Precise Time-synchronization in the Data-Plane using Programmable Switching ASICs

Pravein Govindan Kannan, Raj Joshi, Mun Choon Chan
School of Computing, National University of Singapore

ABSTRACT

Current implementations of time synchronization protocols (e.g. PTP) in standard industry-grade switches handle the protocol stack in the slow-path (control-plane). With new use cases of in-network computing using programmable switching ASICs, global time-synchronization in the data-plane is very much necessary for supporting distributed applications. In this paper, we explore the possibility of using programmable switching ASICs to design and implement a time synchronization protocol, *DPTP* , with the core logic running in the data-plane. We perform comprehensive measurement studies on the variable delay characteristics in the switches and NICs under different traffic conditions. Based on the measurement insights, we design and implement *DPTP* on a Barefoot Tofino switch using the P4 programming language. Our evaluation on a multi-switch testbed shows that *DPTP* can achieve median and 99^{th} percentile synchronization error of 19 ns and 47 ns between 2 switches, 4-hops apart, in the presence of clock drifts and under heavy network load.

CCS CONCEPTS

• **Networks → Programmable networks**.

KEYWORDS

Network Measurement, Programmable Switches, P4

1 INTRODUCTION

Precise network clock synchronization plays a vital role in solutions that tackle various consistency problems related to maintaining distributed databases, applications in the context of e-commerce, trading, and data-mining that run in datacenter settings. Existing standards like NTP [1] achieve only millisecond-level accuracy and require a large number of message exchange. Synchronization mechanisms based on the IEEE 1588 Precise Time Protocol (PTP) [2] can achieve nanosecond-level accuracy under idle network conditions, but achieve only sub-microsecond level accuracy under network congestion [3]. Additionally, PTP's behavior is very tightly coupled to proprietary implementations [4], and hence it is unable to achieve the theoretical performance.

The link speeds in modern datacenters are moving towards 100 Gbps. At 100 Gbps, an 80-byte packet can be received every 6.4 ns. Hence, it is imperative that clock synchronization achieves nanosecond-level precision. Recent works like DTP [3] and HUYGENS [5] have shown that it is possible to provide nanosecond-level precision in datacenters. DTP implements their synchronization logic in the Ethernet PHY. Even though it incurs minimal traffic overhead and is highly precise, DTP needs to be supported by the PHYs across the entire network. On the other hand, HUYGENS implements time synchronization protocol between end-hosts by sending coded probes across the network and using SVM to perform clock estimation. However, HUYGENS synchronizes only the hosts, and while efficient, still incurs a non-trivial amount of bandwidth and processing overhead.

With the arrival of programmable switching ASICs, many distributed algorithms and applications [6–14] are readily implementable in the data-plane of the switch. These algorithms leverage the line-rate processing speed and stateful memory available in the switches. Our work draws its inspiration from these data-plane approaches. With the ability to perform high resolution timestamping and stateful computation on a per-packet basis, we try to answer the question: how far can we go and what does it take to achieve nanosecond-level time synchronization using the data-plane programmability framework? Clearly, implementing network-wide time synchronization in the data-plane is a much more natural way to support existing and future data-plane based distributed algorithms and applications like consistency, caching, load balancing, network updates, and tasks scheduling.
Our contributions are as following:

(1) We perform a comprehensive measurement study on the variable delay characteristics of different stages in the switch pipeline, NICs as well as cables under different network conditions.
(2) Taking insights from the measurement study, we design and implement a precise **D**ata-**Pl**ane **T**ime-synchronization **P**rotocol (*DPTP*) which leverages the flexible packet processing, transactional stateful memory and high-resolution

clocks available in programmable switching ASICs. *DPTP* stores the time in the data-plane and responds to *DPTP* queries entirely in the data-plane.

(3) We have implemented and evaluated *DPTP* on a multi-hop testbed with Barefoot Tofino switches [15].

Our evaluation shows that, in the absence of clock drifts, *DPTP* can achieve median synchronization error of 2 ns and 99th percentile error of 6 ns for two directly connected switches. In the presence of clock drifts across the network switches, *DPTP* can achieve median synchronization error of 19 ns and 99th percentile error of 47 ns for switches 4-hops apart. *DPTP* achieves 99th percentile error of 50 ns between two host NICs 6-hops apart even under heavy network load.

2 RELATED WORK AND MOTIVATION

NTP [1] is a widely used time synchronization protocol. It uses software-based timestamping before sending the requests and after receiving the response, and then calculates the one-way delay by halving the RTT. It achieves synchronization accuracy in the range of few milliseconds [16]. While NTP is easy to deploy, its accuracy is relatively low. The IEEE 1588 Precise Time Protocol (PTP) [2] utilizes hardware timestamping available in switching ASICs or end-host NICs and provides a precision of up to 10's of nanoseconds for networked devices in a datacenter. PTP is an open standard, but it is haunted by implementation-specific artifacts. Most of the proprietary implementations of PTP implement the PTP stack in the host software [17] or in the switch control-plane [18]. This has three main disadvantages. First, the client requires several synchronization rounds and statistical filtering to offset clock drifts suffered during software processing delays. This results in clients requiring up to 10 minutes to achieve an offset below 1 μs [3]. Second, the precision of such implementations has been reported to degrade up to 100's of μs under heavy network load [3, 4]. Finally, a software implementation cannot scale to a large number of clients without adding delays and affecting precision. While there are dedicated PTP hardware appliances [19], they are expensive and not amenable to datacenter-wide deployment. GPS [20] can be used to achieve nanosecond precision [21] by connecting each device to a GPS receiver. However, deployment is cumbersome and not easily scalable since each device is directly synchronized to the GPS satellites.

Datacenter Time Protocol (DTP) [3] leverages the synchronization mechanism in the Ethernet PHY layer to achieve time synchronization. Since it uses the PHY clock, its accuracy depends on the PHY characteristics. For 10 Gbps link speed, the PHY frequency is 156.25 MHz, and thus a single clock cycle is 6.4 ns. The synchronization precision achievable for 10G NIC is bounded by 25.6 ns (4 clock cycles) for a single hop. In addition, DTP requires special hardware at every PHY in the datacenter. HUYGENS [5] performs clock synchronization between the end-hosts using coded probes. It selects the right set of data samples and leverages network effect with measurements among many node pairs to achieve precise synchronization. However, it does not handle synchronization between network devices (in the data-plane) and incurs a non-negligible amount of bandwidth and processing overhead.

Programmable switching is an emerging trend with switching ASICs from vendors such as Cavium [22], Barefoot [23] and Intel [24] being available in the market and more coming up [25–27]. Programmable switches provide flexible packet parsing and header manipulation through reconfigurable match-action pipelines. They also provide transactional stateful memory that allows stateful processing across packets at line-rates. Most importantly, they provide access to high-resolution clocks (~1 ns) in the data-plane using 42-48 bit counters [28] which could maintain timestamps over several hours before *wrap around*. This combination of data-plane programmability and high-resolution clocks have made it possible to add high-resolution timing information to the packets at line-rates enabling some novel applications such as the In-band Network Telemetry (INT) [29].

In this work, we leverage flexible packet processing, transactional stateful memory, and high-resolution clocks provided by programmable switches to design and implement a precise **D**ata-**P**lane **T**ime-synchronization **P**rotocol (DPTP).

3 MEASUREMENTS

Designing a time synchronization protocol requires understanding the various delay components involved in the communication between a reference clock and a requesting client. In particular, it is important to understand which delay components could be accounted for using the timestamping capabilities in the overall system and which delay components remain unaccounted for. Achieving nanosecond-scale synchronization requires accounting for nanosecond-level *variability* in the various delay components. Further, for *maintaining* the nanosecond-level synchronization, it is necessary to keep the total synchronization delay low so that more requests could be processed per-second (for tighter synchronization) and a large number of clients could be supported.

In this section, we consider a request-response model where a *client* sends a request to synchronize with a *server* who maintains the reference clock. We perform measurements to understand the various delay components involved in a request-response packet timeline under different traffic scenarios. The request-response timeline starts when a request packet[1] is generated by a *client* network switch (or host) and ends when the corresponding response packet[1] is

[1] 60 bytes in size comprising of the Ethernet header and a custom *DPTP* header for storing individual component delays.

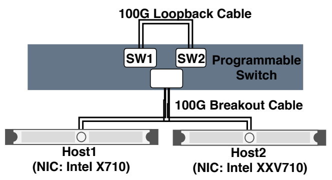


Figure 1: Measurement setup with a Programmable switch, two servers and a loopback cable

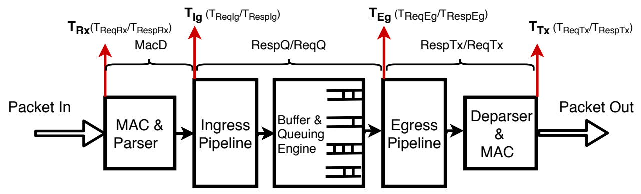

T_{Rx} (T_{ReqRx}/T_{RespRx})
T_{Ig} (T_{ReqIg}/T_{RespIg})
T_{Eg} (T_{ReqEg}/T_{RespEg})
T_{Tx} (T_{ReqTx}/T_{RespTx})
MacD
RespQ/ReqQ
RespTx/ReqTx
Packet In
MAC & Parser
Ingress Pipeline
Buffer & Queuing Engine
Egress Pipeline
Deparser & MAC
Packet Out

Figure 2: Portable Switch Architecture [28]

received from the *server* (another switch or host). Specifically, we measure the delays involved in NIC Tx/Rx, switch's data-plane pipeline processing, and wire propagation. We break down the overall delay into smaller components that can be measured using the precise timestamps provided by the switch data-plane.

Testbed Setup. Our measurement testbed (shown in Figure 1) consists of a Barefoot Tofino switch connected to two Dell servers with Intel XXV710 (25G/10G) and Intel X710 (10G) NIC cards using a 100G breakout cable configured as 4 x 10G. Two switches SW1 and SW2 are emulated on a single physical switch and connected by a 100G QSFP loopback cable (similar to [30]), which can be configured as 4 x 10G or 4 x 25G or 1 x 40G or 1 x 100G for different experiment scenarios. All cables are direct attached copper (DAC) and 1 m in length. To account for the delays involved in switch data-plane pipeline processing, we use the various high-resolution timestamps available in Portable Switch Architecture [28] (Figure 2): (i) T_{Rx}: timestamp when the entire packet is received by the Rx MAC, (ii) T_{Ig}: timestamp when the packet enters the ingress pipeline, (iii) T_{Eg}: timestamp when the packet enters the egress pipeline, and finally (iv) T_{Tx}: timestamp when the packet is transmitted by the Tx MAC. The switch is programmed with a custom P4 program to parse the Ethernet and a custom *DPTP* header. Based on the Ethernet type, our P4 program parses the *DPTP* header and adds timestamps at different stages. Packet forwarding is done based on the destination MAC address. We note that the latency numbers reported in the rest of this section correspond to our custom P4 program[2].

3.1 Switch-to-Switch Synchronization

Figure 3 shows the request-response timeline when a switch SW1 sends a synchronization request to switch SW2. We list the different timestamps and intervals (in bold) of the request-response timeline below (in chronological order):

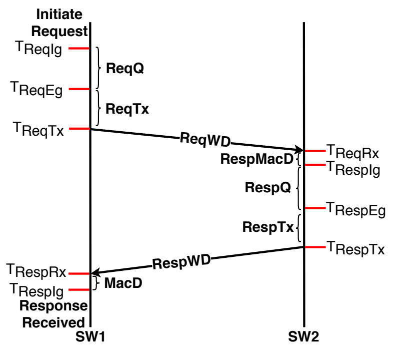


Figure 3: Packet Timeline and delay measurement

(1) T_{ReqIg}: *Timestamp* (Request packet arrival at ingress pipeline of SW1).
ReqQ: *Duration* (Ingress pipeline processing and packet queuing at buffer engine).
(2) T_{ReqEg}: *Timestamp* (Request packet arrival at egress pipeline of SW1).
ReqTx: *Duration* (Egress pipeline processing and deparsing).
(3) T_{ReqTx}: *Timestamp* (Request packet serialization and Tx from SW1).
ReqWD: *Duration* (Wire-delay across DAC cable).
(4) T_{ReqRx}: *Timestamp* (Request packet arrival at SW2).
RespMacD: *Duration* (Input buffering and parsing).
(5) T_{RespIg}: *Timestamp* (Request packet arrival at ingress pipeline and response generation).
RespQ: *Duration* (Ingress pipeline processing and packet queuing in the buffers).
(6) T_{RespEg}: *Timestamp* (Response packet arrival at egress pipeline of SW2).
RespTx: *Duration* (Egress pipeline processing and deparsing).
(7) T_{RespTx}: *Timestamp* (Request packet serialization and Tx from SW2).
RespWD: *Duration* (Wire-delay across DAC cable).
(8) T_{RespRX}: *Timestamp* (Response packet arrival at SW1).
MacD: *Duration* (Input buffering and parsing).
(9) T_{RespIg}: *Timestamp* (Response packet arrival at ingress pipeline of SW1).

From this list, only the wire delays i.e. ReqWD and RespWD cannot be directly measured using the switch data-plane timestamps. However, since there is a single physical clock shared by SW1 and SW2, we can directly compare the timestamps of the request and response packets to precisely construct the request-response timeline shown in Figure 3.

We perform each measurement using 10,000 request packets. Timestamps are added to the packet's header at different points in the switch-pipeline by the P4 program and the delays of various components are calculated when response

[2]the actual switch latency can vary based on the P4 program being used.

Table 1: Delay profiling of components in Switch-to-Switch measurements over 10G links [min - max (avg)]

Component	Idle Links	Line-rate recv traffic(1500b)	Oversubscribed recv traffic(64/1500b)
Request processing and Queuing (**ReqQ**)	306 - 313 ns (307)	306 - 313 ns (306.7)	306 - 313 ns (307)
Request Egress Tx Delay (**ReqTx**)	304 - 356 ns (340.8)	300 - 361 ns (341.1)	296 - 361 ns (341)
*[Wire Delay (1M) (**ReqWD**)]*	67 - 79 ns (72.6)	66 - 79 ns (72.7)	66 - 79 ns (72.4)
MAC & Parser Delay at Server (**RespMacD**)	0 - 57 ns (26.1)	0 - 57 ns (26)	0 - 59 ns (27)
Reply processing and Queuing (**RespQ**)	313 - 323 ns (317)	313 - 481 ns (317.3)	1.552- 1.556 ms (1.554)
Reply Egress Tx Delay (**RespTx**)	296 - 365 ns (341)	298 - 803 ns (555)	1747 - 1914 ns (1783)
*[Wire Delay (1M) (**RespWD**)]*	64 - 77 ns (70)	64 - 76 ns (70)	63 - 76 ns (70)
MAC & Parser Delay at Client (**MacD**)	0 - 55 ns (27)	0 - 58 ns (27)	0 - 90 ns (27))
Total RTT	1462 - 1637 ns (1473)	1460 - 1976 ns (1667)	1.555 - 1.559 ms (1.556)

arrives. We summarize the individual delays in Table 1 under three network conditions: (1) **Idle links**: No other traffic is being sent on the links except for the request-response packet. (2) **Line-rate recv traffic**: Cross-traffic (64/1500 byte UDP packets) at line-rate along the direction of the response packet. (3) **Oversubscribed recv traffic**: Heavy incast cross-traffic (64/1500 byte UDP packets) that induces queuing along the direction of the response packet.

Measurement results under different load conditions. In the scenario with idle links, major delays occur in the queuing and pipeline processing components (ReqQ, ReqTx, RespQ and RespTx) which add up to 84% of the total delay. Somewhat surprisingly, wire delay over the 1 m DAC cable (ReqWD and RespWD) takes around 70 ns, much more than the MAC Delay (RespMacD or MacD). Interestingly, the MAC Delay fluctuates from 0 - 55 ns since the parser takes a variable amount of time to parse the packet and feed it into the ingress pipeline [31]. RespTx also experiences similar amount of fluctuation. RespTx includes egress pipeline processing, deparsing, Tx MAC delay and serialization. Since egress processing has a fixed delay [31], this variation is contributed by the other three components of RespTx. In the line-rate traffic scenario, the RespTx is higher because the line-rate cross-traffic with 1500 byte packets fully saturates the link. The serialization delay for these packets is higher than the egress pipeline processing delay. As a result, the *extra* synchronization packets get queued for serialization in a small buffer after the egress processing. This queuing causes the Egress Tx Delay (RespTx) to increase. Line-rate cross-traffic with 64-byte packets does not observe this behaviour due to faster serialization. All other delay components are similar to the idle link scenario because they result from data-plane (hardware) processing which has similar performance under all conditions.

Under oversubscribed conditions, delay increases in two components – RespQ and RespTx. RespQ increases to 1.55 ms because the queue buffer is always full due to the oversubscribed cross-traffic (1.55 ms is the default maximum per queue buffer in our setup). RespTx in this scenario is even higher than the line-rate traffic scenario. This is because with the oversubscribed traffic, the aforementioned small

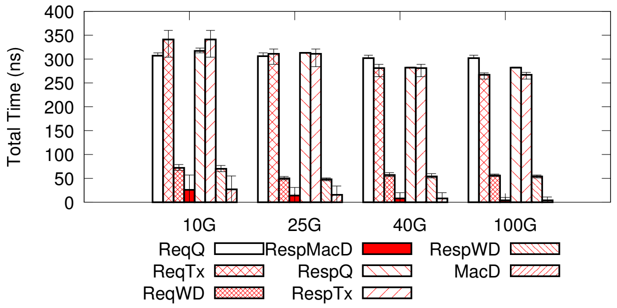


Figure 4: Delays for various link-speeds (idle links)

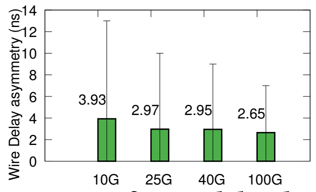


Figure 5: Asymmetry of wire delay between request and response for different link speeds

pre-serialization buffer remains mostly filled with the cross-traffic packets which causes additional delay to the synchronization packets. Due to this, the range of RespTx values falls to a small window of 167 ns. Note that ReqQ and ReqTx remain unaffected in our scenarios due to cross-traffic only along the response direction. However, under the presence of cross-traffic in the request direction, ReqQ and ReqTx exhibit the same behaviour as RespQ and RespTx.

Effect of different link speeds. We re-configured the link speed of the loopback cable between SW1 and SW2 (Figure 1) to 25G, 40G, and 100G and plot the eight delay components as bars for 10G/25G/40G/100G in Figure 4 for the idle traffic condition. We observe that the components involving MAC Delay (ReqTx, RespMacD, RespTx and MacD) reduce significantly with higher link speeds. We also observe decrease in wire delay of upto ~20 ns with 100G compared to 10G. Most importantly, the variation of delays (errorbars) reduce significantly with higher linkspeeds. The average RTTs observed for 25G, 40G and 100G are 1356 ns, 1264 ns and 1044 ns respectively.

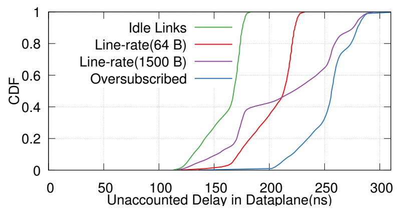


Figure 6: Unaccounted Delay (in ns) when using packet departure timestamp in data-plane

Accounting for the delay components. Normally, it would be reasonable to assume the wire delays to be symmetric between two directly connected switches. However, at a nano-second scale we found that the wire delays are asymmetric. We plot the wire delay asymmetry ($abs(ReqWD - RespWD)$) for different link speeds in Figure 5. For 10G link speed, we observe the asymmetry variation to be ranging from 0-13 ns with an average of 3.93 ns. Asymmetric Wire delays could be due to the low-frequency clocks at PHY [3]. This variation drops to 0-7 ns with an average of 2.65 ns at 100G. Thus, at higher link speeds, clock synchronization would be more accurate, since the unaccounted wire delay would have less asymmetry.

Measuring ReqTx and RespTx. To accurately measure and account for ReqTx/RespTx, we need to know the exact time when the request/response packet left the respective switch i.e. T_{ReqTx}/T_{RespTx}. The timestamp of the exact instant when the packet serialization starts cannot be embedded into the packet itself. This is a fundamental limitation and thus this timestamp needs to be communicated separately via a *follow-up* packet. To avoid such separate follow-up packet, several commercial switches support embedding a *departure timestamp* into the departing packet. The departure timestamp is added when the packet is received by the egress MAC and thus differs from the exact transmit time. The use of this departure timestamp to avoid the follow-up packet thus leads to unaccounted delay during synchronization. We measured this unaccounted delay in our setup under different cross-traffic conditions and the results are shown in Figure 6. At the 99^{th} percentile, oversubscribed traffic leads to unaccounted delay of about 289 ns while it remains about 175 ns for idle links.

Reducing the delays. Even if most of the delay components could be accounted for in a synchronization request-response, it is still necessary to keep the total delay low. A lower total delay allows for more requests to be processed per unit time for tighter synchronization in face of clock drifts. It also enables scaling to a large number of clients. Table 1 shows RespTx and RespQ to be the major contributors of the total delay. We observed that using prioritized queues for synchronization packets doesn't reduce RespTx under line rate or oversubscribed conditions. This is because the small pre-serialization buffer (the cause for higher RespTx) still remains mostly filled with cross-traffic packets. Using prioritized queues does however reduce RespQ significantly (as expected) under oversubscribed conditions. We measured RespQ to be about 2000 ns for 1500-byte packets and 650 ns for 64-byte packets with oversubscribed cross-traffic. RespQ does not come down to the same value as with idle/line-rate traffic conditions because the small pre-serialization buffer once full, back-pressures the port-level scheduler and the egress processing.

Effect of FEC. When FEC (forward error correction) is enabled, we observed that the wire delay (ReqWD and RespWD) component increases to 344 - 356 ns with idle links. The overall RTT is about 2030 ns. At line-rate traffic we observe an increase in the delay for RespQ (314-4452 ns) and RespTx (298 - 1914 ns) in addition to the wire delay components. This behaviour can be attributed to the additional back-pressure created by the serialization overhead of FEC. The overall RTT for line-rate traffic varies from 2026 - 7589 ns. However, it is important to note that the wire-delay asymmetry still remains the same i.e. bounded by 12 ns.

Take-aways:

(1) Hardware supported timestamps and the ability to embed them in packets make programmable data-planes immensely useful to account for the various delay components.
(2) Even when the link is idle, we observe a 175 ns (1462 - 1637 ns) delay variation in the processing time.
(3) Even though there is much variability in the various delay components, since accounting is available for most of them, the variability does not affect clock synchronization. The only unaccounted delay is the wire-delay.
(4) At a nanosecond-scale, the wire-delay exhibits asymmetry (between request and response). The asymmetry reduces with higher link-speeds thus enabling better precision at higher link-speeds. This asymmetry forms the lower bound on the achievable synchronization accuracy.
(5) To accurately account for a packet's exit timestamp, a follow-up packet is required. The follow-up packet could be avoided if the packet *departure timestamp* could precisely capture packet's exit time and could be embedded in the data-plane. This has important implications for the design of a time synchronization protocol.

3.2 Switch-to-Host Synchronization

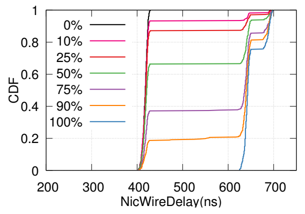
Similar to the earlier switch-to-switch measurement, we profile a request-response packet timeline, where the request packet originates from a host (rack server) and a switch responds to the request. We use MoonGen, a packet generation tool based on DPDK (ver 17.8) to generate request packets. The NIC is configured to capture the hardware timestamp when the packet is sent out and received. We perform the measurement on two NICS: (i) Intel X710 (SFP+), and (ii)

Table 2: Delay profiling of components in Switch-to-Host measurements over 10G links (SFP+)

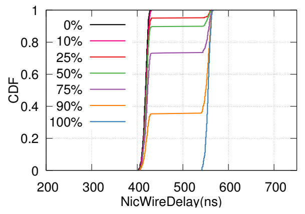
Component	Idle Links	Line-rate recv traffic(64b)	Line-rate send traffic(64b)
Reply MAC & Parser Delay (RespMacD)	0 - 56 ns (26.1)	0 - 58 ns (26.9)	0 - 57 ns (28.1)
Reply processing and Queuing (RespQ)	299 - 306 ns (300.6)	299 - 319 ns (303.8)	299 - 306 ns (300.3)
Reply Egress Tx Delay (RespTx)	297 - 360 ns (328)	303 - 411 ns (356.8)	297 - 360 ns (326)
[Un-accounted (NicWireDelay)]	402 - 429 ns (417.5)	405 - 430 (419) ns	625 - 699 (652) ns
Total RTT	1059 - 1084 ns (1072.4)	1069 - 1149 ns (1107.1)	1280 - 1354 ns (1307.3)

Table 3: Delay profiling of components in Switch-to-Host measurements over 10G links (SFP28)

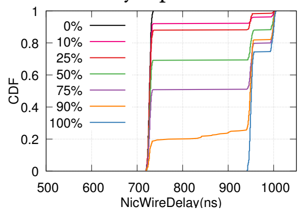
Component	Idle Links	Line-rate recv traffic(64b)	Line-rate send traffic(64b)
Reply MAC & Parser Delay (RespMacD)	0 - 56 ns (27.1)	0 - 60 ns (28.4)	0 - 59 ns (27.8)
Reply processing and Queuing (RespQ)	297 - 304 ns (297)	297 - 316 ns (298.4)	297 - 303 ns (297)
Reply Egress Tx Delay (RespTx)	283 - 356 ns (328)	299 - 356 ns (358)	299 - 364 ns (331)
[Un-accounted (NicWireDelay)]	720 - 736 (727) ns	716 - 735 (725.7) ns	941 - 1005 (962.3) ns
Total RTT	1373 - 1389 ns (1381)	1379 - 1453 ns (1411)	1593- 1661 ns (1661)



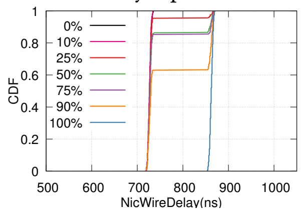
a. Intel X710 NIC (SFP+) for 64-byte packets



b. Intel X710 NIC (SFP+) for 1500-byte packets

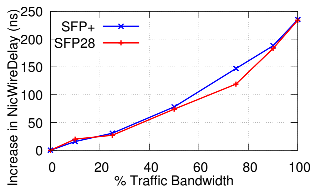


c. Intel XXV710 NIC (SFP28) for 64-byte packets

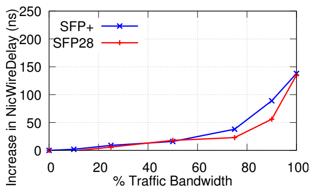


d. Intel XXV710 NIC (SFP28) for 1500-byte packets

Figure 7: CDF of NicWireDelay for idle and cross-traffic conditions with 64-byte and 1500-byte packets



a. 64-byte packets



b. 1500-byte packets

Figure 8: Increase in NicWireDelay w/ cross-traffic

Intel XXV710 (SFP28/SFP+). We configure both NICs to operate at 10G. For each NIC, we perform the measurements under three scenarios: 1) Idle host-switch link, 2) Line-rate receive traffic (64/1500-byte packets) from switch to host, and 3) Line-rate send traffic (64/1500-byte packets) from host to switch. For each scenario, we record the following delay components: 1) MAC and parser delay at the Switch (RespMacD), 2) Reply processing and queuing (RespQ) and 3) Reply egress Tx delay (RespTx). By subtracting the accounted components from the overall RTT using the timestamp captured at NIC hardware, we obtain the total unaccounted delay. Table 2 and Table 3 summarize these measurements for Intel X710 and Intel XXV710, respectively. Due to space constraints, we report only 64-byte packet measurements in the tables. For 1500-byte packets, we observe the measurements to be same for all components except $NicWireDelay$, which will be discussed later in this section.

The overall RTT on NIC X710 is similar to the switch-to-switch measurements without the ReqQ component. The RTT on NIC XXV710 is higher than X710 by about ~300 ns.We suspect that this could be because of running the SFP28 NIC in the SFP+ compatibility mode[3]. The unaccounted delay consists of the two-way wire delay and the MAC/serialization delay incurred at the NIC/transceiver. To validate this, we perform an additional experiment in which we make a loop-back connection between two ports of the NIC from the same host. We measure the timestamps captured when sending and receiving the packet, and observe that the total delay incurred is roughly ~348 ns for Intel X710 and ~660 ns for Intel XXV710. This value is close to the unaccounted delay minus one-way wire delay (roughly ~70 ns). We refer to this unaccounted delay as $NicWireDelay$ in future references.

The Case of NicWireDelay. One interesting observation is that when we *send* cross-traffic of 64-byte packets at line-rate from the same interface as the request packets, we observe an increase in the $NicWireDelay$ by about ≈250 ns in both the NICs. We can infer that this is a one-way delay increase since we do not observe an increase in $NicWireDelay$ when we *receive* line-rate cross traffic in the same interface. In order to understand this delay, we further vary the amount of traffic (using 64-byte packets) being sent out of the host interface from 0% to 100%. Figure 7(a) and 7(c) respectively show the CDF of the $NicWireDelay$ under different volumes of sent traffic for the two NICs. We observe that the overall $NicWireDelay$ fluctuates even under 10% traffic, and the fluctuation (tail) increases with an increase in traffic. The overall range of variation is about 300 ns. Further, Figure

[3]We couldn't measure XXV710 in 25G mode due to NIC timestamp issues.

8(a) shows the average ***increase*** in the $NicWireDelay$ wrt 0% traffic for different traffic volumes. The $NicWireDelay$ increases almost linearly with an increase in the sent traffic. We repeat the same experiment with bigger packets of size 1500-bytes for the sent traffic. As earlier, we plot the CDF of $NicWireDelay$ for the two NICs with 1500-byte packets in Figures 7(b) and 7(d). We observe that the variation is less (~150 ns) compared to the sent traffic with 64-byte packets. This is likely because, with 64-byte packets, there is repeated MAC/serialization delay which adds to the variability. Similar to Figure 8(a), Figure 8(b) shows the average *increase* in the $NicWireDelay$ with 1500-byte packets in the sent traffic. We observe a sluggish increase in the $NicWireDelay$ of about 40 ns till 75% traffic, and then it increases by about 135 ns at line-rate. This could be due to reduced I/O operations with 1500-byte packets as compared to 64-byte packets.

Take-aways:

(1) In spite of NIC hardware timestamping, there is an unaccounted and variable $NicWireDelay$.
(2) The $NicWireDelay$ varies up to 27 ns in Intel X710 (SFP+) and 16 ns in Intel XXV710 (SFP28) under idle conditions.
(3) In the presence of cross-traffic from the host, $NicWireDelay$ increases for both the NICs in a similar fashion.
(4) Time synchronization error can increase in the presence of upstream cross traffic due to asymmetry of $NicWireDelay$. Hence, it is ideal for the host to have knowledge of the cross-traffic volume to minimize the error.

4 DESIGN

DPTP is designed to be a network-level service with networking devices (switches) providing accurate time to hosts. To start, one of the switches in the network is designated the master switch. Next, the switches in the network, communicate with the master switch (directly or indirectly) to get the global time. This step happens periodically since the clocks are known to drift up to 30μs per second [5]. Once the switches in the network are continuously synchronized, they can respond to the time-synchronization queries from the end-hosts. Hence *DPTP* requests from hosts can be completed in a sub-RTT timeframe[4] and in just one hop.

4.1 *DPTP* in Operation

DAG Construction. The network operator designates a switch as the master. Once the master switch is identified, a network-level DAG is constructed leading to master switch in the same way as done by [32]. By leveraging [32], link-failures and fast re-routing can be taken care of at the data-plane. Upon construction of DAG, each switch will query their parent switch for time-synchronization. Eventually, the TOR switches can also maintain accurate global timing in the data-plane and respond to *DPTP* queries of hosts.

[4]sub-RTT wrt requests being responded by a master switch/host

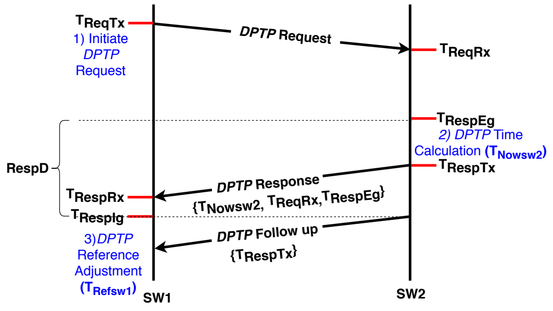

T_{ReqTx}
1) Initiate DPTP Request
DPTP Request
T_{ReqRx}
T_{RespEg}
2) DPTP Time Calculation (T_{Nowsw2})
T_{RespTx}
RespD
T_{RespRx}
DPTP Response {T_{Nowsw2}, T_{ReqRx}, T_{RespEg}}
T_{RespIg}
DPTP Follow up {T_{RespTx}}
3) DPTP Reference Adjustment (T_{Refsw1})
SW1
SW2

Figure 9: *DPTP* Request-Response timeline.

Clock Maintenance. The master switch SW stores the reference clock timestamp from an external global clock source[5] to the data-plane ASIC as $T_{Ref_{SW}}$ and stores the current data-plane timestamp (T_{RespIg}) to a register T_{offset}. Note that $T_{Ref_{SW}}$ and T_{offset} are registers available in the data-plane. We do not disturb (or reset) the existing data-plane clock due to multiple reasons: 1) rewriting the internal data-plane clock is an expensive operation since it involves consistency checks, and 2) other applications like INT [29] may be using the data-plane clock in parallel. Hence, it is necessary to leave the value unchanged to be non-intrusive to other applications.

Era Maintenance. The switch data-plane's internal clock counter used to obtain T_{RespIg} or T_{RespEg} (Figure 3) is usually limited to x bits. For example, a 48-bit counter can account for only up to 78 hours before rolling over. In order to scale above the counter limit, the switch's control-plane program probes the data-plane counter periodically to detect the wrap around. Upon detection, it increments a 64-bit era register T_{era} by $2^x - 1$.

Switch Time-keeping. On receiving an incoming request, the master switch SW reads the initially stored external reference time $T_{Ref_{SW}}$, the era T_{era} and the correction offset T_{offset} in ingress pipeline. At the egress pipeline, it reads the current egress timestamp T_{RespEg} and calculates the current time $T_{Now_{SW}}$ by adding to the reference time, the time that has elapsed since initially storing the reference time at T_{offset}:

$$T_{Now_{SW}} = T_{Ref_{SW}} + (T_{era} + T_{RespEg} - T_{offset}) \qquad (1)$$

Note that, T_{Now} is calculated at the egress pipeline using T_{RespEg} to avoid queuing delays following the T_{Now} calculation. Every time the switch SW receives *DPTP* queries from its child switch/host, it calculates $T_{Now_{SW}}$ using the current T_{RespEg}. The above steps of Clock Maintenance, Era Maintenance, and Switch Time-keeping are executed by every switch in the network.

[5]The details of the interface and maintaining sync between the external global clock and the master switch are beyond the scope of this work.

4.2 Switch-to-Switch *DPTP*

Figure 9 shows the request-response timeline between switch SW1 and the master switch SW2, which has the correct $T_{Ref_{SW2}}$. SW1 initiates *DPTP* Request and sends the packet out at T_{ReqTx}. The Request message is received by SW2 at T_{ReqRx}. SW2 calculates the reference $T_{Now_{SW2}}$ as per Equation 1 in the egress pipeline. $T_{Now_{SW2}}$, T_{ReqRx} and T_{RespEg} are embedded in the packet by the data-plane. The packet is then forwarded back to SW1 after swapping source and destination MAC addresses. The response message is received by SW1 at T_{RespRx} and it starts processing the message at T_{RespIg}. It now stores T_{RespIg} as T_{offset}. To calculate the correct reference time $T_{Ref_{SW1}}$, SW1 also needs to know T_{RespTx} which is the time just before the serialization of the response packet. For the reasons mentioned in §3.1, SW2 can accurately capture T_{RespTx} only *after* sending the response message. SW2 thus sends a *Follow-up* message containing T_{RespTx}. Once SW1 knows T_{RespTx}, it sets the correct reference time $T_{Ref_{SW1}}$ as following:

$$T_{Ref_{SW1}} = T_{Now_{SW2}} + RespD \tag{2}$$

where the response delay $RespD$ is defined as,

$$RespD = \frac{(T_{RespRx} - T_{ReqTx}) - (T_{RespTx} - T_{ReqRx})}{2} + (T_{RespTx} - T_{RespEg}) + (T_{RespIg} - T_{RespRx}) \tag{3}$$

In Equation 3, the first term calculates the approximate one-way wire-delay by removing the switch delays from the RTT. The second term is the time between T_{RespTx} and T_{RespEg}, since $T_{Now_{SW2}}$ is based on T_{RespEg}. The third term adds up the delay at SW1 before the response message is processed. When there is a follow-up packet for obtaining T_{RespTx}, SW1 would initially record the T_{RespIg} (as T_{offset}) and T_{RespRx} when it receives the DPTP response, and then use them for calculation once the follow-up packet arrives with T_{RespTx}. Hence, SW1 does not need to account for the delay due to follow-up packet. Once the reference time $T_{Ref_{SW1}}$ and the corresponding T_{offset} is stored in SW1, T_{Now} can be calculated on-demand as per Equation 1 for use by other data-plane applications or DPTP clients.

4.3 Switch-to-Host *DPTP*

Switch-to-Host *DPTP* is mostly similar to Switch-to-Switch *DPTP* . However, due to variable delay involved when there is outgoing cross-traffic from the host, we design Switch-to-Host *DPTP* to operate in two phases. Phase 1 happens only once during the initialization of the host and phase 2 is the operational phase.

Phase 1: Profiling. During this phase, the host sends *DPTP* probe packets to the switch, which are in the same format as *DPTP* query packets. The switch replies back the current time along with the switch delays ($T_{Now_{SW}}$, T_{ReqRx}, T_{RespTx}). Additionally, the switch replies the current traffic-rate (R) it is receiving from the host. If this rate is close to 0%, then the host calculates the idle $NicWireDelay$ as per Equation 4 (c.f. §3.2), and maintains its average, $AvgNicWireDelay$ over a few seconds of profiling.

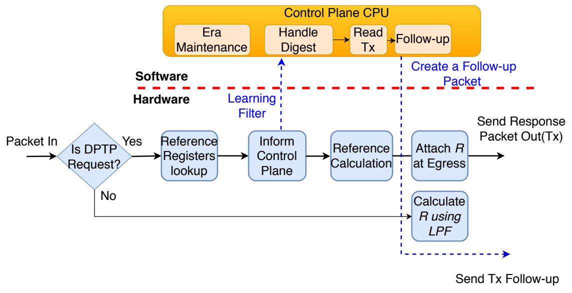


Figure 10: *DPTP* Implementation using P4

$$NicWireDelay = (T_{RespRx} - T_{ReqTx}) - (T_{RespTx} - T_{ReqRx}) \tag{4}$$

Rate Maintenance (R). The incoming traffic-rate from each host is maintained by the ToR switches. For the traffic-rate calculation, we leverage the low-pass filter (essentially used for metering) available in the switch's data-plane. This outgoing traffic rate can also be maintained in the host's smart-nic. However, it is advisable not to calculate the outgoing traffic-rate by the application at the host since a physical interface may be virtualized to several applications. Alternatively, this statistic could be obtained with hypervisor support too.

Phase 2: Synchronization. During this phase, Switch-to-Host synchronization is performed. The host sends *DPTP* query packets, and receives the current time (T_{Now}), and delays incurred in the switch (T_{ReqRx}, T_{RespTx}). Then it calculates the response delay ($RespD$) as following:

$$RespD = OWD + (T_{RespTx} - T_{RespEg}) \tag{5}$$

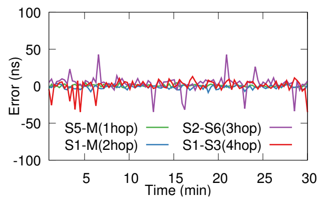
where, OWD is the one-way $NicWireDelay$ calculated as:

$$OWD = \begin{cases} \frac{NicWireDelay}{2} & ; \text{if R} \approx 0\% \\ \frac{AvgNicWireDelay}{2} & ; \text{Otherwise} \end{cases} \tag{6}$$

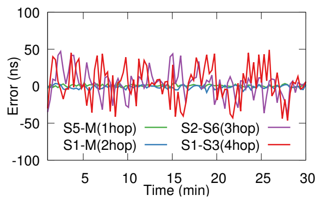
If the outgoing traffic-rate from the host (R) is close to 0%, *DPTP* can use the $NicWireDelay$ calculated from the current *DPTP* query to compute the OWD. However, if R is not close to 0%, it should use the $AvgNicWireDelay$ calculated during the profiling phase. This helps to bound the synchronization error, because the $NicWireDelay$ variation is small (27 ns on X710 and 16 ns on XXV710) when the traffic-rate is close to 0% (§3.2). The host then uses the response delay to calculate the correct reference time as per Equation 2.

5 IMPLEMENTATION

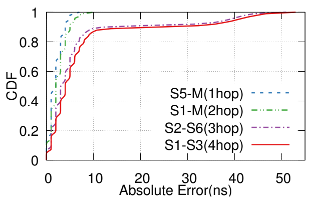
We have implemented *DPTP* on a Barefoot Tofino [15] switch in about 900 lines of P4 code which performs the reference lookup, calculation and era maintenance using stateful memories and ALUs in the data-plane. We store the reference



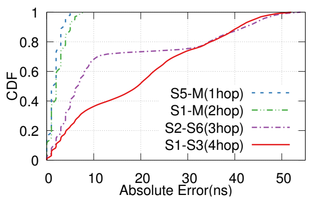
a. Idle Condition



b. Heavy Cross-traffic



c. Idle Condition CDF



d. Heavy Cross-traffic CDF

Figure 11: Error in *DPTP* Switch-to-Switch synchronization

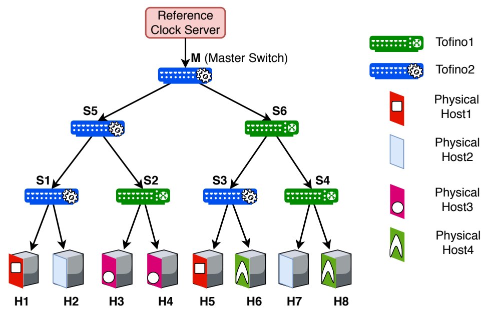


Figure 12: Evaluation Topology

timestamp in the form of two 32-bit registers. We also implement a control-plane program in about 500 lines of C code that runs two threads to perform: (i) Era Maintenance, and (ii) *DPTP* request generation. A Follow-up message handler registers a *learning digest* callback with the data-plane. Each time a *DPTP* request is received, the Follow-up handler gets a learn digest from the data-plane with the following information: 1) host mac-address, 2) *DPTP* reply out-port. The handler then probes the port's register for the transmit timestamp (T_{RespTx}), crafts a *DPTP* follow-up packet containing T_{RespTx} destined to the host mac-address, and sends to the data-plane via PCIe. Note that, the programmable switch allows the transmit timestamp to be recorded for specific packets. Hence, we record it only for the *DPTP* packets. We implement the Follow-up handler in the control-plane since the accurate T_{RespTx} which is available in a port-side register is readable only from the control-plane. The overall implementation of *DPTP* on the programmable switch is shown in Figure 10. The *DPTP* packets are assigned to the *highest priority queue* in the switches to minimize queuing delays (§3.1). The host client is implemented using MoonGen [33].

6 EVALUATION

We perform the evaluation of *DPTP* using two Barefoot Tofino switches, and four servers. Each Tofino switch connects to two servers via two network interfaces per server. Using this physical setup, we form a virtual topology as shown in Figure 12. Switches S1, S3, S5 and M (Master) are implemented on Tofino2. The links S1-S5, S5-M are implemented using loopback cables. Similarly, S2, S4, and S6 are implemented on Tofino1 and the link S4-S6 is implemented using a loopback cable. The links S2-S5, S3-S6 and S6-M are formed using 100G cables connecting Tofino1 and Tofino2. Note that the virtualization is done in data-plane using Match-Action rules similar to [30], and so there is no virtualization overhead. All cables are Direct Attach Copper (DAC) and configured with 10 Gbps link speed. The four physical hosts are virtualized into 8 virtual hosts, with each virtual host (H1, H2, ... H8) assigned a NIC interface, which is connected to either Tofino1 or Tofino2.

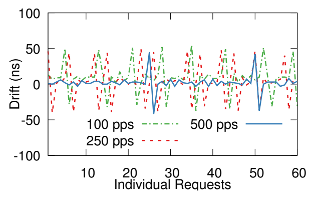
The Master-switch (M) is at the core of the network and is assumed to be synchronized to an external time reference. Other switches synchronize their time with their parent switch. For example, S1 queries S5, S3 queries S6 and so on. Further, S1, S2, S3, and S4 respond to synchronization requests from H1, H2, ... H8.

The experiments are run for at least 1800 secs and each switch synchronizes by sending 500 *DPTP* packets/sec unless otherwise mentioned. We limit to 500 packets/sec (every 2ms) due to accuracy reporting at the control-plane for each synchronization. Each call to accuracy reporting involves calculation of current timestamp from different virtual switches on the same switch and writing to a file. This process takes approximately 1.5 ms. We later show in our evaluation that we can scale up to ~ 2000 DPTP packets/sec/port.

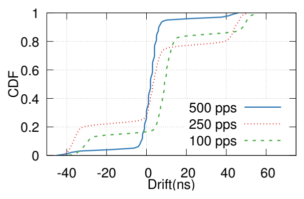
6.1 Switch-to-Switch Synchronization

The virtual switches share a common ground-truth clock when they belong to the same physical switch. As a result, synchronization accuracy calculation is possible at the same time instant in the two virtual switches who belong to the same physical switch. We plot the synchronization error between S5-M (1-hop), S1-M (2-hop), S2-S6 (3-hop) and S1-S3 (4-hop) under idle link condition in Figure 11(a) and with heavy cross-traffic in Figure 11(b). Note that the synchronization still happens over a single hop between adjacent parent-child switches (§4). Figure 11 only captures the accumulated effect of synchronization error in between switches which are 1, 2, 3 and 4 hops away.

Idle Condition. We observe that the synchronization error does not go beyond 6 ns for single-hop (S5-M) and 12 ns for two-hop (S1-M) under both idle and heavy cross-traffic condition. In the case of S2-S6 (3-hop) and S1-S3 (4-hop) we experience higher synchronization error bounded by about ~50 ns due to compounded wire-delay asymmetry error with an increase in hop-count. To understand the distribution of



a.Drift (in ns) between individual DPTP requests



b. CDF of Drift

Figure 13: Drift captured at S6 for different DPTP synchronization intervals

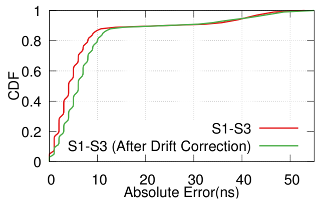


Figure 14: *DPTP* w/ and w/o linear drift correction

synchronization error, we plot the CDF of the absolute value of error in Figure 11(c). We observe the median error to be about 2 ns and the 99th percentile to be about 6 ns for single hop (S5-M). For two-hop (S1-M), the median is about 3 ns and 99thpercentile is about 8 ns. For three-hop (S2-S6) and four-hop (S1-S3), we observe the median to be about 4 ns and a long tail with the 99th percentile about 47 ns. This is due to the effect of drift between the clocks of two different physical switches (explained later). Note that the synchronization paths from the master switch till the switches S1, S2, S3 and S6, include at least one segment where the two synchronizing virtual switches belong to two different physical switches.

Heavy Cross-traffic. When there is heavy cross-traffic across the topology, in Figure 11(b) we observe no effect on the accuracy between S1-M and S5-M when compared to Figure 11(a). Figure 11(d) shows the CDF of the absolute value of error under heavy cross-traffic. While S5-M and S1-M remain unchanged, we observe a higher variation of error in S2-S6 and S1-S3. The median error is about 8 ns for S2-S6 and 19 ns for S1-S3. Note that the maximum buffer size is 1.5 ms. Changing the buffer size will not affect the accuracy since we use prioritized queueing for DPTP packets and queuing delays are accountable. The higher variation could be due to the effect of non-linearity of the clock during stressed conditions [5] and higher variation of drift (explained next).

Effect of Drift. To understand the drift behavior between two physical switches, we measure the drift of the clock at S6 when it synchronizes with the Master. S6 sends a *DPTP* request and uses the T_{Now_M} from the response to calculate T_{Ref1}. At the same time, it stores the current T_{RespIg} as $T_{Elapsed1}$. S6 then sends another *DPTP* request, and calculates T_{Ref2} while storing the current T_{RespIg} as $T_{Elapsed2}$. Now the drift can be calculated as T_{Ref2} - (T_{Ref1}+($T_{Elapsed2} - T_{Elapsed1}$)). We verified that this value is almost zero in the case of two virtual switches implemented on the same physical switch because there is no drift between the two. While our method to measure the drift may not be as accurate as using an external clock reference, the expected errors due to unaccounted wire delay asymmetry are very small. Since we generate the *DPTP* request packets from the control-plane, the packet-arrival rate may not be accurate as configured. In order to make the drift measurement as accurate as possible, we log the interval ($T_{Elapsed2} - T_{Elapsed1}$) for each drift datapoint and normalize. When S6 is synchronized every second (1 DPTP pkt/sec), we observe an average drift of about 775 ns (min:664 ns, max:886 ns). In Figure 13(a), we plot the observed drift at S6 between individual DPTP request/response with the Master at different rates of 100/250/500 requests/sec. We observe that drift between the switches fluctuates and does not increase linearly. While Figure 13(a) shows a 'zoomed in' picture, the trend in the drift remains the same even over a longer period (~30 mins). Additionally in Figure 13(b), we plot the CDF of drift values observed at S6. We observe a high variance in drift values. The median drift is around 11 ns at 100 pkts/sec, 7 ns at 250 pkts/sec and 3 ns at 500 pkts/sec. The drift is about 43 ns at 99th percentile with 500 DPTP pkts/sec and 90th percentile with 250 DPTP pkts/sec. These drift statistics provide insight in to why the synchronization errors between S1-S3 and S2-S6 reach ~50 ns occasionally during idle and heavy cross-traffic conditions (Figures 11(c), 11(d)).

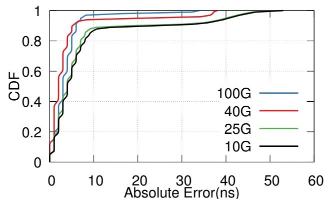
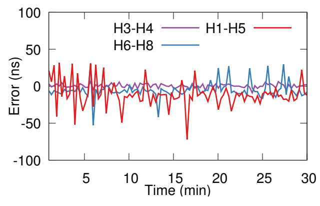
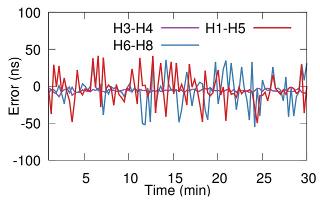


Figure 15: Error of Clock Synchronization (S1-S3) for different link-speeds

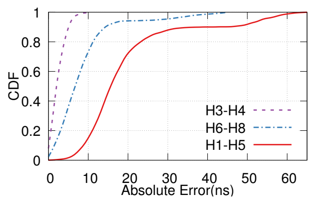
Linear Extrapolation of Drift. HUYGENS performs synchronization every 2 seconds and uses linear extrapolation of past estimates to correct drifts. Based on our measurements of S6's drift wrt Master (Figure 13), the drift may not be linear across small time-scales. We tried applying HUYGEN's method of drift correction, by averaging the past drift estimates (2 ms window) to do drift correction at S6 before responding to requests from S3. Ideally, this should compensate S6's drift (wrt Master) between its consecutive synchronizations with the Master. This should, in turn, improve the synchronization between S1-S3 due to less impact of drift on S3. In Figure 14, we plot the CDF of synchronization error between switches S1-S3 with and without drift correction at S6. We observe that there is no improvement and in fact drift correction reduces the accuracy slightly. This can be explained by our observation that the clock drift over small time-scale is not linear. Hence, we do not use any statistical drift adjustments.



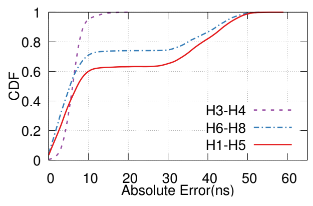
a. Idle Condition



b. Heavy cross-traffic



c. Idle Condition CDF



d. Heavy cross-traffic CDF

Figure 16: Error in *DPTP* Switch-to-Host synchronization

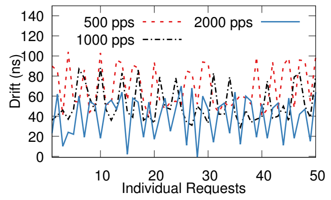


Figure 17: Drift (in ns) at the Host NIC Clock between individual DPTP requests

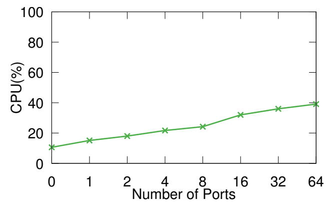


Figure 18: CPU util % to handle follow-up packets

Effect of Link-speeds. We perform switch-to-switch synchronization by changing the link-speeds to 25/40/100G between all the switches. We plot the CDF of error between S1-S3 (longest path in the topology) in Figure 15. As shown in our analysis in §3, we observe a reduction in the error with higher link-speeds. We observe a similar trend for 40G and 100G, with 100G marginally better at the 95th percentile.

6.2 Switch-to-Host Synchronization

We send *DPTP* requests from hosts to their parent switches at 2000 requests/sec. By using NIC timestamps, we capture the synchronization error in between H3-H4 (hosts connected to the same switch), H6-H8 (hosts sharing the same aggregation switch) and H1-H5 (inter-pod hosts). We plot the synchronization error (smoothened) over a 30-minute run under idle condition (Figure 16(a)) and when there is a line-rate upstream cross-traffic (Figure 16(b)). We also plot the CDF of the errors in Figure 16(c) and Figure 16(d) for the two conditions. We observe a median error of 3 ns between H3-H4 under idle condition and 7 ns under heavy cross-traffic condition. The error seems to be higher for the hosts connected to different physical switches due to the drift between the switches as noted earlier. Between H1-H5, we observe a 75th percentile error of 20 ns under idle condition. Under heavy cross traffic, we observe an increase in the error to 31 ns. However, there is minimal change in the 99th percentile error. We observe similar trend in the case of H6-H8 which are separated by four hops. The increase in error from 60th percentile in heavy cross-traffic conditions can be attributed to the usage of the profiled value $AvgNicWireDelay$, upon the feedback of high traffic-rate(R) from the switch.

Drift at NIC. Following similar methodology as in §6.1, we measure the drift at the host's NIC clock. This drift gives an idea of how many requests/sec need to be generated from the host to keep the synchronization error low. At 1 DPTP request/sec, we observe that the host NIC drifts at ~21μs/sec. We also observe that the overall range of the drift is 20874-21061 ns/sec. To understand the drift behavior in the NIC, we plot the drift occurred between individual DPTP requests at different rates in Figure 17. We observe that at short time-scales the drift in the NIC is also not linear and has fluctuations. Therefore, to maintain the error in the order of 10s of ns, it is necessary to perform *DPTP* requests more often. While Figure 17 shows a 'zoomed in' picture, the trend in the drift remains the same even over a longer period (~30 mins).

6.3 Scalability

The scalability and accuracy of *DPTP* is hugely dependent on the follow-up delay. Suppose, there are X *DPTP* request/sec on a specific port. Therefore, the control-plane has to read the port Tx register within $\frac{1}{X}$ sec to get T_{RespTx}. If the control-plane fails to read the T_{RespTx} within that deadline, the next response packet would be sent out through the same port, thus overwriting the value of the previous T_{RespTx}. We use learning filter from data-plane to deliver digests to the control-plane. The learning filter is configured to immediately send the digest notification to the control-plane. We observe that the follow-up delay[6] is about ~429 μs *regardless* of the load of packets on various ports. This means that the control-plane can keep up with requests at the rate, (1 second/ 429μs) = 2331 requests/sec/port. Considering the maximum drift reported by [5] of 30μs/sec, 2331 requests/sec/port can keep the drift of the worst-case clock under 13 ns. Since, the follow-up delay directly impacts the accuracy (in terms of drift), it is ideal to prioritize the follow-up packets to avoid buffer delays. Figure 18 shows the CPU utilization of the control-plane program with 1000 *DPTP* requests/sec from each port. The baseline CPU utilization is about 10 %. We

[6]Digest notification to control-plane + follow-up packet to reach data-plane

Table 4: Hardware resource consumption of *DPTP* compared to the baseline switch.p4

Resource	switch.p4	*DPTP*	Combined
SRAM	29.79%	6.25%	36.04%
Match Crossbar	50.13%	4.62 %	54.75%
TCAM	28.47%	0.0%	28.47%
Stateful ALUs	15.63%	15.63%	31.26%
Hash Bits	32.35%	3.99%	36.34%
VLIW Actions	34.64%	4.43%	39.07%

observe a steady increase in CPU utilization and it reaches only up to 40% with incoming requests from 64 ports concurrently.

6.4 Resource Utilization

We evaluate the hardware resource consumption of *DPTP* compared to the baseline switch.p4 [34]. The switch.p4 is a baseline P4 program that implements various common networking features applicable to a typical datacenter switch. We illustrate the percentage of extra hardware resources consumed by *DPTP* in Table 4. We observe that while *DPTP* consumes a relatively higher proportion of stateful ALUs to maintain the reference clock and to perform the arithmetic operations to calculate and adjust the reference. The requirement of other resources is less than 7%. Hence, *DPTP* can fit easily into datacenter switches on top of switch.p4.

Bandwidth Consumption. A DPTP packet is an Ethernet frame packet with a DPTP header [$Type$, $T_{Now}(hi)$, $T_{Now}(lo)$, T_{ReqRx}, T_{RespEg}, R] of size [1+4+4+4+4+4] bytes = 21 bytes. Due to minimum Ethernet frame size, this translates to 64 bytes "on wire" including the Ethernet FCS. 2000 DPTP requests/sec could keep the clock drift errors low to about 15 ns considering the worst-case clock drift of 30 μs/sec. Hence, total bandwidth usage for 2000 DPTP requests/sec is about 6000 (req + reply + follow-up packets) * 64 bytes = 3.07 Mbps per link, which is 8 times lower than the bandwidth usage (25 Mbps for T-1 testbed) of HUYGENS [5].

7 DISCUSSION AND FUTURE WORK

Currently *DPTP* works the best if the entire network supports *DPTP* in the data-plane. In future, we would like to gauge the behaviour of *DPTP* under partial deployment of programmable switches which could increase the synchronization error. Tackling these uncertainties with statistical extrapolation in the network data-plane can help reduce the error. It would be interesting to implement sophisticated extrapolation techniques like Linear/Polynomial/Log Regression using bit-shifts in data-plane and exponential smoothing using low pass filters. Such statistical extrapolation could further help in reducing the number of *DPTP* requests thereby reducing switch CPU and bandwidth consumption. Future works could also look into more sophisticated ways of estimating $NicWireDelay$ based on the traffic-rate. Currently, the timestamps for follow-up packets are obtained from a port-side register by the control-plane. The reason is because the port-side register is not integrated with the egress pipeline and is accessible only from the control-plane. With hardware enhancements to make the port Tx timestamp accessible at the egress pipeline, a follow-up packet could be generated at the data-plane itself and the Tx timestamp could be embedded in the follow-up packet in the egress pipeline. We believe this is not the case currently since there was no application that demanded this. However, *DPTP* could hugely benefit from this minor hardware enhancement: *Zero* CPU consumption and an *order of magnitude* reduction in the follow-up packet latency. In future, we also plan to expand our measurement study to include other NIC standards, different cable lengths, and cable medium (optic cables).

We believe that *DPTP* will encourage researchers to rethink the implementation of distributed protocols like NetPaxos [6], which have strong assumptions like message ordering in the network. A recent work [35] has shown that designing network-level mechanisms to provide mostly-ordered multicast could improve the latency of Paxos by 40%. If the entire network data plane is time-synchronized, it is possible to observe and capture data-plane events over the network in an ordered manner [36]. This can facilitate building highly accurate in-network debugging tools. SpeedLight [14] which performs network-wide snapshots of switch states to understand network-wide behaviour, uses PTP to synchronize the switches with a median accuracy of 6.4 μs. By employing synchronization in the data-plane, the snapshot synchronization could be brought down to a few nanoseconds.

8 CONCLUSION

In this paper, we design and implement *DPTP*, which to the best of our knowledge, is the first precise time synchronization protocol for the network data-plane. Through a measurement study, we quantify the variability in delays at every stage of data-plane processing. We incorporate these observations to offset the variable delays in the design and implementation of *DPTP* . We evaluate *DPTP* on a hardware testbed and observe synchronization error of within ~50 ns between switches and hosts separated by up to 6 hops and under heavy traffic load.

Acknowledgements. We thank the anonymous reviewers and our shepherd Mina Tahmasbi Arashloo for their valuable feedback. We also thank Vladimir Gurevich and the Barefoot support team for their generous help and support. This work was carried out at the SeSaMe Centre, supported by Singapore NRF under the IRC@SG Funding Initiative. It was also partly supported by the Singapore Ministry of Education tier 1 grant R-252-000-693-114 and National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Programme (grant no. NRF2015NCR-NCR002-001).

REFERENCES

[1] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[2] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, 2008.

[3] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of SIGCOMM*, 2016.

[4] R. Zarick, M. Hagen, and R. BartoÅą. Transparent clocks vs. enterprise ethernet switches. In *Proceedings of IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2011.

[5] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proceedings of NSDI*, 2018.

[6] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proceedings of SOSR*, 2015.

[7] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *SIGCOMM CCR*, 2016.

[8] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of NSDI*, 2018.

[9] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of SOSP*, 2017.

[10] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of ASPLOS*, 2017.

[11] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of SOSP*, 2017.

[12] Amin Tootoonchian, Aurojit Panda, Aida Nematzadeh, and Scoot Shenkar. Distributed shared memory for machine learning. In *Proceedings of SysML*, 2018.

[13] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. P4CEP: Towards In-Network Complex Event Processing. In *Proceedings of NetCompute*, 2018.

[14] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of SIGCOMM*, 2018.

[15] The world's fastest and most programmable networks. https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[16] C. D. Murta, P. R. Torres Jr., and P. Mohapatra. Qrpp1-4: Characterizing quality of time and topology in a time synchronization network. In *Proceedings of Globecom*, 2006.

[17] Meinberg PTP Client. https://www.meinbergglobal.com/english/products/ptp-client-software-win-linux.htm.

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

[19] FSMLabs Timekeeper Appliance. https://www.fsmlabs.com/timekeeper/enterprise-appliance/.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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 OSDI*, 2012.

[21] W. Lewandowski, J. Azoubib, and W. J. Klepczynski. Gps: primary tool for time transfer. *Proceedings of the IEEE*, 1999.

[22] Cavium. Xpliant ethernet switch product family, 2018.

[23] Barefoot Networks. Tofino, 2018.

[24] Intel. Flexpipe, 2018.

[25] Cisco UADP. https://blogs.cisco.com/enterprise/new-frontiers-anti-aging-treatment-for-your-network.

[26] Broadcom Trident 3. https://packetpushers.net/broadcom-trident3-programmable-varied-volume/.

[27] Xilinx SDNet. https://www.xilinx.com/products/design-tools/software-zone/sdnet.html.

[28] Portable Switch Architecture. https://p4.org/p4-spec/docs/PSA-v1.0.0.pdf.

[29] In-band Network Telemetry. https://p4.org/assets/INT-current-spec.pdf.

[30] Pravein Govindan Kannan, Ahmad Soltani, Mun Choon Chan, and Ee-Chien Chang. BNV: Enabling scalable network experimentation through bare-metal network virtualization. In *Proceedings of USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2018.

[31] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of SIGCOMM*, 2013.

[32] Junda Liu, Baohua Yan, Scott Shenker, and Michael Schapira. Data-driven network connectivity. HotNets-X, 2011.

[33] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of IMC*, 2015.

[34] P4 Language Consortium. 2018. Baseline switch.p4. https://github.com/p4lang/switch/blob/master/p4src/switch.p4.

[35] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of NSDI*, 2015.

[36] T. Mizrahi and Y. Moses. The case for Data Plane Timestamping in SDN. In *Proceedings of IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016.