Prism: A Proxy Architecture for Datacenter Networks

Yutaro Hayakawa*
Keio University
river@ht.sfc.keio.ac.jp

Lars Eggert
NetApp
lars@netapp.com

Michio Honda
NEC Laboratories Europe
michio.honda@neclab.eu

Douglas Santry
NetApp
douglas.santry@netapp.com

ABSTRACT

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.

CCS CONCEPTS

• **Networks** → **Network architectures**; *Transport protocols*; *Application layer protocols*; *Middle boxes / network appliances*; *Deep packet inspection*;

KEYWORDS

Datacenter, TCP, proxying, rewriting, load-balancing

ACM Reference Format:
Yutaro Hayakawa, Lars Eggert, Michio Honda, and Douglas Santry. 2017. Prism: A Proxy Architecture for Datacenter Networks. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 8 pages.
https://doi.org/10.1145/3127479.3127480

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 [17, 30], 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. Such proxy-based communication is common and includes distributed storage [30, 43], MapReduce [8] 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 [3] hardware switches, but achieves good performance even when implemented inside a software switch [21].

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 [13], Ananta [32], Duet [16], Rubik [14]—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 [1].

We show that Prism can improve throughputs 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.

The reminder of this paper is organized as follows. Section 2 presents the Prism design, including how Prism breaks out a client TCP connection to multiple backends, and describes the prototype controller, software switch and backend implementation. Section 5

*Most of the research was performed during an internship at NetApp.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3127480

evaluates the performance and overheads of our prototype. Section 7 reviews related work. Section 8 concludes the paper.

2 DESIGN

This section discusses the components involved in the Prism architecture using the packet sequence diagram in Figure 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 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 opening a TCP connection with a server. Step 1 in Figure 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 1 indicate TCP packets sent on the client connection, dashed lines indicate Prism control messages between the controller, switch and backend servers.

2.2 Request Parsing

In step 2 of Figure 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.

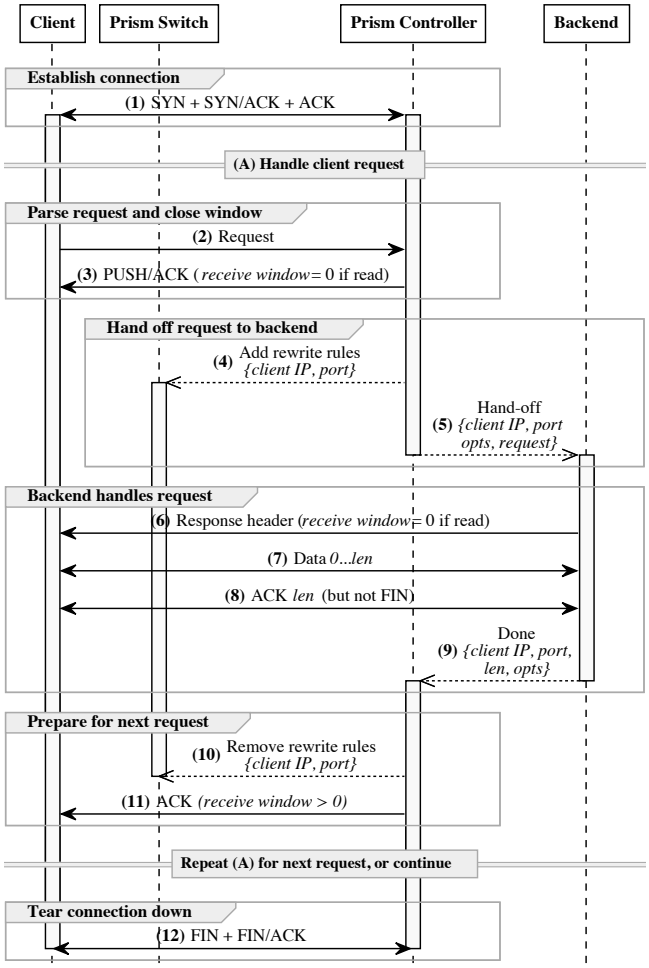


Figure 1: Prism operation.

2.3 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.

2.4 Backend Request Handling

After receiving the hand-off control message from the controller, the backend server handles the client request. Figure 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.

2.5 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.

3 DESIGN DISCUSSION

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

3.1 Supporting TLS

If the application protocol is secured with TLS [10], 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.

3.2 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.

3.3 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.

3.4 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 [3] 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 [26] 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.

3.5 Security Implications

In general, proxies improve attack resilience, acting as firewalls [20, 22] that protect backend servers. The Prism architecture could strengthen this role even more, because it can employ simple and fast user-space TCP/IP stacks at the controller that just handle TCP connection setup, whereas regular proxies usually use the full TCP/IP stack in the OS, which may have a larger attack surface.

Additionally, the Prism architecture could overcome a common problem with user-space TCP stacks, that is a lack of modern features such as current congestion control algorithms [4, 19] and loss recovery mechanisms [5], again because it just handles connection setup. Note that use of SYN cookies [12], a common technique to

mitigate SYN flood attacks, is rather trivial to implement in user-space stacks, because its implementation is even simpler than the normal way to handle SYN packets.

3.6 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.

4 IMPLEMENTATION

Figure 2 illustrates a software architecture of Prism controller, switch and backend. We implemented the Prism controller on Linux, using the netmap [34] fast user-space packet I/O API. It also uses the OS stack for control communication with the backend via the Socket API. The Prism backend uses `TCP_REPAIR` [7] and `TCP_INFO` [33] to implement TCP connection hand-off in userspace, without kernel modifications. For application logic, we implemented a simple HTTP server that serves objects stored in main memory.

The prototype does not use P4, because hardware switches were not yet available, and the performance of the reference software switch implementation [42] is very poor, supporting only 59.8 Mb/s TCP throughput with a mean delay of 204 ms. Instead, we implemented Prism as a packet processing module for mSwitch [21], a kernel software switch that can forward packets at a rate of over 10 Mpps on a single CPU core.

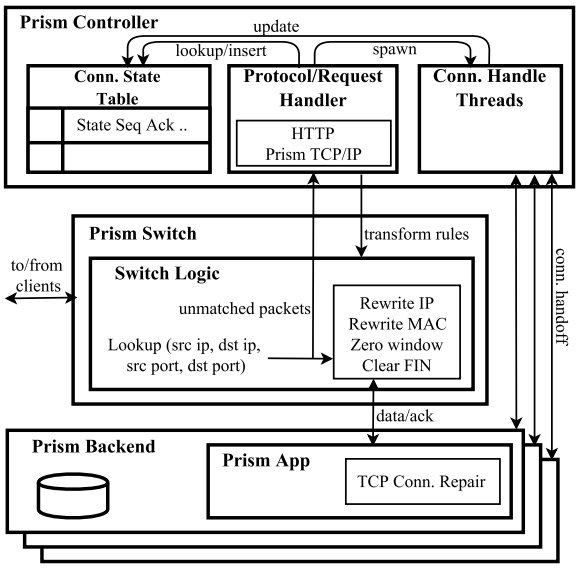
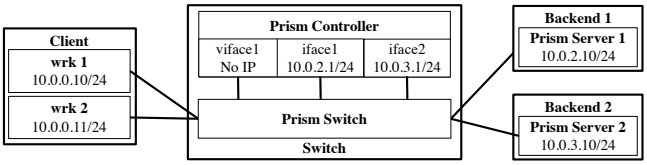
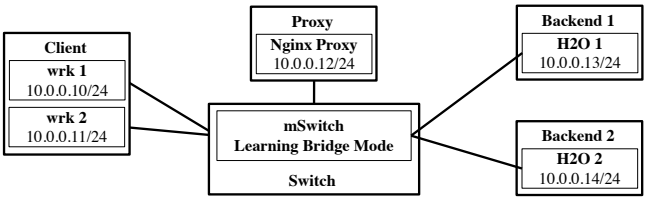


Figure 2: Prism software architecture



(a) Prism topology.



(b) Proxy topology.

Figure 3: Topologies for the evaluation experiments.

Although the prototype deploys this software switch as part of the network fabric in a stand-alone fashion, the Prism design supports other deployments. For example, the Prism packet transformations could also be implemented directly on the backend servers, e.g., in a software switch inside the hypervisor underlying the guest VMs that run the backend servers or even their (guest) operating systems.

The Prism design does not require that all packet transformations occur atomically or at a single location along the path (although the prototype is implemented in this way). Prism allows different transformations to occur at a different points on the path between clients and backend servers. For example, the Prism controller might configure a host firewall or a programmable network interface card on the backend server to zero TCP receive windows and clear FIN bits. The key takeaway is that the general Prism design described in Section 2 can be instantiated in different ways with different trade-offs.

5 EVALUATION

Figure 3 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 3a), the mSwitch [21] 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 3b), we add an additional server that runs the `nginx` proxy [29] and connects to the switch machine via a 10 G Ethernet link; the backend servers run the `H2O` HTTP server [9]. 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.

5.1 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 performance far exceeds the capacity of a 10 G NIC.

Figure 4 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.

5.2 End-to-End Throughput

Figure 5 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 ten packets, and should further increase (esp. for smaller sizes) once that limitation is removed.

Both a traditional proxy and Prism incur some additional management overhead before and after serving a client request (compared to direct backend communication). Table 1 quantifies these overheads. "Startup" overheads incur before the transmission of the first response byte to the client. "Teardown" overheads incur

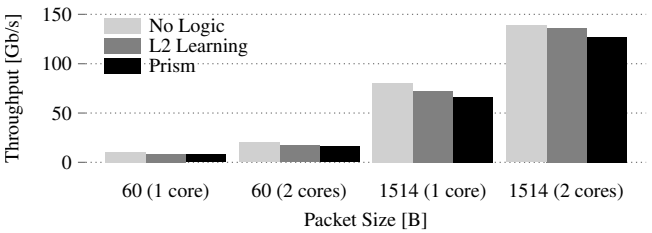


Figure 4: Throughput over Prism packet transformation

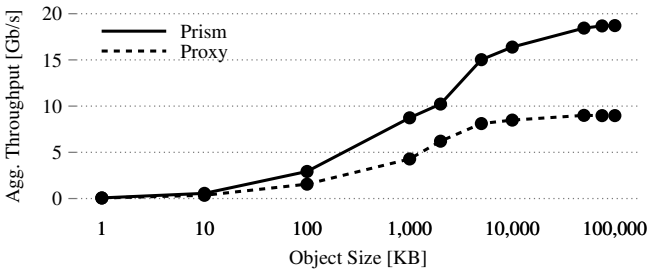


Figure 5: End-to-end throughput.

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 [47].

6 EXAMPLE USE CASES

This section describes some concrete application that would benefit from the deployment of Prism.

The first is backup of home devices to the cloud has become increasingly important as data preservation has become relevant to ordinary families. Data must survive fires and other disasters that occur in homes. Medical records, tax records and family photographs are just a few examples of data that people cannot afford to lose. This need has become a flourishing market, Amazon S3 and Dropbox are examples of vendors in this space.

A common design pattern for backup vendors when building datacenters provides for including an Internet-facing set of machines and placing the storage backends behind them. Clients transfer data by connecting to an Internet facing machine, which acts a proxy for the storage backends. The network traffic is mostly bulk data transfer between clients and storage servers. Thus the Internet facing machines spend a lot of their time simply moving data between sockets; their role is to separate the inside of the datacenter from the outside world. Further, concurrently reading clients lead to bandwidth shortage at the proxy uplink. The protocols used are usually RESTful, reusing TCP connections over multiple HTTP transactions.

Prism would be ideal in this scenario, as it has the potential to dramatically reduce the costs attendant to running the service. The role of the Internet-facing machine is to provide a firewall between the datacenter and the Internet. Prism can provide this by reducing the cost of moving data. Clients would connect to the controller,

	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 1: Latencies of additional request procedures.

which would authenticate the client and verify subscriptions etc., but the bulk data transfer is offloaded. This would result in far fewer machines being required to face the Internet as their primary activity, forwarding bulk data transfer, has been offloaded to where it belongs: the *network*.

A second use case—and one that we are currently implementing—is to use Prism as a high-performance load balancer for the Hyrise-R [36] in-memory database. Hyrise-R can serve read requests from a number of read-only replicas of a central master, which needs to handle all writes. Prism can transparently scale-out any reads over a connection to improve load-balancing, while at the same time directing all writes to the central master. Currently, Hyrise-R either needs to disallow writes per client connection, or use multiple connections for reads and writes.

This integration of Prism into Hyrise-R is transparent to clients, but does require changes to the database servers, which now need to coordinate request hand-off with the Prism controller, as described in Section 2. We hope to demonstrate the performance trade-offs of this approach in a future paper.

7 RELATED WORK

Prism is similar in design to TCP Migrate [39], which provides fail-over functionality for long-running TCP connections to a set of replica servers [38], but does not require client changes and is therefore more deployable.

The way in which Prism splits TCP connections is similar to TCP Splice [25] and related approaches [35, 46] that were developed for the first large-scale web server farms. Some of these approaches have also been implemented in hardware [6, 49]. 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 [40], but is considerably more complex than Prism and requires continuous monitoring of the connection by the forwarder.

TCPR [41] monitors TCP connections at a proxy to increase the fault tolerance of participating applications, which after failure can recover uncommitted state by querying the TCPR proxy using a control channel that is similar in design to the one Prism uses to coordinate request handover between the controller and the backend.

Systems such as Click [27], RouteBricks [11] and xOMB [2] also provide efficient packet and flow transformations, but unlike Prism do not support offloading the handling of payload data to network switches.

The Prism functionality can be seen as a network function, and can hence be implemented in frameworks for network function virtualization (NFV), such as E2 [31]. The Prism approach to handing off connections between different machines, which then transparently serve a client for some time or otherwise operate on data sent to or from the client, could also be used to reduce the overheads of such frameworks when it comes to handling TCP-based protocols.

Maglev [13] and Ananta [32] are load-balancers implemented on commodity hardware. Duet [16] and Rubik [14] are similar, but partially leverage standard hardware switches for improved performance. Unlike Prism, neither of these approaches support request-granularity redirection. Yoda [15], a load-balancer that supports migrating TCP connections between instances, supports request-granularity redirection, but does not address the bandwidth limitation that Prism focuses on. NetKV [48] is a load balancer for memcached, but does not support TCP. Another approach [45] builds a CDN infrastructure that hands over TCP connections to different servers. However, it does not support transaction-granularity redirection.

An OpenFlow-based load-balancing technique that minimizes flow-table occupancy in the switch is described in [44]. This is complementary to Prism; we might extend this approach by pushing more work to backend servers. Problems found in a deployment of transparent TCP proxies (i.e., without application-level logic) in WLAN networks [23] could be helpful to improve Prism's connection handling.

SwitchKV [24] is a distributed key-value store that redirects requests to multiple backends using an OpenFlow switch. To this end, special clients and network protocol based on (only) UDP are needed to embed some hints about requests into a packet header. In contrast, Prism is designed to work with unmodified clients and common network protocols (i.e., TCP) to ease adoption by cloud and service providers.

The explicit communication between the Prism and the backend, which enables transaction-granularity redirections, can also enable support for TLS [10], assuming a hand-off functionality for TLS state (see Section 3). Blindbox [37] can operate on encrypted TLS streams, but is limited to matching operations. Prism requires that the controller is able to parse the cleartext, requiring key sharing or adoption of a scheme like multi-context TLS [28].

8 CONCLUSION AND FUTURE WORK

This paper described Prism, an architecture that addresses the bandwidth utilization problem with proxy-based systems in datacenters. Prism converts much of the traditional proxy-processing functionality into packet-level transformations that can be offloaded to programmable hardware switches or efficiently implemented in 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.

Once P4 hardware switches become available, we plan on comparing the achievable performance and overheads to our software-switch implementation. We are also planning to applying Prism to workloads other than HTTP. One particular area of interest is protocols secured with TLS [10], to investigate if additional benefits can be achieved by offloading TLS processing to the backends.

ACKNOWLEDGMENTS

This paper has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, Chicago, IL, USA, 503–514. https://doi.org/10.1145/2619239.2626316

[2] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. 2012. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'12)*. ACM, Austin, TX, USA, 49–60. https://doi.org/10.1145/2396556.2396566

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *Queue* 14, 5, Article 50 (Oct. 2016), 34 pages. https://doi.org/10.1145/3012426.3022184

[5] Yuchung Cheng and Neal Cardwell. 2016. Making Linux TCP Fast. In *The Technical Conference on Linux Networking (NETDEV 1.2) (Netdev 1.2)*. Tokyo, Japan, 73–80.

[6] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. 1999. 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)*. USENIX Association, Boulder, CO, USA, 1. http://dl.acm.org/citation.cfm?id=1251480.1251491

[7] Jonathan Corbet. 2012. TCP Connection Repair. *LWN.net* Weekly Edition for May 3, 2012 (May 2012). https://lwn.net/Articles/495304/

[8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, San Francisco, CA, USA, 137–149. http://dl.acm.org/citation.cfm?id=1251254.1251264

[9] DeNA Co., Ltd. [n. d.]. H2O - the Optimized HTTP/1, HTTP/2 Server. https://github.com/h2o/h2o. ([n. d.]).

[10] Tim Dierks and Eric Rescorla. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. RFC Editor. https://doi.org/10.17487/rfc5246

[11] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, Big Sky, MT, USA, 15–28. https://doi.org/10.1145/1629575.1629578

[12] Wesley M. Eddy. 2007. *TCP SYN Flooding Attacks and Common Mitigations*. RFC 4987. RFC Editor. https://doi.org/10.17487/rfc4987

[13] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16) (NSDI'16)*. USENIX Association, Santa Clara, CA, USA, 523–535. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[14] Rohan Gandhi, Y. Charlie Hu, Cheng-kok Koh, Hongqiang Liu, and Ming Zhang. 2015. Rubik: Unlocking the Power of Locality and End-point Flexibility in Cloud Scale Load Balancing. In *2015 USENIX Annual Technical Conference (USENIX ATC 15) (ATC'15)*. USENIX Association, Santa Clara, CA, USA, 473–485. https://www.usenix.org/conference/atc15/technical-sessions/presentation/gandhi

[15] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. 2016. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*. ACM, London, UK, Article 21, 16 pages. https://doi.org/10.1145/2901318.2901352

[16] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, Chicago, IL, USA, 27–38. https://doi.org/10.1145/2619239.2626317

[17] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. 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)*. ACM, Heidelberg, Germany, Article 1, 12 pages. https://doi.org/10.1145/2716281.2836086

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

[19] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[20] Mark Handley, Vern Paxson, and Christian Kreibich. 2001. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-end Protocol Semantics. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, 1. http://dl.acm.org/citation.cfm?id=1267612.1267621

[21] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. 2015. mSwitch: A Highly-scalable, Modular Software Switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR'15)*. ACM, Santa Clara, CA, USA, Article 1, 13 pages. https://doi.org/10.1145/2774993.2775065

[22] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is It Still Possible to Extend TCP?. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC'11)*. ACM, Berlin, Germany, 181–194. https://doi.org/10.1145/2068816.2068834

[23] Franck Le, Erich Nahum, Vasilis Pappas, Maroun Touma, and Dinesh Verma. 2015. Experiences Deploying a Transparent Split TCP Middlebox and the Implications for NFV. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox'15)*. ACM, London, United Kingdom, 31–36. https://doi.org/10.1145/2785989.2785991

[24] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16) (NSDI'16)*. USENIX Association, Santa Clara, CA, USA, 31–44. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-xiaozhou

[25] David A. Maltz and Pravin Bhagwat. 2000. TCP Splicing for Application Layer Proxy Performance. *J. High Speed Netw.* 8, 3 (Jan. 2000), 225–240. http://dl.acm.org/citation.cfm?id=339671.339709

[26] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[27] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. 1999. The Click Modular Router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM, Charleston, SC, USA, 217–231. https://doi.org/10.1145/319151.319166

[28] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. ACM, London, United Kingdom, 199–212. https://doi.org/10.1145/2785956.2787482

[29] NGINX Inc. [n. d.]. NGINX: High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/. ([n. d.]).

[30] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat Datacenter Storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12) (OSDI'12)*. USENIX, Hollywood, CA, USA, 1–15. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/nightingale

[31] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, Monterey, CA, USA, 121–136. https://doi.org/10.1145/2815400.2815423

[32] 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. 2013. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*. ACM, Hong Kong, China, 207–218. https://doi.org/10.1145/2486001.2486026

[33] René Pfeiffer. 2012. Measuring TCP Congestion Windows. *Linux Gazette* (March 2012). http://linuxgazette.net/136/pfeiffer.html

[34] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12) (ATC'12)*. USENIX Association, Boston, MA, USA, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[35] Marcel-Catalin Rosu and Daniela Rosu. 2003. Kernel Support for Faster Web Proxies. In *2003 USENIX Annual Technical Conference (USENIX ATC 03) (ATC'03)*. San Antonio, TX, USA, 225–238. http://www.usenix.org/events/usenix03/tech/rosu.html

[36] David Schwalb, Jan Kossmann, Martin Faust, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2015. Hyrise-R: Scale-out and Hot-Standby Through Lazy Master Replication for Enterprise Applications. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics (IMDM'15)*. ACM, Kohala Coast, HI, USA, Article 7, 7 pages. https://doi.org/10.1145/2803140.2803147

[37] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. ACM, London, United Kingdom, 213–226. https://doi.org/10.1145/2785956.2787502

[38] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. 2001. Fine-grained Failover Using Connection Migration. In *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 3 (USITS'01)*. USENIX Association, San Francisco, CA, USA, 221–232. http://dl.acm.org/citation.cfm?id=1251440.1251459

[39] Alex C. Snoeren and Hari Balakrishnan. 2000. An End-to-end Approach to Host Mobility. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom'00)*. ACM, Boston, MA, USA, 155–166. https://doi.org/10.1145/345910.345938

[40] Oliver Spatscheck, Jørgen S. Hansen, John H. Hartman, and Larry L. Peterson. 2000. Optimizing TCP Forwarder Performance. *IEEE/ACM Trans. Netw.* 8, 2 (April 2000), 146–157. https://doi.org/10.1109/90.842138

[41] Robert Surton, Ken Birman, and Robbert van Renesse. 2013. Application-Driven TCP Recovery and Non-Stop BGP. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN'13)*. Budapest, Hungary, 1–12. https://doi.org/10.1109/DSN.2013.6575313

[42] The P4 Language Consortium. [n. d.]. P4 Software Switch. https://github.com/p4lang/behavioral-model. ([n. d.]).

[43] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. 2012. BlueSky: A Cloud-backed File System for the Enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, San Jose, CA, USA, 1. http://dl.acm.org/citation.cfm?id=2208461.2208480

[44] Richard Wang, Dana Butnariu, and Jennifer Rexford. 2011. OpenFlow-based Server Load Balancing Gone Wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE'11)*. USENIX Association, Boston, MA, USA, 6. http://dl.acm.org/citation.cfm?id=1972422.1972438

[45] Matthias Wichtlhuber, Robert Reinecke, and David Hausheer. 2015. An SDN-Based CDN/ISP Collaboration Architecture for Managing High-Volume Flows. *IEEE Transactions on Network and Service Management* 12, 1 (3 2015), 48–60. https://doi.org/10.1109/TNSM.2015.2404792

[46] Chu-Sing Yang and Mon-Yen Luo. 1999. 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)*. USENIX Association, Boulder, CO, USA, 221—232. http://dl.acm.org/citation.cfm?id=1251480.1251500

[47] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16) (ATC'16)*. USENIX Association, Denver, CO, USA, 43–56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

[48] Wei Zhang, Timothy Wood, and Jinho Hwang. 2016. NetKV: Scalable, Self-Managing, Load Balancing as a Network Function. In *2016 IEEE International Conference on Autonomic Computing (ICAC) (ICAC'16)*. Würzburg, Germany, 5–14. https://doi.org/10.1109/ICAC.2016.28

[49] Li Zhao, Yan Luo, Laxmi Bhuyan, and Ravi Iyer. 2005. SpliceNP: A TCP Splicer Using a Network Processor. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems (ANCS'05)*. ACM, Princeton, NJ, USA, 135–143. https://doi.org/10.1145/1095890.1095909