
5	/T0/	1 0	0x87		C1	C2	C3	C4	C5	C6	C7
6	/T1/	1 0	0x99	D0		C2	C3	C4	C5	C6	C7
7	/T2/	1 0	0xaa	D0	D1		C3	C4	C5	C6	C7
8	/T3/	1 0	0xb4	D0	D1	D2		C4	C5	C6	C7
9	/T4/	1 0	0xcc	D0	D1	D2	D3		C5	C6	C7
10	/T5/	1 0	0xd2	D0	D1	D2	D3	D4		C6	C7
11	/T6/	1 0	0xe1	D0	D1	D2	D3	D4	D5		C7
12	/T7/	1 0	0xff	D0	D1	D2	D3	D4	D5	D6	

Figure 2.2: IEEE 802.3 64b/66b block format

characters) between two Ethernet frames. The RS is responsible for additionally inserting or removing idle characters (/I/s) between Ethernet frames to maintain the link speed. The channel between the RS and the PHY is called the 10 gigabit media independent interface (XGMII) and is four bytes wide. Data is transferred at every rising and falling edge. As a result, the channel operates at 156.25 MHz (= 10Gb/32bit/2).

On the transmit path, upon receiving a layer 3 packet, the MAC prepends a preamble, start frame delimiter (SFD), and an Ethernet header to the beginning of the frame. It also pads the Ethernet payload to satisfy a minimum frame-size requirement (64 bytes), computes a CRC value, and places the value in the Frame Check Sequence (FCS) field. On the receive path, the MAC checks the CRC value and passes the Ethernet header and payload to higher layers while discarding the preamble and SFD.

2.1.2 Precise Timestamping and Pacing Need Access to PHY

Accessing the PHY provides the ability to study networks and the network stack at a heretofore inaccessible level: It can help improve the precision of network measurements by orders of magnitude [58]. A 10 GbE network uses one bit per symbol. Since a 10 GbE link operates at 10.3125 Gbaud, each and every symbol length is 97 pico-seconds wide (= $1/(10.3125 * 10^9)$). Knowing the number of bits can then translate into having a precise measure of time at the sub-nanosecond granularity. In particular, depending on the combination of data and control characters in the PCS block (Figure 2.2), the number of bits between data frames is not necessarily a multiple of eight. Therefore, on the RX path, we can tell the exact distance between Ethernet frames in bits by counting *every bit*. On the TX path, we can control time gaps by controlling the number of bits (`idle` characters) between frames.

Unfortunately, current commodity network interface cards do not provide any Application Program Interfaces (API) for accessing and controlling every bit in the PHY. One way of doing it is to use physics equipment such as an oscillator for capturing signals and a laser modulator for transmitting signals as shown in BiFocals [58]. However, BiFocals is not a realtime tool: It can only transmit pre-generated symbols and must perform extensive offline computation to recover symbols from captured signals by the oscillator. As a result, we take a different approach to access and control bits in the PHY: Implement the PHY in software.

The fundamental challenge to perform the PHY functionality in software is maintaining synchronization with hardware while efficiently using system resources. Some important areas of consideration when addressing this challenge include *hardware support, realtime capability, scalability and efficiency, and a usable*

interface.

Hardware support

The hardware must be able to transfer raw symbols from the wire to software at high speeds. This requirement can be broken down into four parts: a) Converting optical signals to digital signals (PMD), b) Clock recovery for bit detection (PMA), and c) Transferring large amounts of bits to software through a high-bandwidth interface. Additionally, d) the hardware should leave recovered bits (both control and data characters in the PHY) intact until they are transferred and consumed by the software. Commercial optical transceivers are available for a). However, hardware that simultaneously satisfies b), c) and d) is not common since it is difficult to handle 10.3125 Giga symbols in transit every second.

NetFPGA 10G [91] does not provide software access to the PHY. In particular, NetFPGA pushes not only layers 1-2 (the physical and data link layer) into hardware, but potentially layer 3 as well. Furthermore, it is not possible to easily undo this design since it uses an on-board chip to implement the PHY which prevents direct access to the PCS sublayer. As a result, we need a new hardware platform to support software access to the PHY.

Realtime Capability

Both hardware and software must be able to process 10.3125 Gigabits per second (Gbps) continuously. The IEEE 802.3 standard [19] requires the 10 GbE PHY to generate a continuous bitstream. However, synchronization between hardware and software and between multiple pipelined cores is non-trivial. The

overheads of interrupt handlers and OS schedulers can cause a discontinuous bitstream which can subsequently incur packet loss and broken links. Moreover, it is difficult to parallelize the PCS sublayer onto multiple cores. This is because the (de-)scrambler relies on state to recover bits. In particular, the (de-)scrambling of one bit relies upon the 59 bits preceding it. This fine-grained dependency makes it hard to parallelize the PCS sublayer. The key takeaway here is that everything must be efficiently pipelined and well-optimized in order to implement the PHY in software while minimizing synchronization overheads.

Scalability and Efficiency

The software must scale to process multiple 10 GbE bitstreams while efficiently utilizing resources. Intense computation is required to implement the PHY and MAC layers in software. (De-)Scrambling every bit and computing the CRC value of an Ethernet frame is especially intensive. A functional solution would require multiple duplex channels that independently perform the CRC, encode/decode, and scramble/descramble computations at 10.3125 Gbps. The building blocks for the PCS and MAC layers will therefore consume many CPU cores. In order to achieve a scalable system that can handle multiple 10 GbE bitstreams, resources such as the PCIe, memory bus, Quick Path Interconnect (QPI), cache, CPU cores, and memory must be efficiently utilized.

User Interface

Users must be able to easily access and control the PHY. Many resources from software to hardware must be tightly coupled to allow realtime access to the PHY. Thus, an interface that allows fine-grained control over them is necessary. The interface must also implement an I/O channel through which users can retrieve data such as the count of bits for precise timing information.

2.1.3 Precise Clock Synchronization Needs Access to PHY

The common mechanism of synchronizing two clocks is similar across different algorithms and protocols: A process *reads* a different process's current clock counter and computes an offset, adjusting its own clock frequency or clock counter by the offset.

In more detail, a process p sends a time request message with its current *local* clock counter (t_a in Figure 2.3) to a process q (q reads p's clock). Then, process q responds with a time response message with its local clock counter and p's original clock counter (p reads q's clock). Next, process p computes the offset between its local clock counter and the *remote clock* counter (q) and round trip time (RTT) of the messages upon receiving the response at time t_d. Finally, p adjusts its clock counter or the rate of its clock to remain close to q's clock.

In order to improve precision, q can respond with two clock counters to remove the internal delay of processing the time request message: One upon receiving the time request (t_b) and the other before sending the time response (t_c). See Figure 2.3. For example, in NTP, the process p computes RTT δ and offset θ,

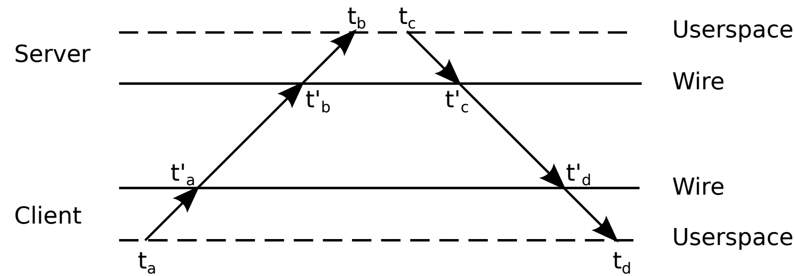


Figure 2.3: Common approach to measure offset and RTT.

as follows [98]:

$$\delta = (t_d - t_a) - (t_c - t_b)$$
$$\theta = \frac{(t_b + t_c)}{2} - \frac{(t_a + t_d)}{2}$$

Then, *p* applies these values to adjust its local clock.

Problems of Clock synchronization

Precision of a clock synchronization protocol is a function of clock skew, errors in reading remote clocks, and the interval between resynchronizations [50, 63, 75]. We discuss these factors in turn below and how they contribute to (reduced) precision in clock synchronization protocols.

Problems with Oscillator skew Many factors such as temperature and quality of an oscillator can affect oscillator skew. Unfortunately, we often do not have control over these factors to the degree necessary to prevent reduced precision. As a result, even though oscillators may have been designed with the same nominal frequency, they may actually run at slightly different rates causing clock counters to diverge over time, requiring synchronization.

Problems with Reading Remote Clocks There are many opportunities where

reading clocks can be inaccurate and lead to reduced precision. In particular, reading remote clocks can be broken down into multiple steps (enumerated below) where each step can introduce random delay errors that can affect the precision of clock synchronization.

1. Preparing a time request (reply) message
2. Transmitting a time request (reply) message
3. Packet traversing through a network
4. Receiving a time request (reply) message
5. Processing a time request (reply) message

Specifically, there are three points where precision is adversely affected: (a) Precision of timestamping affects steps 1 and 5, (b) the software network stack can introduce errors in steps 2 and 4, and (c) network jitter can contribute errors in step 3. We discuss each one further.

First, precise timestamping is not trivial. Before transmitting a message, a process timestamps the message to embed its own local counter value. Similarly, after receiving a message, a process timestamps it for further processing (i.e. computing RTT). Timestamping is often imprecise in commodity systems [81], which is a problem. It can add random delay errors which can prevent the nanosecond-level timestamping required for 10 Gigabit Ethernet (10 GbE) where minimum sized packets (64-byte) arriving at line speed can arrive every 68 nanoseconds. Improved timestamping with nanosecond resolution via new NICs are becoming more accessible [24]. However, random jitter can still be introduced due to the issues discussed below.

Second, transmitting and receiving messages involve a software network stack (e.g., between t_a and t'_a in Figure 2.3). Most clock synchronization protocols (e.g., NTP and PTP) run in a time daemon, which periodically sends and receives UDP packets between a remote process (or a time server). Unfortunately, the overhead of system calls, buffering in kernel and network interfaces, and direct memory access transactions can all contribute to errors in delay [53, 58, 81]. To minimize the impact of measurement errors, a daemon can run in kernel space or kernel bypassing can be employed. Nonetheless, non-deterministic delay errors cannot be completely removed when a protocol involves a network stack.

Third, packet propagation time can vary since it is prone to network jitter (e.g., between t'_a and t'_b or between t'_c and t'_d in Figure 2.3). Two processes are typically multiple hops away from each other and the delay between them can vary over time depending on network conditions and external traffic. Further, time requests and responses can be routed through asymmetric paths, or they may suffer different network conditions even when they are routed through symmetric paths. As a result, measured delay, which is often computed by dividing RTT by two, can be inaccurate.

Problems with Resynchronization Frequency

The more frequent resynchronizations, the more precise clocks can be synchronized to each other. However, frequent resynchronizations require increased message communication, which adds overhead to the network, especially in a datacenter network where hundreds of thousands of servers exist. The interval between resynchronizations can be configured. It is typically configured to resynchronize over a period of once per second [18], which will keep

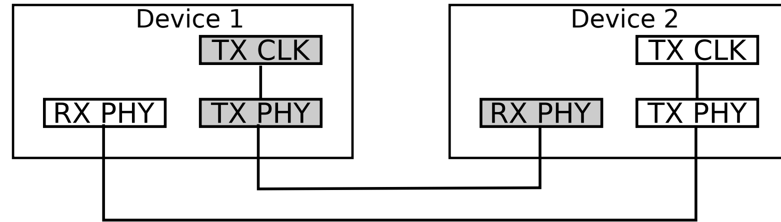


Figure 2.4: Clock domains of two peers. The same color represents the same clock domain.

network overhead low, but on the flip side, will also adversely affect precision of clock synchronization.

Why clock synchronization in the PHY?

Our goal is to achieve synchronizing clocks with nanosecond-level precision and with scalability in a datacenter network, and without *any* network overhead. We achieve this goal by running a decentralized protocol in the PHY.

We exploit the fact that two *peers*[3] are already synchronized in the PHY in order to transmit and receive bitstreams reliably and robustly. In particular, the receive path (RX) of a peer physical layer recovers the clock from the physical medium signal generated by the transmit path (TX) of the sending peer's PHY. As a result, although there are two physical clocks in two network devices, they are virtually in the same circuit (Figure 2.4).

Further, a commodity switch often uses one clock oscillator to feed the sole switching chip in a switch [7], i.e. all TX paths of a switch use the same clock source. Given a switch and N network devices connected to it, there are $N + 1$ physical oscillators to synchronize, and all of them are virtually in the same

[3]two peers are two physically connected ports via a cable.

circuit.

As delay errors from network jitter and a software network stack can be minimized by running the protocol in the lowest level of a system [116], the PHY is the best place to reduce those sources of errors. In particular, we give three reasons why clock synchronization in the PHY addresses the problems discussed previously.

First, the PHY allows precise timestamping at sub-nanosecond scale, which can provide enough fidelity for nanosecond-level precision. Timestamping [58, 81] in the PHY achieves high precision by counting the number of bits between and within packets. Timestamping in the PHY relies on the clock oscillator that generates bits in the PHY, and, as a result, it is possible to read and embed clock counters with a deterministic number of clock cycles in the PHY.

Second, a software network stack is not involved in the protocol. As the PHY is the lowest layer of a network protocol stack, there is always a deterministic delay between timestamping a packet and transmitting it. In addition, it is always possible to avoid buffering in a network device because protocol messages can always be transmitted when there is no other packet to send.

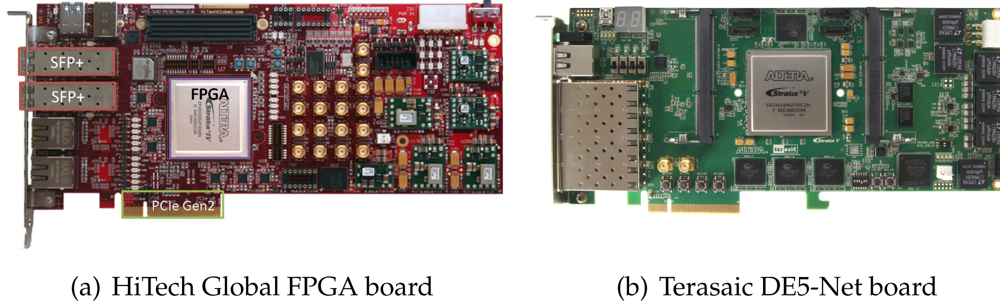
Lastly, there is little to no variation in delay between two peers in the PHY. The only element in the middle of two physically communicating devices is the wire that connects them. As a result, when there is no packet in transit, the delay in the PHY measured between two physically connected devices will be the time to transmit bits over the wire, a few clock cycles required to process bits in the PHY (which can be deterministic), and a clock domain crossing (CDC) which can add additional random delay. A CDC is necessary for passing data between

two clock domains, namely between the transmit and receive paths. Synchronization First-In-First-Out (FIFO) queues are commonly used for a CDC. In a synchronization FIFO, a signal from one clock domain goes through multiple flip-flops in order to avoid meta-stability from the other clock domain. As a result, one random delay could be added until the signal is stable to read.

Operating a clock synchronization protocol in the PHY layer not only provides the benefits of zero to little delay errors, but also zero overhead to a network: There is no need for injection of packets to implement a clock synchronization protocol. As mentioned in Section 2.1.1 a network interface continuously generates either Ethernet frames or special characters (idle characters) to maintain a link connection to its peer. If we use these idle characters to deliver protocol messages (and revert them back to idle characters), no additional packets will be required. Further, we can send protocol messages between every Ethernet frame without degrading the bandwidth of Ethernet and for different Ethernet speeds.

2.2 Methodology: Experimental Environments

We used various types of hardware and network topologies throughout this dissertation to evaluate our approach. We illustrate them in the following subsections.



(a) HiTech Global FPGA board (b) Terasaic DE5-Net board

Figure 2.5: FPGA development boards used for our research.

Name	CPU	# of cores	L3 Cache	Memory	NIC
`SoNIC`	Two Xeon X5670, 2.93GHz	6	12 MB	12 GB	HiTech with SoNIC
`ALT10G`	Two Xeon X5670, 2.93GHz	6	12 MB	12 GB	HiTech with 10 GbE
`Client`	Two Xeon X5670, 2.93GHz	6	12 MB	12 GB	Myricom dual 10G port
`Adversary`	Two Xeon X5670, 2.93GHz	6	12 MB	12 GB	Myricom dual 10G port
`Fractus`	Two Xeon E5-2690, 2.90GHz	8	20 MB	96 GB	Terasic DE5 with DTP Mellanox dual 10G port

Table 2.1: List of servers used for development and experiments

2.2.1 Hardware

For developing and evaluating SoNIC, Chupja and MinProbe, Dell Precision T7500 workstations and Dell T710 servers were used. Each machine was a dual socket, 2.93 GHz six core Xeon X5670 (Westmere [21]) with 12 MB of shared Level 3 (L3) cache and 12 GB of RAM, 6 GB connected to each of the two CPU sockets. We differentiated hardware configurations of the T7500 workstations for various purposes; a machine with an HiTech Global FPGA board [15] (Figure 2.5(a)) with SoNIC firmware (we call this `SoNIC` server), a machine with an FPGA board with an Altera 10G Ethernet design [1] (we call this `ALT10G`), and a machine with a Myricom 10G-PCIE2-8B2-2S dual 10G port NIC (we call this `Client`). Similarly, we also differentiated hardware configurations of the T710 servers for various purposes: We used one 10 GbE Dual-port NIC for receiving

Name	# of 10 GbE ports	Note
HiTech	2	Stratix IV FPGA
DE5-NET	4	Stratix V FPGA
Myricom 10G-PCIE2-8B2-2S	2	HW timestamping
Mellanox ConnectX-3	2	HW timestamping, PTP-enabled

Table 2.2: List of FPGA boards and NICs used for development and experiments

	Type	40G	10G	1G	Full bandwidth	Forwarding	Note
SW1	Core	0	8	0	160 Gbps	SF	
SW2	ToR	4	48	0	1280 Gbps	CT	
SW3	ToR	0	2	48	136 Gbps	SF	
SW4	ToR	0	2	24	105.6 Gbps	SF	
Cisco 6900	Core	0	8	0	160 Gbps	SF	
Cisco 4948	ToR	0	2	48	136 Gbps	SF	
IBM G8264	ToR	4	48	0	1280 Gbps	CT	PTP-enabled
Dell Force10	ToR	4	48	0	1280 Gbps	CT	

Table 2.3: List of evaluated network switches. “SF” is store-and-forward and “CT” is cut-through.

packets for evaluating Chupja (we call this the `adversary`) and used an HiTech Global FPGA board with SoNIC firmware for evaluating MinProbe (we also call this `SoNIC` server).

For evaluating DTP, a cluster of twelve Dell R720 servers were used (we call them `Fractus` nodes). Each server was a dual socket, 2.90 GHz eight core Xeon E5-2690 with 20 MB of shared L3 cache and 96 GB of RAM, 48 GB connected to each of the two CPU sockets. The machines had seven PCIe Gen 3.0 slots, where DTP hardware was plugged in: We used DE5-Net boards from Terasaic [9] (Figure 2.5(b). A DE5-Net board is an FPGA development board with an Altera Stratix V FPGA [5] and four Small Form-factor Pluggable (SFP+) modules. Each server also had a Mellanox ConnectX-3 MCX312A 10G NIC, which supports hardware timestamping for incoming and outgoing packets.

Table 2.3 summarizes the commercial switches that we used for evaluating our systems. SW1 was a core / aggregate router with multiple 10 GbE ports, and we installed two modules with four 10 GbE ports. SW2 was a high-bandwidth 10 GbE top-of-rack (ToR) switch which was able to support forty eight 10 GbE ports at line speed. Moreover, it was a cut-through switch whose latency of forwarding a packet was only a few microseconds. SW3 and SW4 were 1 GbE ToR switches with two 10 GbE uplinks. Other than SW2, all switches were store-and-forward switches. Further, we used Cisco 6500, 4948, IBM G8264, Dell Force10 switches for the evaluation of SoNIC, DTP, and MinProbe. IBM G8264 switch is a PTP-enabled cut-through switch.

2.2.2 Network Topologies

Simple Setup

In order to evaluate pacing and timestamping capabilities of SoNIC, we used two FPGA boards. In particular, we connected the SoNIC board and the ALT10G board directly via optic fibers (Figure 2.6(a)). For evaluating SoNIC's pacing capability, we used SoNIC to generate packets to ALT10G. ALT10G provided detailed statistics such as the number of valid/invalid Ethernet frames, and frames with CRC errors. We compared these numbers from ALT10G with statistics from SoNIC to verify the correctness of SoNIC. Similarly, for evaluating SoNIC's timestamping capability, we used ALT10G for packet generation. ALT10G allowed us to generate random packets of any length and with the minimum interpacket gap. We verified if SoNIC was able to correctly receive and timestamp all packets from ALT10G.

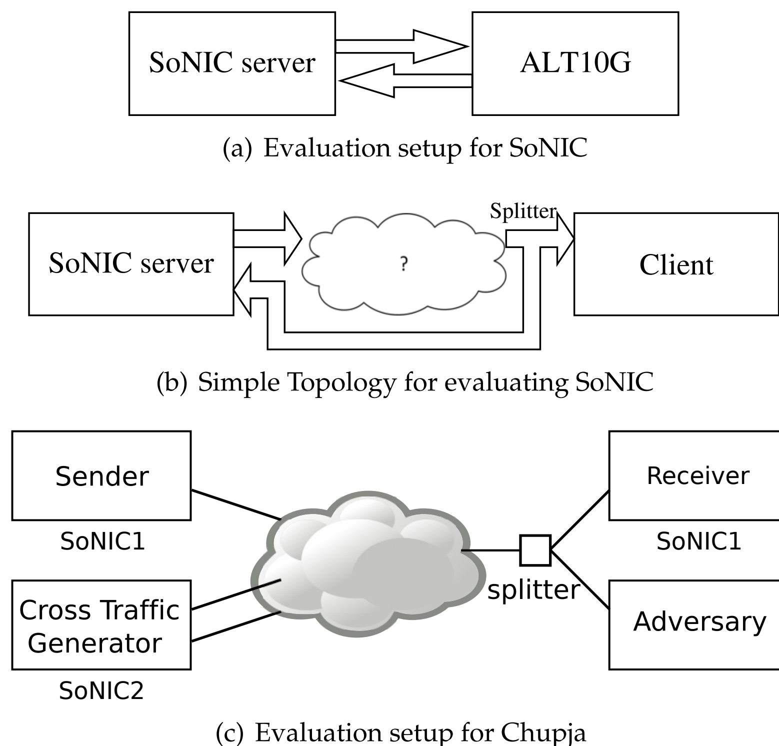


Figure 2.6: Simple evaluation setup

Further, we created a simple topology to evaluate SoNIC: We used port 0 of SoNIC server to generate packets to the Client server via an arbitrary network, and split the signal with a fiber optic splitter so that the same stream can be directed to both the Client and port 1 of the SoNIC server capturing packets (Figure 2.6(b)). We used various network topologies composed of Cisco 4948 [8] and IBM BNT G8264 [17] switches for the network between the SoNIC server and the Client.

For experiments of covert channels called Chupja in Chapter 3.4, we deployed two SoNIC servers each equipped with two 10 GbE ports to connect fiber optic cables. We used one SoNIC server (SoNIC1) to generate packets of the sender destined to a server (the adversary) via a network. We placed a fiber optic splitter at the adversary which mirrored packets to SoNIC1 for capture

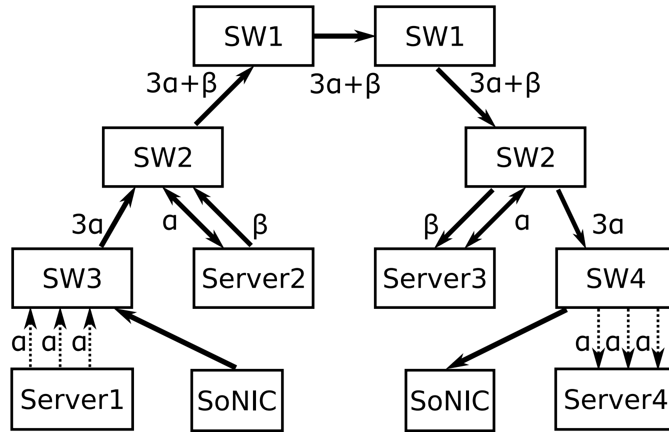


Figure 2.7: A small network. Thick solid lines are 10G connections while dotted lines are 1G connections.

(i.e. SoNIC1 was both the sender and receiver). SoNIC2 was used to generate cross traffic flows when necessary (Figure 2.6(c)). We placed none or multiple commercial switches between the sender and the adversary (the cloud within Figure 2.6(c)). A similar topology was used to evaluate available bandwidth estimation called MinProbe in Chapter 3.5. We placed two network switches between the sender and the receiver.

Small Network in a Lab

We created our own network for evaluating Chupja by connecting six switches, and four servers (See Figure 2.7). The topology resembled a typical network where core routers (SW1) were in the middle and 1 GbE ToR switches (SW3 and SW4) were leaf nodes. Then, SoNIC1 (the sender) generated packets to SW3 via one 10 GbE uplink, which forwarded packets to the receiver which was connected to SW4 via one 10 GbE uplink. Therefore, it was a seven-hop network with 0.154 ms round trip time delay on average. Then, we used four servers (Server1 to 4) to generate cross traffic. Each server had four 1 GbE and two 10 GbE ports. Server1 (Server4) was connected to SW3 (SW4) via three 1

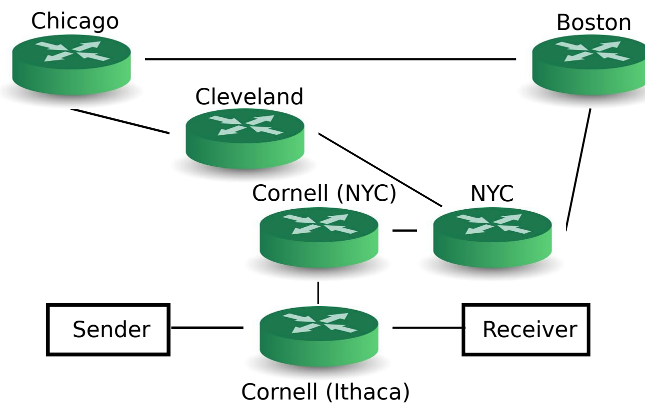


Figure 2.8: Our path on the National Lambda Rail

GbE links, and Server2 (Server3) was connected to SW3 via two 10 GbE links. These servers generated traffic across the network with Linux `pktgen` [106]. The bandwidth of cross traffic over each link between switches is illustrated in Figure 2.7: 1 GbE links were utilized with flows at α Gbps and 10 GbE links at β Gbps.

National Lambda Rail

National Lambda Rail (NLR) was a wide-area network designed for research and had significant cross traffic [26]. We set up a path from Cornell university to NLR over nine routing hops and 2500 miles one-way (Figure 2.8). All the routers in NLR were Cisco 6500 routers. The average round trip time of the path was 67.6 ms and there was always cross traffic. In particular, many links on our path were utilized with 1~4 Gbps cross traffic during the experiment. Cross traffic was not under our control, however we received regular measurements of traffic on external interfaces of all routers.

Fractus Cluster

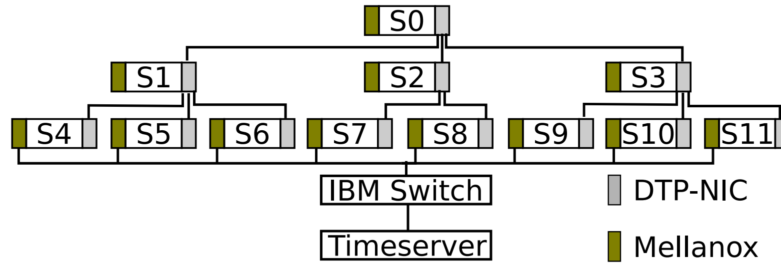


Figure 2.9: Evaluation Setup for DTP

We created a DTP network as shown in Figure 2.9: A tree topology with the height of two, i.e. the maximum number of hops between any two leaf servers was four. DE5-Net boards of the root node, $S0$, and intermediate nodes, $S1 \sim S3$, were configured as DTP switches, and those of the leaves ($S4 \sim S11$) were configured as DTP NICs. We used 10-meter Cisco copper twinax cables to a DE5-Net board's SFP+ modules.

We also created a PTP network with the same servers as shown in Figure 2.9 (PTP used Mellanox NICs). A VelaSync timeserver from Spectracom was deployed as a PTP grandmaster clock. An IBM G8264 cut-through switch was used to connect the servers including the timeserver. As a result, the number of hops between any two servers in the PTP network was always two. Cut-through switches are known to work well in PTP networks [132]. We deployed a commercial PTP solution (Timekeeper [31]) in order to achieve the best precision in 10 GbE.

The timeserver multicasted PTP timing information every second (`sync` message), i.e. the synchronization rate was once per second, which was the recommended sync rate by the provider. Further, we enabled PTP `UNICAST` capability, which allowed the server to send unicast `sync` messages to individual PTP clients once per second in addition to multicast `sync` messages. In our

configuration, a client sent two `Delay_Req` messages per 1.5 seconds.

The PTP network was mostly idle except when we introduced network congestion. Since PTP used UDP datagrams for time synchronization, the precision of PTP could vary relying on network workloads. As a result, we introduced network workloads between servers using `iperf` [22]. Each server occasionally generated MTU-sized UDP packets destined for other servers so that PTP messages could be dropped or arbitrarily delayed.

2.3 Summary

In this chapter, we discussed the challenges of accessing the PHY for precise timestamping and pacing and the challenges of precisely synchronizing clocks. Accessing the PHY in software requires transmitting raw bits from the wire to software (and vice versa) in realtime. Careful design of hardware and software is essential in achieving the goal. Precisely synchronizing clocks is generally difficult due to errors from measuring round trip time. RTTs can be inaccurate because of imprecise timestamping, network stack overhead, and network jitter. Implementing the protocol in the PHY can eliminate non-deterministic errors from measuring round trip time. We also described the servers and switches we used for evaluating our systems and network topologies we created for evaluations in this chapter.

CHAPTER 3

TOWARDS PRECISE TIMESTAMPING AND PACING: SONIC

Precise timestamping and pacing are fundamental for network measurements and many network applications, such as available bandwidth estimation algorithms [87, 88, 120], network traffic characterization [69, 82, 127], and creation and detection of covert timing channels [44, 89, 90]. As a result, in order to improve the performance of network applications, it is important to improve the precision of timestamping and pacing messages. In this chapter, we demonstrate that the precision of timestamping and pacing can significantly improve via access to the physical layer (PHY). We also demonstrate that the precision provided by access to the PHY can improve the performance of network applications such as covert timinig channels and available bandwidth estimation.

Accessing the PHY is not easy, mainly because NICs do not provide any APIs. As a result, we present a new approach for accessing the PHY from *software*. In particular, we present SoNIC, Software-defined Network Interface Card which implemented the PHY in software and as a result improved the precision of timestamping and pacing messages. In essence, in SoNIC, all of the functionality in the PHY that manipulate bits were implemented in software. SoNIC consisted of commodity off-the-shelf multi-core processors and a field-programmable gate array (FPGA) development board with peripheral component interconnect express (PCIe) Gen 2.0 bus. High-bandwidth PCIe interfaces and powerful FPGAs could support full bidirectional data transfer for two 10 GbE ports. Further, we created and implemented optimized techniques to achieve not only high-performance packet processing, but also high-performance 10 GbE bitstream control in software. Parallelism and optimiza-

tions allowed SoNIC to process multiple 10 GbE bitstreams at line-speed.

With software access to the PHY, SoNIC provided two important capabilities for network research applications: Precise packet pacing and precise packet timestamps. First, as a powerful network measurement tool, SoNIC could generate packets at full data rate with minimal interpacket delay. It also provided fine-grained control over interpacket delays; it could inject packets with no variance in interpacket delays. Second, SoNIC accurately captured incoming packets at any data rate including the maximum, while simultaneously timestamping each packet with sub-nanosecond granularity. In other words, SoNIC could capture exactly what was sent. Further, this precise timestamping and pacing could improve the accuracy of research based on interpacket delay. For example, SoNIC could be used to profile network components. It created timing channels that were undetectable from software application and accurately estimated available bandwidth between two endhosts.

In this chapter, we make four contributions:

- We provide precise timestamping and pacing via a new approach enabling software access to the PHY.
- We designed SoNIC with commodity components such as multi-core processors and a PCIe pluggable board and present a prototype of SoNIC.
- We demonstrate that SoNIC could enable flexible, precise, and realtime network research applications. SoNIC increased the flexibility of packet pacing and the precision of packet timestamping.
- We also demonstrate that network research studies based on interpacket delay such as covert timing channels and estimating available bandwidth

could be significantly improved via precise pacing and timestamping offered by SoNIC.

3.1 Design

The design goals of SoNIC were to provide 1) access to the PHY in software, 2) realtime capability, 3) scalability and efficiency, 4) precision, and 5) user interface. As a result, SoNIC must allow users realtime access to the PHY in software, provide an interface to applications, process incoming packets at line-speed, and be scalable. Our ultimate goal was to achieve the same flexibility and control of the entire network stack for a wired network, as a software-defined radio [122] did for a wireless network, while maintaining the same level of precision as BiFocals [58]. Access to the PHY could then enhance the accuracy of timestamping and pacing in network research based on interpacket delay. In this section, we discuss the design of SoNIC and how it addressed the challenges presented in Section 2.1.2.

3.1.1 Access to the PHY in software

An application must be able to access the PHY in software. Thus, our solution must implement the bit generation and manipulation functionality of the PHY in software. The transmission and reception of bits could be handled by hardware. We carefully examined the PHY to determine an optimal partitioning of functionality between hardware and software.

As discussed in Section 2.1.1 and shown in 2.1, the PMD and PMA sublay-

ers of the PHY do not modify any bits or change the clock rate. They simply forward the symbolstream/bitstream to other layers. Similarly, PCS1 only converts the bit width (gearbox) or identifies the beginning of a new 64/66 bit block (blocksync). Therefore, the PMD, PMA, and PCS1 were all implemented in hardware as a forwarding module between the physical medium and SoNIC's software component (See Figure 2.1). Conversely, PCS2 (scramble/descramble) and PCS3 (encode/decode) actually manipulate bits in the bitstream and so they were implemented in SoNIC's software component. SoNIC provided full access to the PHY in software; as a result, all of the functionality in the PHY that manipulate bits (PCS2 and PCS3) were implemented in software.

For this partitioning between hardware and software, we chose an Altera Stratix IV FPGA [4] development board from HiTechGlobal [15] as our hardware platform. The board included a PCIe Gen 2 interface (=32 Gbps) to the host PC and was equipped with two SFP+ (Small Form-factor Pluggable) ports (Figure 2.5(a)). The FPGA was equipped with 11.3 Gbps transceivers which could perform the 10 GbE PMA at line-speed. Once symbols were delivered to a transceiver on the FPGA they were converted to bits (PMA) and then transmitted to the host via PCIe by direct memory access (DMA). This board satisfied all the requirements discussed in the previous Section 2.1.2.

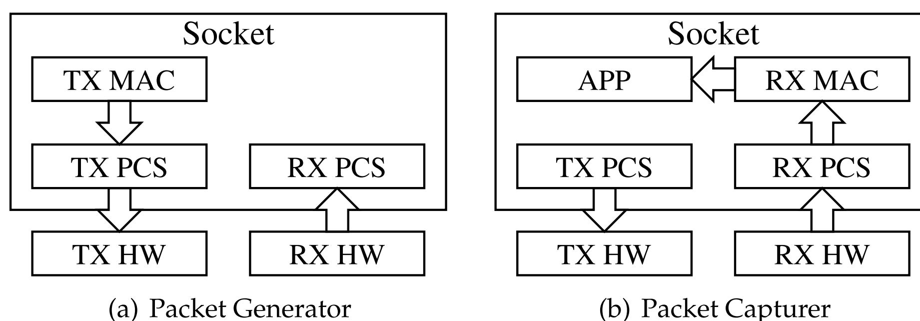
3.1.2 Realtime Capability

To achieve realtime, it was important to reduce any synchronization overheads between hardware and software, and between multiple pipelined cores. In SoNIC, the hardware did not generate interrupts when receiving or transmit-

ting. Instead, the software decided when to initiate a DMA transaction by *polling* a value from a shared data memory structure where only the hardware wrote. This approach was called *pointer polling* and was better than interrupts because there was always data to transfer due to the nature of continuous bitstreams in 10 GbE.

In order to synchronize multiple pipelined cores, a *chasing-pointer* FIFO from Sora [122] was used which supported low-latency pipelining. The chasing-pointer FIFO removed the need for a shared synchronization variable and instead used an additional flag for each entry to indicate whether a FIFO entry was available to reduce the synchronization overheads. In our implementation, we improved the FIFO by avoiding memory operations as well. Memory allocation and page faults were expensive and must have been avoided to meet the realtime capability. Therefore, each FIFO entry in SoNIC was pre-allocated during initialization. In addition, the number of entries in a FIFO was kept small so that the amount of memory required for a port could fit into the shared L3 cache.

We used the Intel Westmere processor to achieve high performance. Intel Westmere is a Non-Uniform Memory Access (NUMA) architecture that was efficient for implementing packet processing applications [54, 65, 95, 112]. It was further enhanced by a instruction `PCLMULQDQ` that performs carry-less multiplication. We used the instruction to implement a fast Cyclic Redundancy Check (CRC) algorithm [61] that the Media Access Control (MAC) requires, making it possible to implement a CRC engine that could process 10 GbE bits at line-speed on a single core.



(a) Packet Generator (b) Packet Capturer

Figure 3.1: Example usages of SoNIC

3.1.3 Scalability and Efficiency

The FPGA board we used was equipped with two physical 10 GbE ports and a PCIe interface that could support up to 32 Gbps. Our design goal was to support two physical ports per board. Consequently, the number of CPU cores and the amount of memory required for one port must be bounded. Further, considering the intense computation required for the PCS and MAC and that processors came with four to six or even eight cores per socket, our goal was to limit the number of CPU cores required per port to the number of cores available in a socket. As a result, for one port we implemented four dedicated kernel threads each running on different CPU cores. We used a PCS thread and a MAC thread on both the transmit and receive paths. We called our threads: TX PCS, RX PCS, TX MAC and RX MAC. Interrupt requests (IRQ) were re-routed to unused cores so that SoNIC threads did not give up the CPU and could meet the realtime requirements.

Additionally, we were careful to use memory efficiently: DMA buffers were preallocated and reused and data structures were kept small to fit in the shared L3 cache. Further, by utilizing memory efficiently, dedicating threads to cores, and using multi-processor QPI support, we could linearly increase the number

of ports with the number of processors. QPI provides enough bandwidth to transfer data between sockets at a very fast data rate (> 100 Gbps).

A significant design issue still remained: Communication and CPU core utilization. The way we pipelined CPUs, i.e. sharing FIFOs depended on the application. In particular, we pipelined CPUs differently depending on the application to reduce the number of active CPUs; unnecessary CPUs were returned to OS. Further, we enhanced communication with a general rule of thumb: Taking advantage of the NUMA architecture and L3 cache and placing closely related threads on the same CPU socket.

Figure 3.1 illustrates examples of how to share FIFOs among CPUs. An arrow was a shared FIFO. For example, a packet generator only required TX elements (Figure 3.1(a)); RX PCS simply received and discarded bitstreams, which was required to keep a link active. On the contrary, a packet capturer required RX elements (Figure 3.1(b)) to receive and capture packets. TX PCS was required to establish and maintain a link to the other end by sending `/I/`s. To create a network profiling application, both the packet generator and packet capturer could run on different sockets simultaneously.

3.1.4 Precision

As discussed in Section 3.1.1, the PCS2 and PCS3 shown in Figure 2.1 were implemented in software. Consequently, the software received the entire raw bitstream from the hardware. While performing PCS2 and PCS3 functionalities, a PCS thread recorded the number of bits in between and within each Ethernet frame. This information could later be retrieved by a user application. More-

```
1: #include "sonic.h"
2:
3: struct sonic_pkt_gen_info info = {
4:     .pkt_num   = 1000000000UL,
5:     .pkt_len   = 1518,
6:     .mac_src   = "00:11:22:33:44:55",
7:     .mac_dst   = "aa:bb:cc:dd:ee:ff",
8:     .ip_src    = "192.168.0.1",
9:     .ip_dst    = "192.168.0.2",
10:    .port_src  = 5000,
11:    .port_dst  = 5000,
12:    .idle   = 12, };
13:
14: fd1 = open(SONIC_CONTROL_PATH, O_RDWR);
15: fd2 = open(SONIC_PORT1_PATH, O_RDONLY);
16:
17: ioctl(fd1, SONIC_IOC_RESET)
18: ioctl(fd1, SONIC_IOC_SET_MODE, SONIC_PKT_GEN_CAP)
19: ioctl(fd1, SONIC_IOC_PORT0_INFO_SET, &info)
20: ioctl(fd1, SONIC_IOC_RUN, 10)
21:
22: while ((ret = read(fd2, buf, 65536)) > 0) {
23:  // process data }
24:
25: close(fd1);
26: close(fd2);
```

Figure 3.2: Example `C` code of SoNIC Packet Generator and Capturer

over, SoNIC allowed users to precisely control the number of bits in between frames when transmitting packets, and could even change the value of any bits. For example, we used this capability to give users fine-grained control over packet generators and could create virtually undetectable covert channels.

3.1.5 User Interface

SoNIC exposed fine-grained control over the path that a bitstream travels in software. SoNIC used the `ioctl` system call for control and the character de-

vice interface to transfer information when a user application needed to retrieve data. Moreover, users could assign which CPU cores or socket each thread run on to optimize the path.

To allow further flexibility, SoNIC allowed additional application-specific threads, called APP threads, to be pipelined with other threads. A character device was used to communicate with these APP threads from userspace. For instance, users could implement a logging thread pipelined with receive path threads (RX PCS and/or RX MAC). Then the APP thread delivered packet information along with precise timing information to userspace via a character device interface. There were two constraints that an APP thread must always meet: Performance and pipelining. First, whatever functionality was implemented in an APP thread, it must be able to perform it faster than 10.3125 Gbps for any given packet stream in order to meet the realtime capability. Second, an APP thread must be properly pipelined with other threads, i.e. input/output FIFO must be properly set. SoNIC supported one APP thread per port.

Figure 3.2 illustrates the source code of an example use of SoNIC as a packet generator and capturer that exhibit precise pacing and timestamping capabilities. After `SONIC_IOC_SET_MODE` was called (line 18), threads were pipelined as illustrated in Figure 3.1(a) and 3.1(b). After `SONIC_IOC_RUN` command (line 20), port 0 started generating packets given the information from `info` (line 3-12) for 10 seconds (line 20) while port 1 started capturing packets with very precise timing information. Captured information was retrieved with `read` system calls (line 22-23) via a character device. As a packet generator, users could set the desired number of `/I/`s between packets (line 12) for precise pacing. For example, twelve `/I/` characters achieved the maximum data rate. Increasing

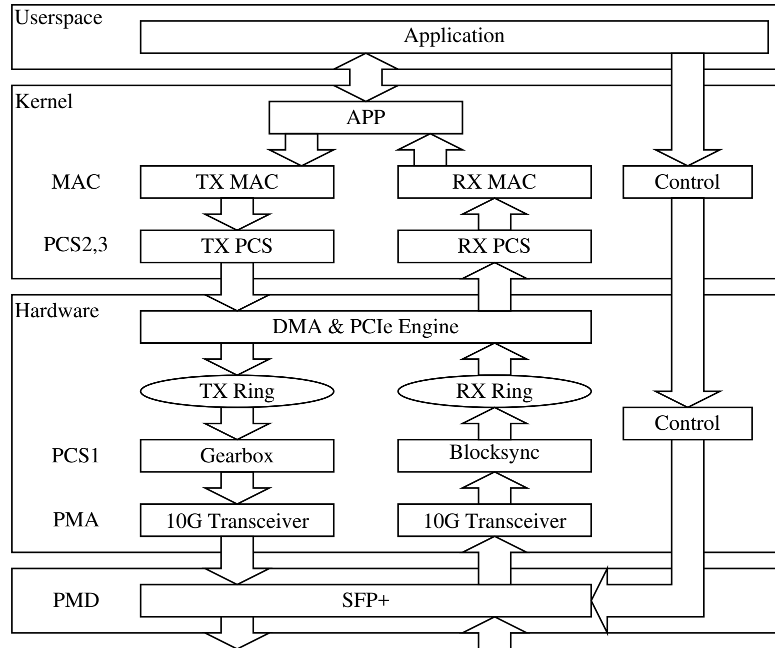


Figure 3.3: SoNIC architecture

the number of `/I/` characters decreased the data rate.

3.1.6 Discussion

We have implemented SoNIC to achieve the design goals described above, namely, software access to the PHY, realtime capability, scalability, high precision, and an interactive user interface that enable precise timestamping and pacing. Figure 3.3 shows the major components of our implementation. From top to bottom, user applications, software as a loadable Linux kernel module, hardware as a firmware in FPGA, and a SFP+ optical transceiver. Although Figure 3.3 only illustrates one physical port, there were two physical ports available in SoNIC. SoNIC software consisted of about 6k lines of kernel module code, and SoNIC hardware consisted of 6k lines of Verilog code excluding auto-

generated source code by Altera Quartus [3] with which we developed SoNIC's hardware modules.

The idea of accessing the PHY in software can be applied to other physical layers with different speeds. The 1 GbE and 40 GbE PHYs are similar to the 10 GbE PHY in that they run in full duplex mode and maintain continuous bitstreams. Especially, the 40 GbE PCS employs four PCS lanes that implements 64B/66B encoding as in the 10 GbE PHY. Therefore, it is possible to access the PHYs of them with appropriate clock cycles and hardware supports. However, it might not be possible to implement four times faster scrambler with current CPUs.

3.2 Implementation

Performance was paramount for SoNIC to achieve its goals and to allow software access to the entire network stack. In this section we discuss the software (Section 3.2.1) and hardware (Section 3.2.2) optimizations that we employed to enable SoNIC. Further, we evaluate each optimization (Sections 3.2.1 and 3.2.2) and demonstrate that they helped to enable SoNIC and network research applications (Section 3.3) with high performance.

3.2.1 Software Optimizations

MAC Thread Optimizations As stated in Section 3.1.2, we used `PCLMULQDQ` instruction which performs carry-less multiplication of two 64-bit quadwords [62] to implement the fast CRC algorithm [61]. The algorithm *folds* a large chunk of

data into a smaller chunk using the `PCLMULQDQ` instruction to efficiently reduce the size of data. We adapted this algorithm and implemented it using inline assembly with optimizations for small packets. See Appendix B for an implementation of the fast CRC algorithm using the `PCLMULQDQ` instruction.

PCS Thread Optimizations Considering there are 156 million 66-bit blocks a second, the PCS must process each block in less than 6.4 nanoseconds. Our optimized (de-)scrambler processed each block in 3.06 nanoseconds which even gave enough time to implement decode/encode and DMA transactions within a single thread.

In particular, the PCS thread needed to implement the (de-)scrambler function, $G(x) = 1 + x^{39} + x^{58}$, to ensure that a mix of 1's and 0's were always sent (DC balance). The (de-)scrambler function could be implemented with Algorithm 1, which was very computationally expensive [58] taking 320 shift and 128 xor operations (5 shift operations and 2 xors per iteration times 64 iterations). In fact, our original implementation of Algorithm 1 performed at 436 Mbps, which was not sufficient and became the bottleneck for the PCS thread. We optimized and reduced the scrambler algorithm to a *total* of 4 shift and 4 xor operations (Algorithm 2) by carefully examining how hardware implements the scrambler function [124]. Both Algorithm 1 and 2 are equivalent, but Algorithm 2 ran 50 times faster (around 21 Gbps). See Appendix C for a more detailed explanation of Algorithm 2.

Memory Optimizations We used *packing* to further improve performance. Instead of maintaining an array of data structures that each contained metadata and a pointer to the packet payload, we packed as much data as possible into a preallocated memory space: Each packet structure contained metadata, packet

Algorithm 1: Scrambler

$s \leftarrow \text{state}$

$d \leftarrow \text{data}$

for $i = 0 \rightarrow 63$ **do**

  $in \leftarrow (d >> i) \,\&\, 1$

  $out \leftarrow (in \oplus (s >> 38) \oplus (s >> 57)) \,\&\, 1$

  $s \leftarrow (s << 1) \mid out$

  $r \leftarrow r \mid (out << i)$

  $\text{state} \leftarrow s$

end for

Algorithm 2: Parallel Scrambler

$s \leftarrow \text{state}$

$d \leftarrow \text{data}$

$r \leftarrow (s >> 6) \oplus (s >> 25) \oplus d$

$r \leftarrow r \oplus (r << 39) \oplus (r << 58)$

$\text{state} \leftarrow r$

payload, and an offset to the next packet structure in the buffer. This packing helped to reduce the number of page faults, and allowed SoNIC to process small packets faster. Further, to reap the benefits of the `PCLMULQDQ` instruction, the first byte of each packet was always 16-byte aligned.

Evaluation We evaluated the performance of the TX MAC thread when computing CRC values to assess the performance of the fast CRC algorithm and packing packets we implemented relative to batching an array of packets. For comparison, we computed the theoretical maximum throughput (Reference throughput) in packets per second (pps) for any given packet length (i.e. the

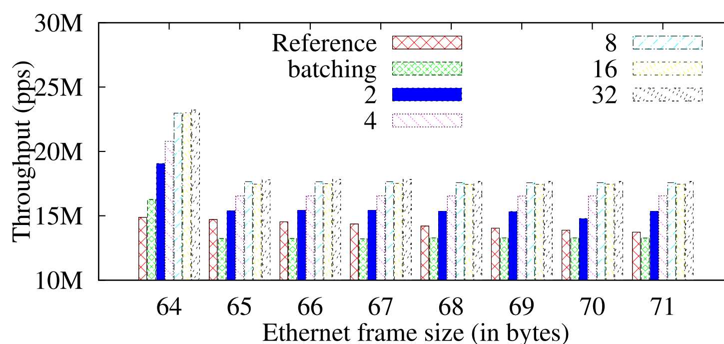


Figure 3.4: Throughput of packing

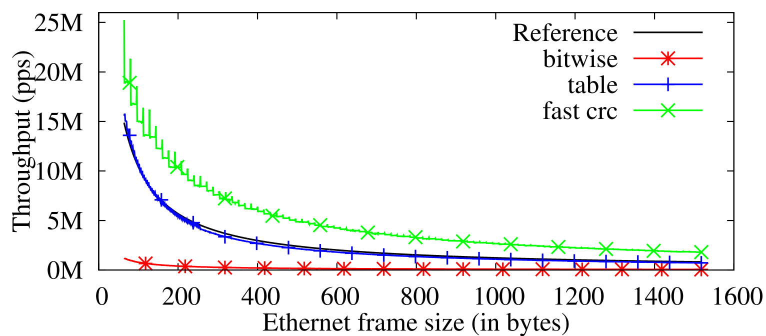


Figure 3.5: Throughput of different CRC algorithms

pps necessary to achieve the maximum throughput of 10 Gbps less any protocol overhead).

If only one packet was packed in the buffer, packing would perform the same as batching since the two were essentially the same in this case. We doubled the factor of packing from 1 to 32 and assessed the performance of packing each time, i.e. we doubled the number of packets written to a single buffer. Figure 3.4 shows that packing by a factor of 2 or more always outperformed the Reference throughput and was able to achieve the max throughput for small packets while batching did not.

Next, we compared our fast CRC algorithm against two CRC algorithms that

the Linux Kernel provided. One of the Linux CRC algorithms was a naive bit computation and the other was a table lookup algorithm. Figure 3.5 illustrates the results of our comparisons. The x-axis is the length of packets tested while the y-axis is the throughput. The Reference line represents the maximum possible throughput given the 10 GbE standard. Packet lengths range the spectrum of sizes allowed by 10 GbE standard from 64 bytes to 1518 bytes. For this evaluations, we allocated 16 pages *packed* with packets of the same length and computed CRC values with different algorithms for 1 second. As we can see from Figure 3.5, the throughput of the table lookup closely followed the Reference line; however, for several packet lengths, it underperformed the Reference line and was unable to achieve the maximum throughput. The fast CRC algorithm, on the other hand, outperformed the Reference line and target throughput for all packet sizes.

Lastly, we evaluated the performance of pipelining and using multiple threads on the TX and RX paths. We tested a full path of SoNIC to assess the performance as packets travel from the TX MAC to the TX PCS for transmission and up the reverse path for receiving from the RX PCS to the RX MAC and to the APP (as a logging thread). All threads performed better than the Reference target throughput. The overhead of FIFO was negligible when we compared the throughputs of individual threads to the throughput when all threads were pipelined together. Moreover, when using two ports simultaneously (two full instances of receive and transmit SoNIC paths), the throughput for both ports achieved the Reference target maximum throughput.

3.2.2 Hardware Optimizations

DMA Controller Optimizations Given our desire to transfer large amounts of data (more than 20 Gbps) over the PCIe, we implemented a high performance DMA controller. There were two key factors that influenced our design of the DMA controller. First, because the incoming bitstream was a continuous 10.3125 Gbps, there must be enough buffering inside the FPGA to compensate for a transfer latency. Our implementation allocated four rings in the FPGA for two ports (Figure 3.3 shows two of the rings for one port). The maximum size of each ring was 256 KB with the size being limited by the amount of SRAM available on hardware.

The second key factor we needed to consider was the efficient utilization of bus bandwidth. The DMA controller operated at a data width of 128 bits. If we send a 66-bit data block over the 128-bit bus every clock cycle, we would waste 49% of the bandwidth, which was not acceptable. To achieve more efficient use of the bus, we created a `sonic_dma_page` data structure and separated the syncheader from the packet payload before storing a 66-bit block in the data structure. Sixteen two-bit syncheaders were concatenated together to create a 32-bit integer and stored in the `syncheaders` field of the data structure. The 64-bit packet payloads associated with these syncheaders were stored in the `payloads` field of the data structure. For example, the *i*-th 66-bit PCS block from a DMA page consisted of the two-bit sync header from `syncheaders[`$i/16$`]` and the 64-bit payload from `payloads[`i`]`. With this data structure there was a 32-bit overhead for every page, however it did not impact the overall performance.

PCI Express Engine Optimizations When SoNIC was first designed, it only

supported a single port. As we scaled SoNIC to support multiple ports simultaneously, the need for multiplexing traffic among ports over the single PCIe link became a significant issue. To solve this issue, we employed a two-level arbitration scheme to provide fair arbitration among ports. A lower level arbiter was a fixed-priority arbiter that worked within a single port and arbitrated between four basic Transaction Level Packet (TLP) types: Memory, I/O, configuration, and message. The TLPs were assigned with fixed priority in favor of the write transaction towards the host. The second level arbiter implemented a virtual channel, where the Traffic Class (TC) field of TLP's were used as demultiplexing keys. We implemented our own virtual channel mechanism in SoNIC instead of using the one available in the PCIe stack since virtual channel support was an optional feature for vendors to comply with. In fact, most chipsets on the market at the time of this dissertation did not support the virtual channel mechanism. By implementing the virtual channel support in SoNIC, we achieved better portability since we did not rely on chip vendors that enabled PCI arbitration.

Evaluation

We examined the maximum throughput for DMA between SoNIC hardware and SoNIC software to evaluate our hardware optimizations. It was important that the bidirectional data rate of each port of SoNIC was greater than 10.3125 Gbps. For this evaluation, we created a DMA descriptor table with one entry, and changed the size of memory for each DMA transaction from one page (4K) to sixteen pages (64KB), doubling the number of pages each time. We evaluated the throughput of a single RX or TX transaction, dual RX or TX transactions, and full bidirectional RX and TX transactions with both one and two ports (see

Configuration	Same Socket?	# pages	Throughput (RX)		# pages	Throughput (TX)		Realtime?
			Port 0	Port 1		Port 0	Port 1	
Single RX		16	25.7851					
Dual RX	Yes	16	13.9339	13.899				
	No	8	14.2215	13.134				
Single TX					16	23.7437		
Dual TX	Yes				16	14.0082	14.048	
	No				16	13.8211	13.8389	
Single RX/TX		16	21.0448		16	22.8166		
Dual RX/TX	Yes	4	10.7486	10.8011	8	10.6344	10.7171	No
		4	11.2392	11.2381	16	12.384	12.408	Yes
		8	13.9144	13.9483	8	9.1895	9.1439	Yes
		8	14.1109	14.1107	16	10.6715	10.6731	Yes
	No	4	10.5976	10.183	8	10.3703	10.1866	No
		4	10.9155	10.231	16	12.1131	11.7583	Yes
		8	13.4345	13.1123	8	8.3939	8.8432	Yes
		8	13.4781	13.3387	16	9.6137	10.952	Yes

Table 3.1: DMA throughput. The numbers are average over eight runs. The delta in measurements was within 1% or less.

the rows of Table 3.1). We also measured the throughput when traffic was sent to one or two CPU sockets.

Table 3.1 shows the DMA throughputs of the transactions described above. We first measured the DMA without using pointer polling (see Section 3.1.2) to obtain the maximum throughputs of the DMA module. For single RX and TX transactions, the maximum throughput was close to 25 Gbps. This was less than the theoretical maximum throughput of 29.6 Gbps for the x8 PCIe interface, but closely matched the reported maximum throughput of 27.5 Gbps [2] from Altera design. Dual RX or TX transactions also resulted in throughputs similar to the reference throughputs of Altera design.

Next, we measured the full bidirectional DMA transactions for both ports varying the number of pages again. As shown in the bottom half of Table 3.1, we had multiple configurations that supported throughputs greater than 10.3125 Gbps for full bidirections. However, there were a few configurations in which the TX throughput was less than 10.3125 Gbps. That was because the TX di-

rection required a small fraction of RX bandwidth to fetch the DMA descriptor. If RX ran at maximum throughput, there was little room for the TX descriptor request to get through. However, as the last column on the right indicates these configurations were still able to support the realtime capability, i.e. consistently running at 10.3125 Gbps, when *pointer polling* was enabled. This was because the RX direction only needed to run at 10.3125 Gbps, less than the theoretical maximum throughput (14.8 Gbps), and thus gave more room to TX. On the other hand, two configurations where both RX and TX ran faster than 10.3125 Gbps for full bidirection were not able to support the realtime capability. For the rest of the chapter, we used 8 pages for RX DMA and 16 pages for TX DMA.

3.3 Evaluation

How can SoNIC enable flexible, precise and novel network research applications? Specifically, what unique value does software access to the PHY buy? SoNIC can literally count the number of bits between and within packets, which can be used for timestamping at the sub-nanosecond granularity (again each bit is 97 ps wide, or about ~0.1 ns). At the same time, access to the PHY allows users control over the number of `idles` (`/I/`s) between packets when generating packets. This fine-grained control over the `/I/`s means we can precisely control the data rate and the distribution of interpacket gaps. For example, the data rate of a 64B packet stream with uniform 168 `/I/`s is 3 Gbps. When this precise packet generation is combined with exact packet capture, also enabled by SoNIC, we can improve the accuracy of any research based on interpacket delays [44, 69, 82, 87, 88, 89, 90, 120, 127, 129].

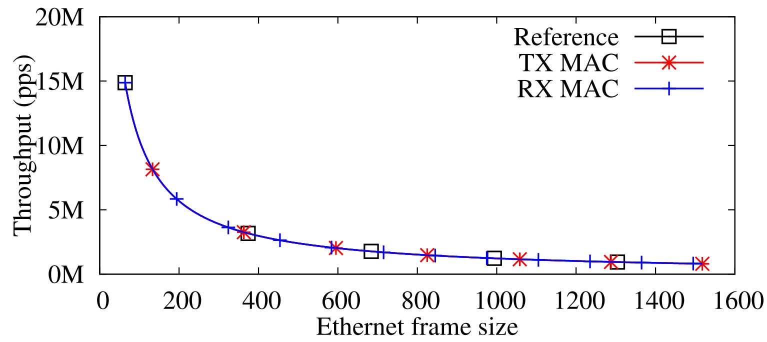


Figure 3.6: Throughput of packet generator and capturer

In this section, we demonstrate SoNIC's accurate packet generation (packet pacing) capability in Section 3.3.1 and packet capture capability in Section 3.3.2, which were unique contributions and could enable unique network research in and of themselves given both the flexibility, control, and precision. Further, we demonstrate that SoNIC could precisely and flexibly characterize and profile commodity network components like routers, switches, and NICs. Section 3.3.3 discusses the profiling capability enabled by SoNIC.

We define *interpacket delay* (IPD) as the time difference between the first bits of two successive packets, and *interpacket gap* (IPG) as the time difference between the last bit of the first packet and the first bit of the next packet. Thus, an interpacket delay between two packets is equal to the sum of transmission time of the first packet and the interpacket gap between the two (i.e. IPD = IPG + packet size). A *homogeneous* packet stream consists of packets that have the same destination, the same size and the same IPGs (IPDs) between them. Furthermore, the variance of IPGs and IPDs of a homogeneous packet stream is always zero.

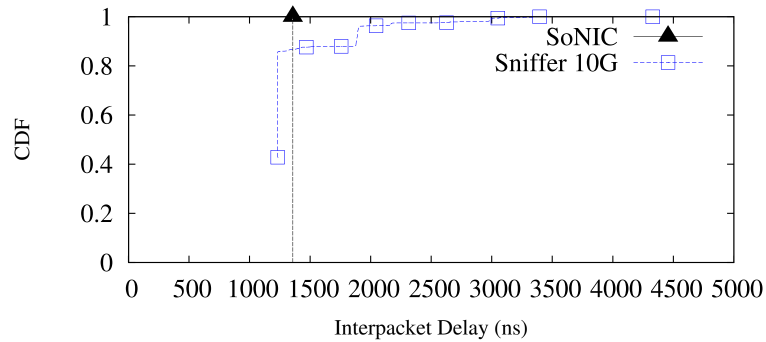


Figure 3.7: Comparison of packet generation at 9 Gbps

3.3.1 Packet Generator (Packet Pacing)

Packet generation is important for network research. It can stress test end-hosts, switches/routers, or a network itself. Moreover, packet generation can be used for replaying a trace, studying distributed denial of service (DDoS) attacks, or probing firewalls.

In order to claim that a packet generator is accurate, packets need to be crafted with fine-grained precision (minimum deviations in IPD) at the maximum data rate. However, this fine-grained control was not usually exposed to users. Further, commodity servers equipped with a commodity NIC often did not handle small packets efficiently and require batching [54, 65, 95, 112]. Thus, the sending capability of servers/software-routers were determined by the network interface devices. Myricom Sniffer 10G [25] provided line-rate packet injection capability, but did not provide fine-grained control of interpacket gaps (IPG). Hardware based packet generators such as ALT10G did not provide any interface for users to flexibly control IPGs.

We evaluated SoNIC as a packet generator (Figure 2.6(a) and 3.1(a)). Figure 3.7 compares the performance of SoNIC to that of Sniffer 10G. Note, we did

not include ALT10G in this evaluation since we could not generate packets at 9 Gbps. We used two servers with Sniffer 10G enabled devices to generate 1518B packets at 9 Gbps between them. We split the stream so that SoNIC could capture the packet stream in the middle (we describe this capture capability in the following section). As the graph shows, Sniffer 10G allowed users to generate packets at desired data rate, however, it did not give the control over the IPD; that is, 85.65% packets were sent in a burst (instantaneous 9.8 Gbps and minimum IPG (14 `/I/`s)). SoNIC, on the other hand, could generate packets with uniform distribution. In particular, SoNIC generated packets with no variance for the IPD (i.e. a single point on the CDF, represented as a triangle). Moreover, the maximum throughput perfectly matched the Reference throughput (Figure 3.6) while the TX PCS consistently ran at 10.3125 Gbps (which is not shown). In addition, we observed no packet loss, bit errors, or CRC errors during our experiments.

SoNIC packet generator could easily achieve the maximum data rate, and allowed users to precisely control the number of `/I/`s to set the data rate of a packet stream. Moreover, with SoNIC, it was possible to inject less `/I/`s than the standard. For example, we could achieve 9 Gbps with 64B packets by inserting only eight `/I/`s between packets. This capability was not possible with any other (software) platform. In addition, if the APP thread was carefully designed, users could flexibly inject a random number of `/I/`s between packets, or the number of `/I/`s from captured data.

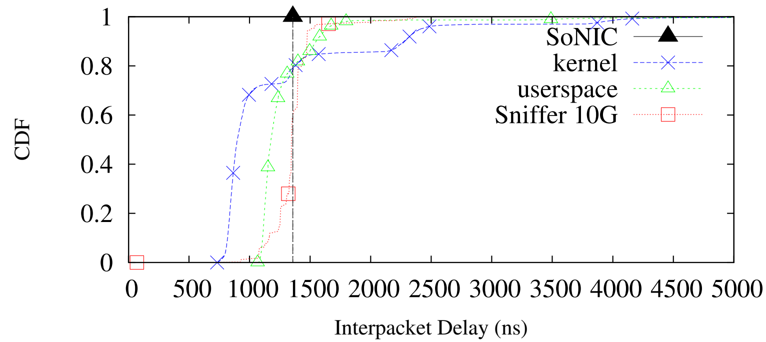


Figure 3.8: Comparison of timestamping

3.3.2 Packet Capturer (Packet Timestamping)

A packet capturer (a.k.a. packet sniffer, or packet analyzer) plays an important role in network research; it is the opposite side of the same coin as a packet generator. It can record and log traffic over a network which can later be analyzed to improve the performance and security of networks. In addition, capturing packets with precise timestamping is important for High Frequency Trading [74, 115] or latency sensitive applications.

Similar to the sending capability, the receiving capability of servers and software routers is inherently limited by the network adapters they use; it has been shown that some NICs are not able to receive packets at line speed for certain packet sizes [112]. Furthermore, if batching was used, timestamping was significantly perturbed if done in kernel or userspace [58]. High-performance devices such as Myricom Sniffer10G [25, 72] provided the ability of sustained capture of 10 GbE by bypassing kernel network stack. It also provided timestamping at 500 ns resolution for captured packets. SoNIC, on the other hand, could receive packets of any length at line-speed with precise timestamping.

Putting it all together, when we used SoNIC as a packet capturer (Fig-

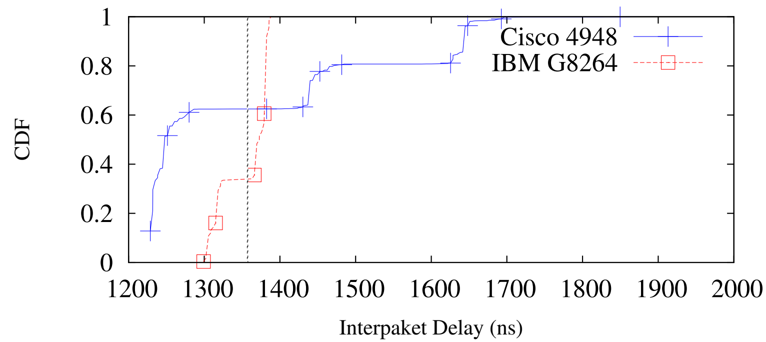


Figure 3.9: IPDs of Cisco 4948 and IBM G8264. 1518B packets at 9 Gbps

ure 2.6(a) and 3.1(b)), we were able to receive at the full Reference data rate (Figure 3.6). For the APP thread, we implemented a simple logging application which captured the first 48 bytes of each packet along with the number of `/I/`s and bits between packets. Because of the relatively slow speed of disk writes, we stored the captured information in memory. This required about 900MB to capture a stream of 64 byte packets for 1 second, and 50 MB for 1518 byte packets. We used ALT10G to generate packets for 1 second and compared the number of packets received by SoNIC to the number of packets generated.

SoNIC had perfect packet capture capabilities with flexible control in software. In particular, Figure 3.8 shows that given a 9 Gbps generated traffic with uniform IPD (average IPD=1357.224ns, stdev=0), SoNIC captured what was sent; this is shown as a single triangle at (1357.224, 1). All the other packet capture methods within userspace, kernel or a mixture of hardware timestamping in userspace (Sniffer 10G) failed to accurately capture what was sent. We received similar results at lower bandwidths as well.

3.3.3 Profiler

Interpacket delays are a common metric for network research. It can be used to estimate available bandwidth [87, 88, 120], increase TCP throughput [129], characterize network traffic [69, 82, 127], and detect and prevent covert timing channels [44, 89, 90]. There are a lot of metrics based on IPD for these areas. SoNIC increased the accuracy of those applications because of its precise control and capture of IPDs. In particular, when the SoNIC packet generator and capturer were combined, i.e. one port transmitted packets while the other port captured, SoNIC became a flexible platform for various studies. As an example, we demonstrate how SoNIC was used to profile network switches.

Switches can be generally divided into two categories: store-and-forward and cut-through switches. Store-and-forward switches decode incoming packets and buffers them before making a routing decision. On the other hand, cut-through switches route incoming packets before entire packets are decoded to reduce the routing latency. We generated 1518B packets with uniform 1357.19 ns IPD (=9 Gbps) to a Cisco 4948 (store-and-forward) switch and a IBM BNT G8264 (cut-through) switch. These switches showed different characteristics as shown in Figure 3.9. The x-axis is the interpacket delay; the y-axis is the cumulative distribution function. The long dashed vertical line on the left is the original IPD injected to the packet stream.

There are several takeaways from this experiment. First, the IPD for generated packets had no variance; none. The generated IPD produced by SoNIC was *always* the same. Second, the cut-through switch introduced IPD variance (stdev=31.6413), but less than the IPD on the store-and-forward switch (stdev=161.669). Finally, the average IPD was the same for both switches

since the data rate was the same: 1356.82 (cut-through) and 1356.83 (store-and-forward). This style of experiment can be used to profile and fingerprint network components as different models show different packet distributions.

3.4 Application 1: Covert Timing Channel

In this chapter, we demonstrated that access to the PHY enabled significantly improved precision for timestamping and pacing messages. In this section, we demonstrate that the improved precision of timestamping and pacing could also improve the performance of network applications such as covert channels. We designed and implemented a covert timing channel which relied on the precision of timestamping and pacing of the PHY and advances the state of the art.

Covert channels are defined as channels that are not intended for information transfer, but can leak sensitive information [79]. In essence, covert channels provide the ability to hide the transmission of data within established network protocols [130], thus hiding their existence. Covert channels are typically classified into two categories: Storage and timing channels. In *storage channels*, a sender modulates the value of a storage location to send a message. In *timing channels*, on the other hand, a sender modulates system resources over time to send a message [32].

Network covert channels send hidden messages over *legitimate* packets by modifying packet headers (storage channels) or by modulating interpacket delays (timing channels). Because network covert channels can deliver sensitive messages across a network to a receiver multiple-hops away, they impose serious threats to the security of systems.

In a network covert channel, the *sender* has secret information that she tries to send to a *receiver* over the Internet. The sender has control of some part of a network stack including a network interface (L1~2), kernel network stack (L3~4) and/or user application (L5 and above). Thus, the sender can modify protocol headers, checksum values, or control the timing of transmission of packets. The sender can either use packets from other applications of the system or generate its own packets. Although it is also possible that the sender can use packet payloads to directly embed or encrypt messages, we do not consider this case because it is against the purpose of a covert channel: *hiding the existence of the channel*. The *adversary* (or warden), on the other hand, wants to detect and prevent covert channels. A *passive* adversary monitors packet information to detect covert channels while an *active* adversary employs network appliances such as network jammers to reduce the possibility of covert channels.

In network *storage* channels, the sender changes the values of packets to secretly encode messages, which is examined by the receiver to decode the message. This can be easily achieved by using unused bits or fields of protocol headers. The IP Identification field, the IP Fragment Offset, the TCP Sequence Number field, and TCP timestamps are good places to embed messages [76, 102, 113, 114]. As with the easiness of embedding messages in packet headers, it is just as easy to detect and prevent such storage channels. The adversary can easily monitor specific fields of packet headers for detection, or *sanitize* those fields for prevention [57, 66, 93].

In network *timing* channels, the sender controls the timing of transmission of packets to deliver hidden messages. The simplest form of this channel is to send or not send packets in a pre-arranged interval [44, 107]. Because interpacket de-

lays are perturbed with noise from a network, synchronizing the sender and receiver is a major challenge in these on/off timing channels. However, synchronization can be avoided when each interpacket delay conveys information, i.e. a long delay is zero, and a short delay is one [41]. JitterBugs encodes bits in a similar fashion, and uses the remainder of modulo operation of interpacket delays for encoding and decoding [117]. These timing channels naturally create patterns of interpacket delays which can be analyzed with statistical tests for detection. For example, *regularity* tests [41, 44], *shape* tests [109], or *entropy* tests [60] are widely used for covert timing channel detection. On the other hand, to avoid detection from such statistical tests, timing channels can mimic patterns of legitimate traffic, or use random interpacket delays. Liu et al., demonstrated that with spreading codes and a shared key, a timing channel can be robust against known statistical tests [89]. They further developed a method to use independent and identically distributed (i.i.d) random interpacket delays to make the channel less detectable [90].

In this section, we demonstrate that a sophisticated covert timing channel can be created via precise pacing and timestamping and that such a channel is *high-bandwidth*, *robust* against cross traffic, and *undetectable* by software endhosts. The channel effectively delivers 81 kilobits per second with less than 10% errors over nine routing hops, and thousands of miles over the National Lambda Rail (NLR). Chupja advances the state-of-the-art covert timing channels which can only deliver thousands of bits per second while avoiding detection. We empirically demonstrate that we could create such a timing channel by modulating interpacket gaps at sub-microsecond scale: A scale at which sent information was preserved through multiple routing hops, but statistical tests could not differentiate the channel from legitimate traffic. The idea was to con-

trol (count) the number of `/I/`s to encode (decode) messages exhibiting packet pacing capabilities, i.e. to modulate interpacket gaps in nanosecond resolution.

3.4.1 Design

Design Goal

The design goal of our timing channel, called Chupja, was to achieve *high-bandwidth*, *robustness* and *undetectability* by precisely pacing and timestamping packets. By high-bandwidth, we mean a covert rate of many tens or hundreds of thousands of bits per second. Robustness is how to deliver messages with minimum errors. In particular, we set as our goal for robustness to a bit error rate (BER) of less than 10%, an error rate that was small enough to be compensated with forward error correction such as Hamming code, or spreading code [89]. We define BER as the ratio of the number of bits incorrectly delivered from the number of bits transmitted. Undetectability is how to hide the existence of it.

In order to achieve these goals, we precisely paced packets by modulating the number of `/I/`s between packets in the PHY. If the modulation of `/I/`s was large, the channel could effectively send messages in spite of noise or perturbations from a network (robustness). At the same time, if the modulation of `/I/`s was small, an adversary would not be able to detect regularities (undetectability). Further, Chupja embedded one timing channel bit per interpacket gap to achieve high-bandwidth. Thus, higher overt packet rates would achieve higher covert timing channel rates. We focused on finding an optimal modulation of interpacket gaps to achieve high-bandwidth, robustness, and undetectability (Section 3.4.2).

Model

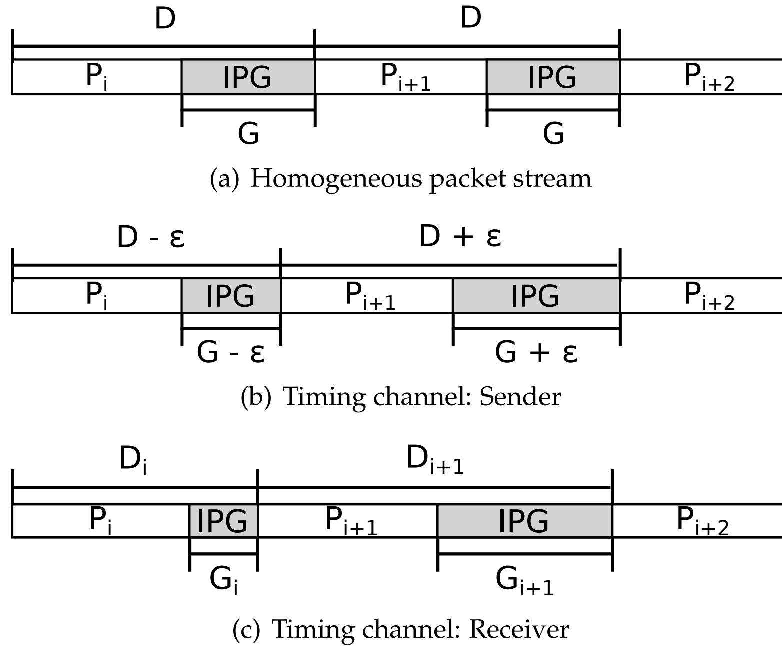
In our model, the sender of Chupja had control over a network interface card or a switch[1] with access to and control of the PHY. In other words, the sender could easily control the number of `/I/` characters of outgoing packets. The receiver was in a network multiple hops away, and tapped/sniffed on its network with access to the PHY. Then, the sender modulated the number of `/I/`s between packets destined to the receiver's network to embed secret messages.

Our model included an adversary who ran a network monitoring application and was in the same network with the sender and receiver. We assumed that the adversary was built from a commodity server and commodity NIC. As a result, the adversary did not have direct access to the PHY. Since a commodity NIC discards `/I/`s before delivering packets to the host, the adversary could not monitor the number of `/I/`s to detect the possibility of covert timing channels. Instead, it ran statistical tests with captured interpacket delays.

Encoding and Decoding

Chupja embedded covert bits into interpacket gaps of a homogeneous packet stream of an *overt* channel. In order to create Chupja, the sender and receiver must share two parameters: G and W. G was the number of `/I/`s in the IPG that was used to encode and decode hidden messages, and W (*Wait time*) helped the sender and receiver synchronize (Note that interpacket delay $D = G +$ packet size). Figure 3.10 illustrates our design. Recall that IPGs of a homogeneous packet stream are all the same (=G, Figure 3.10(a)). For example, the IPG of a

[1]We use the term *switch* to denote both bridge and router.



(a) Homogeneous packet stream

(b) Timing channel: Sender

(c) Timing channel: Receiver

Figure 3.10: Chupja encoding and decoding.

homogeneous stream with 1518 byte packets at 1 Gbps is always 13738 `/I/`s; the variance is zero. To embed a secret bit sequence $\{b_i, \ b_{i+1}, \ \cdots\}$, the sender encoded 'one' ('zero') by increasing (decreasing) the IPG (G) by ϵ `/I/`s (Figure 3.10(b)):

$$G_i = G - \epsilon \text{ if } b_i = 0$$

$$G_i = G + \epsilon \text{ if } b_i = 1$$

where G_i was the i-th interpacket gap between packet i and $i + 1$. When G_i was less than the minimum interpacket gap (or 12 `/I/` characters), it was set to twelve to meet the standard requirement.

Interpacket gaps (and delays) are perturbed as packets go through a number of switches. However, as we will see in Section 3.4.2, many switches did not significantly change interpacket gaps. Thus, we could expect that if ϵ was large enough, encoded messages would be preserved along the path. At the same

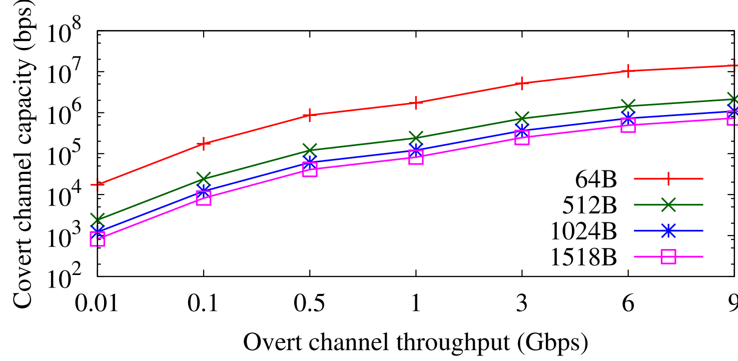


Figure 3.11: Maximum capacity of PHY timing channel

time, ϵ must be small enough to avoid detection by an adversary. We evaluated how big ϵ must be with and without cross traffic and over multiple hops of switches over thousands of miles in a network path (Section 3.4.2).

Upon receiving packet pairs, the receiver decoded bit information as follows:

$$b_i' = 1 \text{ if } G_i \geq G$$

$$b_i' = 0 \text{ if } G_i < G$$

b_i' might not be equal to b_i because of network noise. We used BER to evaluate the performance of Chupja (Section 3.4.2).

Because each IPG corresponded to a signal, there was no need for synchronization between the sender and the receiver [41]. However, the sender occasionally needed to pause until the next covert packet was available. W was used when there was a pause between signals. The receiver considered an IPG that was larger than W as a pause, and used the next IPG to decode the next signal.

The capacity of this PHY timing channel was equal to the number of packets being transmitted from the sender when there was no pause. Given a packet

size, the maximum capacity of the channel is illustrated in Figure 3.11. For example, if an overt channel sent at 1 Gbps with 1518 byte packets, the maximum capacity of the covert channel was $81,913$ bits per second (bps). We demonstrate in Section 3.4.2 that Chupja delivered 81 kilobits per second (kbps) with less than 10% BER over nine routing hops and thousands of miles over National Lambda Rail (NLR) where many links on our path were utilized with 1~4 Gbps cross traffic.

Discussion

Chupja used homogeneous packet streams to encode messages, which created a regular pattern of IPGs. Fortunately, as we discuss in the following section, the adversary was not able to accurately timestamp incoming packets when the data rate was high. This means that it did not matter what patterns of IPGs were used for encoding at above a certain data rate. Therefore, we chose the simplest form of encoding for Chupja. The fact that the PHY timing channel worked over multiple hops means that a non-homogeneous timing channel would work as well. For instance, consider the output after one routing hop as the sender, then the PHY timing channel worked with a non-homogeneous packet stream. If, on the other hand, the sender wanted to use other patterns for encoding and decoding, other approaches could easily be applied [44, 89, 90, 117]. For example, if the sender wanted to create a pattern that looked more random, we could also use a shared secret key and generate random IPGs for encoding and decoding [90]. However, the focus of this chapter is to demonstrate that even this simplest form of timing channel can be a serious threat to a system and not easily be detected.

3.4.2 Evaluation

In this section, we evaluated Chupja over real networks. We investigated to answer following questions.

- How *robust* was Chupja? How effectively could it send secret messages over the Internet?
- Why was Chupja *robust*? What properties of a network did it exploit?
- How *undetectable* was Chupja? Why was it hard to detect it and what was required to do so?

Evaluation Setup

Packet size [Bytes]	Data Rate [Gbps]	Packet Rate [pps]	IPD [ns]	IPG [/I/]
1518	9	737028	1356.8	170
1518	6	491352	2035.2	1018
1518	3	245676	4070.4	3562
1518	1	81913	12211.2	13738
64	6	10416666	96.0	48
64	3	5208333	192.0	168
64	1	1736111	576.0	648

Table 3.2: IPD and IPG of homogeneous packet streams.

ϵ (/I/s)	16	32	64	128	256	512	1024	2048	4096
ns	12.8	25.6	51.2	102.4	204.8	409.6	819.2	1638.4	3276.8

Table 3.3: Evaluated ϵ values in the number of /I/s and their corresponding time values in nanosecond.

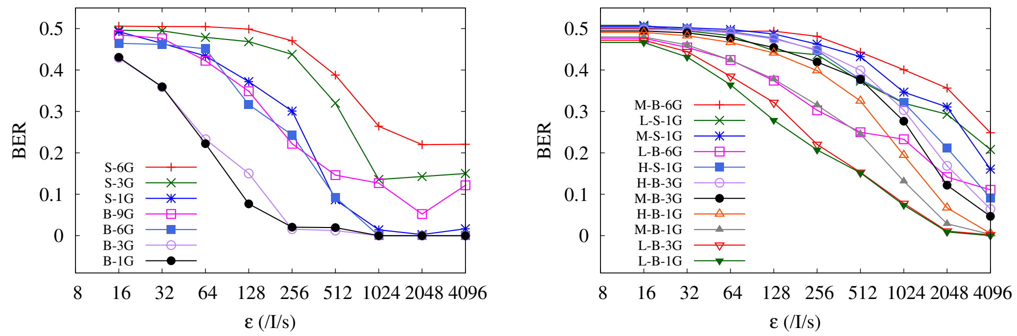
For most of the evaluation, we used 1518 byte and 64 byte packets for simplicity. We define packet size as the number of bytes from the first byte of the

Ethernet header to the last byte of the Ethernet frame check sequence (FCS) field (i.e. we exclude seven preamble bytes and start frame delimiter byte from packet size). Then, the largest packet allowed by Ethernet is 1518 bytes (14 byte header, 1500 payload, and 4 byte FCS), and the smallest is 64 bytes. In this section, the data rate refers to the data rate of the overt channel that Chupja was embedded. Interpacket delays (IPDs) and interpacket gaps (IPGs) of homogeneous packet streams at different data rates and with different packet sizes are summarized in Table 3.2. Table 3.3 shows the number of `/I/`s (ϵ) we modulated to create Chupja and their corresponding time values in nanosecond. We set ϵ starting from 16 `/I/`s (= 12.8 ns), doubling the number of `/I/`s up to 4096 `/I/`s (= 3276.8 ns). We use a tuple (s, r) to denote a packet stream with s byte packets running at r Gbps. For example, a homogeneous stream with (1518B, 1Gbps) is a packet stream with 1518 byte packets at 1 Gbps.

Efficiency of Chupja

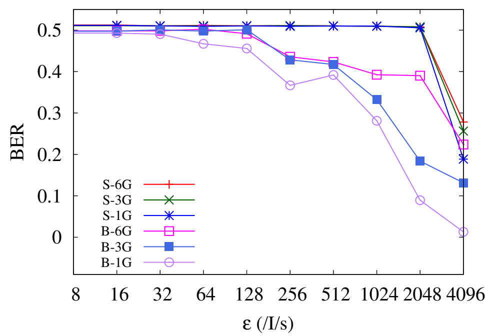
The goal of a covert timing channel is to send secret messages to the receiver with minimum errors (robustness). As a result, Bit Error Rate (BER) and the achieved covert bandwidth are the most important metrics to evaluate a timing channel. Our goal was to achieve BER less than 10% over a network with high bandwidth. As a result, we evaluated Chupja over two networks, a small network and the Internet (NLR), focusing on the relation between BER and the number of `/I/`s being modulated (ϵ).

A small network Before considering cross traffic, we first measured BER with no cross traffic in our small network (Figure 2.7). Figure 3.12(a) illustrates the



(a) Without cross traffic over a small network

(b) With cross traffic over a small network



(c) Over National Lambda Rail

Figure 3.12: BER of Chupja over a small network and NLR. X-Y-Z means that workload of cross traffic is X (H-heavy, M-medium, or L-light), and the size of packet and data rate of overt channel is Y (B-big=1518B or S-small=64B) and Z (1, 3, or 6G).

result. The x-axis is ϵ modulated in the number of idle (`/I/`) characters (see Table 3.3 to relate `/I/`s to time), and the y-axis is BER. Figure 3.12(a) clearly illustrates that the larger ϵ, the smaller BER. In particular, modulating 128 `/I/`s (=102.4 ns) was enough to achieve BER=7.7% with (1518B, 1Gbps) (filled in round dots). All the other cases also achieved the goal BER except (64B, 6Gbps) and (64B, 3Gbps). Recall that Table 3.2 gives the capacity of the covert channel. The takeaway is that when there was no cross traffic, modulating small number of `/I/`s (128 `/I/`s, 102.4 ns) was sufficient to create a timing channel. In addition, it was more efficient with large packets.

We also evaluated Chupja with cross traffic. We created three workloads where (α, β) = (0.333, 0.333), (0.9, 0.9), and (0.9, 3.7) (Figure 2.7), and we call them Light, Medium and Heavy workloads. Packets of cross traffic were always maximum transmission unit (MTU) sized. Then SoNIC1 generated timing channel packets at 1, 3, and 6 Gbps with 1518 and 64 byte packets. Figure 3.12(b) illustrates the result. At a glance, because of the existence of cross traffic, ϵ must be larger to transmit bits correctly compared to the case without cross traffic. There are a few takeaways. First, regardless of the size of workloads, timing channels with (1518B, 1Gbps) and (1518B, 3Gbps) worked quite well, achieving the goal BER of less than 10% with $\epsilon \geq 1024$. On the other hand, channels at a data rate higher than 6 Gbps were not efficient. In particular, $\epsilon = 4096$ was not sufficient to achieve the goal BER with (1518B, 6Gbps). Second, creating timing channels with small packets was more difficult. Generally, BER was quite high even with $\epsilon = 4096$ except H-S-1G case (BER=9%).

National Lambda Rail Figure 3.12(c) illustrates the results from NLR. Again, we changed the size and the data rate of overt packets. In NLR, it was more difficult to create a timing channel. In particular, only (1518B, 1Gbps) achieved BER less than 10% when ϵ was larger than 2048 (8.9%). All the other cases had higher BERs than our desired goal, although BERs were less than 30% when ϵ is 4096. Creating a channel with 64 byte packet was no longer possible in NLR. This was because more than 98% of IPGs were minimum interpacket gaps, i.e. most of bit information was discarded because of packet train effects [58].

Sensitivity Analysis

Network devices change interpacket gaps while forwarding packets; switches add randomness to interpacket gaps. We discuss how Chupja could deliver secret messages via a PHY timing channel in spite of the randomness added from a network. In particular, we discuss the following observations.

- A single switch did not add significant perturbations to IPDs when there was no cross traffic.
- A single switch treated IPDs of a timing channel's encoded 'zero' bit and those of an encoded 'one' bit as uncorrelated distributions; ultimately, allowing a PHY timing channel receiver to distinguish an encoded 'zero' from an encoded 'one'.
- The first and second observations above held for multiple switches and cross traffic.

In other words, we demonstrate that timing channels could encode bits by modulating IPGs by a small number of `/I/` characters (hundreds of nanoseconds) and these small modulations could be effectively delivered to a receiver over multiple routing hops with cross traffic.

In order to understand and appreciate these observations, we must first define a few terms. We denote the interpacket delay between packet i and $i + 1$ with the random variable D_i. We use superscript on the variables to denote the number of routers. For example, D_i^1 is the interpacket delay between packet i and $i + 1$ after processed by one router, and D_i^0 is the interpacket delay before packet i and $i + 1$ are processed by any routers. Given a distribution of D,

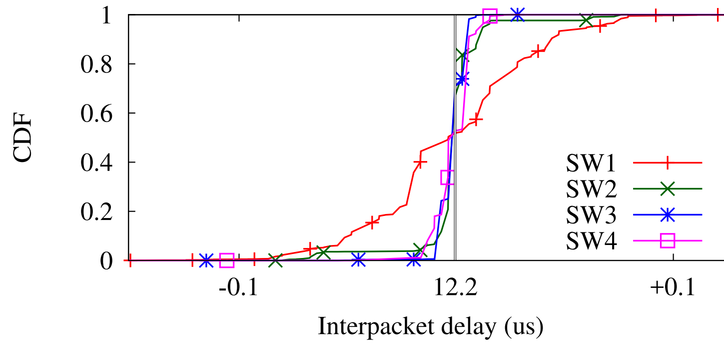


Figure 3.13: Comparison of IPDs after switches process a homogeneous packet stream (1518B, 1Gbps)

and the average interpacket delay μ, we define I_{90} as the smallest ϵ that satisfies $P(\mu - \epsilon \leq D \leq \mu + \epsilon) \geq 0.90$. In other words, I_{90} is the interval from the average interpacket delay, μ, which contains 90% of D (i.e. at least 90% of distribution D is within $\mu \pm I_{90}$). For example, I_{90} of a homogeneous stream (a delta function, which has no variance) that leaves the sender and enters the first router is zero; i.e. D^0 has $I_{90} = 0$ and $P(\mu - 0 \leq D \leq \mu + 0) = 1$ since there is no variance in IPD of a homogeneous stream. We use I_{90} in this section to quantify perturbations added by a network device or a network itself. Recall that the goal of Chupja was to achieve a BER less than 10%, and, as a result, we were interested in the range where 90% of D observed by a timing channel receiver was contained.

First, *switches did not add significant perturbations to IPDs when there was no cross traffic*. In particular, when a homogeneous packet stream was processed by a switch, I_{90} was always less than a few hundreds of nanoseconds, i.e. 90% of the received IPDs were within a few hundreds of nanoseconds from the IPD originally sent. Figure 3.13 displays the received IPD distribution measured after packets were processed and forwarded by one switch: The x-axis is the received

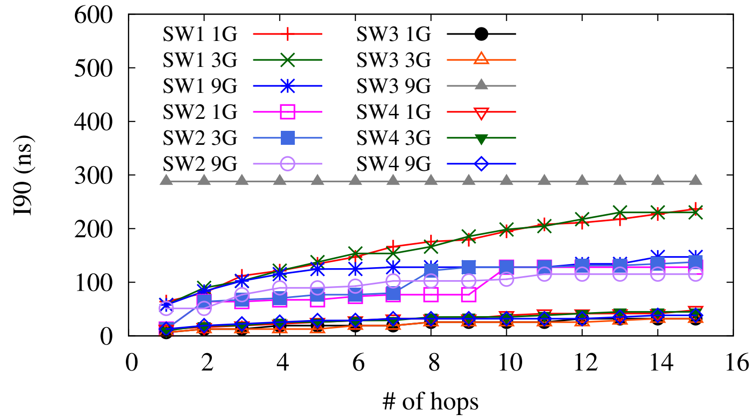
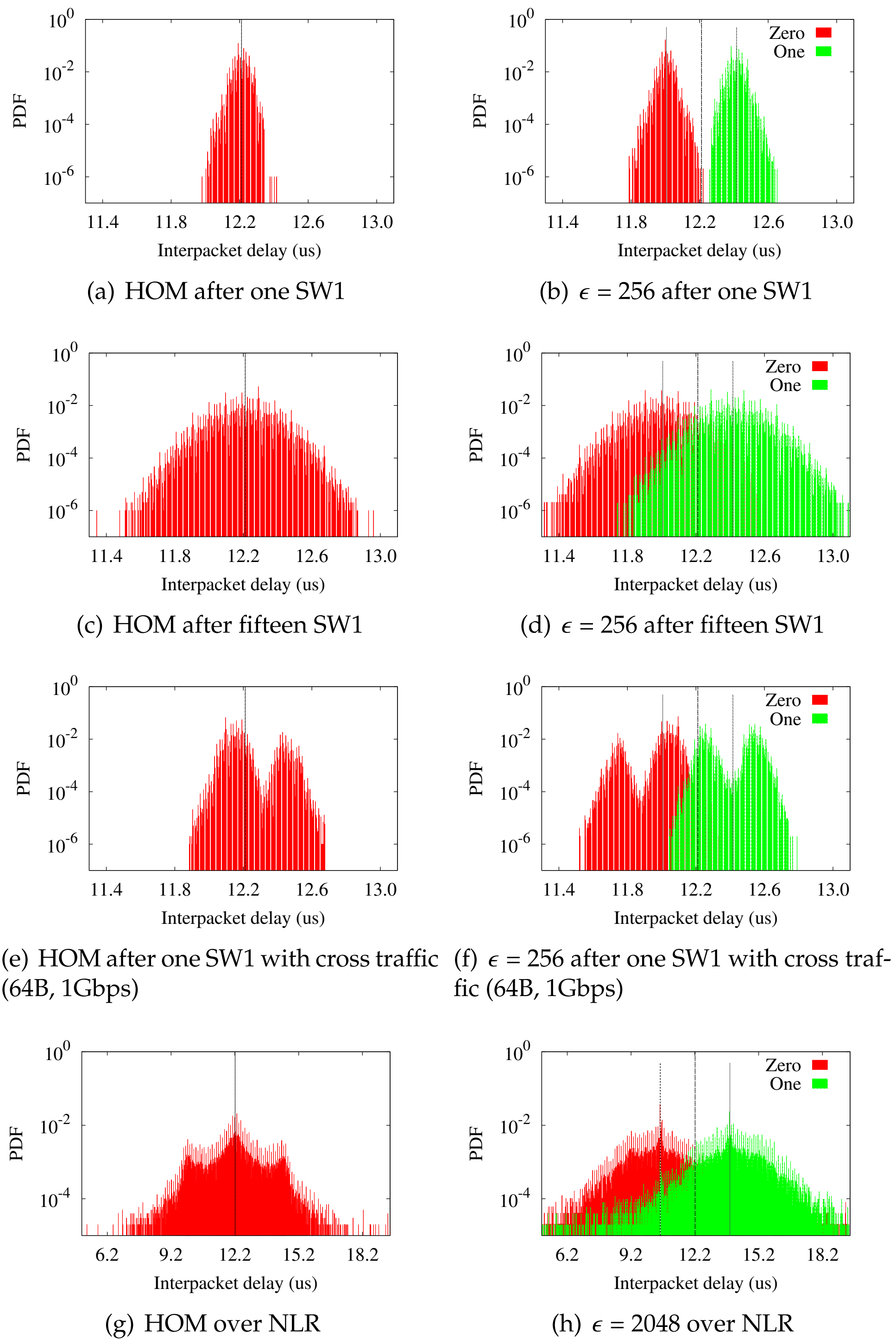


Figure 3.14: I_{90} comparison between various switches

IPD and the y-axis is the cumulative distribution function (CDF). Further, different lines represent different switches from Table 2.3. The characteristics of the original sender's homogeneous packet stream was a data rate of 1 Gbps with 1518B size packets, or (1518B, 1Gbps) for short, which resulted in an average IPD of 12.2 us (i.e. μ = 12.2 us). We can see in Figure 3.13 that when μ was 12.2 us and ϵ was 0.1 us, $P(12.2 - 0.1 < D < 12.2 + 0.1) \geq 0.9$ was true for all switches. In general, the range of received IPDs was always bounded by a few hundreds of nanoseconds from the original IPD, regardless of the type of switch.

Moreover, when a packet stream was processed by the same switch type, but for multiple hops, I_{90} increased linearly. Each packet suffered some perturbation, but the range of perturbation was roughly constant at every hop over different packet sizes [85] resulting in a linear increase in I_{90}. In Figure 3.14, we illustrate I_{90} for different switches, at different data rates (1, 3, and 9G), and as we increased the number of hops: The x-axis is the number of routing hops, y-axis is measured I_{90}, and each line is a different type of switch with a different packet stream data rate. Packet size was 1518B for all test configurations. One important takeaway from the graph is that I_{90} for the same switch showed



(a) HOM after one SW1 (b) ϵ = 256 after one SW1

(c) HOM after fifteen SW1 (d) ϵ = 256 after fifteen SW1

(e) HOM after one SW1 with cross traffic (64B, 1Gbps) (f) ϵ = 256 after one SW1 with cross traffic (64B, 1Gbps)

(g) HOM over NLR (h) ϵ = 2048 over NLR

Figure 3.15: Comparison of homogeneous streams and covert channel streams of (1518B, 1Gbps)

similar patterns regardless of data rates, except SW3 9 Gbps. In particular, the degree of perturbation added by a switch was not related to the data rate (or the average interpacket delay). Instead, IPD perturbation was related to the number of hops, and the size of packet. Further, a second takeaway is that I_{90} values after one hop were all less than 100 ns, except SW3 9 Gbps, and still less than 300 ns after fifteen hops. 300 ns is slightly greater than 256 /I/s (Table 3.3). Unfortunately, we did not have a definitive explanation on why I_{90} of SW3 9 Gbps was larger than any other case, but it was likely related to how SW3 handled high data rates.

Second, *switches treated IPDs of an encoded 'zero' bit and those of an encoded 'one' bit as uncorrelated distributions.* After encoding bits in a timing channel, there will be only two distinctive IPD values leaving the sender: $\mu + \epsilon$ for 'one', and $\mu - \epsilon$ for 'zero'. Let a random variable $D+$ be the IPDs of signal 'one', and $D-$ be those of signal 'zero'. We observed that the encoded distributions after one routing hop, D^1+ and D^1-, looked similar to the unencoded distribution after one routing hop, D^1. The similarity was likely due to the fact that at the sender the encoded distributions, D^0+ and D^0-, were each homogeneous packet streams (i.e. D^0, D^0+, and D^0- are all delta functions, which have no variance). For instance, using switch SW1 from Table 2.3, Figure 3.15(a) illustrates D^1 (the unencoded distribution of IPDs after one routing hop) and Figure 3.15(b) illustrates D^1+ and D^1- (the encoded distribution after one routing hop). The data rate and packet size was 1Gbps and 1518B, respectively, with ϵ = 256 /I/s for the encoded packet stream. We encoded the same number of 'zeros' and 'ones' randomly into the packet stream. Note the similarity in distributions between D^1 in Figure 3.15(a) and D^1+ and D^1- in Figure 3.15(b). We observed a similarity in distributions among D^1, D^1+, and D^1- throughout different data rates

and switches. We conjectured that $D+$ and $D-$ were uncorrelated because the computed correlation coefficient between $D+$ and $D-$ was always very close to zero.

Because the distributions of D^1+ and D^1- were uncorrelated, we could effectively deliver bit information with appropriate ϵ values for one hop. If ϵ was greater than I_{90} of D^1, then 90% of IPDs of D^1+ and D^1- will not overlap. For example, when I_{90} is 64 ns, and ϵ is 256 `/I/`s (=204.8 ns), two distributions of D^1+ and D^1- were clearly separated from the original IPD (Figure 3.15(b)). On the other hand, if ϵ was less than I_{90} of D^1, then many IPDs overlapped, and thus the BER increased. For instance, Table 3.4 summarizes how BER of the timing channel varied with different ϵ values. From Table 3.4, we can see that when ϵ was greater than 64 `/I/`s, BER was always less than 10%. The key takeaway is that BER was *not* related with the data rate of the overt channel, rather it was related to I_{90}.

ϵ (`/I/`s)		16	32	64	128	256	512	1024
BER	1G	0.35	0.24	0.08	0.003	10^{-6}	0	0
	3G	0.37	0.25	0.10	0.005	10^{-5}	0	0
	6G	0.35	0.24	0.08	0.005	0.8×10^{-6}	0	0
	9G	0.34	0.24	0.07	0.005	0.0005	0.0004	0.0005

Table 3.4: ϵ and associated BER with (1518B, 1Gbps)

Third, *switches treated IPDs of an encoded 'zero' bit and those of an encoded 'one' bit as uncorrelated distributions over multiple switches and with cross traffic.* In particular, distributions D^n+ and D^n- were uncorrelated regardless of the number of hops and the existence of cross traffic. However, I_{90} became larger as packets went through multiple routers with cross traffic. Figures 3.15(c) and 3.15(d) show the distributions of D^{15}, $D^{15}+$, and $D^{15}-$ without cross traffic (Note that the y-axis is log-scale). The data rate and packet size was 1 Gbps and 1518B,

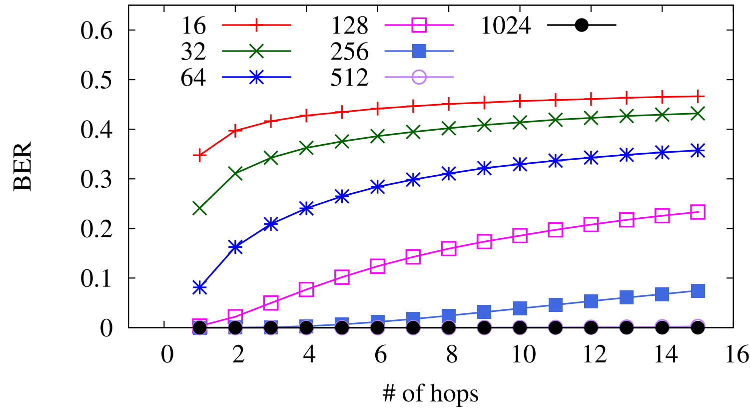


Figure 3.16: BER over multiple hops of SW1 with various ϵ values with (1518B, 1Gbps)

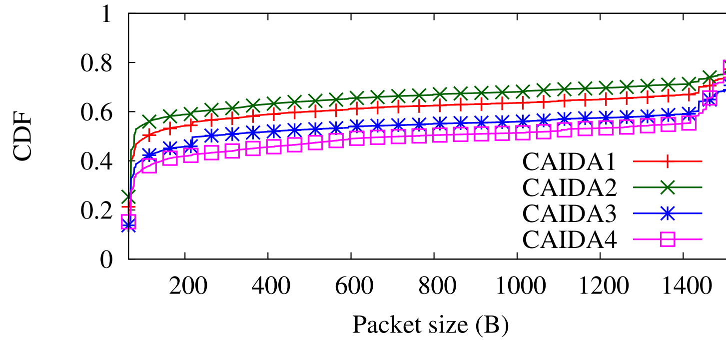


Figure 3.17: Packet size distributions of CAIDA traces

respectively, with ϵ = 256 /I/s for the encoded packet stream. We conjectured that the distributions were still uncorrelated without cross traffic and after multiple hops of routers: The computed correlation coefficient was close to zero. Further, the same observation was true with the other switches from Table 2.3. Figure 3.16 shows BER over multiple hops of SW1. When ϵ was greater than 256 /I/s (=204.8 ns), BER was always less than 10% after fifteen hops. Recall that I_{90} of D after 15 hops of SW1 was 236 ns (Figure 3.14).

Figures 3.15(e) and 3.15(f) show the distributions after one routing hop when

	Data Rate [Gbps]	Packet Rate [pps]	I_{90} [ns]	BER	
				$\epsilon = 512$	$\epsilon = 1024$
CAIDA1	2.11	418k	736.0	0.10	0.041
CAIDA2	3.29	724k	848.1	0.148	0.055
CAIDA3	4.27	723k	912.1	0.184	0.071
CAIDA4	5.12	798k	934.4	0.21	0.08

Table 3.5: Characteristics of CAIDA traces, and measured I_{90} and BER

there was cross traffic: Distributions D^1, D^1+, and D^1- using overt data rate and packet size 1 Gbps and 1518B, respectively. The cross traffic was (64B, 1Gbps). We can see in the figures that there was still a similarity in D^1+ and D^1- even with cross traffic. However, I_{90} became larger due to cross traffic when compared to without cross traffic. Table 3.6 summarizes how I_{90} changed with cross traffic. We used five different patterns of cross traffic for this evaluation: 10-clustered (10C), 100-clustered (100C), one homogeneous stream (HOM), two homogeneous streams (HOM2), and random IPD stream (RND). A N-clustered packet stream consisted of multiple clusters of N packets with the minimum interpacket gap (96 bits, which is 12 `/I/` characters) allowed by the standard [19] and a large gap between clusters. Note that a larger N meant the cross traffic was *bursty*. For the RND stream, we used a geometric distribution for IPDs to create bursty traffic. In addition, in order to understand how the distribution of packet sizes affected I_{90}, we used four CAIDA traces [30] at different data rates to generate cross traffic (Table 3.5). With packet size and timestamp information from the traces, we reconstructed a packet stream for cross traffic with SoNIC. In the CAIDA traces, the distribution of packet sizes was normally a bimodal distribution with a peak at the lowest packet size and a peak at the highest packet size (Figure 3.17).

We observe that I_{90} increased with cross traffic (Table 3.6). In particular,

bursty cross traffic at higher data rates significantly impacted I_{90}, although they were still less than one microsecond except 100C case. The same observation was also true using the CAIDA traces with different data rates (Table 3.5). As a result, in order to send encoded timing channel bits effectively, ϵ must increase as well. Figure 3.15(g) and 3.15(h) show the distributions of IPDs over the NLR. It demonstrates that with external traffic and over multiple routing hops, sufficiently large ϵ could create a timing channel.

Data Rate [Gbps]	Packet Size [Byte]	I_{90}				
		10C	100C	HOM	HOM2	RND
0.5	64	79.9	76.8	166.45	185.6	76.8
	512	79.9	79.9	83.2	121.6	86.3
	1024	76.8	76.8	80.1	115.2	76.8
	1518	111.9	76.8	128.0	604.7	83.2
1	64	111.9	108.8	236.8	211.2	99.3
	512	115.2	934.4	140.8	172.8	188.9
	1024	111.9	713.5	124.9	207.9	329.5
	1518	688.1	1321.5	64.0	67.1	963.3

Table 3.6: I_{90} values in nanosecond with cross traffic.

Summarizing our sensitivity analysis results, I_{90} was determined by the characteristics of switches, cross traffic, and the number of routing hops. Further, I_{90} could be used to create a PHY timing channel like Chupja. In particular, we can refine the relation between I_{90} and ϵ^* (the minimum ϵ measured to achieve a BER of less than 10%). Let I_{90}^+ be the minimum ϵ that satisfies $P(D > \mu - \epsilon) \geq 0.90$ and let I_{90}^- be the minimum ϵ that satisfies $P(D < \mu + \epsilon) \geq 0.90$ given the average interpacket delay μ. Table 3.7 summarizes this relationship between $\epsilon*$, I_{90} and $\max(I_{90}^+, I_{90}^-)$ over the NLR (Figure 2.8) and our small network (Figure 2.7).

In our small network, BER was always less than 10% when ϵ is greater than $\max(I_{90}^+, I_{90}^-)$. On the other hand, we were able to achieve our goal BER over the NLR when ϵ^* was slightly less than $\max(I_{90}^+, I_{90}^-)$. Because we did not have

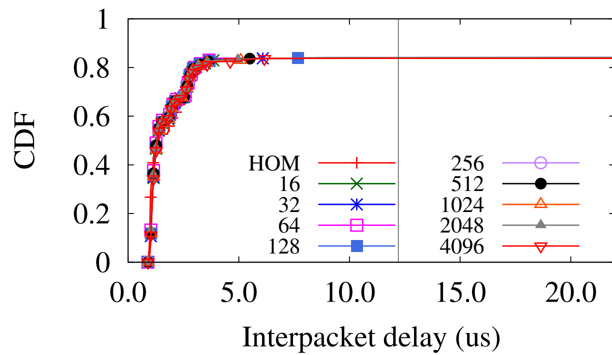
Network	Workload	I_{90}	$\max(I_{90}^{+}, I_{90}^{-})$	ϵ^{*} (ns)
Small network	Light	1065.7	755.2	819.2
Small network	Medium	1241.6	1046.3	1638.4
Small network	Heavy	1824.0	1443.1	1638.4
NLR		2240.0	1843.2	1638.4

Table 3.7: Relation between ϵ, I_{90}, and $\max(I_{90}^{+}, I_{90}^{-})$ over different networks with (1518B, 1Gbps). Values are in nanosecond.

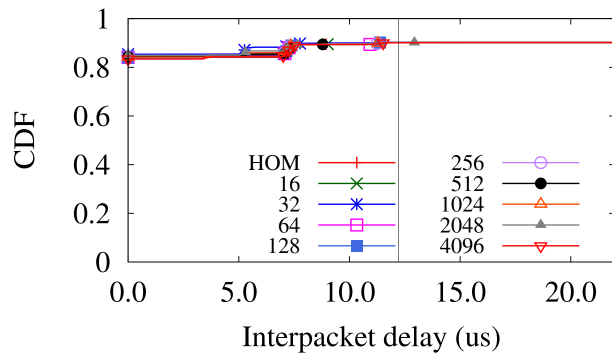
control over cross traffic in the NLR, I_{90} varied across our experiments.

Detection

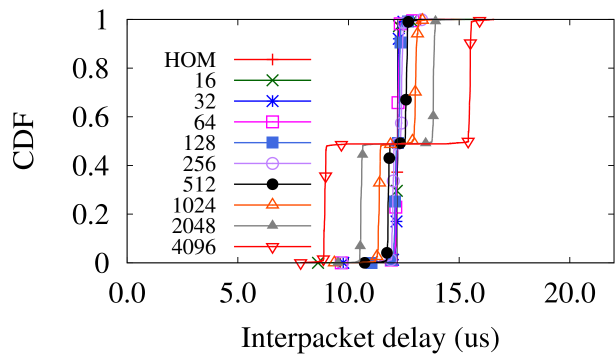
In order to detect timing channels, applying statistical tests to captured IPDs is widely used. For example, the adversary can use regularity, similarity, shape test, and entropy test of IPDs in order to detect potential timing channels [41, 44, 60, 109]. The same strategies could be applied to Chupja. Since our traffic was very *regular*, those algorithms could be easily applied to detect Chupja. However, we argue that none of these algorithms will work if the IPG modulations cannot be observed at all. In particular, endhost timestamping was too inaccurate to observe fine-grained IPG modulations whereas Chupja modulated IPGs in hundreds of nanoseconds to create a timing channel. In fact, the accuracy of endhost timestamping was at most microsecond resolution. Specialized NICs can provide a few hundreds of nanosecond resolution. In this section, we demonstrate that endhost or hardware timestamping was not sufficient to detect Chupja timing channels. We focused on measuring and comparing an endhost's ability to accurately timestamp arrival times (i.e. accurately measure IPDs) since the ability to detect a PHY timing channel was dependent upon the ability to accurately timestamp the arrival time of packets. As a result, we do not discuss statistical approaches further.



(a) Kernel timestamping.



(b) Zero-copy timestamping.



(c) Hardware timestamping.

Figure 3.18: Comparison of various timestamping methods. Each line is a covert channel stream of (1518B, 1Gbps) with a different ϵ value.

As discussed in Section 1.2.2, timestamping messages in kernel or userspace is inaccurate mainly due to the overhead from a network stack. To reduce the

overhead of a network stack between kernel and userspace and between hardware and kernel, a technique called zero-copy could be employed to improve the performance of userspace network applications. An example of a zero-copy implementation was Netmap [112]. In Netmap, packets were delivered from a NIC directly to a memory region which was shared by a userspace application. This zero-copy removed expensive memory operations and bypassed the kernel network stack. As a result, Netmap was able to inject and capture packets at line speed in a 10 GbE network with a single CPU. Therefore, detection algorithms could exploit a platform similar to Netmap to improve the performance of network monitoring applications. We call this *zero-copy timestamping*. We also included in our evaluation a specialized NIC with hardware timestamping capability, the Sniffer 10G [25], which provided 500 ns resolution for timestamping.

In order to compare *kernel*, *zero-copy*, and *hardware* timestamping, we connected a SoNIC server and a network monitor directly via an optical fiber cable, generated and transmitted timing channel packets to a NIC installed in the network monitor, and collected IPDs using different timestamping methods. The network monitor was a passive adversary built from a commodity server. Further, we installed Netmap in the network monitor. Netmap originally used the `do_gettimeofday` for timestamping packets in kernel space, which provided only microsecond resolution. We modified the Netmap driver to support nanosecond resolution instead. For this evaluation, we always generated ten thousand packets for comparison because some of the approaches discarded packets when more than ten thousand packets were delivered at high data rates.

Figure 3.18 illustrates the results. Figure 3.18(a) demonstrates the effective-

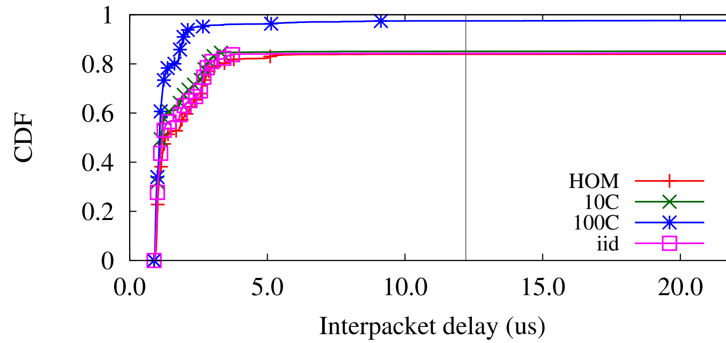


Figure 3.19: Kernel timestamping with (1518B, 1Gbps).

ness of *kernel timestamping* of a timing channel with various IPG modulation (ϵ) values. The data rate of the overt channel was 1 Gbps and the packet size was 1518 bytes. The x-axis is interpacket delays (IPDs) in microsecond and y-axis is a cumulative distribution function (CDF). The vertical line in the middle is the original IPD (=12.2 us) of Chupja. In order to detect Chupja, the timestamping CDF would be centered around the vertical line at ≈12.2 us. Instead, as can be seen from the graph, all measured kernel timestamps were nowhere near the vertical line regardless of ϵ values (ϵ varied between ϵ=0 [HOM] to ϵ=4096 `/I/`s). As a result, kernel timestamping could not distinguish a PHY covert channel like Chupja. In fact, even an i.i.d random packet stream was inseparable from other streams (Figure 3.19). Unfortunately, *zero-copy timestamping* did not help the situation either (Figure 3.18(b)). Netmap did not timestamp every packet, but assigned the same timestamp value to packets that were delivered in one DMA transaction (or polling). This is why there were packets with zero IPD. Nonetheless, Netmap still depended on underlying system's timestamping capability, which was not capable.

On the other hand, *hardware timestamping* using the Sniffer 10G demonstrated enough fidelity to detect Chupja when modulation (ϵ) values were larger

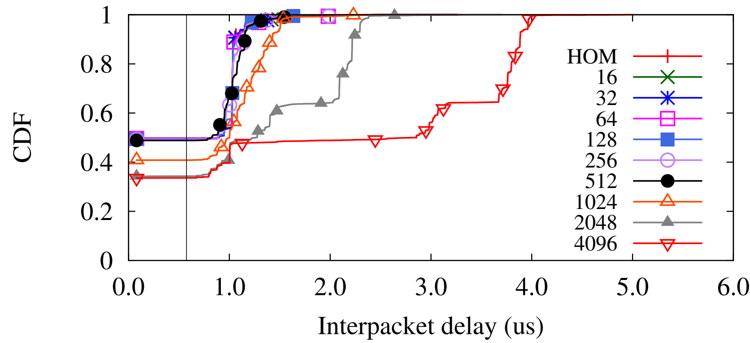


Figure 3.20: Hardware timestamping with (64B, 1Gbps)

than 128 `/I/s` (Figure 3.18(c)). However, hardware timestamping still could not detect smaller changes in IPDs (i.e. modulation, ϵ, values smaller than 128 `/I/s`), which was clear with a timing channel with smaller packets. A timing channel with 64 byte packets at 1 Gbps was not detectable by hardware timestamping (Figure 3.20). This was because packets arrived much faster with smaller packets making IPGs too small for the resolution of hardware to accurately detect small IPG modulations.

The takeaway is that to improve the possibility of detecting Chupja, which modulated IPGs in a few hundreds of nanoseconds, a network monitor (passive adversary) must employ hardware timestamping for analysis. However, using better hardware (more expensive and sophisticated NICs) still might not be sufficient; i.e. for much finer timing channels. Therefore, we concluded that a PHY timing channel such as Chupja was invisible to a software endhost. However, a hardware based solutions with fine-grained capability [10] might be able to detect Chupja.

3.4.3 Summary

In this section, we demonstrated how precise pacing and timestamping could create a covert timing channel that was high-bandwidth, robust and undetectable. The covert timing channel embedded secret messages into interpacket gaps in the physical layer by pacing packets at sub-microsecond scale. We empirically demonstrated that our channel could effectively deliver 81 kilobits per second over nine routing hops and thousands miles over the Internet, with a BER less than 10%. As a result, a Chupja timing channel worked in practice and was undetectable by software endhosts since they were not capable of precise timestamping to detect such small modulations in interpacket gaps employed by Chupja.

3.5 Application 2: Estimating Available Bandwidth

Another network application that relies on the precision of timestamping and pacing is available bandwidth estimation: The sender must carefully control the time gaps of probe packets. Similarly, the receiver must carefully examine the time gaps of probe packets to infer the available bandwidth between two processes. As a result, the accuracy of available bandwidth estimation algorithms depends on the precision of timestamping and pacing. In this section, we discuss how precise pacing and timestamping via access to the PHY can improve the performance of available bandwidth estimation algorithms.

Available bandwidth estimation is motivated by a simple problem: What is the maximum data rate that a user could send down a given network path

without going over the available capacity? This data rate is equal to the available bandwidth on the *bottlenecked link,* which has the minimum available bandwidth among all links on the network path.

An available bandwidth estimation algorithm typically sends probe packets with a predefined interval along the path under measurement (pacing), and observe change in certain packet characteristics at the receiver to infer the amount of cross traffic in the network by timestamping probe packets. For instance, when the probing rate exceeds the available bandwidth, the observed packet characteristics undergo a significant change. Estimation algorithms infer the available bandwidth at the receiving end by examining timestamps of probe packets [68, 111, 120, 97]. As a result, the precision of pacing and timestamping is the key to accurately estimating the available bandwidth.

In this section, we present MinProbe that performed available bandwidth estimation with high-fidelity for high-speed networks, while maintaining the flexibility of userspace control and compatibility with existing bandwidth estimation algorithms. MinProbe greatly increased measurement fidelity via precise timestamping and pacing provided by SoNIC. We evaluated MinProbe on a testbed that consisted of multiple 10 GbE switches and on the National Lambda Rail (NLR). It achieved high accuracy: Results illustrated that the difference between the estimated available bandwidth and the actual available bandwidth was typically no more than 0.4 Gbps in a 10 Gbps network. MinProbe advances the state-of-the-art algorithms which do not perform well in high speed networks such as 10 Gbps networks.

3.5.1 Design

The key insight and capability of how MinProbe achieved high-fidelity is from precise pacing and timestamping via its direct access to the `/I/` characters in the physical layer. In particular, MinProbe measured (counted) and generated (inserted or removed) an exact number of `/I/` characters between each subsequent probe packet to measure the relative time elapsed between packets or generate a desired interpacket gap.

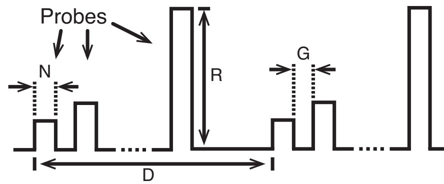


Figure 3.21: Generalized probe train model

Figure 3.21 shows a generalized probe train model we developed to emulate a number of existing bandwidth estimation algorithms and used for the rest of the section. The horizontal dimension is time. The vertical dimension is the corresponding (instantaneous) probe rate. Each pulse contains multiple probe packets sent at a particular rate, as depicted by the height. In particular, the parameter N represents the number of packets in each train and parameter R represents the (instantaneous) probe rate of the train (i.e. we are able to change the interpacket gap between N packets to match a target probe rate R). Packet sizes are considered in computing the probe rate. For a mixed-size packet train, MinProbe was able to adjust the space between all adjacent packets within the train to achieve the desired probe rate R. The gaps between successive probe trains are specified with parameter G (gap). Finally, each measurement *sample* consists of a set of probe trains with increasing probe rate. The distance in time

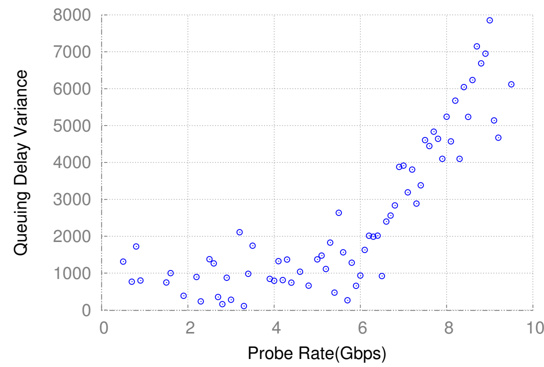
Algorithm	N	R	G	D
Pathload	20	[0.1:0.1:9.6] Gbps	Variable	Variable
Pathchirp	1	[0.1:0.1:9.6] Gbps	Exponential decrease	Variable
Spruce	2	500Mbps	1.2ns	48us
IGI	60	[0.1:0.1:9.6] Gbps	30s	30s

Table 3.8: Parameter setting for existing algorithms. G is the gap between packet trains. R is the rate of probe. N is the number of probe packets in each sub-train. D is the gap between each sub-train.

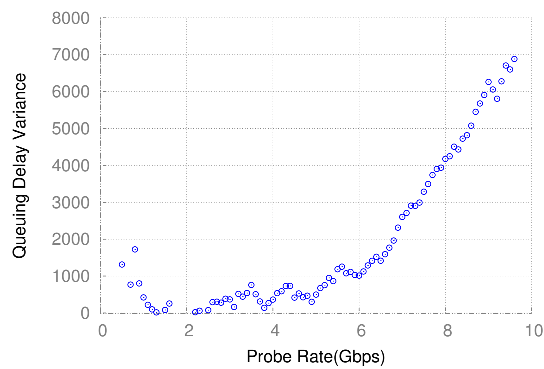
between each measurement sample is identified with parameter D (distance). With these four parameters, we could emulate most probe traffic used in prior work, as shown in Table 3.8. For example, to emulate Spruce probes, we set the parameters (N, R, G, D) to the values (2, 500Mbps, 1.2ns, 48us), which would generate a pair of probe packets every 48 us with minimal inter-packet gap (12 `/I/` characters or 1.2ns) at 500Mbps (5% of the capacity of a 10Gbps link), as intended by the original Spruce Algorithm [120]. Similarly, to reproduce the IGI experiment results [67], we set the parameters (N, R, G, D) to the values (60, [0.1:0.1:9.6]Gbps, 30s, 30s) where $[a : b : c]$ is a range of numbers from a to c and the difference between each number increment is b. In summary, most types of existing probe trains can be emulated with the generalized model we developed for MinProbe, where different points in the parameterization space can represent different points in the entire design space covering prior art and possibly new algorithms.

3.5.2 Evaluation

We evaluated MinProbe over our simple network topology and the National Lambda Rail (NLR). For our experiments, we generated probe packets using parameter (N, R, G, D) = (20, [0.1 : 0.1 : 9.6] Gbps, 10 us, 4 ms). We used 792-



(a) Raw measurement



(b) After moving average

Figure 3.22: Bandwidth estimation of CAIDA trace, the figure on the top is the raw data trace, the figure on the bottom is the moving average data.

bytes as the packet size which was close to the average packet size observed in the Internet [30]. We adjusted the traffic data rate by varying interpacket gaps (IPG). In particular, we inserted a specific number of `/I/`s between packets to generate a specific data rate.

CAIDA Trace First, we evaluated MinProbe with traces extracted from CAIDA anonymized OC192 dataset [30] recorded using a DAG card [10] with nanosecond scale timestamps, and replayed by SoNIC as a traffic generator. Figure 3.22(a) shows an example of the raw data captured by MinProbe. The x-axis

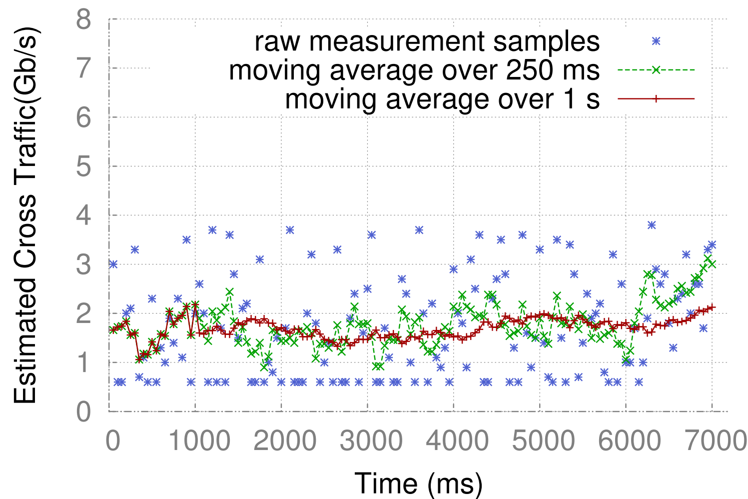


Figure 3.23: Measurement result in NLR.

is the rate (R) at which the probe trains were sent. The y-axis is the variance of the queuing delay, which was computed from the one-way delay (OWD) experienced by the packets within the same probe train. The available bandwidth was estimated to be the turning point where the delay variance showed a significant trend of increasing. We observed that the traffic traces were bursty. To compensate, we used standard exponential moving average (EMA) method to smooth out the data in Figure 3.22(a). For each data points in Figure 3.22(a), the value was replaced by the weighted average of the five data points preceding the current data point. Figure 3.22(b) shows the result. The difference between the estimated available bandwidth and the actual available bandwidth was typically within 0.4 Gbps of the true available bandwidth.

NLR Second, we evaluated MinProbe over NLR. The traffic was routed through the path shown in Figure 2.8. Since it was difficult to obtain accurate readings of cross traffic at each hop, we relied on the router port statistics to obtain a 30-second average of cross traffic. We observed that the link between

Cleveland and NYC experienced the most cross traffic compared to other links, and was therefore the bottleneck (tight) link of the path. The amount of cross traffic was 1.87 Gbps on average, with a maximum of 3.58 Gbps during our experiments.

Figure 3.23 shows the result of MinProbe in NLR. We highlight two important takeaways: First, the average of our estimation was close to 2 Gbps, which was consistent with the router port statistics collected every 30 seconds. Second, we observed that the readings were very bursty, which means the actual cross traffic on NLR exhibited a good deal of burstiness. The available bandwidth estimation by MinProbe agreed with the link utilization computed from switch port statistical counters: The mean of the MinProbe estimation was 1.7 Gbps, which was close to the 30-second average 1.8 Gbps computed from switch port statistical counters. Moreover, MinProbe estimated available bandwidth at higher sample rate and with finer resolution.

3.5.3 Summary

In this section, we presented MinProbe which accurately estimated available bandwidth in 10 Gbps networks via precise pacing and timestamping. MinProbe performed well under bursty traffic, and the estimation was typically within 0.4 Gbps of the true value in a 10 Gbps network.

3.6 Summary

Precise pacing and timestamping is important for many network research applications. In this chapter, we demonstrated that access to the physical layer could improve the precision of pacing and timestamping. In particular, SoNIC implements the physical layer in software and allows users to precisely pace and timestamp packets. At its heart, SoNIC utilizes commodity-off-the-shelf multicore processors to implement part of the physical layer in software and employs an FPGA board to transmit optical signal over the wire. As a result, SoNIC allows cross-network-layer research explorations by systems programmers.

We also presented Chupja and MinProbe as examples of a covert channel and an available bandwidth estimation algorithm that benefit from precise pacing and timestamping. Chupja is a PHY covert timing channel that is high-bandwidth, robust, undetectable. Chupja was created by precisely pacing packets to control interpacket gaps at sub-microsecond scale in the physical layer. MinProbe was an available bandwidth estimation algorithm that accurately measure available bandwidth by precisely pacing and examining timestamps of probe packets. As a result, the approach discussed in this chapter demonstrates an advance in the state-of-the-art in network measurements via improved precision of pacing and timestamps.

CHAPTER 4

TOWARDS PRECISE CLOCK SYNCHRONIZATION: DTP

Precisely synchronized clocks are essential for many network and distributed applications. Importantly, an order of magnitude improvement in synchronized precision can improve performance. Synchronized clocks with 100 ns precision allow packet level scheduling of minimum sized packets at a finer granularity, which can minimize congestion in rack-scale systems [49] and in datacenter networks [110]. Moreover, taking a snapshot of forwarding tables in a network requires synchronized clocks [134]. In software-defined networks (SDN), synchronized clocks with microsecond level of precision can be used for coordinated network updates with less packet loss [100] and for real-time synchronous data streams [55]. In distributed systems, consensus protocols like Spanner can increase throughput with tighter synchronization precision bounds on TrueTime [48]. In this chapter, we demonstrate that the precision of synchronized clocks can significantly improve via access to the physical layer.

Synchronizing clocks with nanosecond level precision is a difficult problem. It is challenging due to the problem of measuring round trip times (RTT) accurately, which many clock synchronization protocols use to compute the time difference between a timeserver and a client. RTTs are prone to variation due to characteristics of packet switching networks: Network jitter, packet buffering and scheduling, asymmetric paths, and network stack overhead. As a result, any protocol that relies on RTTs must carefully handle measurement errors.

In this chapter, we present the DTP, Datacenter Time Protocol, which improves the precision of synchronized clocks by running the protocol in the physical layer. In particular, DTP provides nanosecond precision in hardware and

tens of nanosecond precision in software, and at virtually no cost to the datacenter network (i.e. no protocol message overhead). DTP achieved better precision than other protocols and provided strong bounds on precision: By running in the physical layer of a network stack, it eliminated non-determinism from measuring RTTs and it introduced zero Ethernet packets on the network. It was decentralized and synchronized clocks of every network device in a network including network interfaces and switches.

In practice, in a 10 Gbps network, DTP achieved a bounded precision of 25.6 nanoseconds between any directly connected nodes, and 153.6 nanoseconds within an entire datacenter network with six hops at most between *any* two nodes, which is the longest distance in a Fat-tree [36] (i.e. no two nodes [clocks] would differ by more than 153.6 nanoseconds). In software, a DTP daemon could access its DTP clock with usually better than $4T$ nanosecond precision resulting in an end-to-end precision better than $4TD + 8T$ nanoseconds where D was the longest distance between any two servers in a network in terms of number of hops and T was the period of the fastest clock ($\approx$ 6.4ns). DTP's approach applies to full-duplex Ethernet standards such as 1, 10, 40, 100 Gigabit Ethernet. It did require replacing network devices to support DTP running in the physical layer of the network. But, it could be incrementally deployed via DTP-enabled racks and switches. Further, incrementally deployed DTP-enabled racks and switches could work together and enhance other synchronization protocols such as Precise Time Protocol (PTP) [18] and Global Positioning System (GPS) by distributing time with bounded nanosecond precision within a rack or set of racks without any load on the network.

In this chapter, we make three contributions:

- We provide precise clock synchronization at nanosecond resolution with bounded precision in hardware and tens of nanosecond precision in software via access to the PHY.
- We demonstrate that DTP works in practice. DTP can synchronize all devices in a datacenter network.
- We evaluated PTP as a comparison. PTP did not provide bounded precision and was affected by configuration, implementation, and network characteristics such as load and congestion.

4.1 Design

In this section, we present the DTP, Datacenter Time Protocol: Assumptions, protocol, and analysis. The design goals for the protocol were the following:

- Internal synchronization with nanosecond precision.
- No network overhead: No packets are required for the synchronization protocol.

4.1.1 Assumptions

We assume, in a 10 Gigabit Ethernet (10 GbE) network, all network devices are driven by oscillators that run at slightly different rates due to oscillator skew, but operate within a range defined by the IEEE 802.3 standard. The standard requires that the clock frequency f_p be in the range of $[f - 0.0001f, f + 0.0001f]$[1]

[1]This is ±100 parts per million (ppm).

where f is 156.25 MHz in 10 GbE (See Section 2.1.1).

We assume that there are no "two-faced" clocks [77] or Byzantine failures which can report different clock counters to different peers.

We further assume that the length of Ethernet cables is bounded and, thus, network propagation delay is bounded. The propagation delay of optic fiber is about 5 nanoseconds per meter (2/3 × the speed of light, which is 3.3 nanoseconds per meter in a vacuum) [71]. In particular, we assume the longest optic fiber inside a datacenter is 1000 meters, and as a result the maximum propagation delay is at most 5 us. Most cables inside a datacenter are 1 to 10 meters as they are typically used to connect rack servers to a Top-of-Rack (ToR) switch; 5 to 50 nanoseconds would be the more common delay.

4.1.2 Protocol

In DTP, every network port (of a network interface or a switch) has a local counter in the physical layer that increments at every clock tick. DTP operates via protocol messages between peer network ports: A network port sends a DTP message timestamped with its current *local* counter to its peer and adjusts its local clock upon receiving a *remote* counter value from its peer. We show that given the bounded delay and frequent resynchronizations, local counters of two peers can be precisely synchronized in Section 4.1.3.

Since DTP operates and maintains local counters in the physical layer, switches play an important role in scaling up the number of network devices synchronized by the protocol. As a result, synchronizing across all the network

Algorithm 3: DTP inside a network port

STATE:

gc : global counter, from Algorithm 4

$lc \leftarrow 0$: local counter, increments at every clock tick

$d \leftarrow 0$: measured one-way delay to **peer** p

TRANSITION:

T0: After the link is established with p

$lc \leftarrow gc$

Send $(Init, lc)$

T1: After receiving $(Init, c)$ from p

Send $(Init\text{-}Ack, c)$

T2: After receiving $(Init\text{-}Ack, c)$ from p

$d \leftarrow (lc - c - \alpha)/2$

T3: After a timeout

Send $(Beacon, gc)$

T4: After receiving $(Beacon, c)$ from p

$lc \leftarrow \max(lc, c + d)$

ports of a switch (or a network device with a multi-port network interface) requires an extra step: DTP needs to synchronize the local counters of all local ports. Specifically, DTP maintains a *global* counter that increments every clock tick, but also always picks the *maximum* counter value between it and all of the local counters.

DTP follows Algorithm 3 to synchronize the local counters between two peers. The protocol runs in two phases: `INIT` and `BEACON` phases.

INIT phase The purpose of the `INIT` phase is to measure the one-way delay

between two peers. The phase begins when two ports are physically connected and start communicating, i.e. when the link between them is established. Each peer measures the one-way delay by measuring the time between sending an `INIT` message and receiving an associated `INIT-ACK` message, i.e. measure RTT, then divide the measured RTT by two (T0, T1, and T2 in Algorithm 3).

As the delay measurement is processed in the physical layer, the RTT consists of a few clock cycles to send / receive the message, the propagation delays of the wire, and the clock domain crossing (CDC) delays between the receive and transmit paths. Given the clock frequency assumption, and the length of the wire, the only non-deterministic part is the CDC. We analyze how they affect the accuracy of the measured delay in Section 4.1.3. Note that α in Transition 2 in Algorithm 3 is there to control the non-deterministic variance added by the CDC (See Section 4.1.3).

BEACON phase During the `BEACON` phase, two ports periodically exchange their local counters for resynchronization (T3 and T4 in Algorithm 3). Due to oscillator skew, the offset between two local counters will increase over time. A port adjusts its local counter by selecting the maximum of the local and remote counters upon receiving a `BEACON` message from its peer. Since `BEACON` messages are exchanged frequently, hundreds of thousands of times a second (every few microseconds), the offset can be kept to a minimum.

Scalability and multi hops Switches and multi-port network interfaces have two to ninety-six ports in a single device that need to be synchronized within the device[2]. As a result, DTP always picks the maximum of all local counters $\{lc_i\}$ as the value for a global counter gc (T5 in Algorithm 4). Then, each port

[2]Local counters of a multi-port device will not always be the same because remote clocks run at different rates. As a result, a multi-port device must synchronize local counters.

Algorithm 4: DTP inside a network device / switch

STATE:

gc: global counter

$\{lc_i\}$: local counters

TRANSITION:

T5: at every clock tick

$gc \leftarrow \max(gc + 1, \{lc_i\})$

transmits the global counter gc in a `BEACON` message (T3 in Algorithm 3).

Choosing the maximum allows any counter to increase monotonically at the same rate and allows DTP to scale: The maximum counter value propagates to all network devices via `BEACON` messages, and frequent `BEACON` messages keep global counters closely synchronized (Section 4.1.3).

Network dynamics When a device is turned on, the local and global counters of a network device are set to zero. The global counter starts incrementing when one of the local counters starts incrementing (i.e., a peer is connected), and continuously increments as long as one of the local counters is incrementing. However, the global counter is set to zero when all ports become inactive. Thus, the local and global counters of a newly joining device are always less than those of other network devices in a DTP network. We use a special `BEACON_JOIN` message in order to make large adjustments to a local counter. This message is communicated after `INIT_ACK` message in order for peers to agree on the maximum counter value between two local counters. When a network device with multiple ports receives a `BEACON_JOIN` message from one of its ports, it adjusts its global clock and propagates `BEACON_JOIN` messages with its new

global counter to other ports. Similarly, if a network is partitioned and later restored, two subnets will have different global counters. When the link between them is re-established, BEACON_JOIN messages allow the two subnets to agree on the same (maximum) clock counter.

Handling failures There are mainly two types of failures that need to be handled appropriately: Bit errors and faulty devices. IEEE 802.3 standard supports a Bit Error Rate (BER) objective of 10^{-12} [19], which means one bit error could happen every 100 seconds in 10 GbE. However, it is possible that a corrupted bit coincides with a DTP message and could result in a big difference between local and remote counters. As a result, DTP ignores messages that contain remote counters off by more than eight (See Section 4.1.3), or bit errors not in the three least significant bits (LSB). Further, in order to prevent bit errors in LSBs, each message could include a parity bit that is computed using three LSBs. As BEACON messages are communicated very frequently, ignoring messages with bit errors does not affect the precision.

Similarly, if one node makes too many *jumps* (i.e. adjusting local counters upon receiving BEACON messages) in a short period of time, it assumes the connected peer is faulty. Given the latency, the interval of BEACON messages, and maximum oscillator skew between two peers, one can estimate the maximum offset between two clocks and the maximum number of jumps. If a port receives a remote counter outside the estimated offset too often, it considers the peer to be faulty and stops synchronizing with the faulty device.

4.1.3 Analysis

The precision of clock synchronization is determined by oscillator skew, interval between resynchronizations, and errors in reading remote clocks [50, 63, 75]. In this section, we analyze DTP to understand its precision in regards to the above factors. In particular, we analyze the bounds on precision (clock offsets) and show the following:

- Bound of two tick errors due to measuring the one-way delay (OWD) during the `INIT` phase.
- Bound of two tick errors due to the `BEACON` interval. The offset of two synchronized peers can be up to two clock ticks if the interval of `BEACON` messages is less than 5000 ticks.
- As a result, the offset of two peers is bound by four clock ticks or $4T$ where T is 6.4ns. In 10 GbE the offset of two peers is bound by 25.6ns.
- Multi hop precision. As each link can add up to four tick errors, the precision is bounded by $4TD$ where 4 is the bound for the clock offset between directly connected peers, T is the clock period and D is the longest distance in terms of the number of hops.

For simplicity, we use two peers p and q, and use T_p (f_p) and T_q (f_q) to denote the period (frequency) of p and q's oscillator. We assume for analysis p's oscillator runs faster than q's oscillator, i.e. $T_p < T_q$ (or $f_p > f_q$).

Two tick errors due to OWD. In DTP, the one-way delay (OWD) between two peers, measured during the `INIT` phase, is assumed to be stable, constant, and symmetric in both directions. In practice, however, the delay can be measured

differently depending on *when* it is measured due to oscillator skew and *how* the synchronization FIFO between the receive and transmit paths interact. Further, the OWD of one path (from p to q) and that of the other (from q to p) might not be symmetric due to the same reasons. We show that DTP still works with good precision despite any errors introduced by measuring the OWD.

Suppose p sends an `INIT` message to q at time t, and the delay between p and q is d clock cycles. Given the assumption that the length of cables is bounded, and that oscillator skew is bounded, the delay is d cycles for both directions. The message arrives at q at $t + T_p d$ (i.e. the elapsed time is $T_p d$). Since the message can arrive in the middle of a clock cycle of q's clock, it can wait up to T_q before q processes it. Further, passing data from the receipt path to the transmit path requires a synchronization FIFO between two clock domains, which can add one more cycle randomly, i.e. the message could spend an additional T_q before it is received. Then, the `INIT-ACK` message from q takes $T_q d$ time to arrive at p, and it could wait up to $2T_p$ before p processes it. As a result, it takes up to a total of $T_p d + 2T_q + T_q d + 2T_p$ time to receive the `INIT-ACK` message after sending an `INIT` message. Thus, the measured OWD, d_p, at p is,

$$d_p \leq \lfloor \frac{T_p d + 2T_q + T_q d + 2T_p}{T_p} \rfloor / 2 = d + 2$$

In other words, d_p could be one of d, $d + 1$, or $d + 2$ clock cycles depending on when it is measured. As q's clock is slower than p, the clock counter of q cannot be larger than p. However, if the measured OWD, d_p, is larger than the actual OWD, d, then p will think q is faster and adjust its offset more frequently than necessary (See Transition $T4$ in Algorithm 3). This, in consequence, causes the global counter of the network to go faster than necessary. As a result, α in T2 of Algorithm 3 is introduced.

Setting α to be 3 allows d_p to be always less than d. In particular, d_p will be $d-1$ or d; however, d_q will be $d-2$ or $d-1$. Fortunately, a measured delay of $d-2$ at q does not make the global counter go faster, but it can increase the offset between p and q to be two clock ticks most of the time, which will result in q adjusting its counter by one only when the actual offset is two.

Two tick errors due to the `BEACON` interval. The `BEACON` interval, period of resynchronization, plays a significant role in bounding the precision. We show that a `BEACON` interval of less than 5000 clock ticks can bound the clock offset to two ticks between peers.

Let $C_p(X)$ be a clock that returns a real time t at which $c_p(t)$ changes to X. Note that the clock is a discrete function. Then, $c_p(t) = X$ means, the value of the clock is stably X at least after $t - T_p$, i.e. $t - T_p < C_p(X) \leq t$.

Suppose p and q are synchronized at time t_1, i.e. $c_p(t_1) = c_q(t_1) = X$. Also suppose $c_p(t_2) = X + \Delta P$, and $c_q(t_2) = X + \Delta Q$ at time t_2, where ΔP is the difference between two counter values of clock p at time t_1 and t_2. Then,

$$t_2 - T_p < C_p(X + \Delta P) = C_p(X) + \Delta P T_p \leq t_2$$

$$t_2 - T_q < C_q(X + \Delta Q) = C_q(X) + \Delta Q T_q \leq t_2$$

Then, the offset between two clocks at t_2 is,

$$\Delta t(f_p - f_q) - 2 < \Delta P - \Delta Q < \Delta t(f_p - f_q) + 2$$

where $\Delta t = t_2 - t_1$.

Since the maximum frequency of a NIC clock oscillator is $1.0001f$, and the minimum frequency is $0.9999f$, $\Delta t(f_p - f_q)$ is always smaller than 1 if Δt is less than 32 us. As a result, $\Delta P - \Delta Q$ can be always less than or equal to 2, if the

interval of resynchronization (Δt) is less than 32 us ($\approx$ 5000 ticks). Considering the maximum latency of the cable is less than 5 us ($\approx$ 800 ticks), a beacon interval less than 25 us ($\approx$ 4000 ticks) is sufficient for any two peers to synchronize with 12.8 ns (= 2 ticks) precision.

Multi hop Precision. Note that DTP always picks the maximum clock counter of all nodes as the global counter. All clocks will always be synchronized to the fastest clock in the network, and the global counter always increases monotonically. Then, the maximum offset between any two clocks in a network is between the fastest and the slowest. As discussed above, any link between them can add at most two offset errors from the measured delay and two offset errors from BEACON interval. Therefore, the maximum offset within a DTP-enabled network is bounded by $4TD$ where D is the longest distance between any two nodes in a network in terms of number of hops, and T is the period of the clock as defined in the IEEE 802.3 standard ($\approx 6.4ns$).

4.2 Implementation

4.2.1 DTP-enabled PHY

The control logic of DTP in a network port consists of Algorithm 3 from Section 4.1 and a local counter. The local counter is a 106-bit integer (2×53 bits) that increments at every clock tick (6.4 ns = 1/156.25 MHz), or is adjusted based on received BEACON messages. Note that the same oscillator drives all modules in the PCS sublayer on the transmit path and the control logic that increments the local counter. i.e. they are in the same clock domain. As a result, the DTP

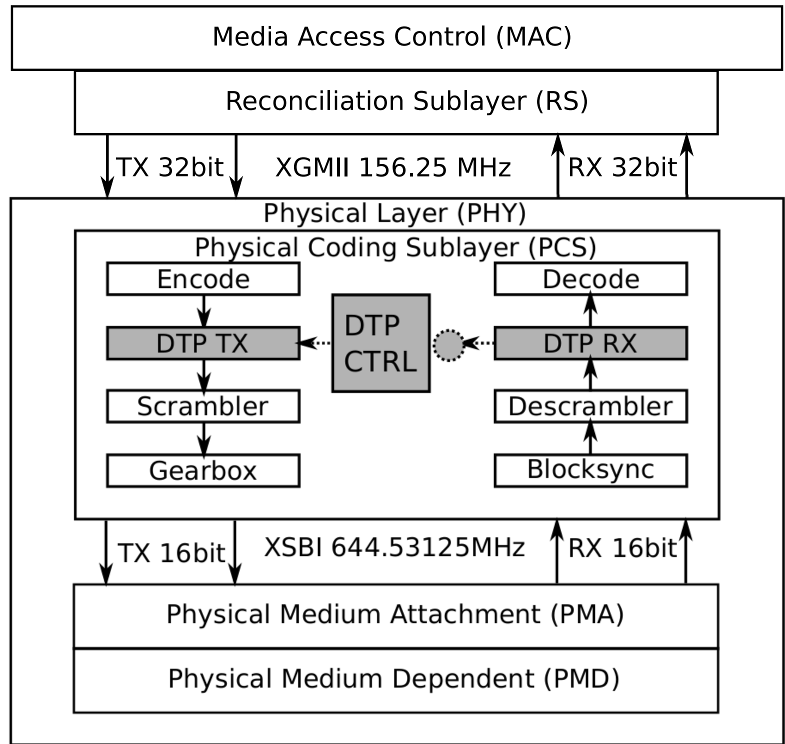


Figure 4.1: Low layers of a 10 GbE network stack. Grayed rectangles are DTP sublayers, and the circle represents a synchronization FIFO.

sublayer can easily insert the local clock counter into a protocol message with no delay.

The DTP-enabled PHY is illustrated in Figure 4.1. Figure 4.1 is exactly the same as the PCS from the standard, except that Figure 4.1 has DTP control, TX DTP, and RX DTP sublayers shaded in gray. Specifically, on the transmit path, the TX DTP sublayer inserts protocol messages, while, on the receive path, the RX DTP sublayer processes incoming protocol messages and forwards them to the control logic through a synchronization FIFO. After the RX DTP sublayer receives and uses a DTP protocol message from the Control block (`/E/`), it replaces the DTP message with idle characters (`/I/`s, all 0's) as required by the standard such that higher network layers do not know about the existence of

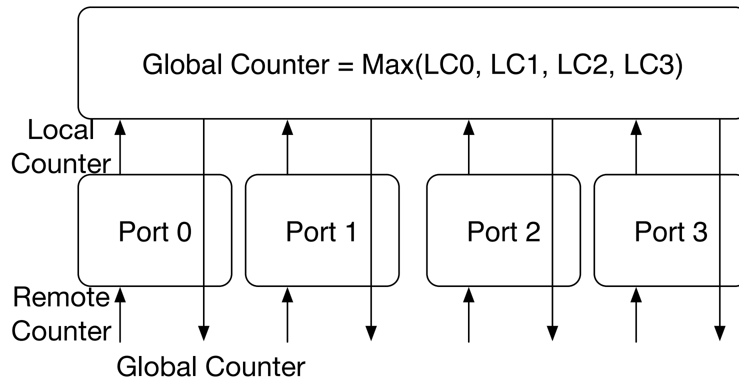


Figure 4.2: DTP enabled four-port device.

the DTP sublayer. Lastly, when an Ethernet frame is being processed in the PCS sublayer in general, DTP simply forwards blocks of the Ethernet frame unaltered between the PCS sublayers.

4.2.2 DTP-enabled network device

A DTP-enabled device (Figure 4.2) can be implemented with additional logic on top of the DTP-enabled ports. The logic maintains the 106-bit global counter as shown in Algorithm 4, which computes the maximum of the local counters of all ports in the device. The computation can be optimized with a tree-structured circuit to reduce latency, and can be performed in a deterministic number of cycles. When a switch port tries to send a `BEACON` message, it inserts the global counter into the message, instead of the local counter. Consequently, all switch ports are synchronized to the same global counter value.

4.2.3 Protocol messages

DTP uses /I/s in the /E/ control block to deliver protocol messages. There are eight seven-bit /I/s in an /E/ control block, and, as a result, 56 bits total are available for a DTP protocol message per /E/ control block. Modifying control blocks to deliver DTP messages does not affect the physics of a network interface since the bits are scrambled to maintain DC balance before sending on the wire (See the scrambler/descrambler in Figure 4.1). Moreover, using /E/ blocks do not affect higher layers since DTP replaces /E/ blocks with required /I/s (zeros) upon processing them.

A DTP message consists of a three-bit message type, and a 53-bit payload. There are five different message types in DTP: INIT, INIT-ACK, BEACON, BEACON-JOIN, and BEACON-MSB. As a result, three bits are sufficient to encode all possible message types. The payload of a DTP message contains the local (global) counter of the sender. Since the local counter is a 106-bit integer and there are only 53 bits available in the payload, each DTP message carries the 53 least significant bits of the counter. In 10 GbE, a clock counter increments at every 6.4 ns (=1/156.25MHz), and it takes about 667 days to overflow 53 bits. DTP occasionally transmits the 53 most significant bits in a BEACON-MSB message in order to prevent overflow.

As mentioned in Section 2.1.1, it is always possible to transmit one protocol message after/before an Ethernet frame is transmitted. This means that when the link is fully saturated with Ethernet frames DTP can send a BEACON message every 200 clock cycles (≈ 1280 ns) for MTU-sized (1522B) frames[3] and 1200 clock cycles (≈ 7680 ns) at worst for jumbo-sized (≈9kB) frames. The PHY re-

[3] It includes 8-byte preambles, an Ethernet header, 1500-byte payload and a checksum value.

quires about 191 66-bit blocks and 1,129 66-bit blocks to transmit a MTU-sized or jumbo-sized frame, respectively. This is more than sufficient to precisely synchronize clocks as analyzed in Section 4.1.3 and evaluated in Section 4.3. Further, DTP communicates frequently when there are no Ethernet frames, e.g every 200 clock cycles, or 1280 ns: The PHY continuously sends `/E/` when there are no Ethernet frames to send.

4.2.4 DTP Software Clock

The *DTP daemon* implements a software clock where applications can access the DTP counter. The value of a software clock should not change dramatically (discontinue) or go back in time. As a result, a DTP daemon adjusts the *rate* of the DTP software clock carefully in order not to introduce discontinuity. A DTP daemon regularly (e.g., once per second) reads the DTP counter of a network interface card via a memory-mapped IO in order to minimize errors in reading the counter. Further, Time Stamp Counters (`TSC` counters) are employed to estimate the frequency of the DTP counter. A `TSC` counter is a reliable and stable source to implement software clocks [108, 123, 53]. Modern systems support *invariant* `TSC` counters that are not affected by CPU power states [20].

The top part of Algorithm 5 illustrates the polling. In order to read both the `TSC` counter and DTP counter reliably, the daemon attempts to read counters with interrupts disabled. Then, the daemon computes the frequency ratio between two counters (*tmp* in Algorithm 5). Note that although the latency of reading `TSC` counter is small and does not vary much [16], the latency of reading DTP counter could vary a lot mainly because of data communication over PCIe.

Algorithm 5: DTP Daemon

procedure POLLING

disable interrupts

$tsc \leftarrow$**READ** TSC

$dtpc \leftarrow$**READ** DTP

$tmp \leftarrow \frac{tsc - ptsc}{dtpc - pdtpc}$

$r \leftarrow tmp * \beta + r * (1 - \beta)$

$ptsc \leftarrow tsc$

$pdtpc \leftarrow dtpc$

enable interrupts

end procedure

function `get_DTP_counter`

$t \leftarrow$**READ** TSC

return $pdtpc + (t - ptsc) * r + delay$

end function

As a result, the actual ratio the daemon uses for interpolation is smoothed. It is a weighted sum of the previously computed ratio and the newly computed ratio (r in Algorithm 5).

Then, applications can read the DTP clock via a `get_DTP_counter` API that interpolates the DTP counter at any moment using `TSC` counters and the estimated ratio r (The bottom part of Algorithm 5). Similar techniques are used to implement `gettimeofday()`. Note that the latency of reading the DTP counter is not negligible; it can be hundreds of nanoseconds to a few microseconds [43]. As a result, the read counter ($dtpc$) is actually some number of cycles behind the counter from hardware. The latency between the daemon and hardware must be taken into account when `get_DTP_counter` returns an in-

terpolated value, which is shown as *delay* in Algorithm 5. A DTP daemon approximates *delay* by a special `send` instruction that DTP hardware provides. In particular, a DTP daemon sends its estimated DTP time, t_1, to hardware periodically, which is then timestamped (t_2) and sent to its peer by DTP hardware via a DTP protocol message in the physical layer. Upon receiving the message, the receiver daemon computes the offset between $t_2 - t_1$, computes a weighted sum of previous computed offset and newly computed offset, applies moving average to smooth large changes, and reports it back to the original sender. By doing so, the reported delay from the receiver does not change much and allows a DTP daemon to precisely estimate the offset between software and hardware. We demonstrate in the following section that a moving average over 10 samples is sufficient to prevent certain change of *delay*. Although *delay* was stable and did not change often in our deployment, it could still introduce a discontinuity. One approach to prevent a discontinuity from happening is to take *delay* into account when computing the rate of the DTP clock.

Note that DTP counters of each NIC are running at the same rate on every server in a DTP-enabled network and, as a result, software clocks that DTP daemons implement are also tightly synchronized.

4.3 Evaluation

In this section, we attempt to answer following questions:

- *Precision:* In Section 4.1.3, we showed that the precision of DTP is bounded by $4TD$ where D is the longest distance between any two nodes in terms

of number of hops. In this section, we demonstrate and measure that precision was indeed within the $4TD$ bound via a prototype and deployed system.

- *Scalability:* We demonstrate that DTP scaled as the number of hops of a network increases.

Further, we measured the precision of accessing DTP from software and compared DTP against PTP. The DTP sublayer and the 10 GbE PHY were implemented using the Bluespec language [6] and Connectal framework [73].

4.3.1 Methodology

Measuring offsets at nanosecond scale is a very challenging problem. One approach is to let hardware generate pulse per second (PPS) signals and compare them using an oscilloscope. Another approach, which we used, is to measure the precision directly in the PHY. Since we were mainly interested in the clock counters of network devices, we developed a *logging* mechanism in the PHY.

Each leaf node generated and sent a 106-bit log message twice per second to its peer, a DTP switch. DTP switches also generated log messages between each other twice per second. A log message contained a 53-bit estimate of the DTP counter generated by the DTP daemon, t_0, which was then timestamped in the DTP layer with the lower 53-bits of the global counter (or the local counter if it is a NIC). The 53-bit timestamp, t_1, was appended to the original message generated by the DTP daemon, and, as a result, a 106-bit message was generated by the sender. Upon arriving at an intermediate DTP switch, the log message was timestamped again, t_2, in the DTP layer with the receiver's global counter.

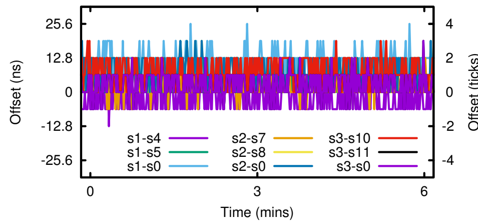
Then, the original 53-bit log message (t_0) and two timestamps (t_1 from the sender and t_2 from the receiver) were delivered to a DTP daemon running on the receiver. By computing $\text{offset}_{hw} = t_2 - t_1 - \text{OWD}$ where OWD was the one-way delay measured in the `INIT` phase, we could estimate the precision between two peers. Similarly, by computing $\text{offset}_{sw} = t_1 - t_0$, we could estimate the precision of a DTP daemon. Note that offset_{hw} included the non-deterministic variance from the synchronization FIFO and offset_{sw} included the non-deterministic variance from the PCIe bus. We could accurately approximate both the offset_{hw} and offset_{sw} with this method.

For PTP, the Timekeeper provided a tool that reported measured offsets between the timeserver and all PTP clients. Note that our Mellanox NICs had PTP hardware clocks (PHC). For a fair comparison against DTP that synchronized clocks of NICs, we used the precision numbers measured from a PHC. Also, note that a Mellanox NIC timestamped PTP packets in the NIC for both incoming and outgoing packets.

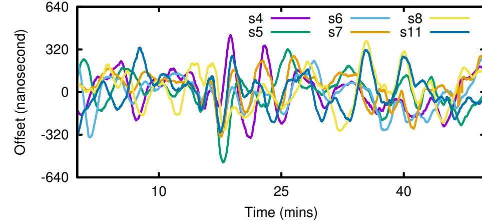
To measure how DTP responded to varying network conditions, we changed the `BEACON` interval during experiments from 200 to 1200 cycles, which changed the Ethernet frame size from 1.5kB to 9kB. Recall that when a link is fully saturated with MTU-sized (Jumbo) packets, the minimum `BEACON` interval possible is 200 (1200) cycles.

4.3.2 Results

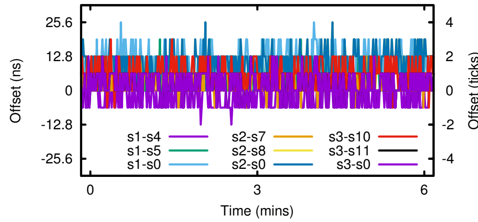
Figure 4.3 and 4.4 show the results: We measured precision of DTP in Figure 4.3a-c, PTP in Figure 4.3d-f, and the DTP daemon in Figure 4.4. For all



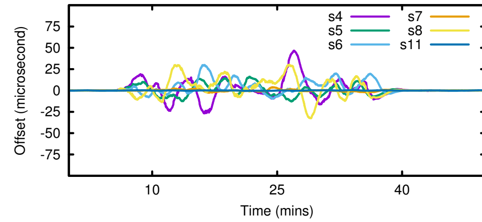
(a) DTP: `BEACON` interval = 200. Heavily loaded with MTU packets.



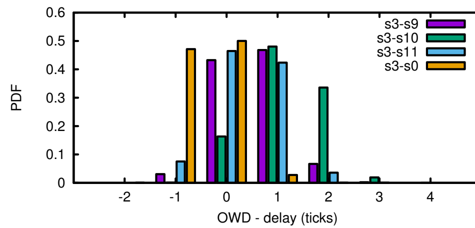
(b) PTP: Idle network



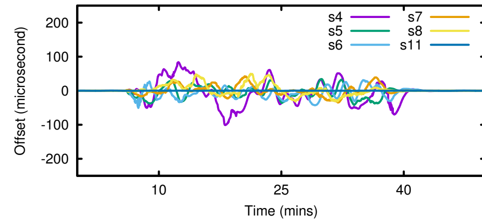
(c) DTP: `BEACON` interval = 1200. Heavily loaded with Jumbo packets.



(d) PTP: Medium loaded



(e) DTP: Offset distribution from S3. (`BEACON` interval = 1200 cycles)

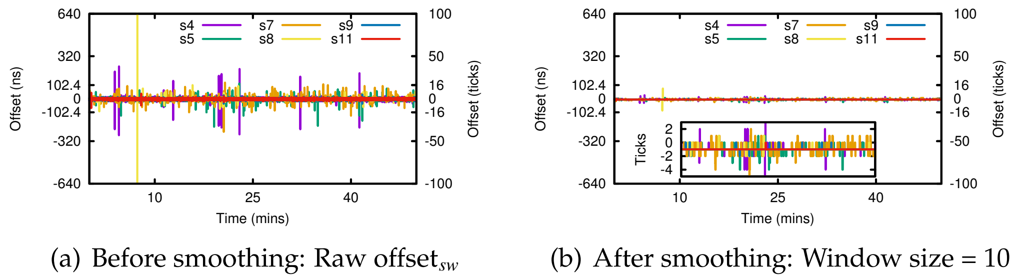


(f) PTP: Heavily loaded

Figure 4.3: Precision of DTP and PTP. A *tick* is 6.4 nanoseconds.

results, we continuously synchronized clocks and measured the precision (clock offsets) over at least a two-day period in Figure 4.3 and at least a few-hour period in Figure 4.4.

Figures 4.3a-b demonstrate that the clock offsets between any two directly connected nodes in DTP never differed by more than four clock ticks; i.e. offsets never differed by more than 25.6 nanoseconds ($4TD = 4 \times 6.4 \times 1 = 25.6$): Figures 4.3a and b show three minutes out of a two-day measurement period and Figure 4.3(e) shows the distribution of the measured offsets with node S3 for



(a) Before smoothing: Raw offset$_{sw}$ (b) After smoothing: Window size = 10

Figure 4.4: Precision of DTP daemon.

the entire two-day period. The network was always under `heavy load` and we varied the Ethernet frame size by varying the `BEACON` interval between 200 cycles in Figure 4.3(a) and 1200 cycles in Figure 4.3(c). DTP performed similarly under `idle` and `medium load`. Since we measured all pairs of nodes and no offset was ever greater than four, the results support that precision was bounded by $4TD$ for nodes D hops away from each other. Figure 4.4 shows the precision of accessing a DTP counter via a DTP daemon: Figure 4.4(a) shows the raw offset$_{sw}$ and Figure 4.4(b) shows the offset$_{sw}$ after applying a moving average algorithm with a window size of 10. We applied the moving average algorithm to smooth the effect of the non-determinism from the PCIe bus, which is shown as occasional spikes. The offset between a DTP daemon in software and the DTP counter in hardware was usually no more than 16 clock ticks ($\approx 102.4ns$) before smoothing, and was usually no more than 4 clock ticks ($\approx 25.6ns$) after smoothing.

Figures 4.3d-f show the measured clock offsets between each node and the grandmaster timeserver using PTP. Each figure shows minutes to hours of a multi-day measurement period, enough to illustrate the precision trends. We varied the load of the network from `idle` (Figure 4.3(b)), to `medium load` where five nodes transmitted and received at 4 Gbps (Figure 4.3(d)), to `heavy`

load where the receive and transmit paths of all links except S11 were fully saturated at 9 Gbps (Figure 4.3(f)). When the network was idle, Figure 4.3(b) shows that PTP often provided hundreds of nanoseconds of precision, which matched literature [14, 34]. When the network was under medium load, Figure 4.3(d) shows the offsets of $S4 \sim S8$ became unstable and reached up to 50 microseconds. Finally, when the network was under heavy load, Figure 4.3(f) shows that the maximum offset degraded to hundreds of microseconds. Note that we measured, but do not report the numbers from the PTP daemon, ptpd, because the precision with the daemon was the same as the precision with the hardware clock, PHC. Also, note that all reported PTP measurements included smoothing and filtering algorithms.

There are multiple takeaways from these results.

1. DTP synchronized clocks more tightly than PTP.
2. The precision of DTP was not affected by network workloads. The maximum offset observed in DTP did not change either when load or Ethernet frame size (the BEACON interval) changed. PTP, on the other hand, was greatly affected by network workloads and the precision varied from hundreds of nanoseconds to hundreds of microseconds depending on the network load.
3. DTP scales. The precision of DTP only depended on the number of hops between any two nodes in the network. The results show that precision (clock offsets) were always bounded by $4TD$ nanoseconds.
4. DTP daemons could access DTP counters with tens of nanosecond precision.

5. DTP synchronized clocks in a short period of time, within two `BEACON` intervals. PTP, however, took about 10 minutes for a client to have an offset below one microsecond. This was likely because PTP needed history to apply filtering and smoothing effectively.
6. PTP's performance was dependent upon network conditions, configuration such as transparent clocks, and implementation.

4.4 Discussion

4.4.1 External Synchronization

We discuss one simple approach that extends DTP to support external synchronization, although there could be many other approaches. One server (either a timeserver or a commodity server that uses PTP or NTP) periodically (e.g., once per second) broadcasts a pair, DTP counter and universal time (UTC), to other servers. Upon receiving consecutive broadcast messages, each DTP daemon estimates the frequency ratio between the received DTP counters and UTC values. Next, applications can read UTC by interpolating the current DTP counter with the frequency ratio in a similar fashion as the method discussed in Section 4.2.4. Again, DTP counters of each NIC are running at the same rate, and as a result, UTC at each server can also be tightly synchronized with some loss of precision due to errors in reading system clocks. It is also possible to combine DTP and PTP to improve the precision of external synchronization further: A timeserver timestamps `sync` messages with DTP counters, and delays between the timeserver and clients are measured using DTP counters.

4.4.2 Incremental Deployment

DTP requires the physical layer to be modified. As a result, in order to deploy DTP, network devices must be modified. As there is usually a single switching chip inside a network device [7], the best strategy to deploy DTP is to implement it inside the switching chip. Then network devices with DTP-enabled switching chips can create a DTP-enabled network. This would require updating the firmware, or possibly replacing the switching chip. PTP uses a similar approach in order to improve precision: PTP-enabled switches have a dedicated logic inside the switching chip for processing PTP packets and PTP-enabled NICs have hardware timestamping capabilities and PTP hardware clocks (PHC). Therefore, the cost of achieving the best configuration of PTP is essentially the same as the cost of deploying DTP, as both require replacing NICs and switches.

An alternative way to deploy DTP is to use FPGA-based devices. FPGA-based NICs and switches [11, 101] have more flexibility of updating firmware. Further, customized PHYs can be easily implemented and deployed with modern FPGAs that are equipped with high-speed transceivers.

One of the limitations of DTP is that it is not possible to deploy DTP on routers or network devices with multiple line cards without sacrificing precision. Network ports on separate line cards typically communicate via a bus interface. As a result, it is not possible to maintain a single global counter with high precision over a shared bus, although each line card can have its own separate global counter. Fortunately, as long as all switches and line cards form a connected graph, synchronization can be maintained.

Replacing or updating switches and NICs in a datacenter at once is not pos-

sible due to both cost and availability. Importantly, DTP can be incrementally deployed: NICs and a ToR switch within the same rack are updated at the same time, and aggregate and core switches are updated incrementally from the lower levels of a network topology. Each DTP-enabled rack elects one server to work as a master for PTP / NTP. Then, servers within the same rack will be tightly synchronized, but servers from different racks are less tightly synchronized depending on the performance of PTP / NTP. When two independently DTP-enabled racks start communicating via a DTP-enabled switch, servers from two racks will be tightly synchronized both internally and externally after communicating `BEACON_JOIN` messages.

4.4.3 Following The Fastest Clock

DTP assumes that oscillators of DTP-enabled devices operate within a range defined by IEEE 802.3 standard (Section 4.1.1). However, in practice, this assumption can be broken, and an oscillator in a network could run at a frequency outside the range specified in the standard. This could lead to many jumps from devices with slower oscillators. More importantly, the maximum offset between two devices could be larger than 4TD. One approach to address the problem is to choose a network device with a reliable and stable oscillator as a master node. Then, through DTP daemons, it is possible to construct a DTP spanning tree using the master node as a root. This is similar to PTP's best master clock algorithm. Next, at each level of the tree, a node uses the remote counter of its parent node as the global counter. If an oscillator of a child node runs faster than its parent node, the local counter of a child should *stall* occasionally in order to keep the local counter monotonically increasing. We leave this design as

Data Rate	Encoding	Data Width	Frequency	Period	Δ
1G	8b/10b	8 bit	125 MHz	8 ns	25
10G	64b/66b	32 bit	156.25 MHz	6.4 ns	20
40G	64b/66b	64 bit	625 MHz	1.6 ns	5
100G	64b/66b	64 bit	1562.5 MHz	0.64 ns	2

Table 4.1: Specifications of the PHY at different speeds

a future work.

4.4.4 What about 1G, 40G or 100G?

In this chapter we discussed and demonstrated how we can implement and deploy DTP over a datacenter focusing on 10 GbE links. However, the capacity of links in a datacenter is not homogeneous. Servers can be connected to Top-of-Rack switches via 1 Gbps links, and uplinks between switches and routers can be 40 or 100 Gbps. Nonetheless, DTP is still applicable to these cases because the fundamental fact still holds: Two physically connected devices in high-speed Ethernet (1G and beyond) are already synchronized to transmit and receive bitstreams. The question is how to modify DTP to support thousands of thousands of devices with different link capacities.

DTP can be extended to support 40 GbE and 100 GbE in a straight forward manner. The clock frequency required to operate 40 or 100 Gbps is multiple of that of 10 Gbps (Table 4.1). In fact, switches that support 10 Gbps and beyond normally use a clock oscillator running at 156.25 MHz to support all ports [27]. As a result, incrementing clock counters by different values depending on the link speed is sufficient. In particular, see the last column of Table 4.1, if a counter tick represents 0.32 nanoseconds, then DTP will work at 10, 40, and 10GbE by

adjusting a counter value to match the corresponding clock period (i.e. $20 \times 0.32 = 6.4$ ns, $5 \times 0.32 = 1.6$ ns, and $2 \times 0.32 = 0.64$ ns, respectively).

Similarly, DTP can be made to work with 1 GbE by incrementing the counter of a 1 GbE port by 25 at every tick (see the last column of Table 4.1). However, the PHY of 1 Gbps is different, it uses a 8b/10b encoding instead of a 64b/66b encoding, and we need to adapt DTP to send clock counter values with the different encoding.

4.5 Summary

Precisely synchronizing clocks is not trivial. In this chapter, we demonstrated that access to the physical layer can improve the precision of synchronized clocks. In particular, DTP exploited the fundamental fact that two physically connected devices are already synchronized to transmit and receive bitstreams. DTP synchronized clocks of network components at tens of nanoseconds of precision, scaled up to synchronize an entire datacenter network, and was accessed from software with usually better than twenty five nanosecond precision. As a result, the end-to-end precision was the precision from DTP in the network (i.e. 25.6 nanoseconds for directly connected nodes and 153.6 nanoseconds for a datacenter with six hops) plus fifty nanosecond precision from software. The approach demonstrated in this chapter for precise synchronized clocks represents an advance in the state-of-the-art.

CHAPTER 5
RELATED WORK

In this dissertation, we first investigated how to improve the precision of timestamping and pacing via an augmented physical layer that provides timing information. Related work that used physics equipment and FPGA boards for network measurements inspired precise timestamping and pacing. Although there are other efforts to provide network programmability or precise timestamping, their approaches do not allow the physical layer to be programmable or provide reliable packet timestamps and pacing. We also researched how to precisely synchronize clocks in a network. Existing clock synchronization protocols do not provide the same level of precision as DTP which provides bounded precision regardless of network conditions. In this chapter, we discuss work related to our contributions.

5.1 Programmable Network Hardware

Programmable network hardware allows for the experimentation of novel network system architectures. Previous studies on reconfigurable NICs [128] showed that programmable network hardware is useful for exploring new I/O virtualization technique in VMMs. NetFPGA [91, 136] allows users to experiment with FPGA-based routers and switches for research in new network protocols and intrusion detection [39, 45, 103, 133]. Further, there are specialized NICs that are programmable and support P4 language [28] which is a dataplane programming language. In P4, forwarding elements such as switches or NICs perform user-defined actions such as modifying headers, discarding, or

forwarding on packets whose header fields match to flow tables. Netronome's FlowNIC [13] and Xilinx's SDNet [29] support P4. While programmable NICs allow users to access the layer 2 and above, SoNIC allows users to access the PHY. This means that users can access the entire network stack in software using SoNIC.

5.2 Hardware Timestamping

The importance of timestamping has long been established in the network measurement community. Prior work either does not provide precise enough timestamping, or requires special devices. Packet stamping in user-space or kernel suffers from the imprecision introduced by the OS layer [52]. Many commodity NICs support hardware timestamping with various levels of accuracy ranging from nanoseconds [10, 24, 11] to hundreds of nanoseconds [25]. Further, NICs can be combined with GPS receiver or PTP-capability to use reference time for timestamping. However, as we discussed in Section 2.1.2, hardware timestamping could still be imprecise.

Although BiFocals [58] is able to provide an exact timestamping, it has limitations that prevented it from being a portable and realtime tool. BiFocals can store and analyze only a few milliseconds worth of a bitstream at a time due to the small memory of an oscilloscope. Furthermore, it requires thousands of CPU hours for converting raw optic waveforms to packets. Lastly, the physics equipment used by BiFocals is expensive and not easily portable. Its limitations motivated us to design SoNIC to achieve the realtime exact precision timestamping. Note that both BiFocals and SoNIC only support delta timestamping.

However, combining SoNIC and DTP can achieve fine-grained timestamping that is synchronized with other network devices.

5.3 Software Defined Radio

The Software Defined Radio (SDR) allows easy and rapid prototyping of wireless network in software. Open-access platforms such as the Rice University's WARP [37] allow researchers to program both the physical and network layer on a single platform. Sora [122] presented the first SDR platform that fully implements IEEE 802.11b/g on a commodity PC. AirFPGA [133] implemented a SDR platform on NetFPGA, focusing on building a chain of signal processing engines using commodity machines. SoNIC is similar to Sora in that it allows users to access and modify the PHY and MAC layers. The difference is that SoNIC must process multiple 10 Gbps channels which is much more computationally intensive than the data rate of wireless channels. Moreover, it is harder to synchronize hardware and software because a 10 GbE link runs in a full duplex mode, unlike a wireless network.

5.4 Software Router

Although SoNIC is orthogonal to software routers, it is worth mentioning software routers because they share common techniques. SoNIC pre-allocates buffers to reduce memory overhead [65, 112], polls huge chunks of data from hardware to minimize interrupt overhead [54, 65], packs packets in a similar fashion to batching to improve performance [54, 65, 95, 112]. Software routers

	Precision	Scalability	Overhead (pckts)	Extra hardware
NTP	us	Good	Moderate	None
PTP	sub-us	Good	Moderate	PTP-enabled devices
GPS	ns	Bad	None	Timing signal receivers, cables
DTP	ns	Good	None	DTP-enabled devices

Table 5.1: Comparison between NTP, PTP, GPS, and DTP

normally focus on scalability and hence exploit multi-core processors and multi-queue supports from NICs to distribute packets to different cores for packet processing. On the other hand, SoNIC pipelines multiple CPUs to handle continuous bitstreams.

5.5 Clock Synchronization

In this section, we describe widely-used clock synchronization protocols, NTP, PTP, GPS, and some other protocols.

Network Time Protocol (NTP)

The most commonly used time synchronization protocol is the Network Time Protocol (NTP) [98]. NTP provides millisecond precision in a wide area network (WAN) and microsecond precision in a local area network (LAN). In NTP, time servers construct a tree, and top-level servers (or stratum 1) are connected to a reliable external time source (stratum 0) such as satellites through a GPS receiver, or atomic clocks. A client communicates with one of the time servers via UDP packets. As mentioned in Section 2.1.3, four timestamps are used to account for processing time in the time server.

NTP is not adequate for a datacenter. It is prone to errors that reduce precision in clock synchronization: Inaccurate timestamping, software network stack (UDP daemon), and network jitter. Furthermore, NTP assumes symmetric paths for time request and response messages, which is often not true in reality. NTP attempts to reduce precision errors via statistical approaches applied to network jitter and asymmetric paths. Nonetheless, the precision in NTP is still low.

Precise Time Protocol (PTP)

The IEEE 1588 Precise Time Protocol (PTP) [18][1] is an emerging time synchronization protocol that can provide tens to hundreds of nanosecond precision in a LAN when properly configured. PTP picks the most accurate clock in a network to be the *grandmaster* via the best master clock algorithm and others synchronize to it. The grandmaster could be connected to an external clock source such as a GPS receiver or an atomic clock. Network devices including PTP-enabled switches form a tree with the grandmaster as the root. Then, at each level of the tree, a server or switch behaves as a slave to its parent and a master to its children. When PTP is combined with Synchronous Ethernet, which syntonizes frequency of clocks (SyncE, See below), PTP can achieve sub-nanosecond precision in a carefully configured environment [86], or hundreds of nanoseconds with tens of hops in back-haul networks [84].

The protocol normally runs as follows: The grandmaster periodically sends timing information (`Sync`) with IP multicast packets. Upon receiving a `Sync` message which contains time t_0, each client sends a `Delay_Req` message to the timeserver, which replies with a `Delay_Res` message. The mechanism of com-

[1] We use PTPv2 in this discussion.

municating `Delay_Req` and `Delay_Res` messages is similar to NTP and Figure 2.3. Then, a client computes the offset and adjusts its clock or frequency. If the timeserver is not able to accurately embed t_0 in the `Sync` message, it emits a `Follow_Up` message with t_0, after the `Sync` message, to everyone.

To improve the precision, PTP employs a few techniques. First, PTP-enabled network switches can participate in the protocol as *Transparent clocks* or *Boundary clocks* in order to eliminate switching delays. Transparent clocks timestamp incoming and outgoing packets, and correct the time in `Sync` or `Follow_Up` to reflect switching delay. Boundary clocks are synchronized to the timeserver and work as masters to other PTP clients, and thus provide scalability to PTP networks. Second, PTP uses hardware timestamping in order to eliminate the overhead of network stack. Modern PTP-enabled NICs timestamp both incoming and outgoing PTP messages [24]. Third, a PTP-enabled NIC has a PTP hardware clock (PHC) in the NIC, which is synchronized to the timeserver. Then, a PTP-daemon is synchronized to the PHC [47, 105] to minimize network delays and jitter. Lastly, PTP uses smoothing and filtering algorithms to carefully measure one way delays.

As we demonstrate in Section 4.3, the precision provided by PTP is about few hundreds of nanoseconds at best in a 10 GbE environment, and it can change (decrease) over time even if the network is in an idle state. Moreover, the precision could be affected by the network condition, i.e. variable and/or asymmetric latency can significantly impact the precision of PTP, even when cut-through switches with priority flow control are employed [131, 132]. Lastly, it is not easy to scale the number of PTP clients. This is mainly due to the fact that a timeserver can only process a limited number of `Delay_Req` messages per sec-

ond [18]. Boundary and Transparent clocks can potentially solve this scalability problem. However, precision errors from Boundary clocks can be cascaded to low-level components of the timing hierarchy tree, and can significantly impact the precision overall [70]. Further, it is shown that Transparent clocks often are not able to perform well under network congestion [132], although a correct implementation of Transparent clocks should not degrade the performance under network congestion.

Global Positioning System (GPS)

In order to achieve nanosecond-level precision, GPS can be employed [10, 48]. GPS provides about 100 nanosecond precision in practice [83]. Each server can have a dedicated GPS receiver or can be connected to a time signal distribution server through a dedicated link. As each device is directly synchronized to satellites (or atomic clocks) or is connected via a dedicated timing network, network jitter and software network stack is not an issue.

Unfortunately, GPS based solutions are not realistic for an entire datacenter. It is not cost effective and scalable because of extra cables and GPS receivers required for time signals. Further, GPS signals are not always available in a datacenter as GPS antennas must be installed on a roof with a clear view to the sky. However, GPS is often used in concert with other protocols such as NTP and PTP and also DTP.

Other Clock Synchronization Protocols

Because NTP normally does not provide precise clock synchronization in a local area network (LAN), much of the literature has focused improving NTP without extra hardware. One line of work was to use `TSC` instructions to implement precise software clocks called TSCclock, and later called RADclock [53, 108, 123]. It was designed to replace `ntpd` and `ptpd` (daemons that run NTP or PTP) and provide sub-microsecond precision without any extra hardware support. Other software clocks include Server Time Protocol (STP) [104], Coordinated Cluster Time (CCT) [59], AP2P [118], and skewless clock synchronization [94], which provide microsecond precision.

Implementing clock synchronization in hardware has been demonstrated by Fiber Channel (FC) [12] and discussed by Kopetz and Ochsenreiter [75]. FC embeds protocol messages into interpacket gaps similar to DTP. However, it is not a decentralized protocol and the network fabric simply forwards protocol messages between a server and a client using physical layer encodings. As a result, it does not eliminate non-deterministic delays in delivering protocol messages.

Frequency Synchronization

Synchronous optical networks (SONET/SDH) is a standard that transmits multiple bitstreams (such as Voice, Ethernet, TCP/IP) over an optical fiber. In order to reduce buffering of data between network elements, SONET requires precise frequency synchronization (i.e., *syntonization*). An atomic clock is commonly deployed as a Primary Reference Clock (PRC), and other network elements are synchronized to it either by external timing signals or by recovering clock sig-

nals from incoming data. DTP does not synchronize frequency of clocks, but values of clock counters.

Synchronous Ethernet (SyncE) [23] was introduced for reliable data transfer between synchronous networks (e.g. SONET/SDH) and asynchronous networks (e.g. Ethernet). Like SONET, it synchronizes the frequency of nodes in a network, not clocks (i.e. *syntonization*). It aims to provide a synchronization signal to all Ethernet network devices. The idea is to use the recovered clock from the receive (RX) path to drive the transmit (TX) path such that both the RX and TX paths run at the same clock frequency. As a result, each Ethernet device uses a phase locked loop to regenerate the synchronous signal. As SyncE itself does not synchronize clocks in a network, PTP is often employed along with SyncE to provide tight clock synchronization. One such example is White Rabbit which we discuss below.

Frequency and Clock Synchronization

White Rabbit [101, 80, 86] has by far the best precision in packet-based networks. The goal of White Rabbit (WR) [101] was to synchronize up to 1000 nodes with sub-nanosecond precision. It uses SyncE to syntonize the frequency of clocks of network devices and WR-enabled PTP [80] to embed the phase difference between a master and a slave into PTP packets. WR demonstrated that the precision of a non-disturbed system was 0.517ns [86]. WR also requires WR-enabled switches and synchronizes slaves that are up to four-hops apart from the timeserver. WR works on a network with a tree topology and with a limited number of levels and servers. Furthermore, it supports 1 Gigabit Ethernet only and it is not clear how WR behaves under heavy network loads as it uses PTP packets.

DTP does not rely on any specific network topology, and can be extended to protocols with higher speeds.

Similarly, BroadSync [42] and ChinaMobile [84] also combine SyncE and PTP to provide hundreds of nanosecond precision. The Data Over Cable Service Interface Specification (DOCSIS) is a frequency synchronized network designed to time divide data transfers between multiple cable modems (CM) and a cable modem termination system (CMTS). The DOCSIS time protocol [46] extends DOCSIS to synchronize time by approximating the internal delay from the PHY and asymmetrical path delays between a reference CM and the CMTS. We expect that combining DTP with frequency synchronization, SyncE, will also improve the precision of DTP to sub-nanosecond precision as it becomes possible to minimize or remove the variance of the synchronization FIFO between the DTP TX and RX paths.

Fault Tolerant Clock Synchronization

Synchronizing clocks in the presence of failures has also been studied in distributed systems [51, 64, 77, 78, 92, 116, 119] where processes could fail or show Byzantine behaviors including reporting different clock values to different processes ("two-faced" clocks). These protocols can tolerate some number of faulty processes when there are enough number of non-faulty processes by periodically communicating their time with each other and, as a result, converging to a common time. These protocols assume upper limit on message transmission delays when reading other processes' clock to converge, use different convergence functions and thus provide different levels of precision. There are also some other protocols without upper bounds on message delivery that use timeouts

to detect failures [63, 96]. Cristian [50] discussed an external synchronization between a time server and multiple clients and used the characteristics of the distribution of delays to deal with network jitter.

CHAPTER 6

FUTURE WORK

In this dissertation, we described two systems that provide three fundamental capabilities for network measurements: Precise timestamping, pacing, and clock synchronization. Further, we illustrated how they can improve network applications such as creating and detecting covert timing channels and estimating available bandwidth. In the future, we plan to perform network measurements to understand the behavior of NICs, switches, routers and distributed systems and improve the performance of network applications that can benefit from precise timestamping, pacing and clock synchronization.

Further, there are many interesting research directions. First, it is possible to combine DTP with Synchronous Ethernet (SyncE) to achieve better precision. Second, DTP can be used to implement a packet scheduler that achieves high-throughput and low-latency packet flows. Lastly, by combining SoNIC, DTP and programmable dataplane (P4), we can build a network measurement tool with customizable packet processing algorithms. We explore each idea in this chapter.

6.1 Synchronous DTP

As briefly discussed in Chapter 5, the goal of SyncE is to synchronize the *frequency* of clocks, and the goal of DTP and PTP is to synchronize the *time* of clocks. WhiteRabbit combines SyncE and PTP and achieves sub-nanosecond precision in a carefully configured environment. Similarly, it is possible to com-

bine DTP with SyncE. Using SyncE essentially means that the synchronization FIFO between the DTP RX path and the DTP TX path is no longer necessary, and it will allow clocks to be synchronized precisely at sub-nanosecond scale as in WhiteRabbit. Implementing SyncE on an FPGA board requires a high-performance Phase Locked Loop (PLL) for jitter attenuation, which is not readily available in current FPGA development boards. As a result, it is required to carefully design a new FPGA board to combine SyncE and DTP.

6.2 Packet Scheduler

DTP provides a globally synchronized time among network devices including NICs and switches and SoNIC provides precise timestamping and pacing. Further, DTP can also provide precise RTT and lower bound on OWD between two nodes in a network. The properties that DTP and SoNIC provide can be used to implement various types of packet schedulers in a network. First, a centralized controller can control when an end-host can send a packet through the network. Fastpass [110] showed that this approach could improve the throughput of TCP streams and reduce the switch queue size in a datacenter network. Fastpass relies on PTP and software-based pacing algorithm [121], which provides microsecond-scale precision while pacing packets. We expect that using DTP and SoNIC can potentially allow more fine-grained control and achieve a better throughput. Second, coordination-free time-division scheduling can be implemented where each physical layer of participating nodes can transmit one packet to one destination per time slot. This type of scheduling could be useful for rack-scale systems where thousands of micro-servers are connected via simple switches without buffers. Lastly, precise RTT could lead to high throughput

and low latency in transport layer protocols (TCP) in a datacenter network [99]. TIMELY uses RTT as an indicator of congestion in a network and it exploits hardware support from NICs to achieve it. Similarly, TCP can use OWD offered by DTP and pacing offered by SoNIC for congestion control.

6.3 SoNIC, DTP, and P4

Testing network devices or network protocols is not easy. For example, measuring packet processing latency of a switch under stress requires a multi-port packet generator that is able to send traffic at line speed with arbitrary packet size, and a multi-port packet capturer that is able to capture traffic at line speed. The generator and capturer could be a single machine or multiple machines that are synchronized. Further, generating protocol specific packets requires programmability in the packet generator. Unfortunately, commercial network testers are expensive and are not readily available for network researchers. Open Source Network Tester (OSNT) [38] is an effort to provide an open-source packet generator and capturer for testing various network components using NetFPGA 10G cards. Multiple OSNT devices could also be synchronized via GPS signal or PTP.

Similarly, we can implement a network tester that combines DTP, SoNIC and P4 [28]. In particular, the physical layer of the tester combines DTP and SoNIC. Note that SoNIC is a software implementation of the physical layer with complete control over every bit. However, achieving the same level of control and access in a hardware implementation of the physical layer is not trivial. It is necessary to define a new set of APIs that will allow users to control interpacket

gaps by specifying the number of `/I/`s and capture interpacket gaps / delays. By doing so, arbitrary traffic pattern could be generated and captured. Further, P4 provides programmability in dataplane. Many protocols that current network switches provide can be expressed using P4 language. As a result, integrating P4 into the network tester can also provide flexibility for testing various protocols.

CHAPTER 7
CONCLUSION

Precise network measurements are fundamental for network research, monitoring, and applications. For instance, if no clock differs by more than 100 nanoseconds compared to 1 microsecond, one-way delay can be measured precisely due to the tight synchronization. Unfortunately, it is challenging to precisely perform network measurements because it is difficult to precisely access and control time in a network of computer systems.

In this dissertation, we investigated how to improve the precision of timestamping, pacing and clock synchronization. In particular, we discussed how to augment the physical layer of a network stack to provide precise timing information. The principle observation was that when two physical layers are connected, each physical layer continuously generates a sequence of bits for clock and bit recovery and for maintaining the link connectivity. Then, controlling and accessing every single bit in the physical layer allows precise timestamping and pacing. Further, precise timestamping and pacing can improve the performance of network applications. Similarly, exploiting the bit-level synchronization in the physical layer allows precisely synchronized clocks with bounded precision. Our approach is an advance in the state-of-the-art in network measurements.

In order to validate the idea of the augmented physical layer, we presented the design and implementation of multiple systems that provide and exploit precise time. SoNIC provides precise timestamping and pacing by implementing the physical layer in software. A covert timing channel, Chupja, and an available bandwidth estimation algorithm, MinProbe, are two network applica-

tions that demonstrate how precise timestamping and pacing can improve the performance of network applications. Chupja can deliver hundreds of thousands of bits per second while avoiding detection, and MinProbe can accurately estimate available bandwidth in a high-speed network. DTP, Datacenter Time Protocol, provides precise clock synchronization by running a clock synchronization protocol in the physical layer. DTP can synchronize clocks in a datacenter network with bounded precision at hundreds of nanosecond scale. Precise access and control of time via approaches exemplified in this dissertation marks an important step towards precise network measurements.

APPENDIX A

PHY TUTORIAL

In this dissertation, we demonstrated that access to the physical layer of a network stack could improve the performance of network measurements. Yet, the physical layer is often considered as a black box and how it works is generally unknown. The tutorial below was designed to help undergraduate students in computer science to understand how the physical layer works by implementing the physical layer in software.

A.1 Physical layer: Encoder and Scrambler

A.1.1 Introduction

According to the IEEE 802.3 standard, the PHY layer of 10 GbE consists of three sublayers: the Physical Coding Sublayer (PCS), the Physical Medium Attachment (PMA) sublayer, and the Physical Medium Dependent (PMD) sublayer (See Figure A.1). The PMD sublayer is responsible for transmitting the outgoing symbolstream over the physical medium and receiving the incoming symbolstream from the medium. The PMA sublayer is responsible for clock recovery and (de-)serializing the bitstream. The PCS performs the blocksync and gearbox (we call this PCS1), scramble/descramble (PCS2), and encode/decode (PCS3) operations on every Ethernet frame.

When Ethernet frames are passed to the PHY layer, they are reformatted before being sent across the physical medium. On the transmit (TX) path, the PCS

encodes every 64-bit of an Ethernet frame into a 66-bit *block* (PCS3), which consists of a two bit *synchronization header* (syncheader) and a 64-bit *payload*. As a result, a 10 GbE link actually operates at 10.3125 Gbaud ($10G \times \frac{66}{64}$). The PCS also scrambles each block (PCS2) and adapts the 66-bit width of the block to the 16-bit width of the PMA interface (PCS1) before passing it down the network stack to be transmitted. The entire 66-bit block is transmitted as a continuous stream of *symbols* which a 10 GbE network transmits over a physical medium (PMA & PMD). On the receive (RX) path, the PCS performs block synchronization based on two-bit syncheaders (PCS1), descrambles each 66-bit block (PCS2) before decoding it (PCS3).

We will implement the TX path of the PCS, especially the encoder and scrambler today and the RX path of the PCS (the decoder and descrambler) tomorrow. Let us illustrate how the PCS sublayer works in detail.

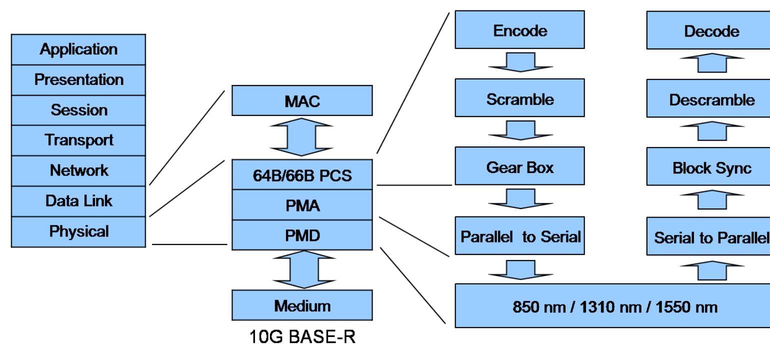


Figure A.1: 10G Network Stack

A.1.2 Encoding

In this section, for better understanding of how 64/66b Encoding works, we will use two sample packets: “Welcome to Ithaca!!!” (P1) and “SoNIC Work-

shop" (P2). For simplicity, we will not discuss preambles, Ethernet headers and checksums.

Once the PCS receives a packet from higher network layers, the packet is reformatted to a sequence of one /S/ block (Start of an Ethernet frame), multiple data blocks, and one /T/ block (End of an Ethernet frame). The PCS inserts /E/ blocks (Idle frame) between packets if necessary. /S/, /T/, and /E/ blocks are called *control blocks* and their syncheaders are always '0x01', while the syncheader of data blocks are always '0x2'. A control block includes a 8-bit block type field which indicates the type of the block such as /S/, /T/ and /E/. /E/ consists of eight *Idle characters*, /I/, which are used to fill the inter-packet gaps. The standard requires that there must be at least twelve /I/s between any packets. Figure A.2 illustrates all possible blocktypes.

Let's assume we picked the fifth row of the table for our /S/ of P1, i.e. /S/ with the blocktype 0x78. Then, the control block will be constructed as the first row of Figure A.3. Notice that byte-ordering is reversed to follow the little-endian format of x86 processors. A data block solely consists of eight data bytes (64bit) with a 0x01 syncheader. Therefore, a data block of " to Itha" will be constructed as the second row of Figure A.3. Finally, as we now have five data bytes left, we can use the eleventh row from Figure A.2 for our /T/ block, i.e. /T/ with the blocktype 0xd2. The rest of the block is filled with /I/s which is a 7-bit long control character (its value is zero). In our case, we can insert two /I/s in the /T/ block (the third row of Figure A.3).

Now, we have to insert at least *twelve* /I/s between two packets to conform to the IEEE 802.3 standard. Interestingly, the blocktype field of a /T/ block is also considered as one /I/ character. Therefore, as we have inserted three /I/s

so far, we just need to insert nine more /I/s. A special control block /E/ can be used to insert eight /I/s (the third row of Figure A.2 and the fourth row of Figure A.3). Since we now have to insert at least one more /I/, we can pick /S/ with 0x33 (the fourth row of Figure A.2) to encode the next packet P2, since this /S/ block contains four /I/s. The final sequence of 66-bit blocks will look like Figure A.3.

	Block Payload									Sync
	63	56	48	40	32	24	16	8	0	
Data Block	D7	D6	D5	D4	D3	D2	D1		D0	10
									Block Type	
/E/	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x1e	01
/S/	D7	D6	D5		0x0	0x0	0x0	0x0	0x33	01
/S/	D7	D6	D5	D4	D3	D2	D1		0x78	01
/T/	0x0	0x0	0x0	0x0	0x0	0x0	0x0		0x87	01
/T/	0x0	0x0	0x0	0x0	0x0	0x0		D0	0x99	01
/T/	0x0	0x0	0x0	0x0	0x0		D1	D0	0xaa	01
/T/	0x0	0x0	0x0	0x0		D2	D1	D0	0xb4	01
/T/	0x0	0x0	0x0		D3	D2	D1	D0	0xcc	01
/T/	0x0	0x0		D4	D3	D2	D1	D0	0xd2	01
/T/	0x0		D5	D4	D3	D2	D1	D0	0xe1	01
/T/	D6	D5	D4	D3	D2	D1	D0		0xff	01

Figure A.2: 66b block format

Block Payload									Sync
63	56	48	40	32	24	16	8	0	
e	m	o	c	l	e	W		0x78	01
a	h	t	l		o	t			10
0x0	0x0		!	!	!	a	c	0xd2	01
0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x1e	01
N	o	S		0x0	0x0	0x0	0x0	0x33	01
s	k	r	o	W		C		l	10
0x0	0x0	0x0	0x0		p	o	h	0xb4	01
0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x1e	01

Figure A.3: Example

A.1.3 Scrambler

The scrambler runs continuously on all block payload while the syncheader bypasses the scrambler. The payload of each block (i.e. 64 bit data) is scrambled

with the following equation:

$$G(x) = 1 + x^{39} + x^{58}$$

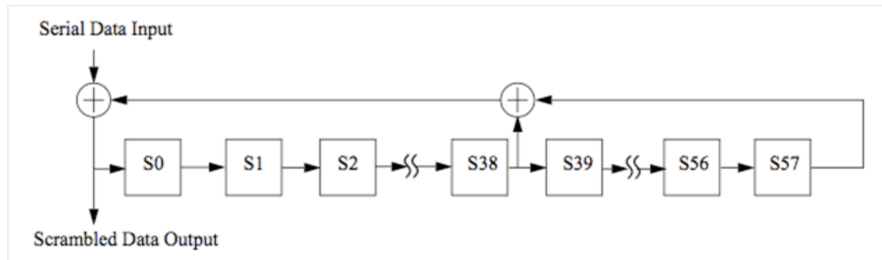


Figure A.4: Scrambler

while x^n is the (n)th least significant bit (LSB) of the internal state of the scrambler, '1' is the input bit, and + is an XOR operation.

To understand what this equation means, let's assume that the initial state of the scrambler is `0x03ff ffff ffff ffff` and the input block is `0xdead beef dead beed`. Notice both x^{58} and x^{39} of the scrambler are '1'. Now, starting from the LSB of the input block, the scrambler takes one bit at a time to calculate the corresponding output bit. For example, the LSB of the input block, '1', is XORed with $x^{58} = 1$ and $x^{39} = 1$ to generate '1', the LSB of the output block. Then, all bits of the scrambler shift to the left by one bit (`0x07ff ffff ffff fffe`), and is XORed with the output of the previous computation '1' (`0x07ff ffff ffff ffff`). x^{58} and x^{39} are again '1' and XORed with the second LSB of the input block, '0', to produce the second LSB of the output block, '0'. The scrambler continues this process until the entire input block is scrambled.

A.1.4 Task

You will implement a black box that functions as a PCS encoder/scrambler today; the black box encodes a sequence of packets given as an input and generates an array of scrambled 66-bit blocks. By default, the black box inserts 12 Idle characters between packets which you can change with "-g" option.

For example,

```
$> ./encode -i packets
 10, 0x0123456789abcdef
   ...
```

We will provide a skeleton source code and you are required to fill at least two functions `scrambler` and `encode` with your partner. In addition, a decoder and sample packets are provided to check the correctness of your implementation.

Scrambler

First, you will need to implement the scrambler. You can check the correctness of your scrambler by running:

```
$> ./encode -d
01, a1fe788405060708
01, 60a77dbee226551e
state = 78aa64477dbee506
```

If your output matches the above, your scrambler is working correctly.

Encoder

Now, you need to implement the Encoder. For each packet, generate /S/, Data blocks, /T/ and /E/ if necessary in sequence. Make sure to insert the correct number of /I/s between packets (If you can not meet the exact number of /I/s, try to minimize the number of them). After implementing the Encoder, try to run the following:

```
$> make fun
```

which will display something to your screen. If you can recognize it, you correctly implemented the encoder.

Note Think carefully about the byte / bit ordering.

A.2 Physical Layer: Decoder and Descrambler

A.2.1 Introduction

Today, we will implement the Decoder and the Descrambler.

A.2.2 Task

Descrambler

First, you will need to implement the descrambler. The descrambler is implemented as illustrated in Figure A.5.

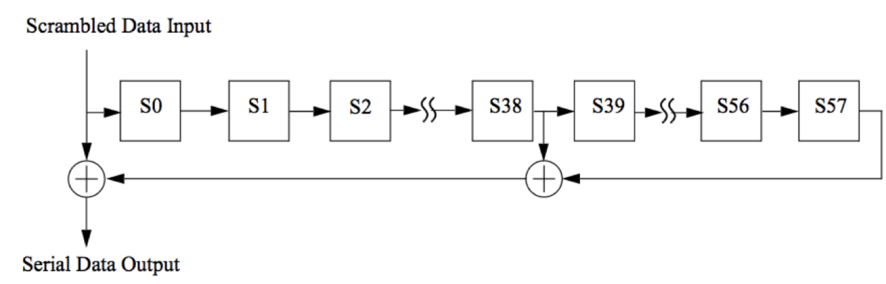


Figure A.5: Descrambler

How does it differ from the scrambler?

To check the correctness of your descrambler, run the following:

```
$> ./decode -d
0102030405060708
090a0b0c0d0e0f00
state = 78aa64477dbee506
```

If your output matches with the above, your descrambler is correct.

Decoder

The decoder does two things: recovers original Ethernet frames and counts the number of /I/s between frames. After descrambling a block, the decoder

should check the syncheader first to distinguish data blocks and control blocks. For control blocks, your decoder must perform appropriate actions based on the block type. As a reminder, an Ethernet frame starts at /S/ block and ends at /T/ block. In addition, the decoder must count the number of /I/s preceding an Ethernet frame. (i.e. start counting it when you see /T/ until /S/ appears) Call `print_decoded` once the Ethernet frame is recovered.

After implementing the Decoder, try to run the following:

```
$> make fun
```

Discussion

Try to run the following:

```
$> make fun IDLE=100
```

What happened? The command above changes the number of /I/s between packets to 100. You can actually see the /I/s by running the following which changes idle characters to ‘x’.

```
$> make fun IDLE=100 IDLEC="x"
```

Run the above command again changing 100 to different values (larger or smaller). What does this imply in terms of throughput?

A.3 Pseudo C code for 64/66b Encoder and Decoder

```
1 struct Frame {
2     uint32_t idles; // # of idles preceding this frame
3     uint8_t data;
4     uint32_t len;
5 };
6
7 uint64_t scrambler (uint64_t *state, uint8_t sh, uint64_t payload)
8 {
9     for ( i = 0 ; i < 64 ; i ++) {
10         in_bit = (payload >> i) & 0x1;
11         out_bit = (in_bit ^ (*state >> 38) ^ (*state >> 57)) & 0x1;
12         *state = (*state << 1) | out_bit;
13         scrambled |= (out_bit << i);
14     }
15
16     // pass syncheader and scrambled to pma
17     return scrambled;
18 }
19
20 uint64_t scrambler_opt (uint64_t *state, uint8_t sh, uint64_t payload)
21 {
22     scrambled = (*state >> 6) ^ (*state >> 25) ^ payload;
23     *state = scrambled ^ (scrambled << 39) ^ (scrambled << 58);
24
25     //pass syncheader and scrambled to pma
26     return scrambled;
27 }
28
29 // Encode a Ethernet Frame
30 void encode (struct Frame *frame, uint64_t *state)
```

```
31 {
32     terminal[8] = {0x87, 0x99, 0xaa, 0xb4, 0xcc, 0xd2, 0xe1, 0xff};
33     data = frame->data;
34     len = frame->len;
35     idles = frame->idles;
36
37     /* /E/ */
38     while (idles >= 8) {
39         scrambler(state, 0x1e);
40         idles -= 8;
41     }
42
43     /* /S/ */
44     tmp = 0;
45     if (idles > 0 && idles <= 4) {
46         block_type = 0x33;
47         tmp = *(uint64_t *) data;
48         tmp <<= 40;
49         tmp |= block_type;
50         data += 3;
51         len -= 3;
52     } else {
53         if (idles != 0)
54             scrambler(state, 0x1, e_frame);
55
56         block_type = 0x78;
57         tmp = *(uint64_t *) data;
58         tmp <<= 8;
59         tmp |= block_type;
60         data += 7;
61         len -= 7;
62     }
```

```
63     state = scrambler(state, 0x1, tmp);
64
65     /* /D/ */
66     while ( len >= 8) {
67         tmp = *(uint64_t *) data ;
68         state = scrambler(state, 0x2, tmp);
69         data += 8;
70         len -= 8;
71     }
72
73     /* /T/ */
74     block_type = terminal[len];
75     tmp = 0;
76     if (len != 0) {
77         tmp = *(uint64_t *) data;
78         tmp <<= (8-len) * 8;
79         tmp >>= (7-len) * 8;
80     }
81     tmp |= block_type;
82
83     state = scrambler(state, 0x1, tmp);
84 }
85
86 struct block66b {
87     uint8_t syncheader;
88     uint64_t payload;
89 };
90
91 uint64_t descrambler (uint64_t *state, uint64_t payload)
92 {
93     for ( i = 0 ; i < 64 ; i ++ ) {
94         in_bit = (payload >> i) & 0x1;
```

```
95          out_bit = (in_bit ^ (*state >> 38) ^ (*state >> 57)) & 0x1;
96          *state = (*state << 1) | in_bit;
97          descrambled |= (out_bit << i);
98      }
99
100     return descrambled;
101 }
102
103 uint64_t descrambler_opt (uint64_t *state, uint64_t payload)
104 {
105     descrambled = (*state >> 6) ^ (*state >> 25) ^ payload;
106     *state = descrambled ^ (payload << 39) ^ (payload << 58);
107
108     return descrambled;
109 }
110
111 void decode(struct block66b block, struct Frame *frame, uint64_t *state)
112 {
113     descrambled = descrambler(state, block.payload);
114
115     p = frame->data;
116     tail = 0;
117
118     /* /D/ */
119     if (block.sync_header != 1) {
120         * (uint64_t *) p = descrambled;
121         p += 8;
122         frame->len += 8;
123     /* control block */
124     } else {
125         switch(descrambled & 0xff) {
126         /* /S/ */
```

```
127         case 0x33:
128             frame->idles += 4;
129             descrambled >>= 40;
130             * (uint64_t *) p = descrambled;
131             p += 3;
132             frame->len += 3;
133             break;
134         case 0x78:
135             descrambled >>= 8;
136             * (uint64_t *) p = descrambled;
137             p += 7;
138             frame->len += 7;
139             break;
140         /* /T/ */
141         case 0xff:
142             tail ++;
143         case 0xe1:
144             tail ++;
145         case 0xd2:
146             tail ++;
147         case 0xcc:
148             tail ++;
149         case 0xb4:
150             tail ++;
151         case 0xaa:
152             tail ++;
153         case 0x99:
154             tail ++;
155         case 0x87:
156             descrambled >>= 8;
157             * (uint64_t *) p = descrambled;
158             frame->len += tail;
```

```
159         /* /E/ */
160         case 0x1e:
161             frame->idles += 8;
162             break;
163         }
164     }
165 }
```

APPENDIX B
FAST CRC ALGORITHM

A cyclic redundancy check (CRC) is widely used for detecting bit errors. For example, the Media Access Control (MAC) layer of the network stack computes a CRC value of each Ethernet frame and appends it to the end of each frame so that the receiver can detect any bit errors by re-computing the CRC value of the received frame. In this section, we discuss how to improve the performance of CRC computation in order to meet the performance requirements for implementing SoNIC.

Mathematically, the CRC value of an Ethernet frame is the remainder of the binary polynomial of the frame divided by a CRC polynomial. A polynomial here is defined over the Galois field, $GF(2)$[1], where the first bit of a frame corresponds to the x^{n-1} term and the last bit to the x^0 term. The IEEE 802.3 standard defines the 32-bit CRC value of an Ethernet frame MSG as follows:

$$CRC = MSG \cdot x^{32} \ \% \ P$$

where '$\cdot$' is carry-less multiplication, % is the modulo operator, and P is `0x104C11DB7` [19]. A CRC engine is simple to implement in hardware, but difficult to make efficient if implemented in software. Table lookup algorithms and parallel updates for 8-bits are commonly used to compute a CRC value in software. The Linux Kernel implements both of these algorithms but neither of them are fast enough for 10 GbE as we showed in Section 3.2.1 especially for small packets.

In order to improve the performance of software-based CRC implementation, the Fast CRC algorithm [61] from Intel uses the `PCLMULQDQ` instruction

[1] A Galois field is a field with a finite number of elements

which performs carry-less multiplication of two 64-bit quadwords [62]. The algorithm *folds* a large chunk of data into a smaller chunk using the `PCLMULQDQ` instruction to efficiently reduce the size of data. Here we only discuss the basic idea of how the algorithm works.

Let $H(x)$, $L(X)$ and $G(x)$ be the most significant 64-bit, second most significant 64-bit and remaining T-bit (T ≥ 128) chunks of MSG :

$$MSG = H(x) \cdot x^{T+64} + L(x) \cdot x^{T} + G(x)$$

where + is XOR operation. Then, 128 bits can be folded while maintaining the same CRC value because of the following fact:

$$\begin{aligned} MSG\%P =& \{H(x) \cdot (x^{T+64} \ \% \ P)\}+ \\ & \{L(x) \cdot (x^{T} \ \% \ P)\} + G(x) \ \% \ P \end{aligned}$$

If T is known, 32-bit $c_1 = x^{T+64} \ \% \ P$ and $c_2 = x^{T} \ \% \ P$ can be pre-computed. Then, the results of $H(x) \cdot c_1$ and $L(x) \cdot c_2$ become both 96-bit numbers. With this idea, the algorithm iteratively folds the message by 512 bits or 128 bits into a small 64-bit chunk which is then passed to Barrett Reduction algorithm [40] for final CRC computation. We adapted this algorithm and implemented it using inline C assembly with a few optimizations for smaller packets. We present the pseudocode implementation of the algorithm below. The details of the algorithm is discussed in [61].

B.1 Psuedo assembly code for Fast CRC algorirthm

```
1 static uint64_t k[] = {
2   0x653d9822, 0x111a288ce, // k1, k2
3   0x65673b46, 0x140d44a2e, // k3, k4
4   0xccaa009e, 0x163cd6124, // k5, k6
```

```
5   0x1f7011641, 0x1db710641, // u, p
6 };
7
8 // len must be >= 64
9 uint32_t fast_crc(uint8_t *data,int len)
10 {
11  uint64_t tmp[6];
12
13  __asm__ (
14   movdqa xmm15, k[0]; // k1 | k2;
15   movdqa xmm14, k[2]; // k3 | k4;
16   movdqa xmm13, k[4]; // k5 | k6;
17   movdqa xmm12, k[6]; // u | p
18   movq rcx, data;
19   movl ebx, len;
20
21   // first 32bit needs to be complemented
22   movl ecx, $0xffffffff;
23   movq xmm4, rcx4;
24   movdqa xmm2, [rcx];
25   pxor xmm4, xmm2;
26
27   movdqa xmm5, [rcx+16];
28   movdqa xmm6, [rcx+32];
29   movdqa xmm7, [rcx+48];
30
31   add rcx, 64;
32   sub rbx, 64;
33   cmp rbx, 64;
34   jl fold_by_4_done;
35
36   // while (len >= 128) folding by 4
37 fold_by_4_begin:
38   movdqa xmm0, xmm4;
39   movdqa xmm8, [rcx];
40   pclmulqdq xmm4, xmm15, 0x00;
41   pclmulqdq xmm0, xmm15, 0x11;
42   pxor xmm4, xmm0;
43   pslldq xmm4, 4;
44   pxor xmm4, xmm8;
45   movdqa xmm1, xmm5;
46   movdqa xmm9, [rcx+16];
47   pclmulqdq xmm5, xmm15, 0x00;
48   pclmulqdq xmm1, xmm15, 0x11;
49   pxor xmm5, xmm1;
50   pslldq xmm5, 4;
51   pxor xmm5, xmm9;
52   movdqa xmm2, xmm6;
53   movdqa xmm10, [rcx+32];
54   pclmulqdq xmm6, xmm15, 0x00;
55   pclmulqdq xmm2, xmm15, 0x11;
56   pxor xmm7, xmm2;
57   pslldq xmm6, 4;
58   pxor xmm6, xmm10;
59   movdqa xmm3, xmm7;
60   movdqa xmm11, [rcx+48];
61   pclmulqdq xmm7, xmm15, 0x00;
62   pclmulqdq xmm3, xmm15, 0x11;
63   pxor xmm8, xmm3;
64   pslldq xmm7, 4;
65   pxor xmm7, xmm11;
66
67   sub rbx, 64;
68   add rcx, 64;
69   cmp rbx, 64;
70   jge fold_by_4_begin;
```

```
71
72 fold_by_4_done:
73   movdqa xmm2, xmm4;
74   pclmulqdq xmm4, xmm14, 0x00;
75   pclmulqdq xmm2, xmm14, 0x11;
76   pxor xmm4, xmm2;
77   pslldq xmm4, 4;
78   pxor xmm4, xmm5;
79   movdqa xmm2, xmm4;
80   pclmulqdq xmm4, xmm14, 0x00;
81   pclmulqdq xmm2, xmm14, 0x11;
82   pxor xmm4, xmm2;
83   pslldq xmm4, 4;
84   pxor xmm4, xmm6;
85   movdqa xmm2, xmm4;
86   pclmulqdq xmm4, xmm14, 0x00;
87   pclmulqdq xmm2, xmm14, 0x11;
88   pxor xmm4, xmm2;
89   pslldq xmm4, 4;
90   pxor xmm4, xmm7;
91
92   cmp rbx, 16;
93   jl fold_by_1_done;
94
95 fold_by_1_begin:
96   movdqa xmm2, xmm4;
97   movdqa xmm1, [rcx];
98   pclmulqdq xmm4, xmm14, 0x00;
99   pclmulqdq xmm2, xmm14, 0x11;
100  pxor xmm4, xmm2;
101  pslldq xmm4, 4;
102  pxor xmm4, xmm1;
103
104  sub rbx, 16;
105  add rcx, 16;
106  cmp rbx, 16;
107  jge fold_by_1_begin;
108
109 fold_by_1_done:
110  cmp rbx, 0;
111  je final_reduction;
112
113  movdqa xmm1, [rcx];
114  pxor xmm3, xmm3;
115  movdqu [tmp], xmm3;
116  movdqu [tmp+16], xmm3;
117  movdqu [tmp+32], xmm3;
118  movdqu [tmp+32], xmm1;
119  movdqu [tmp+16], xmm4;
120
121  movdqu xmm1, [tmp+rbx];
122  movdqu xmm2, [tmp+rbx + 16];
123
124  movaps xmm4, xmm1;
125  pclmulqdq xmm1, xmm14, 0x00;
126  pclmulqdq xmm4, xmm14, 0x11;
127  pxor xmm4, xmm1;
128  pslldq xmm4, 4;
129  pxor xmm4, xmm2;
130
131 final_reduction:
132  movdqa xmm2, xmm4;
133  pxor xmm3, xmm3;
134  punpckldq xmm4, xmm3;
135  movdqa xmm1, xmm4;
136  pclmulqdq xmm4, xmm13, 0x00;
```

```
137  pclmulqdq xmm1, xmm13, 0x11;
138  psrldq xmm2, 8;
139  pxor xmm4, xmm2;
140  pxor xmm4, xmm1;
141
142  // Barrett Reduction algorithm
143  movdqa xmm2, xmm4;
144  pslldq xmm4, 4;
145  pclmulqdq xmm4, xmm12, 0x00;
146  pclmulqdq xmm4, xmm12, 0x10;
147  psrldq xmm4, 4;
148  pxor xmm4, xmm2;
149  pextrd crc, xmm4, 1;
150  );
151 }
```

APPENDIX C

OPTIMIZING SCRAMBLER

The scrambler is used in the physical coding sublayer of the physical layer of the network stack. It randomizes bits to eliminate a long sequence of the same symbol before they are transmitted. In this section, we discuss how we optimized the scrambler in order to meet the performance requirements for implementing SoNIC.

64b/66b encoder implements a self-synchronous scrambler in which the prior 59 bits of the transmitted data is the state of the scrambler. The following binary polynomial[1] implements the scrambler of 64b/66b encoder:

$$G(x) = 1 + x^{39} + x^{58} \tag{C.1}$$

where the polynomial is defined over the Galois field, $GF(2)$, and + is XOR operation. Considering that x^{39} is equal to the 39th output bit preceding the current one, and x^{58} to the 58th output bit, we can rewrite the equation as

$$y_n = x_n + y_{n-39} + y_{n-58}$$

where y_i is the i-th output bit of an 64-bit payload, and x_i is the i-th input bit. Both x and y are 64-bit data. Let s_i denote y_{i-64}, then the equation can be rewritten generally as,

$$y_n = x_n + s_{n+25} + s_{n+6}$$

For an input block x where $0 \leq n < 64$, the output block y can be divided into

[1]The most recent bit is the lowest order term following the convention

three small blocks,

$$
\begin{aligned}
y_{0\cdots 38} &= x_{0\cdots 38} + s_{25\cdots 63} + s_{6\cdots 44} \\
y_{39\cdots 57} &= x_{39\cdots 57} + s_{64\cdots 82} + s_{45\cdots 63} \\
&= x_{39\cdots 57} + y_{0\cdots 18} + s_{45\cdots 63} \\
y_{58\cdots 63} &= x_{58\cdots 63} + s_{83\cdots 88} + s_{64\cdots 69} \\
&= x_{58\cdots 63} + y_{19\cdots 24} + y_{0\cdots 5}
\end{aligned} \tag{C.2}
$$

Let s'_i be s_{i+6} (6-bit shift to right), s''_i be s_{i+25} (20-bit shift to right), y'_i be y_{i-39} (39-bit shift to left), and y''_i be y_{i-58} (58-bit shift to left). Note that y' and y'' can be known after Equation C.2 is computed. Then above equations become

$$
\begin{aligned}
y_{0\cdots 38} &= x_{0\cdots 38} + s''_{0\cdots 38} + s'_{0\cdots 38} \\
y_{39\cdots 57} &= x_{39\cdots 57} + y'_{39\cdots 57} + s'_{39\cdots 57} \\
y_{58\cdots 63} &= x_{58\cdots 63} + y'_{58\cdots 63} + y''_{58\cdots 63}
\end{aligned}
$$

Finally, considering shift operations pad zeros, $s'_{58\cdots 63}$, $s''_{39\cdots 63}$, $y'_{0\cdots 38}$, and $y''_{0\cdots 57}$ are all zeros. This fact leads to following equations:

$$
\begin{aligned}
y_{0\cdots 38} &= x_{0\cdots 38} + s''_{0\cdots 38} + s'_{0\cdots 38} \\
y_{39\cdots 57} &= x_{39\cdots 57} + (s''_{39\cdots 57}) + s'_{39\cdots 57} + y'_{39\cdots 57} + (y''_{39\cdots 57}) \\
y_{58\cdots 63} &= x_{58\cdots 63} + (s''_{58\cdots 63} + s'_{58\cdots 63}) + y'_{58\cdots 63} + y''_{58\cdots 63}
\end{aligned}
$$

where values in () are zeros . The first three terms on the right hand side of each equation (x, s', and s'') can be computed all together, and then y' and y'' can be

XORed to produce the final result. In other words,

$$
\begin{aligned}
t &= x + s'' + s' \\
&= x + (s >> 25) + (s >> 6) \\
y &= t + y' + y'' \\
&= t + (t << 39) + (t << 58)
\end{aligned}
$$

The result y is the final output block and becomes s for the next block computation. Actual implementation in C is shown in Line 20~27 in Appendix A.3

APPENDIX D

GLOSSARY OF TERMS

accuracy: Closeness of a measured value to a standard or known value. In clock synchronization, it is the closeness of a local time to a reference time. In available bandwidth estimation, it is the closeness of an estimated bandwidth to the actual bandwidth available. See also *clock synchronization* and *available bandwidth*.

available bandwidth: The maximum data rate that a system can send down a network path to another system without going over the capacity between the two.

bit error rate (BER): The ratio of the number of bits incorrectly delivered from the number of bits transmitted.

bitstream: A sequence of bits.

clock domain crossing (CDC): delivering a signal from one clock domain into another in a digital circuit.

clock skew: The time difference between two clocks. See also *offset*.

clock synchronization: Synchronizing clocks of multiple systems to a common time.

clock recovery: In high-speed data communications such as Ethernet, bitstreams do not carry clock signals. As a result, the receiving device recovers clock from the transitions in the received bitstream. There must be enough transitions (one to zero or zero to one) for easier clock synchronization, which is achieved by scrambler in Ethernet.

computer network: A collection of computers and network devices that communicate data via data links.

covert channel: A channel that is not intended for information transfer, but can leak sensitive information.

covert storage channel: The sender of a covert channel modulates the value of a storage location to send a message.

covert timing channel: The sender of a covert channel modulates system resources over time to send a message.

cyclic redundancy check (CRC): A check value that is computed from data and sent along with data to detect data corruption from the receiver.

datacenter: A facility that consists of racks of servers and a network connecting servers along with the physical infrastructure and power.

device driver: A computer program that controls a hardware device attached to a system.

direct memory access (DMA): Transfer of data in and out of main memory without involving central processing unit.

Ethernet: A family of standards specified in IEEE 802.3 that is used for data communication between network components in local area networks.

Ethernet frame: A unit of data in Ethernet. See also *Ethernet*.

field-programmable gate array (FPGA): An integrated circuit that can be configured and programmed after manufacture.

first in first out (FIFO): A data structure where the oldest entry is processed first.

hardware timestamping: Timestamping performed by network interface cards before transmitting a packet to a network or after receiving a packet from a network. See also *timestamp*.

homogeneous packet stream: A stream of packets that have the same destination, the same size and the same IPGs (IPDs) between them. See also *interpacket delay*, *interpacket gap* and *network packet*.

interpacket delay (IPD): The time difference between the first bits of two successive packets.

interpacket gap (IPG): The time difference between the last bit of one packet and the first bit of the following packet.

kernel: The core component of operating systems, managing systems resources and hardware devices, providing interface to userspace programs for accessing system resources.

meta-stability: An unstable state in a digital circuit where the state does not settle into '0' or '1' within a clock cycle.

network: See *computer network*.

network application: A program running on a host that is communicating with other programs running on other machines over a network.

network device: See *network component*.

network component: Network interface cards, switches, routers, or any other devices that build a network and send, receive, forward or process network packets. See also *computer network* and *network packet*.

network covert channel: A covert channel that sends hidden messages over legitimate packets by modifying packet headers or by modulating interpacket delays. See also *covert channel*.

network interface card (NIC): Device attached to a computer that connects the host computer to a network.

network measurement: Measuring the amount and type of network traffic on a network.

network node: See *network component*.

network packet: A unit of data being transferred in a network.

network switch: A multi-port network device that forwards network packets to other network devices.

cut-through switch: A switch that starts forwarding a frame once the destination port is known without checking CRC value. It does not store an entire frame before forwarding. Switching latency is lower than store-and-forward switches. See also *Cyclic Redundancy Check*.

store-and-forward switch: A switch that stores an entire frame, verifies CRC value, and forwards it to destination port. See also *Cyclic Redundancy Check*.

network traffic: Data being moved in a network.

offset: In clock synchronization, the time difference between two clocks. See also *clock skew*.

oscillator: A circuit or device that generates a periodically oscillating signal.

one way delay (OWD): The time it takes for a message to travel across a network from source to destination.

operating system (OS): A software that manages hardware and software resources and provides abstracts and services for programs.

pacing: Controlling time gaps between network packets.

packet: See *network packet*.

peripheral device: Hardware device that is attached to a computer.

peripheral component interconnect express (PCIe): A standard for serial communication bus between a computer and peripheral devices.

precision: Closeness of two or more measurements to each other. In clock synchronization, it is the degree to which clocks are synchronized, or the maximum offset between any two clocks. In timestamping packets, it is the closeness of the timestamp to the actual time it was received. In pacing packets, it is the closeness of an intended time gap to the actual time gap between two messages. See also *clock synchronization*, *offset*, *pacing*, and *timestamp*.

process: An instance of program that is being executed.

robustness: In covert channels, delivering messages with minimum errors.

round trip time (RTT): The sum of the length of time it takes for a request to be sent and the length of time it takes for a respond to be received.

router: See *network switch*.

symbol: A pulse in the communication channel that persists for a fixed period of time and that represents some number of bits.

symbol stream: A sequence of symbols. See also *bitstream* and *symbol*.

synchronization FIFO: A special FIFO that delivers data between two clock domains. See also *clock domain crossing* and *first-in-first-out*.

system clock: The number of seconds since the epoch (1 January 1970 00:00:00 UTC).

system time: What a system clock reads. See also *system clock*.

undetectability: An ability to hide the existence of a covert channel.

userspace: All code that run outside the operating systems' kernel. It includes programs and libraries that provide interface to the operating system. See also *operating system* and *kernel*.

time synchronization protocol: A protocol that achieves clock synchronization in a network. See also *clock synchronization*.

timeserver: A server that reads the reference time from an atomic clock or a GPS receiver and distributes the time to other systems in a network.

timestamp: The time at which an event is recorded, such as when a packet is transmitted or received.

timestamp counter (TSC): A register in Intel processors that counts the number of cycles since reset or boot.

transceiver: A device that can both transmit and receive signals.

BIBLIOGRAPHY

[1] Altera. 10-Gbps Ethernet Reference Design. http://www.altera.com/literature/ug/10G_ethernet_user_guide.pdf.

[2] Altera. PCI Express High Performance Reference Design. http://www.altera.com/literature/an/an456.pdf.

[3] Altera Quartus II. http://www.altera.com/products/software/quartus-ii/subscription-edition.

[4] Altera. Stratix IV FPGA. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/stxiv-index.jsp.

[5] Altera. Stratix V FPGA. http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp.

[6] Bluespec. www.bluespec.com.

[7] Broadcom. http://http://www.broadcom.com/products/Switching/Data-Center.

[8] Cisco Catalyst 4948 Switch. http://www.cisco.com/c/en/us/products/switches/catalyst-4948-switch/index.html.

[9] DE5-Net FPGA development kit. http://de5-net.terasic.com.tw.

[10] Endace DAG network cards. http://www.endace.com/endace-dag-high-speed-packet-capture-cards.html.

[11] Exablaze. https://exablaze.com/.

[12] Fibre Channel. http://fibrechannel.org.

[13] FlowNIC. http://netronome.com/product/flownics/.

[14] Highly accurate time synchronization with ConnectX-3 and Timekeeper. http://www.mellanox.com/pdf/whitepapers/WP_Highly_Accurate_Time_Synchronization.pdf.

[15] HitechGlobal. http://hitechglobal.com.

[16] How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[17] IBM RackSwitch G8264. http://www.lenovo.com/images/products/system-x/pdfs/datasheets/rackswitch_g8264_ds.pdf.

[18] IEEE Standard 1588-2008. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757.

[19] IEEE Standard 802.3-2008. http://standards.ieee.org/about/get/802/802.3.html.

[20] Intel 64 and IA-32 architectures software developer manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[21] Intel Westmere. http://ark.intel.com/products/codename/33174/Westmere-EP.

[22] iperf. https://iperf.fr.

[23] ITU-T Rec. G.8262. http://www.itu.int/rec/T-REC-G.8262.

[24] Mellanox. www.mellanox.com.

[25] Myricom Sniffer10G. http://www.myricom.com/sniffer.html.

[26] National Lambda Rail. http://www.nlr.net/.

[27] Open compute project. http://www.opencompute.org.

[28] P4. http://p4.org.

[29] SDNet. http://www.xilinx.com/products/design-tools/software-zone/sdnet.html.

[30] The CAIDA UCSD Anonymized Internet Traces. `http://www.caida.org/datasets/`.

[31] Timekeeper. `http://www.fsmlabs.com/timekeeper`.

[32] Trusted computer system evaluation criteria. Technical Report DOD 5200.28-STD, National Computer Security Center, December 1985.

[33] `gettimeofday`*(2) Linux Programmer's manual*, 2012.

[34] IEEE 1588 PTP and Analytics on the Cisco Nexus 3548 Switch. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white-paper-c11-731501.html`, 2014.

[35] *Oxford English Dictionary Online*. Oxford University Press, June 2016.

[36] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2008.

[37] Kiarash Amiri, Yang Sun, Patrick Murphy, Chris Hunter, Joseph R. Cavallaro, and Ashutosh Sabharwal. WARP, a unified wireless network testbed for education and research. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, 2007.

[38] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, Andrew W. Moore, and Philippe Owezarski. OSNT: Open source network tester. *IEEE Network*, 28(5):6–12, 2014.

[39] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. SwitchBlade: A platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM conference on Special Interest Group on Data Communication*, 2010.

[40] Paul D. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings of the International Cryptology Conference*, 1986.

[41] Vincent Berk, Annarita Giani, and George Cybenko. Detection of covert

channel encoding in network packet delays. Technical Report TR2005-536, Department of Computer Science, Dartmouth College, November 2005.

[42] Broadcom. Ethernet time synchronization. `http://www.broadcom.com/collateral/wp/StrataXGSIV-WP100-R.pdf`.

[43] Timothy Broomhead, Julien Ridoux, and Darryl Veitch. Counter availability and characteristics for feed-forward based synchronization. In *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2010.

[44] Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP covert timing channels: Design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[45] Martin Casado. Reconfigurable networking hardware: A classroom tool. In *Proceedings of Hot Interconnects 13*, 2005.

[46] John T. Chapman, Rakesh Chopra, and Laurent Montini. The DOCSIS timing protocol (DTP) generating precision timing services from a DOCSIS system. In *Proceedings of the Spring Technical Forum*, 2011.

[47] Richard Cochran, Cristian Marinescu, and Christian Riesch. Synchronizing the Linux system time to a PTP hardware clock. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2011.

[48] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, 2012.

[49] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2C2: A network stack for rack-scale computers. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, 2015.

[50] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, September 1989.

[51] Flaviu Cristian, Houtan Aghili, and Ray Strong. Clock synchronization in the presence of omission and performance failures, and processor joins. In *Proceedings of 16th International Symposium on Fault-Tolerant Computing Systems*, 1992.

[52] Mark Crovella and Balachander Krishnamurthy. *Internet Measurement: Infrastructure, Traffic and Applications*. John Wiley and Sons, Inc, 2006.

[53] Matthew Davis, Benjamin Villain, Julien Ridoux, Anne-Cecile Orgerie, and Darryl Veitch. An IEEE-1588 Compatible RADclock. In *Proceedings of International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2012.

[54] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[55] Thomas G. Edwards and Warren Belkin. Using SDN to facilitate precisely timed actions on real-time data streams. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, 2014.

[56] Albert Einstein. Zur elektrodynamik bewegter kőrper. *Annalen der Physik*, 17(10):891–921, 1905. Translated by W. Perrett and G.B. Jeffery.

[57] Gina Fisk, Mike Fisk, Christos Papadopoulos, and Joshua Neil. Eliminating steganography in Internet traffic with active wardens. In *Revised Papers from the 5th International Workshop on Information Hiding*, 2003.

[58] Daniel A. Freedman, Tudor Marian, Jennifer H. Lee, Ken Birman, Hakim Weatherspoon, and Chris Xu. Exact temporal characterization of 10 Gbps optical wide-area network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[59] Steven Froehlich, Michel Hack, Xiaoqiao Meng, and Li Zhang. Achieving precise coordinated cluster time in a cluster environment. In *Proceedings of International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2008.

[60] Steven Gianvecchio and Haining Wang. Detecting covert timing channels: An entropy-based approach. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[61] Vinodh Gopal, Erdinc Ozturk, Jim Guilford, Gil Wolrich, Wajdi Feghali, Martin Dixon, and Deniz Karakoyunlu. Fast CRC computation for generic polynomials using PCLMULQDQ instruction. White paper, Intel, `http://download.intel.com/design/intarch/papers/323102.pdf`, December 2009.

[62] Shay Gueron and Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. White paper, Intel, `http://software.intel.com/file/24918`, January 2010.

[63] Riccardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, 15(7):847–853, July 1989.

[64] Joseph Y. Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-tolerant clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, 1984.

[65] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM conference on Special Interest Group on Data Communication*, 2010.

[66] Mark Handley, Christian Kreibich, and Vern Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[67] Ningning Hu, Li Erran Li, Zhuoqing Morley Mao, Peter Steenkiste, and Jia Wang. Locating Internet bottlenecks: Algorithms, measurements, and implications. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, 2004.

[68] Manish Jain and Constantinos Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. In *Proceedings of Passive and Active Measurements Workshop*, 2002.

[69] Raj Jain and Shawn A. Routhier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal On Selected Areas in Communications*, 4:986–995, 1986.

[70] Jurgen Jasperneite, Khaled Shehab, and Karl Weber. Enhancements to the time synchronization standard IEEE-1588 for a system of cascaded bridges. In *Proceedings of the IEEE International Workshop in Factory Communication Systems*, 2004.

[71] Christoforos Kachris, Keren Bergman, and Ioannis Tomkos. *Optical Interconnects for Future Data Center Networks*. Springer, 2013.

[72] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the ACM Symposium on Cloud Computing*, 2012.

[73] Myron King, Jamey Hicks, and John Ankcorn. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[74] Ramana Rao Kompella, Kirill Levchenko, Alex C. Snoeren, and George Varghese. Every microsecond counts: Tracking fine-grain latencies with a lossy difference aggregator. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009.

[75] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36:933–940, Aug 1987.

[76] Deepa Kundur and Kamran Ahsan. Practical internet steganography: Data hiding in ip. In *Proceedings of Texas workshop on Security of Information Systems*, 2003.

[77] Leslie Lamport and P. M. Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, 1984.

[78] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. Number 1, pages 52–78, January 1985.

[79] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[80] Maciej Lapinski, Thomasz Wlostowki, Javier Serrano, and Pablo Alvarez. White Rabbit: a PTP application for robust sub-nanosecond synchronization. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2011.

[81] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. SoNIC: Precise real-time software access and control of wired networks. In *Proceedings of the*

10th USENIX Symposium on Networked Systems Design and Implementation, 2013.

[82] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transaction on Networking*, 2(1), February 1994.

[83] W. Lewandowski, Jacques Azoubib, and William J. Klepczynski. GPS: primary tool for time transfer. *Proceedings of the IEEE*, 87:163–172, January 1999.

[84] Han Li. IEEE 1588 time synchronization deployment for mobile backhaul in China Mobile, 2014. Keynote speech in the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication.

[85] Chiun Lin Lim, Ki Suh Lee, Han Wang, Hakim Weatherspoon, and Ao Tang. Packet clustering introduced by routers: Modeling, analysis and experiments. In *Proceedings of the 48th Annual Conference on Information Sciences and Systems*, 2014.

[86] Maciej Lipinski, Tomasz Wlostowski, Javier Serrano, Pablo Alvarez, Juan David Gonzalez Cobas, Alessandro Rubini, and Pedro Moreira. Performance results of the first White Rabbit installation for CNGS time transfer. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2012.

[87] Xiliang Liu, Kaliappa Ravindran, Benyuan Liu, and Dmitri Loguinov. Single-hop probing asymptotics in available bandwidth estimation: sample-path analysis. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[88] Xiliang Liu, Kaliappa Ravindran, and Dmitri Loguinov. Multi-hop probing asymptotics in available bandwidth estimation: Stochastic analysis. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, 2005.

[89] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Hide and seek in time: Robust covert timing channels. In *Proceedings of the 14th European conference on Research in computer security*, 2009.

[90] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Robust and undetectable steganographic timing channels for i.i.d. traffic. In *Proceedings of the 12th international conference on Information hiding*, 2010.

[91] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proceedings of Microelectronics Systems Education*, 2007.

[92] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–204, 1984.

[93] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and application protocol scrubbing. In *Proceedings of IEEE Conference on Computer Communications*, 2000.

[94] Enrique Mallada, Xiaoqiao Meng, Michel Hack, Li Zhang, and Ao Tang. Skewless network clock synchronization. In *Proceedings of the 21st IEEE International Conference on Network Protocols*, 2013.

[95] Tudor Marian, Ki Suh Lee, and Hakim Weatherspoon. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2012.

[96] Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, 1983.

[97] Bob Melander, Mats Bjorkman, and Per Gunningberg. A new end-to-end probing and analysis method for estimating bandwidth bottlenecks. In *Proceedings of the IEEE Global Telecommunications Conference*, 2000.

[98] David L. Mills. Internet time synchronization: The network time protocol. *IEEE transactions on Communications*, 39:1482–1493, October 1991.

[99] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, 2015.

[100] Tal Mizrahi and Yoram Moses. Software Defined Networks: It's about time. In *Proceedings of the IEEE International Conference on Computer Communications*, 2016.

[101] Pedro Moreira, Javier Serrano, Tomasz Wlostowski, Patrick Loschmidt, and Georg Gaderer. White Rabbit: Sub-nanosecond timing distribution over Ethernet. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2009.

[102] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into TCP/IP. In *Proceedings of 7th Information Hiding workshop*, 2005.

[103] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[104] Bill Ogden, Jose Fadel, and Bill White. IBM system Z9 109 technical introduction. July 2005.

[105] Patrick Ohly, David N. Lombard, and Kevin B. Stanton. Hardware assisted Precision Time Protocol. Design and case study. In *Proceedings of the 9th LCI International Conference on High-Performance Clustered Computing*, 2008.

[106] Robert Olsson. pktgen the Linux packet generator. In *Proceeding of the Linux symposium*, 2005.

[107] M. A. Padlipsky, D. W. Snow, and P. A. Karger. Limitations of end-to-end encryption in secure computer networks. Technical Report ESD-TR-78-158, Mitre Corporation, August 1978.

[108] Attila Pásztor and Darryl Veitch. PC based precision timing without GPS. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002.

[109] Pai Peng, Peng Ning, and Douglas S. Reeves. On the secrecy of timing-based active watermarking trace-back techniques. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[110] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and

Hans Fugal. Fastpass: A centralized ”zero-queue” datacenter network. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, 2014.

[111] Vinay Ribeiro, Rudolf Riedi, Richard Baraniuk, Jiri Navratil, and Les Cottrell. pathchirp: Efficient available bandwidth estimation for network paths. In *Proceedings of Passive and Active Measurements Workshop*, 2003.

[112] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[113] Craig H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, July 1997.

[114] Joanna Rutkowska. The implementation of passive covert channels in the Linux kernel. In *Proceedings of Chaos Communication Congress*, 2004.

[115] David Schneider. The microsecond market. *IEEE Spectrum*, 49(6):66–81, 2012.

[116] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report TR87-859, Cornell University, August 1987.

[117] Gaurav Shah, Andres Molina, and Matt Blaze. Keyboards and covert channels. In *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

[118] Ahmed Sobeih, Michel Hack, Zhen Liu, and Li Zhang. Almost peer-to-peer clock synchronization. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2007.

[119] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[120] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A measurement study of available bandwidth estimation tools. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, 2003.

[121] Ryousei Takano, Tomohiro Kudoh, Yuetsu Kodama, and Fumihiro Okazaki. High-resolution timer-based packet pacing mechanism on the Linux operating system. *IEICE Transactions on Communications*, 94(8):2199–2207, 2011.

[122] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: high performance software radio using general purpose multi-core processors. In *Proceedings of the 6th USENIX symposium on Networked Systems Design and Implementation*, 2009.

[123] Darryl Veitch, Satish Babu, and Attila Pàsztor. Robust synchronization of software clocks across the Internet. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, 2004.

[124] Rick Walker, Birdy Amrutur, and Tom Knotts. 64b/66b coding update. `grouper.ieee.org/groups/802/3/ae/public/mar00/walker_1_0300.pdf`.

[125] Han Wang, Ki Suh Lee, Erluo Li, Chiun Lin Lim, Ao Tang, and Hakim Weatherspoon. Timing is everything: Accurate, minimum overhead, available bandwidth estimation in high-speed wired networks. In *Proceedings of the 2014 Conference on Internet Measurement*, 2014.

[126] David X. Wei, Pei Cao, and Steven H. Low. TCP pacing revisited. In *Proceedings of IEEE International Conference on Computer Communications*, 2006.

[127] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, 1995.

[128] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[129] Takeshi Yoshino, Yutaka Sugawara, Katsushi Inagami, Junji Tamatsukuri, Mary Inaba, and Kei Hiraki. Performance optimization of TCP/IP over 10 gigabit Ethernet by precise instrumentation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[130] Sebastian Zander, Grenville Armitage, and Philip Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys and Tutorials*, 9(3):44–57, 2007.

[131] Ryan Zarick, Mikkel Hagen, and Radim Bartos. The impact of network latency on the synchronization of real-world IEEE 1588-2008 devices. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2010.

[132] Ryan Zarick, Mikkel Hagen, and Radim Bartos. Transparent clocks vs. enterprise ethernet switches. In *Proceedings of the International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2011.

[133] Hongyi Zeng, John W. Lockwood, G. Adam Covington, and Alexander Tudor. AirFPGA: A software defined radio platform based on NetFPGA. In *NetFPGA Developers Workshop*, 2009.

[134] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[135] Lixia Zhang, Scott Shenker, and David D Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *ACM SIGCOMM Computer Communication Review*, 21(4):133–147, 1991.

[136] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W Moore. NetFPGA SUME: Toward research commodity 100Gb/s. 34:32–41, July 2014.