Graduation Thesis Academic Year 2016

Prism: A Proxy Architecture for Datacenter Networks

Advisors:
Professor Hideyuki Tokuda
Professor Jun Murai
Professor Osamu Nakamura
Professor Hiroyuki Kusumoto
Associate Professor Kazunori Takashio
Associate Professor Rodney D. Van Meter III
Associate Professor Keisuke Uehara
Professor Jin Mitsugi
Associate Professor Jin Nakazawa
Professor Keiji Takeda

Keio University
Faculty of Policy Management
Yutaro Hayakawa

river@ht.sfc.keio.ac.jp

Keio University Academic Year 2016

Prism: A Proxy Architecture for Datacenter Networks

Keio University Faculty of Policy Management

Yutaro Hayakawa

Abstract

Prism: A Proxy Architecture for Datacenter Networks

Summary

In datacenters, workload throughput is often constrained by the attachment bandwidth of proxy servers, despite the much higher aggregate bandwidth of backend servers. We introduce a novel architecture that addresses this problem by combining programmable network switches with a controller that together act as a network "Prism" that can transparently redirect individual client transactions to different backend servers. Unlike traditional proxy approaches, with Prism, transaction payload data is exchanged directly between clients and backend servers, which eliminates the proxy bottleneck. Because the controller only handles transactional metadata, it should scale to much higher transaction rates than traditional proxies. An experimental evaluation with a prototype implementation demonstrates correctness of operation, improved bandwidth utilization and low packet transformation overheads even in software.

Keywords:

1 Proxy 2 Software Defined Network 3 Software Switch 4 High Performance Networking 5 TCP

Keio University Faculty of Policy Management

Yutaro Hayakawa

卒業論文要旨　　2016年度(平成28年度)

Prism: A Proxy Architecture for Datacenter Networks

論文要旨

近年のデータセンターにおいて, プロキシサーバのような複数のサーバのトラフィックをすべて中継するようなサーバがデータセンター自体のキャパシティにかかわらずネットワークのボトルネックとなることがある. 本研究ではこの問題をプログラマブルなSDNスイッチとそのコントローラ及びカスタマイズされたバックエンドによって解決できるシステム,Prismを設計・実装した. 既存のプロキシアーキテクチャと同じくクライアントのリクエストは透過的に複数のバックエンドにリダイレクトされるが, プロトコルのペイロード転送部分はプロキシサーバを介さずバックエンド-クライアント間で直接行われる. これによって既存のプロキシアーキテクチャにおけるボトルネックを解消し, スループットの向上を図ることができる. 本研究において実装したプロトタイプは既存のプロキシアーキテクチャにおいて使用不可能であったネットワークの帯域幅を使用可能にし,Prismが行うパケット処理はソフトウェアによる実装においても少ないオーバーヘッドで実現できることを明らかにすることができた.

キーワード：

1 Proxy　2 Software Defined Network　3 Software Switch　4 High Performance Networking　5 TCP

慶應義塾大学総合政策学部

早川 侑太朗

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter discusses the background and the motivation of this research. First, overview our research area. Second, bring up problem that we have attacked, and at last we introduce our proposal system named Prism briefly.

1.1 Introduction

A datacenter fabric interconnects network switches, to provide capacity for many servers to communicate at the same time. The trend has been towards topologies that isolate communications between one server pair from those between others, often providing full bisection bandwidth [1, 2], to provide a more predictable service.

However, applications may still experience limited throughput even on topologies with full bisection bandwidth. When one server proxies traffic to and from multiple other servers, its attachment bandwidth to the core limits the aggregate throughput of the workload Figure 1.1. Such proxy-based communication is common and includes distributed storage [2, 3], MapReduce [4] and web workloads, all of which require stateful application-level logic to operate on application transactions at the proxy. Naive approaches to alleviate this problem simply increase the fabric attachment bandwidth of proxy servers, by installing additional and/or faster NICs. This complicates hardware configuration, increases cabling costs, and reduces provisioning flexibility—all for limited returns and leaving backend bandwidth under-utilized.

This paper presents the Prism architecture, which provides a superior solution. It recognizes that one role of a proxy—relaying transaction payload data over TCP connections—can be separated from its application-level processing, when such processing only involves the metadata (e.g., request and response headers) of a transaction. Prism offloads the relaying of transaction payload data to the network fabric, by utilizing programmable network switches to transform payload packets at line rate. It was originally designed for forthcoming P4 [5] hardware switches, but achieves good performance even when implemented inside a software switch [6].

Prism remains a true proxy architecture with transaction-granularity operation, even when applications reuse TCP connections to issue long streams of transactions. This is not just challenging but essential to support legacy and modern application protocols such as HTTP, memcached, iSCSI and

NFS. Many related proposals in this space—Maglev [7], Ananta [8], Duet [9], Rubik [10]—merely load-balance a connection to a backend server once upon establishment, but are unable to execute subsequent transactions against different backend servers. This causes significant load imbalance over time [11].

We show that Prism can improve throughput for data transfers larger than 2 MB and demonstrate that its packet transformations are cheap enough to forward traffic at tens of Gb/s even when implemented in a software switch. This allows datacenter operators to initially deploy Prism via a software switch upstream of the leaf switches, instead of requiring programmable hardware switches.

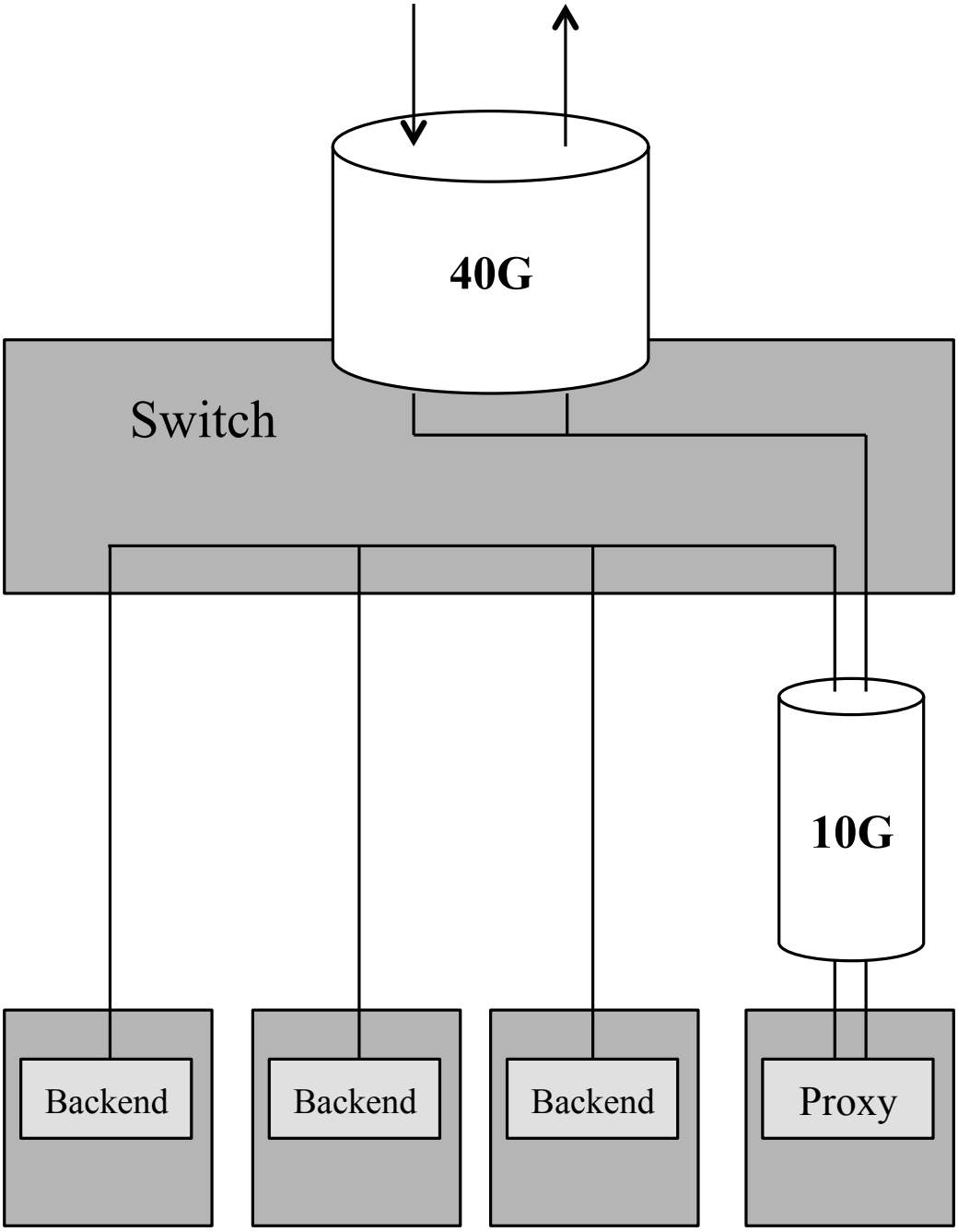


Figure 1.1: All incoming and outgoing traffics will be limited bandwidth to 10Gbps in this case

Chapter 2

Design

This chapter describes Prism's design. Prism system is consists of several components which act together over network. First, we show Prism's behavior using sequence diagram and at the end of this chapter, we show some design limitation, other possible features and so on.

2.1 Design

This section discusses the components involved in the Prism architecture using the packet sequence diagram in Figure 2.1; it also discusses some design alternatives.

Prism uses a controller application that uses software-defined networking (SDN) interfaces to dynamically program a set of SDN network switches to transparently redirect the transactions a client issues towards a logical server IP address to different physical backend servers. Prism can migrate a TCP connection between the controller and a different backend server for each client transaction, by instructing the programmable switches to rewrite TCP headers. Backend servers communicate over these already-established, migrated connections. At any point in time, the end point that is handling the client connection (controller or backend server) is responsible for maintaining TCP semantics by ACK'ing, retransmitting, etc. A connection is only migrated when it is guaranteed that there is no un-ACK'ed data in flight.

Although this paper always talks about a single controller and a single switch, an actual deployment will use multiple controller instances and switches together with a suitable consistency protocol to increase scalability and fault-tolerance.

2.1.1 Connection Establishment

The use of Prism is transparent to the clients, which are unmodified and execute their normal protocol implementation. Clients connect to a "logical" server IP address that is initially forwarded to the Prism controller. The Prism controller handles TCP connection establishment and teardown with the clients and maintains sufficient metadata to determine which backend server should handle a given client request. It also parses request headers and programs the Prism switch to rewrite the packet headers of TCP segments carrying request and response payload data.

A client begins a transaction sequence in its usual way, that is, by open-

ing a TCP connection with a server. Step 1 in Figure 2.1 illustrates that the client performs the required TCP three-way handshake with the Prism controller, negotiating any desired TCP options. Solid arrows in Figure 2.1 indicate TCP packets sent on the client connection, dashed lines indicate Prism control messages between the controller, switch and backend servers.

2.1.2 Request Parsing

In step 2 of Figure 2.1, the client begins a transaction by sending a request, which the controller receives and parses. When the controller determines that it has received the entire request header, it consults the metadata it maintains about the backend servers to select one to handle the request. It sends PUSH/ACK in step 3, setting the TCP receive window to zero if the request is a read. This prevents the client from issuing additional requests while the controller has handed off the request to the backend. If the client already included some request payload data after its request header, the controller ACK's the reception of the request header *only*, forcing the client to retransmit any request payload data, so it will reach the backend.

2.1.3 Request Hand-Off

In step 4, the controller instructs the Prism switch to rewrite the destination IP address of packets sent from the client to that of the chosen backend server, and to rewrite the source IP address of packets sent from that backend server to the client to that of the logical IP address. The consequence is that any following (payload) packets will be exchanged directly between the client and the backend, with the switch fabric performing the required rewriting (in hardware, once P4 switches are available.)

After the switch is configured, the controller contacts the chosen backend server in step 5 and passes it sufficient information about the TCP connection state and the client request so that the server can take over the connection and serve the request. This includes application-level information about the

client request as well as TCP port, sequence and ACK numbers and TCP options negotiated for both directions of the connection.

2.1.4 Backend Request Handling

After receiving the hand-off control message from the controller, the backend server handles the client request. Figure 2.1 illustrates a client read, where the backend server first sends a response header in step 6, followed by the payload data in step 7. (For a client write, the order would be opposite; first payload data would be read and then a response header would be sent.)

The backend server needs to send and receive TCP packets that, after header rewriting by the switch, are accepted by the client as belonging to the already-established connection between the controller and the client. Because the backend server is aware of the header rewriting the switch performs, it must only make sure that TCP source and destination ports as well as sequence and ACK numbers and any TCP options that the controller negotiated with the client are correct in transmitted segments.

When the client request is a write, the backend must only ACK the payload data of that request (step 8), and not any additional data the client have sent, such as a next request. For client reads, the backend sets the TCP receive window to zero to prevent the client from sending any further data, but this is not possible for writes. Additionally, the backend must ignore (i.e., not ACK) any TCP FIN the client sends, to prevent the client from closing the connection before it can be handed back to the controller. Handing a connection back to the controller is required for proper connection tracking and metadata maintenance.

After the main data exchange has completed, the server notifies the controller in step 9 and includes sufficient information about the progression of the connection (i.e., new TCP sequence and ACK numbers, timestamp options, etc.) so that the controller can take over the connection. For the backend server, this concludes serving the request.

If an unforeseen event prevents the backend server from serving the client

request, it needs to notify the controller about this (step 9). The controller can then reset the TCP connection to the client, in order to signal a failure. In addition, the controller may want to set time-outs for handed-off requests to handle crashing backend servers.

2.1.5 Preparing for Next Request

After the controller receives the request completion notification from the backend in step 9, it removes the header rewrite rules from the switch (step 10). Then, it synthesizes an ACK to the client in step 11 that re-opens the receive window (if it was closed for a prior read request). This allows the client to issue its next request.

If the client sends a new request, operation resumes at step 2. If the client closes the connection by sending a FIN, the controller continues the FIN handshake to close the connection in step 12. The controller may also itself initiate the connection teardown by sending a FIN.

2.1.6 Design Discussion

This section discusses aspects of the Prism design, including variants and future extensions.

Supporting TLS: If the application protocol is secured with TLS [12], the client will begin a TLS handshake over the connection after step 1. To support TLS, the controller needs to be extended to complete this handshake. It must also pass sufficient information about the state of the TLS session to the backend server in step 5, the TLS implementation at the backend servers must be augmented to support bringing up a TLS session directly into the "handshake finished" state, and the backend must pass sufficient information about the progression of the TLS session to the controller in step 9.

Eliminating controller notifications: If the controller knows the size of the payload data for a given client request, e.g., based on the request headers or the metadata it maintains, some of the notification delay in step 9 may

be reduced. The controller could configure the switch to monitor progression of the respective TCP connection, e.g., by using counters to track the TCP sequence and ACK numbers. Once the configured amount of data has been exchanged, the switch would notify the controller, or the switch itself could revert the connection back to the controller by removing the respective rewrite rules. Either of those two approaches may be faster than explicit notifications by the backend.

Speculative caching of rewrite rules: After step 9, the controller may want to direct the next client request to the same backend server, it could speculatively postpone the removal of the switch rules until after it has parsed the next request. In such a case, the controller could skip step 5 on the next request, reducing latency.

Packet transformations: Prism uses a programmable switch to transform packets as they are forwarded through the fabric. It needs to modify TCP and IP headers, so P4 [5] switches appear to offer a simple way to implement the needed functionality, due to their ability to perform operations on arbitrary, application-defined headers. More readily available OpenFlow [13] switches do not support modification of all the required TCP header fields.

The Prism design does not require that all packet transformations occur atomically or even at a single location along the path. Instead of in a network switch, packet transformations could also be implemented directly on the backend servers, e.g., in a software switch or host firewall inside the hypervisor or the guest OS of the backend servers, or a programmable NIC that provides fabric attachment—or any combination thereof. The key takeaway here is that the general Prism design can be instantiated in different ways with different trade-offs.

Design limitations: For a small-message transactional workload, i.e., where requests and responses fit into few TCP packets, Prism may not be a suitable solution. In such cases, the overheads associated with Prism—receive window management, rule addition and removal, controller notifications—cannot be sufficiently amortized. This may include protocols such as

HTTP/2 that allow aggressive interleaved pipelining of chunked data, which Prism currently needs to treat as individual requests. Supporting such workloads will require further modifications to the backend and Prism design, so that the controller can let a connection remain at a single backend server while several concurrent transactions are being executed.

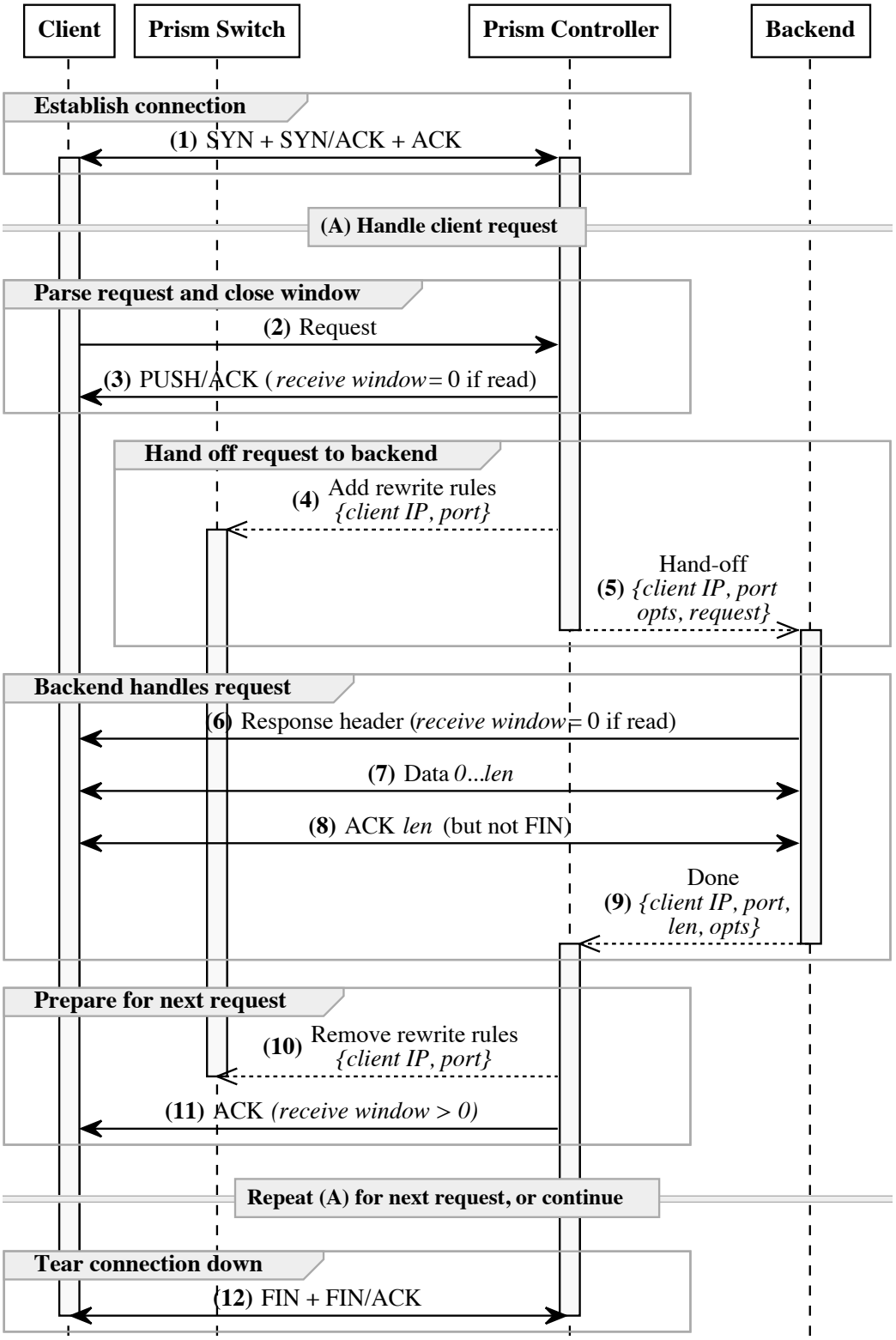


Figure 2.1: Prism operation.

Chapter 3

Implementation

This chapter discusses the current implementation for our prototype implementation that can split HTTP/1.1 `GET` request sequences over IPv4 and Ethernet.

3.1 Prism Controller

Figure 3.1 shows the structure of Prism controller. Prism controller has two important roles in Prism system. First role is managing control part of protocol transaction like TCP connection establishment, HTTP request parsing, and controlling TCP receive window size. For implementing these features, standard OS's socket API is not suitable, because some of operation like controlling window size is not supported in general OSs like Linux or FreeBSD. Instead of using standard socket API, we use netmap [14] and implemented simple TCP and HTTP on top of it.

Prism TCP implements only part of TCP functionality. It can perform three-way handshake, connection teardown and acking to clients segments and never do retransmit, flow control or congestion control. It have connection hash table which takes source/destination IP address, source/destination TCP port as a key, for storing TCP state information.

Prism HTTP doesn't have full HTTP header parsing functionality. It emulates HTTP header parsing by looking at first GET strings of client's request.

Second role is handing off TCP connections to backend servers. Connection handoff is triggered when client sends request to controller. When controller receives request from client, controller sends PUSH/ACK to client with receive window size zero and start hand off operation.

Backend server migrates TCP connection using `TCP_REPAIR` [15] functionality of Linux which requires sequence number, acknowledge number, peer IP address, peer TCP port number, self IP address, self TCP port number and optionally TCP options information.

All these information are already collected while controller communicate with client, so, controller only needs to configure switch, establishes TCP connection via standard socket and sends required information with client's HTTP request data.

Switch configuring is done by sending special packet to switch. Connection hand off threads synthesize it and send it to switch via netmap API.

While hand off operations contain expensive IO, and our simple TCP uses non-blocking IO multiplexing with single thread for raw packet IO, they are separated to other threads.

3.2 Prism Switch

Figure 3.2 shows structure of Prism Switch. Prism switch can be implemented on programmable hardware switch like P4 [5], however, while P4 hardware switch is not available, we implemented Prism switch on top of the mSwitch [6] as a module.

Prism Switch defines two tables. One is for Prism functionality and another is just for default routing table based on static IPv4 address look up. First table looks at TCP connection four-tuple; source and destination IP address as well as source and destination port numbers, if table entry found, it rewrites source or destination IP address and MAC address. If not, just fall back to IP routing.

Due to the `TCP_REPAIR` limitation that can't control receive window size or manipulate TCP flags, the switch also zeroes the receive window for packets from the backend and clears the FIN flag for packets from the client.

3.3 Prism Backend

As described in section 3.1, Prism backend migrates TCP connection between clients and Prism controller using `TCP_REPAIR`. It requires sequence number, acknowledge number, client's IP address, client's TCP port number, Prism virtual IP address, Prism virtual TCP ports and optionally TCP options information.

After migrating TCP connection, Prism backend sends HTTP response header and payload data to client. While Prism backend needs to guarantee there are no unacked bytes on the fly before send response to controller, Prism backend check acked bytes using `TCP_INFO` socket option.

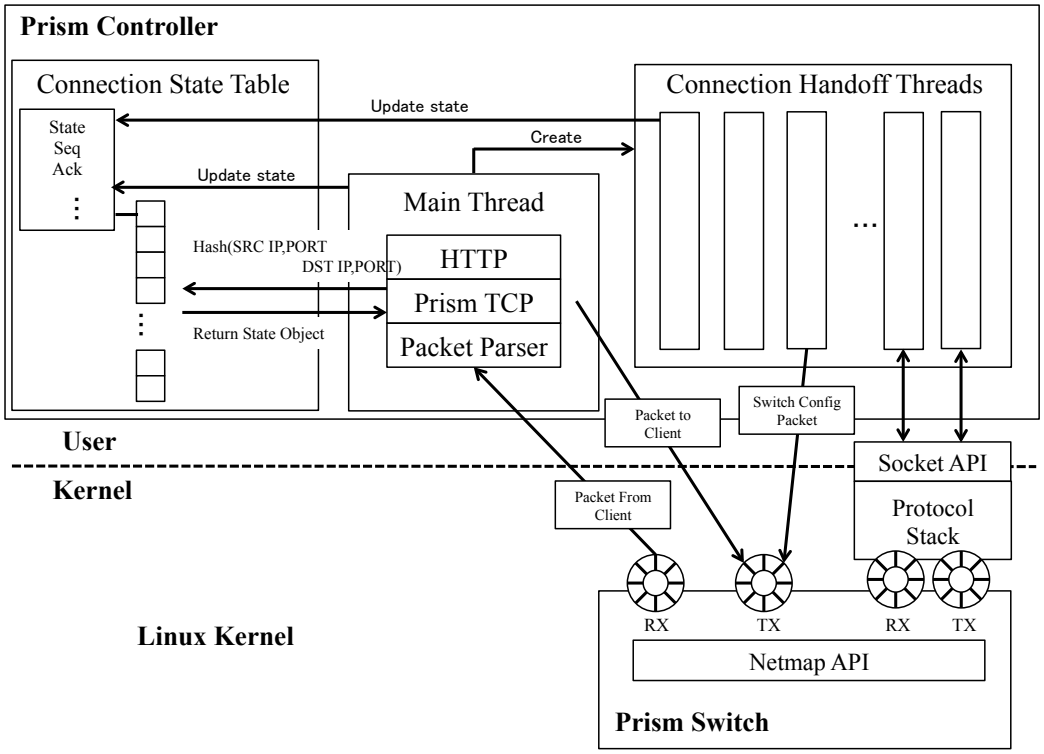


Figure 3.1: Structure of Prism Controller

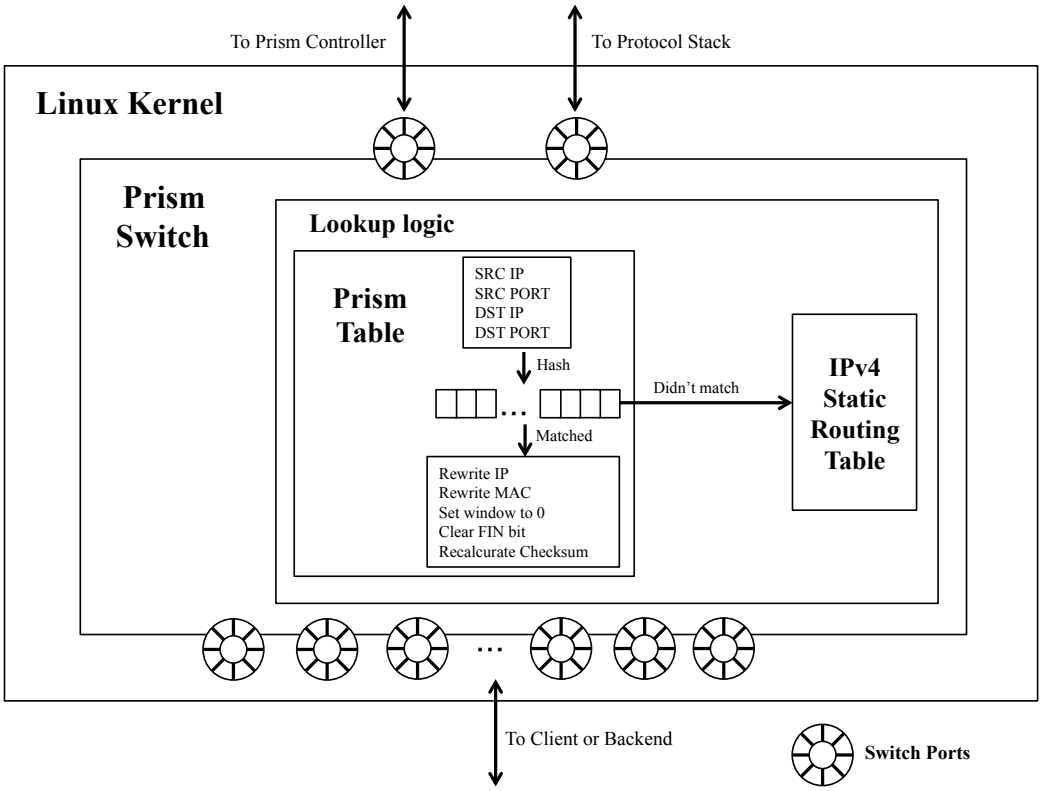


Figure 3.2: Structure of Prism Switch

Chapter 4

Evaluation

This chapter describes our experiment and evaluation for Prism. We evaluate end-to-end throughput of Prism against legacy proxy architecture and overhead of Prism switch.

4.1 Evaluation

Figure 4.1 illustrates the topologies used during the evaluation. The client machine connects into the fabric (emulated by a switch) via two 10 G Ethernet links, emulating a well-connected datacenter. In the Prism case (Figure 4.1a), the mSwitch [6] server also runs the controller. The switch connects to two backend machines via two disjoint 10 G Ethernet links. Note that the controller ideally would run on a separate server, but collocating it with the switch has a negligible performance impact, because the controller only handles a very small amount of traffic and the switch configuration latency is masked by connection hand-off procedures.

To compare Prism against a traditional proxy (Figure 4.1b), we add an additional server that runs the `nginx` proxy [16] and connects to the switch machine via a 10 G Ethernet link; the backend servers run the `H2O` HTTP server [17]. The switch and controller machine are equipped with Intel Core i7-4790K CPUs clocked at 3.5 GHz, the others with Intel Xeon E5630 CPUs clocked at 2.53 GHz. All machines have at least 16 GB RAM; Intel x540 NICs provide all links. The client always runs two `wrk` [18] instances to generate HTTP/1.1 traffic over persistent TCP connections.

First of all, we confirmed our server equipment never be a bottleneck in our experiment. We used pkt-gen a packet generator application which implemented on top of the netmap [14] and easily achieves line rate of 10Gbps ethernet links even for smallest 60B packets. Our initial experiment for all of links in experiment environment shows that our equipment achieves around 13.3Mpps for size of 64bytes packet and around 814Kpps for 1514bytes packet which are enough rate for our experiment workloads.

4.1.1 Packet Transformation Overhead

We benchmark the overhead of the Prism packet transformation to gain insight into the forwarding capacity obtainable with in software. We measure across two virtual ports of an mSwitch instance, since even single-core per-

formance far exceeds the capacity of a 10 G NIC.

Figure 4.2 illustrates forwarding throughput for three different packet processing modules: the Prism packet transformations, an L2 learning bridge and a "no logic" module that statically forwards packets without modifying them. The results show that the Prism module can forward packets on a single CPU core at 7.98 Gb/s for 60 B packets (a rate of 16.63 Mpps) and 66.33 Gb/s for 1514 B packets (a rate of 5.48 Mpps). These numbers translate into 60 ns and 183 ns of per-packet processing cost, respectively, most of which is spent on recomputing the TCP checksum. Once mSwitch supports checksum offloading (to physical NICs), we expect Prism overheads to be similar to the L2 learning module. Forwarding performance can easily be increased by using additional CPU cores. We measure 16.03 Gb/s and 127.1 Gb/s for the two packet sizes when a second CPU is used.

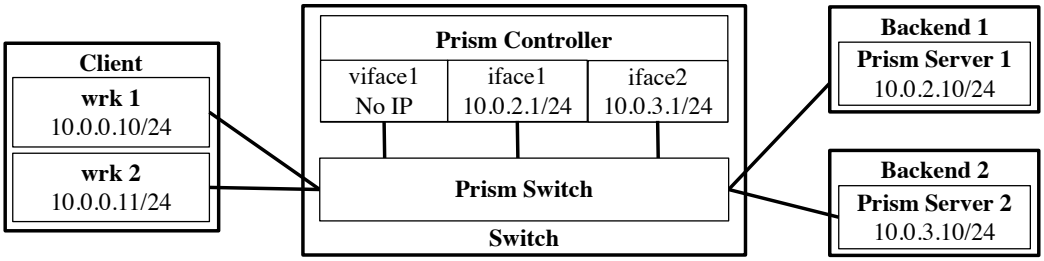
We conclude that the packet transformation overhead of Prism is very low, even when implemented in software, allowing immediate deployment of Prism even before P4 hardware switches are available.

	Startup [µs]		Teardown [µs]		Total [µs]	
Direct	446	$\sigma = 65$	52	$\sigma = 36$	3883	$\sigma = 786$
Proxy	1015	$\sigma = 171$	50	$\sigma = 35$	5142	$\sigma = 839$
Prism	754	$\sigma = 74$	185	$\sigma = 62$	3990	$\sigma = 837$

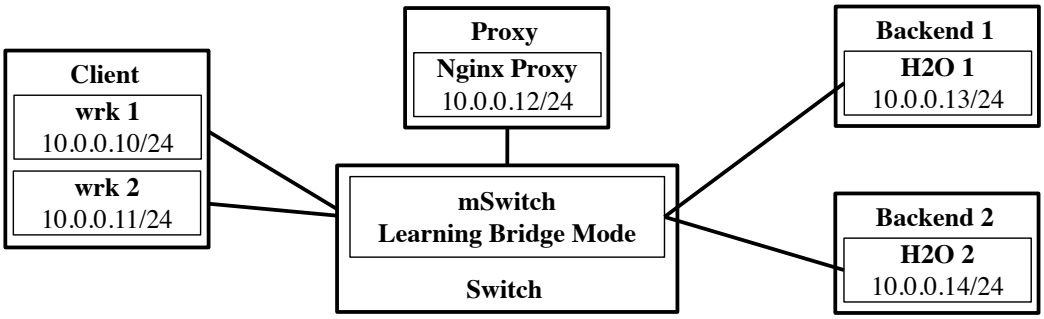
Table 4.1: Latencies of additional request procedures.

4.1.2 End-to-End Throughput

Figure 4.3 illustrates the client-observed end-to-end HTTP/1.1 throughput for different HTTP "OK" response sizes. The experiments use two concurrent TCP connections, each assigned to one path between the client and switch. Throughput of Prism starts exceeding the 10 Gb/s maximum performance achievable with a traditional proxy with object sizes of 2 MB. Due to `TCP_REPAIR` deficiencies, Prism performance is currently limited by starting each response transmission with a default initial TCP window size of



(a) Prism topology.



(b) Proxy topology.

Figure 4.1: Topologies for the evaluation experiments.

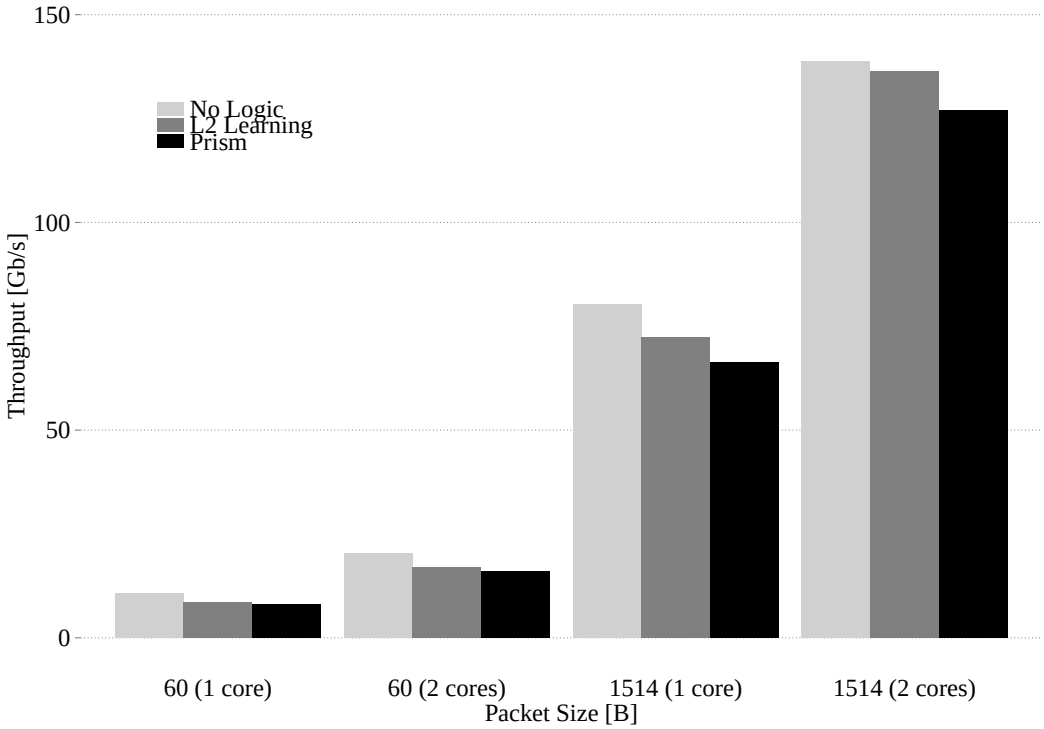


Figure 4.2: Throughput over Prism packet transformation

ten packets, and should further increase (esp. for smaller sizes) once that limitation is removed.

The plot also shows that Prism already beats proxy's throughput from 10KB data transfer. However, this result currently should be just a reference, because Prism controller and Nginx proxy don't have same processing cost. For example they don't implement same HTTP parser and Prism controller uses netmap which improves packet IO. Prism backend and H2O also don't have exact same functionality, so it is not a fare comparison.

Nevertheless, we suppose it is because proxy's payload data forwarding which receives data from backend and sends them via socket API is expensive and Prism's payload forwarding using switch per packet processing is cheap. Detail analysis for this hypothesis would be a future work for this research.

Both a traditional proxy and Prism incur some additional management overhead before and after serving a client request (compared to direct backend communication). Table 4.1 quantifies these overheads. "Startup" overheads incur before the transmission of the first response byte to the client. "Teardown" overheads incur after the last byte of a response has been ACK'ed. To illustrate the relative impact, the "Total" column shows the sum of these times together with the transmission time of a 2 MB response.

At this point, we report these numbers for reference and leave a detailed analysis for future work. We expect higher latencies for Prism (on the order of tens of µs) due to the additional network round trip and OS overheads [19].

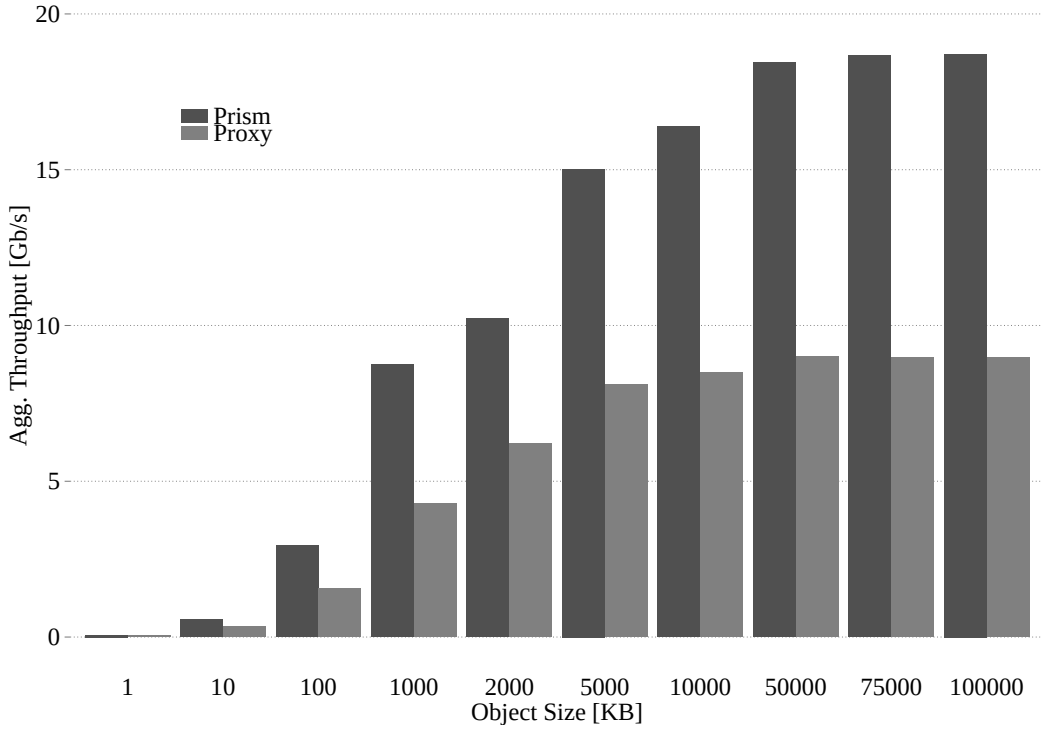


Figure 4.3: End-to-end throughput.

Chapter 5

Related Works

This chapter introduce related works for this research attacking to similar problem and similar approach that introduced in past.

5.1 Load Balancer

Load balancers are historically hardware appliances [7, 8]. However, recent years because of its costs or lack of fault tolerance, software load balancers implemented on top of commodity hardware arises. While they are deployed on servers, they would have same problem as we claimed at beginning of this paper. In this section, we introduce recent works for several layer load balancing method.

5.1.1 L4 Load Balancer

Maglev [7] and Ananta [8] are L4 load balancer which implemented on commodity hardware. Both of them use technique called Direct Server Return (DSR) that after TCP connection spread to particular server, server spoofs their IP address to that of load balancer's virtual address and communicate directly with client. Unlike legacy Network Address Translation (NAT) like approach, load balancer wouldn't be a bottleneck. However, recent research [11] shows L4 load balance i.e., balance TCP connection cause significant load imbalance in datacenter environment. Prism or legacy proxy supports per request load balancing which can spread load more precisely.

5.1.2 L7 Load Balancer

Yoda [20] is an application level load balancer, it means it takes a form of legacy proxy. They don't solve bandwidth limitation problem, but they uses IP tunneling inside load balancer for efficient payload forwarding. However, it is not a essential solution for the problem.

5.2 TCP Related Techniques

Some of the similar TCP related techniques are already introduced in old research. In this section describes those works.

5.2.1 TCP Migration

Prism is similar in design to how TCP Migrate [21] was used to provide fail-over functionality for long-running TCP connections to a set of replica servers [22], but does not require client TCP modifications and is therefore more deployable.

5.2.2 TCP Splicing

The way in which Prism splits TCP connections is similar to TCP Splice [23] and related approaches [24, 25] that were developed for the first large-scale web server farms. Some of these approaches have also been implemented in hardware [26, 27]. Unlike these monolithic approaches, Prism combines a programmable switch for efficient in-network data transformations with a general purpose controller to implement arbitrary request forward logic. In addition, Prism can hand off individual requests arriving over a single TCP connection to different backend servers, whereas many earlier approaches are limited to handing off a connection once. One earlier proposal offered "unsplicing" functionality [28], but is considerably more complex than Prism and requires continuous monitoring of the connection by the forwarder.

Chapter 6

Conclusion and Future Work

In this chapter, we will conclude this paper and discuss about future work of this research.

6.1 Conclusion

This paper described Prism, an architecture that addresses the bandwidth utilization problem with proxy-based systems in datacenter. Prism converts much of the traditional proxy processing functionality into packet-level transformations that can be offloaded to programmable hardware switches or efficiently implemented software switches, lowering processing overheads and increasing bandwidth utilization. We confirmed that Prism improves bandwidth utilization without breaking TCP semantics, by utilizing an unmodified TCP/IP client stack and application.

6.2 Future Work

In this research, we only evaluate very limited functionality of Prism. We need to make fare performance comparison of payload forwarding against proxy. Also, we need to test Prism with more clients or with more uplink capacity. In addition to these, we plan to add some feature to Prism like following,

Implement Prism switch on hardware switch: Once P4 hardware switches become available, we plan on comparing the achievable performance and overheads to our software-switch implementation.

Fault tolerance: Usually, proxy servers has mechanism for fault tolerance. Recent work Yoda [20] proposes fault tolerant L7 load balancer. As described in chapter 5, their idea can be combined to Prism system. This would make our proposal system more solid.

Supporting encryption: For deploying Prism to real world environment, especially web use cases, it is important to support encryption. One particular area of interest is protocols secured with TLS [12], to investigate if additional benefits can be achieved by offloading TLS processing to the backends.

Acknowledgments

Japanese

本研究は慶應義塾大学徳田研究室, 及び私のインターンシップ先である NetApp Deutschland GmbH において行われました. そのため, 本論文は英語で執筆いたしましたが, 私の最大限の謝意を表現するために謝辞は日英併記とさせて頂きます.

はじめに本論文を執筆するまでに至る 4 年間, 私の研究活動をご支援いただいた徳田英幸教授に深く感謝いたします.4 年間の学部生生活においてコンピュータ・サイエンス, こと困難なシステムの研究に関して全くの初学者であった私がその分野に粘り強く取り組めたことは偏に徳田先生のご理解があってのことと存じます. また,4 年間の徳田研学部生生活の随所にて私の活動を支援してくださった高汐一紀准教授, 中澤仁准教授, 米沢拓郎特任助教, 大越匡特任講師に感謝致します.

私の徳田研究室生活は研究グループ LINK から始まりました. 非常に生意気であった新人の私を時に励まし, 時に窘めて下さった寺山淳基先輩, 豊田智也先輩に感謝いたします.LINK から研究グループ MEMSYS が発足した際に私を快く MEMSYS に迎え入れて下さった当時の KG リーダー, 伊藤瑛先輩にはシステムの研究をするにあたって多くの知識を分けていただきました. また,MEMSYS の宇佐美真之介先輩にはよく自宅前まで車で送っていただき, その車中で沢山の実のあるお話をしていただきました.

HTroot として徳田研のインフラ管理に携わるにあたり, 高木慎介先輩には手厚いご指導をいただきました. サーバオペレーション, ネットワーク管理の経験は私の研究においても非常に役立ちました.

現 MEMSYS の KGL である小町芳樹先輩には幾度となく笑いをいただきました. また, どんな逆境にあっても諦めない粘り強さは小町先輩より私にも受け継がれています.

徳田研に同時期に入った仲間の中で唯一 4 年間を共にした池田貴匡君は 1

学年上であるのにも関わらずタメ口で接するなど, 失礼な態度を取り続ける私に寛容に接してくれました.

3 年次よりの MEMSYS の数少ないメンバーのうちの一人であり, 私にドイツに行くきっかけを下さった本多奈々子氏には研究室にいてはわからない外の世界のことを沢山教えていただきました.

また, 卒論の時期を共に乗り切った同期の大戸君, 森重君, 山田君, 小出君, 佐藤君, 稲見君,s103 号室の人々, 博士の坂村美奈先輩, 西山勇毅先輩, その他, 徳田研すべての方々に感謝いたします.

アルバイト先であった株式会社 Studio Ousia においては世界最高水準の自然言語処理技術にいち早く触れ, デモアプリケーションを書かせていただくなど, 貴重な機会をたくさんいただきました.CEO 渡邉安弘氏,CTO 山田育矢氏, 徳田研究室の先輩としてもお世話になった伊藤友隆氏並びに関係者の方々に感謝いたします.

今回この論文を執筆するにあたり, 多大なる技術的, 研究的なご助言を頂き, インターンシップ先のミュンヘンにおいてはプライベート, 仕事共に大変なご支援を頂いた本多倫夫博士に最大の感謝の意をお伝えいたします. 本田博士のご協力なしにはこの研究は決してあらず, 私がドイツでの大変な成長の機会をいただくことはなかったでしょう.

また, サークルでの出会いから私をこの道に誘い, 研究活動においても大きな影響を私に与えて下さった安形憲一先輩に特別の感謝の意を申し上げます.

English

At last, I would like to express my sincere appreciation to Dr. Lars Eggert and Dr. Douglas Santry of NetApp Deutschland GmbH. They gave me a lot of precious experience which would be a lifetime assets for me.

Yutaro Hayakawa February 10, 2017

Bibliography

[1] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 1:1–1:12, New York, NY, USA, 2015. ACM.

[2] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 1–15, Hollywood, CA, USA, 2012. USENIX.

[3] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, 2012. USENIX Association.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George

Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[6] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. mSwitch: A highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 1:1–1:13, New York, NY, USA, 2015. ACM.

[7] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, USA, 2016. USENIX Association.

[8] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[9] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 27–38, New York, NY, USA, 2014. ACM.

[10] Rohan Gandhi, Y. Charlie Hu, Cheng-kok Koh, Hongqiang Liu, and Ming Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *2015 USENIX Annual Tech-*

nical Conference (USENIX ATC 15), pages 473–485, Santa Clara, CA, USA, July 2015. USENIX Association.

[11] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM.

[12] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, RFC Editor, August 2008.

[13] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[14] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, USA, 2012. USENIX Association.

[15] Jonathan Corbet. TCP connection repair. May 2012.

[16] NGINX Inc. NGINX: High performance load balancer, web server, & reverse proxy. https://www.nginx.com/.

[17] DeNA Co., Ltd. H2O - the optimized HTTP/1, HTTP/2 server. https://github.com/h2o/h2o.

[18] Will Glozer. Modern HTTP benchmarking tool. https://github.com/wg/wrk.

[19] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-latency networking with the OS stack and dedicated

NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 43–56, Denver, CO, USA, 2016. USENIX Association.

[20] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 21:1–21:16, New York, NY, USA, 2016. ACM.

[21] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, MobiCom '00, pages 155–166, New York, NY, USA, 2000. ACM.

[22] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-grained failover using connection migration. In *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 3*, USITS'01, pages 221—232, Berkeley, CA, USA, 2001. USENIX Association.

[23] David A. Maltz and Pravin Bhagwat. TCP splicing for application layer proxy performance. *J. High Speed Netw.*, 8(3):225–240, January 2000.

[24] Chu-Sing Yang and Mon-Yen Luo. Efficient support for content-based routing in web server clusters. In *Proceedings of the 2nd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, USITS'99, pages 221—232, Berkeley, CA, USA, 1999. USENIX Association.

[25] Marcel-Catalin Rosu and Daniela Rosu. Kernel support for faster web proxies. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, pages 225–238, San Antonio, TX, USA, June 2003.

[26] Li Zhao, Yan Luo, Laxmi Bhuyan, and Ravi Iyer. SpliceNP: A TCP splicer using a network processor. In *Proceedings of the 2005 ACM Sym-*

posium on Architecture for Networking and Communications Systems, ANCS '05, pages 135–143, New York, NY, USA, 2005. ACM.

[27] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. On the performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, USITS'99, Berkeley, CA, USA, 1999. USENIX Association.

[28] Oliver Spatscheck, Jørgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Trans. Netw.*, 8(2):146–157, April 2000.