Prism: Proxies without the Pain

Yutaro Hayakawa
LINE Corporation

Michio Honda
University of Edinburgh

Douglas Santry
Unaffiliated

Lars Eggert
NetApp

Abstract

Object storage systems, which store data in a flat name space over multiple storage nodes, are essential components for providing data-intensive services such as video streaming or cloud backup. Their bottleneck is usually either the compute or the network bandwidth of customer-facing frontend machines, despite much more such capacity being available at backend machines and in the network core. Prism addresses this problem by combining the flexibility and security of traditional frontend proxy architectures with the performance and resilience of modern key-value stores that optimize for small I/O patterns and typically use custom, UDP-based protocols inside a datacenter. Prism uses a novel connection hand-off protocol that takes the advantages of a modern Linux kernel feature and programmable switch, and supports both unencrypted TCP and TLS, and a corresponding API for easy integration into applications. Prism can improve throughput by a factor of up to 3.4 with TLS and by up to 3.7 with TCP, when compared to a traditional frontend proxy architecture.

1 Introduction

A scale-out architecture for object storage systems is essential not only for supporting large storage capacities but also to incorporate sufficient compute and network bandwidth so the system can offer a predictable high-throughput and low-latency service to clients. Non-volatile memories (NVM) now fill the performance gap between networking and storage [28] with measured throughputs of 39.4 Gb/s and access latencies of 305 ns [34], emphasizing the importance of minimizing storage stack overheads for use with NVM [33].

A common design pattern seen in object storage systems [2, 3, 22, 35, 41, 65] uses a set of frontend machines to mediate client requests and relay them to a set of backend machines, as illustrated in the left diagram in Figure 1. The frontend often acts as a cache and/or load balancer to sharded or replicated backends. In this architecture, handling small objects incurs severe network inefficiencies, and handling large objects is limited by both network and compute resource availability because of data movement and encryption [2]. However, such systems have been widely adopted because of practical tractability when encryption and filtering are required, and because the performance characteristics of traditional disk-based backends are so poor that the CPUs and network typically are under-utilized, i.e., are not bottlenecks [67].

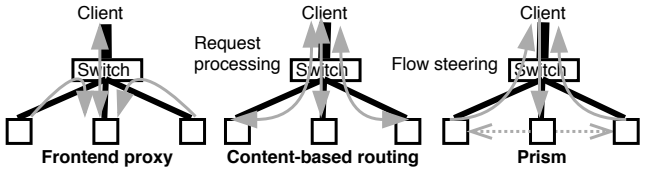

Figure 1: **Scale-out architecture variants.**

Recent approaches such as SwitchKV [45], NetCache [37] and Pegasus [44] use a content-based routing architecture, where servers interact with programmable switches, as illustrated in the central diagram in Figure 1. These approaches significantly improve throughput, latency and resilience to skew that is prevalent in realistic workloads, because the switches can redirect traffic that would otherwise be arriving at congested backend nodes.

While this overall concept is promising, these systems can only handle unencrypted single-packet-sized objects of up to 1.5 to 9 KB (with jumbogram fabrics), because the switch data path must understand the application logic. Also, clients need to use custom UDP-based protocols and implement loss recovery and congestion control functionality by themselves, which is non-trivial especially when serving clients over the Internet. Other content-routing approaches support large objects [13, 40, 71], but do not fully support TCP and hence cannot support industry-standard TLS security or popular application-level protocols (e.g., HTTP and Amazon S3).

This paper presents Prism, a framework that enables the new object storage architecture that combines the flexibility of frontend proxies with the efficiencies and resilience of content-routing approaches, as illustrated in the right diagram in Figure 1. It transparently scales out to many clients and supports arbitrarily-sized objects, TCP and TLS, allowing applications to secure industry-standard protocols such as S3 over HTTPS.

A key concept of Prism is *repeatable* TCP connection hand-off, which allows a TCP connection to be re-homed to different machines over its lifetime. This enables the frontend to examine (even encrypted) requests without requiring it to then also relay object payloads (i.e., bulk data), addressing one of the main drawbacks of traditional proxy architectures (Section 2). At the expense of a fixed per-request cost, repeatable connection hand-off smoothly distributes the I/O, compute and network bandwidth usage across the backends, avoiding bottlenecks in the data path.

The novelty and viability of Prism are based on two other recent innovations. Prism is novel because of its new TCP hand-off protocol that conforms to the TCP state serialization feature available in modern Linux, which is also under development in FreeBSD [66], and overcomes the limitation of this feature (Section 4). Prism is now viable because of the availability of scalable, fine-grained state management techniques for programmable switches, such as SilkLoad [51] and FlowBlaze [8, 58], which enable Prism to control a large number of concurrent flows with fast switch rule management.

Prism has been implemented on Linux hosts and eBPF-based software switches, and for this paper was instantiated as an S3-compatible object store in order to demonstrate much better throughput and latency when compared to a frontend proxy architecture.

This paper makes three main contributions:

- Characterization of network and CPU utilization in the current proxy-based architecture with TLS (Section 2.3).
- A robust TCP hand-off protocol for commodity programmable switches and network stack (Section 4.2).
- Improvement of resource utilization in the replicated or sharded backend architecture (Section 5).

In the remainder of this paper, Section 2 reviews how object storage systems work and characterizes their performance. Section 3 describes our high level approach and design challenges. Section 4 details the design and implementation of Prism, including its connection hand-off protocol and software stack. Section 5 evaluates Prism, and Section 6 discusses implications of our work. Section 7 describes related work. The paper concludes with Section 8.

2 Background and Problem

The primary focus of this paper is commercial object storage systems. This section describes the concepts behind such systems, then characterizes their performance.

2.1 Motivation: Object Storage Systems

Object storage systems serve huge amounts of data, both when instantiated as public cloud services, such as Amazon S3, Azure Storage, Dropbox and others, or as private installations, such as NetApp StorageGRID or Dell/EMC ECS. Cloud storage systems are also being deployed in edge clouds [4, 43], which are smaller but closer to clients, compared to hyperscalar public clouds, but can still generate a terabit of data per second [4]. Scale-out object storage systems are also used in many other scenarios. For example, OpenStack, a popular multi-tenant cloud platform, uses them as a primary data repository [65], and optionally supports bandwidth isolation and fine-grained filtering [22, 23]. IBM has deployed them to build a scale-out Docker registry that maintains Docker images and other data [3].

A common design pattern for such object storage systems uses a set of client-facing *frontend* machines that arbitrate access to a set of *backend* storage machines. The frontend typically does not persistently store any data, but may in some instantiations locally cache some objects. Clients that connect to the storage system, over the wider Internet or from within an enterprise or datacenter network, will be routed to one of the frontend machines via DNS round-robin or L4 load-balancers. The designated frontend machine then acts as a proxy for the set of storage backends. The role of the proxy includes application-level firewalling the internals of the storage system from the outside, possibly TCP and/or TLS termination, client and/or request authentication and authorization, in addition to relaying requests and responses.

Once the TCP connection is established, the network traffic consists mostly of bulk data transfer between client and the backend servers, relayed by a frontend machine. Consequently, a frontend machine spends the majority of their resources relaying traffic between clients and backends. The protocols used by clients are usually RESTful, reusing TCP connections for many individual storage transactions that can be served by the different backends. The frontends are responsible for concurrently handling many clients, which can lead to congestion at the client-facing links [2]. Modern storage devices, such as NVMe SSD and persistent memory, further stress frontend machines, because unlike slow spinning disks [6, 54] these devices do not constrain the networking throughput.

2.2 System Model and Components

In summary, the frontend machine in scale-out object storage systems:

1. terminates client TCP and/or TLS connections
2. receives requests that contain a target object identifier, e.g., a key or URL
3. redirects the request to a suitably chosen backend
4. forwards request data from the client to the backend
5. forwards response data from the backend to the client

Subsequent requests over an already-established connection can be directed to different backends. This re-homing of the connection incurs a cost that is made up of several components, which we will review here to better understand the measured end-to-end performance in Section 2.3.

Data movement: A proxy relays data between two TCP sockets issuing two system calls: `read()` on one socket and `write()` on another, each copying data to and from the kernel. The costs of these system calls increase with the number of bytes read or written. When the data is very small (a few hundreds of bytes or smaller), the fixed *per-call* costs, i.e., the context switch overhead (several tens of ns) dominates the total cost—the cost of moving the data is negligible. For large data sizes, the total cost is dominated by the *per-byte* cost of moving the data. Proposed optimizations include TCP Splice [48, 63] and `tproxy` in recent Linux kernels, which move data between

two TCP sockets within the kernel by swapping socket buffer pointers, without actually copying any data. However, these approaches struggle to support encryption and other more complex application logic.

Data encryption: For confidentiality and authentication, TLS has become the standard in today's Internet and datacenters. Therefore, offloading techniques for TLS have attracted considerable attention. Some TLS libraries can take advantage of hardware acceleration for various ciphers (e.g., AES-NI CPU instructions). In-kernel TLS support was recently added to Linux and FreeBSD to use cryptographic engines available on some NICs. Nevertheless, encrypting traffic comes at significant *per-byte* processing costs, similar to the data movement costs.

Application logic: Proxies typically perform application-level processing when relaying data. For example, they look up a key or URL embedded in the request data to select a suitable backend for the target value or object, or they may scan data to filter out particular requests. Since such information is typically included in the application-level protocol headers, the costs of such application logic usually does not increase with the size of transferred object. (Although it can if this information is contained in the request payload.)

Network stack: Modern kernel TCP/IP stacks can send and receive bulk data at the rate of tens of Gb/s by utilizing NIC offload features (in particular, checksum, TCP segmentation and large receive offloads). TCP and especially TLS connection establishment is an expensive operation due to needing multiple network round-trips, and the required updates to shared resources in the kernel impair multi-core parallelism [57]. However, modern application protocols are usually already designed to maximize connection reuse to amortize these costs. Hence, this paper does not concern itself with application protocols that have high connection-open rates; various improvements that are complementary to our work have already been proposed for high-rate connection openings [46, 57] and small data transfers [24, 36].

Network topology: In rack-scale storage in large datacenters [44, 53] or edge clouds such as used for content delivery networks (CDNs) [4], operators wire servers uniformly so that they can assign the role of servers (e.g., frontend and backend) flexibly depending on the node failure or service demands, as it is very expensive to rewrite cables [73] or the physical space is at a premium in the edge clouds [4]. Each rack consists of one or more top-of-rack (ToR) switches equipped with high bandwidth uplinks. These uplinks are a Clos [16] network fabric in datacenters and Internet exchanges in CDNs, respectively. In such a deployment, individual machines cannot be "scaled up" to create additional proxying capacity.

2.3 End-to-End Performance

To characterize performance of the proxy-based systems, we imitate the aforementioned network topology using a six

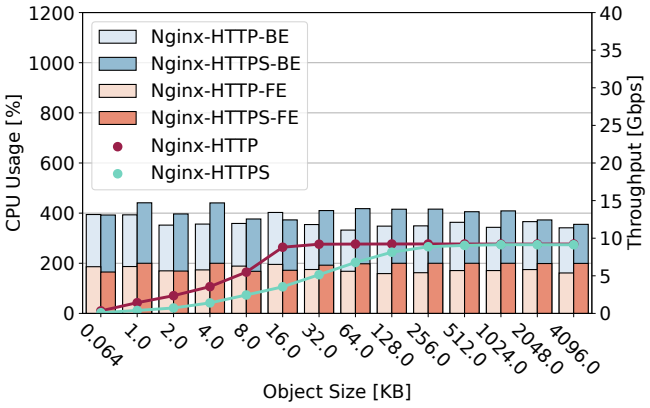

Figure 2: **Throughput and CPU usage of an `nginx` cluster.** One node acts as a proxy and the other five act as backends. Every node connects to a switch with a 10 Gb/s link. The switch connects to a client machine with a 40 Gb/s link (See Figure 8 for illustration). CPU usage can be at most 1200 % due to six dual-core servers.

logical node cluster, each of which connects to a cluster switch over a 10 Gb/s link, and a client machine that is connected to the cluster switch over a 40 Gb/s link that serves as a high bandwidth uplink (see Section 5.1 for hardware details). We install `nginx`, a high-performance, popular web server, to every server cluster node; one acts as a frontend (also generally called reverse proxy) and the other five act as backends. The client node runs `wrk` HTTP benchmark tool to generate requests to retrieve objects from the server cluster.

Figure 2 plots throughput measured by the client and CPU utilization monitored at the servers. The requested object sizes vary, from 64 B to 4 MB. Although the real object storage systems handle a wider range of object sizes (e.g., hundreds of KB to a few MB for photos, and tens or hundreds of MB for videos, deep learning models and VM images), we select the range that characterizes the performance of the system.

With unencrypted HTTP, the frontend ("Nginx-HTTP") can serve up to 9.2 Gb/s, which is close to the 10 Gb/s line rate, taking into account protocol header and framing overheads. At the same time, the frontend CPU resources ("Nginx-HTTP-FE") are also fully utilized. When we allocate one more CPU core to the frontend (not plotted), the network bandwidth becomes the bottleneck (i.e., it results in idle CPU cycles).

When using HTTPS, the frontend ("Nginx-HTTPS") is able to serve only up to 9.1 Gb/s ("Nginx-HTTPS-FE"). Further, it requires larger objects to achieve that throughput than HTTP cases. This performance reduction is due to performing TLS cryptographic operations at the frontend that acts as a proxy, which fully utilizes its CPU resources.

These experiments confirm that either the fabric attachment bandwidth or the CPU resources of the frontend proxy become the bottleneck for this workload, depending on the hardware setup and the use of TLS. Since the Internet-facing capacity of

the switch and backend CPU resources are left underutilized in these experiments, ideally the backends would circumvent the proxy and the switch would forward data directly between clients and backends. The Prism architecture enables this design.

3 Approach and Challenges

These observations confirm the need for reducing CPU usage at the frontend and increasing network utilization at the backends and switch uplink. Is it possible to exclude the frontend from the end-to-end data path for the majority of a transaction, while allowing it to perform its necessary tasks? We will show that it indeed is possible, but that doing so requires a different request-redirection approach than traditionally used. In this section we describe the high-level approach and highlight the main challenges, then describe our resulting design of Prism in Section 4.

3.1 Request-Granularity Redirection

The fundamental problem with a proxy architecture is that all traffic is mediated by the frontend; it relays all traffic between clients and backends. If the ToR switch itself could be instructed to relay data between clients and backends, that forwarding would happen at the faster core network speeds, and—more importantly—eliminate the traditional frontend participation in bulk data relaying. In other words, the frontend could focus on the control-plane aspects of relaying, and the fabric would focus on the data-plane aspects, which optimizes their relative strengths. A frontend confined to the control plane would have a great deal more network and CPU bandwidth available to support more clients per machine, and would thus reduce the overall number of frontend machines needed to support a given client population, making the service more cost effective.

Specialized examples if such a general architecture have been realized in the narrow domain of small-object key-value stores, such as SwitchKV [45], NetCache [37] and Pegasus [44]. However, these systems handle only single-packet-sized transactions and require clients to use custom, unencrypted protocols. It should be noted that for large objects, it is not trivial to implement congestion control and loss recovery that are able to cope with various Internet conditions, such as tail loss [9], incast [1] and phase effects [17].

General, commercially viable object storage systems of course need to support objects larger than a single packet. They also need to be able to secure their client communications with industry standard protocols such as TLS. One example is Amazon S3, the de facto industry standard for object storage, which runs over HTTPS, i.e., TCP and TLS. For such a protocol, *after* the frontend has processed application-level connection setup (e.g., user authentication), it is hence necessary to migrate the entire TCP and TLS connection state to one of the storage backends and have the fabric switches redirect traffic to it—based on *flow-level* information rather than application-level information.

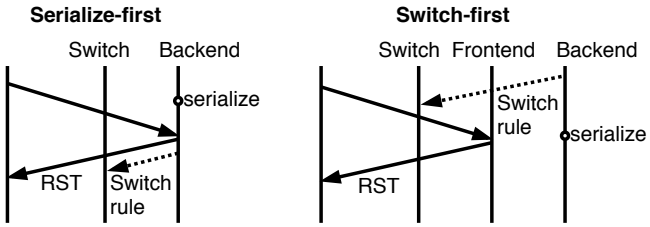

Figure 3: **Breakages with naive TCP hand-off designs.** Leaked packets trigger a connection reset (Section 3.2).

3.2 TCP Hand-Off

Basic TCP hand-off provides a starting point for the request-granularity TCP redirection of Prism. Although TCP hand-off was already explored over twenty years ago [5, 55] based on a custom TCP stack, it has not seen much real-world deployment or open source availability. However, TCP connection serialization, one of the essential features to enable TCP hand-off, was added to the Linux kernel in 2012 [12]. The identical feature is also under development in FreeBSD [66]. Therefore, designing a new TCP hand-off protocol based on this feature could ease the deployment of Prism approach.

When TCP connection state is serialized and then migrated to another machine, it is essential to carefully coordinate the updates to the necessary fabric switch rules, so that no packets "leak" to machines that do not hold the required state—such leakage would generate TCP RST (reset) messages, impacting client operation. Figure 3 depicts two example scenarios where such a connection reset occurs, because the hand-off protocol is incorrectly designed. On the left, an already-migrated connection receives a packet at the original machine, because the switch does not yet to redirect packets to the new target another server. On the right, the switch begins forwarding packets to the new target before the connection has been migrated there. These scenarios can happen with the hand-off protocol designed in the past [5]. The Prism migration protocol avoids any such problems, using the two-phase hand-off protocol described in Section 4.2.

Also, a fabric-based connection hand-off raises concerns about latency and scalability, because it requires manipulating fabric switch rules on a per-request basis (or at least every time the connection migrates). Previous studies imply that this might pose a significant hurdle: In 2014, Liu et al. [47] report that configuring a hardware switch can take hundreds of µs, meaning that for short backend transactions, the hand-off cost could significantly increase the overall request processing cost. In 2011, Yu et al. [69] show maintaining fine-grained flow state in hardware switches to be infeasible because of limited on-chip memory.

However, more recent work has addressed some of these constraints. SilkLoad [51] and FlowBlaze [8, 58] store state for millions of flows in hardware switches; the latter inserts hardware switch rules in a few µs. The Prism hand-off protocol incorporates these more recent observations.

4 Design

Based on the challenges above, the goals of Prism are to (1) design an efficient connection hand-off protocol that works for both TCP and TLS without breaking client sessions, and (2) to build a software stack that implements the protocol and provides a suitably abstract API to applications. We start with describing what an end-to-end data TCP transfer looks like, using the example of a read request that is received by a frontend and served by a backend. We then detail our connection hand-off protocol, which performs a two-phase switch configuration, and our software stack, which ensures correct kernel- and application-level operations.

4.1 Prism in Action

Figure 4 illustrates the Prism hand-off protocol in a packet sequence diagram. Solid arrows indicate packets sent on the TCP connection; dashed lines indicate Prism control messages between the frontend, switch and backends.

Establish connection: As illustrated in Figure 4, a client opens a TCP connection with a Prism frontend server, optionally followed by a TLS handshake.

Parse request: The client begins a transaction by sending a request, which the frontend receives and parses. When the frontend determines that it has received the entire request, it consults the metadata it maintains about the backend servers to select one to handle the request.

Hand-off request to backend: The frontend serializes the TCP connection and TLS session state. TCP state includes ports, sequence and ACK numbers and the TCP options negotiated for both directions of the connection; the TLS state includes the exchanged shared secrets. The frontend then contacts the chosen backend and passes the serialized states, the client IP address, and the client request, so that the backend can take over the connection and serve the request.

The backend 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 frontend. The switch also rewrites the destination or source MAC address, if the client resides in the same broadcast domain. The consequence of this is that any subsequent packets on this connection will be exchanged directly between the client and the backend, with the switch performing the required rewriting. Since the inserted rules only affect a single TCP connection, other connections, either

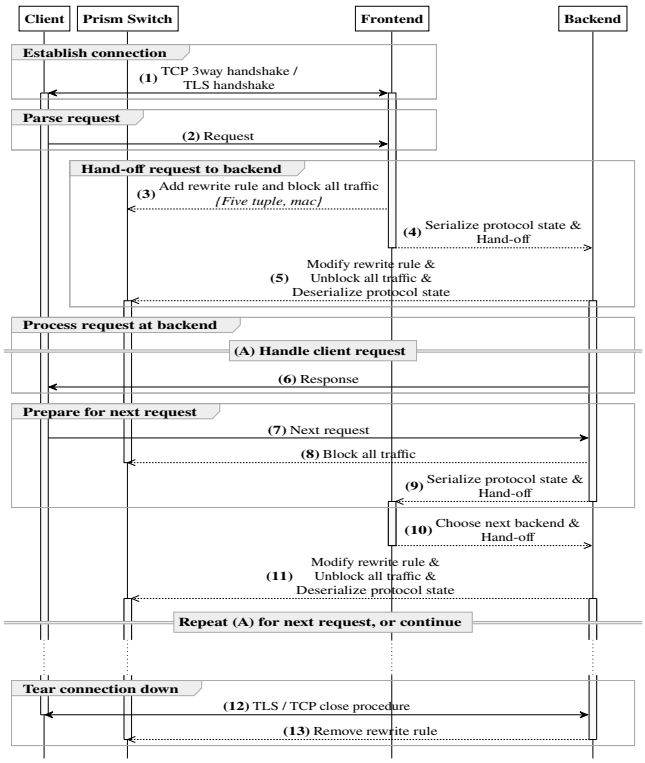

Figure 4: **End-to-end Prism operation.** Solid arrows indicate TCP packets, dashed ones control messages. Step (8)–(11) indicates our two-phase hand-off protocol described in Section 4.2.

to the same frontend or other destinations, remain unaffected. We describe detailed procedures later in this section.

The backend then de-serializes the TCP and TLS state by instantiating a TCP socket based on the information in the serialized connection state and its local IP address (i.e., not frontend's). Because of the active switch rules, the client sees the traffic coming from this backend as if it was coming from the original frontend.

Process request at backend: The backend serves the client, sending back the response over the migrated connection.

Prepare for next request: After a transaction has completed, the backend may return the connection to the frontend and remove the corresponding switch rules, if it wishes subsequent requests on the same connection to be handled by the original frontend. The backend may parse the next request by itself and hand off the request to another backend.

Tear connection down: When the client or the backend itself closes the connection, the backend withdraws the corresponding switch rules.

4.2 Two-Phase Hand-Off Protocol

As shown in Figure 3, a deficient TCP hand-off protocol design breaks client connections. We therefore develop a

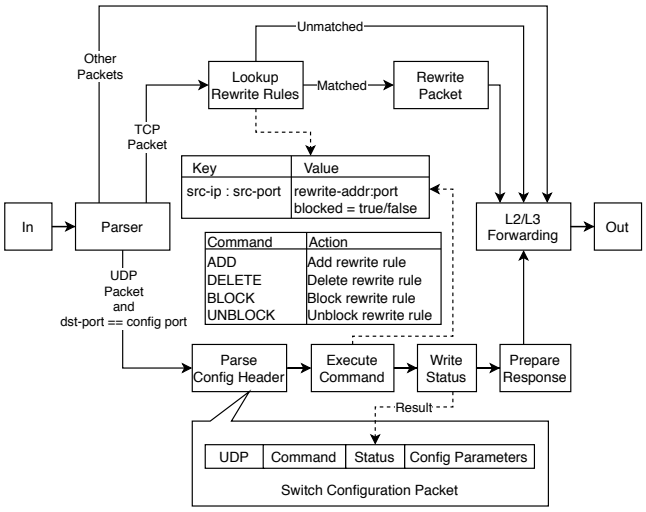

Figure 5: **Prism switch logic.** Custom UDP packets update switch rules.

new connection hand-off protocol that works with the TCP serialization feature available in Linux.

One "hack" would be to drop reset segments sent on a connection under migration with host firewall rules. However, this design requires maintaining flow steering state across the switch fabric *and* the servers, which complicates failure handling. We thus reject this design option.

Our solution instead employs a two-phase switch configuration. First, the host instructs the fabric switch to *drop* all traffic that belongs to the connection being migrated. This prevents this connection from receiving any further packets that might then lead to RSTs. It should be noted that this does not affect performance, because what may be dropped are only unusual packets, such as spurious retransmissions.

Then, a machine serializes a TCP connection and its TLS session and hands this serialized state off to another machine. The target machine then restores the TCP and TLS state. Finally, the target machine sends three commands to the switch atomically. The first one inserts a new rule that rewrites the source IP address of outgoing packets sent from the target machine. The second updates the existing switch rule to redirect any inbound packets to the target machine (instead of the original one). The third removes the drop rule. This two-phase hand-off procedure is depicted over step (8)–(11) in Figure 4.

Prism inserts or withdraws switch rules over a simple, stateless UDP control protocol that triggers in-switch rule manipulation without control-plane involvement. The switch logic that enables the two-phase connection hand-off is illustrated in Figure 5. This protocol implements a simple timeout-based retransmission mechanism, because we assume the communication over the shared links with client data traffic, which can be congested.

4.3 Stack and API

The hand-off protocol described above dictates that many individual commands and application I/O, which runs asynchronously, be executed in coordination with each other. Thus, we need a *software stack* that ensures the correct system state transitions, rather than just API extensions. This stack adds a loadable kernel module that allows applications to detect completed connection removal, an event-based execution engine that drives both the hand-off protocol and application I/O, and high-level APIs for applications to read and write data, and open, close and redirect connections. Figure 6 illustrates the Prism stack; the rest of this section details key components.

TCP state tracking. Before the withdrawal of a switch rule that rewrites the source IP address, the in-kernel connection state must have been freed *completely* to ensure the connection does not transmit any further packets. Unfortunately, this happens silently, long after an application closes a socket. Since the kernel does not notify applications of such events, we implemented a new kernel module to do so, using the Linux `eventfd` framework and socket destructor ("Conn. dtor" in Figure 6). This approach is suitable because the event loop component in the stack, described later, can monitor connection removal events together with any other events, such as new data read from the kernel and requests issued by the application. A similar method is possible also in FreeBSD.

TCP and TLS state serialization. Prism relies on the Linux `TCP_REPAIR` feature [12] to serialize TCP connection state. Based on the option parameter, `getsockopt()` serializes send and receive buffer data, sequence and ACK numbers and negotiated TCP options, which are restored using `setsockopt()` with the same option name. Prism uses the `tlse` library for TLS handshake and serialization, but for the data path, it uses the in-kernel TLS stack of recent Linux kernels, in order to benefit from future hardware offload support. We implemented a new `getsockopt()` option to retrieve the in-kernel TLS state, and upstreamed it to the mainline Linux kernel.

Switch communication. The stack is in charge of communication with the switch using the custom UDP-based packets (Section 4.2). It schedules and sends switch rule update commands to the switch when the application requests connection hand-off or restoration, and waits for the response that includes a status code.

Event loop. Because of the need to perform migration operations for application sockets, Prism discourages applications from using their sockets directly. The stack needs to coordinate and execute these operations for multiple file descriptors in parallel. Therefore, Prism augments `libuv`, a popular event-based I/O library that hides low-level system calls, like `epoll`, `kqueue`, `read` and `write`, from applications. An application associates its own callbacks with events, such as new connection establishment or network or file descriptor readiness for I/O. Prism extends `libuv` to allow applications

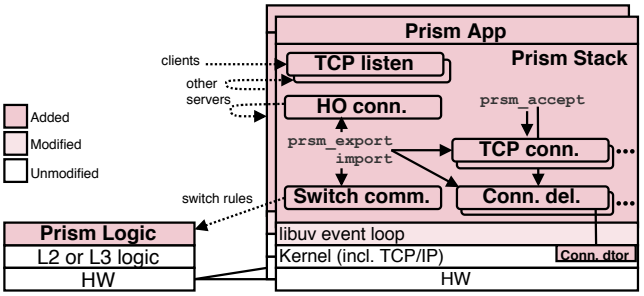

Figure 6: **Prism stack in the host (right) and switch (left)**. Rounded rectangles represent file descriptors, and monospaced font corresponds to the calls used in Figure 7.

to associate their callbacks on top of TLS connections, and to coordinate the application requests and its hand-off protocol, including TCP state tracking and switch communication described above.

API. The programming model of Prism is based on `libuv` with two new methods to export or import migrating connections, and a TLS connection abstraction. Figure 7 implements the frontend and backend roles of the example object store, which statically partitions data across the servers by key, and covers the vast majority of Prism APIs. The `main()` function initiates the event loop with two TCP ports to monitor: one for client requests and the other for hand-offs from other servers. `on_accept()` runs upon establishment of a client TCP connection and initiates the TLS handshake via `uv_tls_init()`. When new client data arrives, it is decrypted and `on_read()` is executed. `who_has_val()` identifies whether the current server hosts the requested content, based on its key. The server either redirects the request to another server using `prsm_export()`, or returns the content using `uv_write()`, which schedules transmission in the event loop.

Although the use of these APIs ensures the correct hand-off state transitions, we do not prevent applications from the use of regular socket APIs. A regular application based on an `epoll` event loop would monitor three additional types of file discriptors: one for the terminated connection notification, another for switch communication and the other for connections to other servers. The application then needs to coordinate the events of these descriptors, for example, to serialize and hand-off a connection only after configuring the switch.

Switch. Many conventional frameworks, including eBPF and P4, are suitable to implement the Prism switching logic described in Figure 5. Although OpenFlow can also be used, this option incurs higher connection hand-off latency due to the access to remote control plane.

The Prism packet transformation logic can be implemented *on top of* an existing L2 switching or L3 forwarding logic without having to disrupt the existing network addressing, as shown in Figure 6. This also allows the Prism switch to be deployed alongside a non-programmable switch as a software switch, as we will show in our experiments.

```
1  on_read(client, buf) { // buf contains a decrypted request
2    const uint64_t key = get_key(buf);
3    who = who_has_val(key);
4    if (who != me)
5      prsm_export(client, who); // prsm_* are Prism methods
6    else {
7      (char *obj, int objlen) = get_objp(key);
8      req = new uv_buf_t(.base = obj, .len = objlen);
9      uv_write(req, client, on_write);
10   }
11 }
13 on_write(req) {
14   free(req->base);
15 }
17 on_accept(server) {
18   client = new uv_tls_t;
19   loop = server->loop;
20   uv_tls_init(loop, client); // uv_* are libuv
       objects/subclasses
21   prsm_accept(server, client);
22   uv_read_start(client, on_read);
23 }
25 main() {
26   uv_tls_t server, internal; // extend uv_tcp_t
27   loop = uv_default_loop();
28   uv_tls_init(loop, &server);
29   uv_tcp_bind(&server, "0.0.0.0:50000");
30   uv_tcp_bind(&internal, "0.0.0.0:60000");
31   uv_listen(&server, on_accept);
32   uv_listen(&internal, prsm_import);
33   uv_run(loop);
34 }
```

Figure 7: **Prism application pseudo code.** Prism extends `libuv`. Server returns requested object (line 9) if present, otherwise hands off request to actual custodian (line 5). The code thus serve the role of both frontend and backend.

4.4 Limitations

Concurrent requests. When a frontend or backend receives parallel requests being handled by different backends in the same TCP connection, it must serialize these requests using one of the following options. The first option is for a backend to simply block any arriving subsequent requests. If these requests need to be processed by different backends, the server hands off the connection after processing its current request. The second option is for the backend to send a TCP "zero window" advertisement to the client. This method turns out to be rather complex, because the backend must do so *before* TCP acknowledges received data, which may already have contained a subsequent request.

To maximize the performance, it is ideally the responsibility of the application-level protocol to prevent the client from issuing another request that might need to be handled by a different backend before an ongoing transaction on the same connection has completed. Traditional proxies can process parallel requests faster than Prism, if the frontend has enough attachment network bandwidth and spare CPU cycles to aggregate the responses from multiple backends; we leave this analysis as future work.

Small transfers. For a small-message transactional workload, i.e., where requests and responses fit into a few TCP packets, Prism may not be a good solution. In such cases, the overheads—switch configuration and connection hand-off—

cannot be sufficiently amortized. We analyze this trade-off in Section 5, which shows that 8 K B (i.e., 6 packets) is sufficient to amortize these overheads.

4.5 Implementation

The Prism server stack consists of 2193 Lines-of-Code (LoC): 612 LoC for TCP and TLS state serialization, 255 LoC for the switch configuration protocol, 167 LoC for active connection tracking, 130 LoC for the loadable kernel module to detect TCP connection removal and 1029 LoC for integration of these components. We modify a single line of `libuv`, and port `tlse` for our TLS abstraction.

We also implement a high performance software switch that makes up the Prism logic in Figure 5. To run the same code in hardware in the future, and to prevent the system from unexpected crash caused by software bug that affects many servers, we implement an eBPF execution environment as a switching logic module of mSwitch [29], a scalable, modular software switch that runs in the kernel. eBPF is popular these days and known that some switch vendors will support hardware offloading of eBPF processing. This software switch never becomes a bottleneck throughout the experiments in the next section.

Our source code is publicly available at https://github.com/YutaroHayakawa/Prism-HTTP.

5 Evaluation

This section evaluates Prism and reports the following main results:

- Prism improves throughput by a factor of up to 3.7 (with HTTP) and 3.4 (with HTTPS), utilizing the switch uplink and backend CPU resources efficiently.
- Prism's throughput increases with the number of backends due to the very light remaining load at the frontend, in terms of both network bandwidth and CPU usage.
- Prism improves object retrieval latency by up to 74 % and 96 % in the 50th and 90th percentile, respectively.
- Prism's connection hand-off latency is 232 µs, which is a win when transferring at least 2 K B with HTTP or 16 K B with HTTPS.
- Prism can be used to build object storage systems with partitioned or replicated backends.

5.1 Experiment Setup

Hardware and OS: Figure 8 depicts the testbed setup used for the experiments. Each machine has a quad-core Xeon E-1231v3 CPU clocked at 3.4 GHz, 16 GB of RAM and a dual-port Intel X540-T2 10 Gb/s NIC, running Linux kernel 4.18. We partition it into two logical servers, dedicating one 10 Gb/s port and two CPU cores to each. The switch has a ten-core Xeon E5-2690v4 CPU clocked at 2.6 GHz, 64 GB

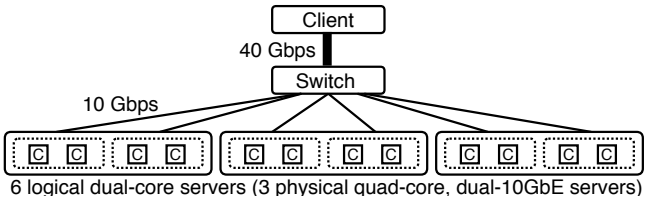

Figure 8: **Experimental topology.**

of RAM, three dual-port Intel 10 Gb/s NICs where each port connects to a logical server, and one Intel XL710 40 Gb/s NIC that connects to the client. We confirmed that this switch never becomes a bottleneck during the experiments. The client has two Xeon E5-2640v4 CPUs, 64 GB of RAM and the same 40 Gb/s NIC as the switch. Our network does not use jumbo frames. The connection hand-off traffic shares the same links with the data traffic.

Software: In the baseline experiment, all of the logical servers run a single `nginx` process. One server acts as a reverse proxy, the others as backends. In the Prism experiment, each logical server runs our custom application implemented on top of the Prism stack described in Section 4.3. Unlike the pseudo code in Figure 7, we use a static frontend setup where the same frontend always establishes incoming TCP connections and examines every request. Therefore, a backend that receives the next request always returns the connection to the frontend.

The communication protocol is always S3, either over HTTP or HTTPS (i.e., with or without TLS). The client continuously generates read or write requests using the `wrk` HTTP benchmark tool [21], instrumented to issue S3 requests (using the Lua scripting support) over 100 parallel persistent TCP connections per backend.

5.2 Link and CPU Utilization

Figure 9 plots the throughput and CPU utilization of Prism and the `nginx` baseline. Both systems redirect requests to the backends in round-robin fashion and each backend serves static, in-memory content.

For HTTP, Prism saturates the 40 Gb/s switch uplink (with the protocol headers and framing overheads) for object transfers of 256 K B and above, whereas `nginx` throughput is constrained by the 10 Gb/s attachment link capacity. Furthermore, Prism reduces the CPU utilization of the frontend by 23 to 53 % in comparison to `nginx`, which utilizes nearly the entire CPU to relay data, and uses the same amount of backend CPU resources as the Prism backends.

For HTTPS, Prism achieves a throughput of 31.4 Gb/s whereas the `nginx` baseline achieves only 9.2 Gb/s. Prism consequently leads to a much higher utilization of the backend CPUs, whereas `nginx` leaves around 75 % of backend CPU resources idle. Since the Prism frontend offloads and load-balances data encryption to the backends and avoids relaying data, its CPU usage reaches at most 70 %, which is spent on

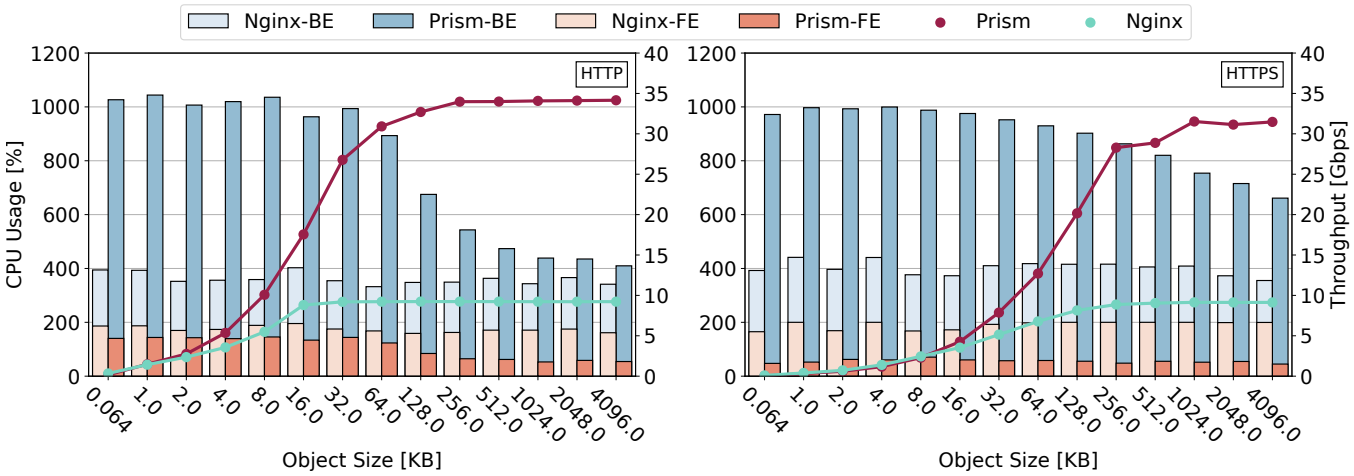

Figure 9: **Throughput and CPU usage with Prism and `nginx`.** FE and BE stand for frontend and backend, respectively. Prism utilizes backend CPU and network resources while keeping the frontend load low.

request redirection. These results imply that Prism would allow operators to provision a fewer number of frontend machines than traditional proxy architectures.

It may seem odd that overheads of the Prism frontend appear low even for 1 K B object sizes, but there is an obvious explanation. With `nginx`, all requests and responses over all 500 parallel connections are relayed by the frontend. Prism's frontend hands off connections to the backends instead of relaying requests, but it does *not* relay responses from the backends. Our results indicate that the advantage of not relying (nor encrypting) the responses at the frontend outweighs the connection hand-off costs.

Figure 10 plots the throughput and CPU utilization of Prism and baseline on a fewer number of backends for a subset of the object sizes, which are 16 , 256 and 512 K B. It confirms that throughputs of Prism have increased almost proportionally to the number of backends until they reach the limit of client processing capacity, which is lower with HTTPS (34 Gb/s and 28 Gb/s with HTTP and HTTPS, respectively) because of decryption and framing overheads.

Overall, Prism improves throughput by a factor of up to 3.7 without TLS, and 3.4 with TLS. It improves CPU utilization by factors of 2.4 and 2.6, respectively.

5.3 End-to-End Latency

Figure 11 plots the 50th and 90th percentile transaction latencies to retrieve 16, 256 or 512 K B objects (same experiments as Figure 10). These latencies decrease when adding more backends, except for the 90th percentile latencies with five backends and 256 or 512 K B objects, where the throughputs reach the maximum and queues start building up (see Figure 10). With `nginx`, those latencies do not change much with the number of backends, and are always higher than Prism, except for those 90th percentile latencies with 256 or 512 K B object sizes and five backends.

5.4 Connection Hand-Off Latency

Operation	Latency [µs]	Std. dev. [µs]
Block all traffic	22	13.1
Serialize TCP	7	1.1
Serialize TLS	5	1.1
Serialize HTTP	2	0.6
Close TCP socket	9	5.0
Hand-off (to frontend)	21	14.4
Hand-off (to backend)	21	14.4
Deserialize HTTP	3	1.5
Deserialize TLS	109	64.3
Deserialize TCP	11	1.0
Modify rewrite rule	22	9.4
Total	**232**	-

Table 1: **Connection hand-off latency breakdown.** The sequence starts from a backend returning the current connection to the frontend.

The largest concern about Prism is the connection hand-off overheads. As described in Section 4.2, the hand-off protocol takes two network round trips to the switch, in addition to the connection state transfer between frontend and backends.

Table 1 reports a breakdown of the latencies of a single connection hand-off cycle in which a backend returns a connection to a frontend, which then hands off the connection to another backend. The total is 232 µs. Each operation takes 21 to 22 µs if the network is involved, otherwise 2 to 11 µs, except for TLS deserialization that takes 109 µs and whose improvement is left as future work. The hand-off latency could be improved if the backend examined the next request after serving the current one, as in described in Figure 7, bypassing the frontend and saving 21 µs. Nevertheless, since Prism out-

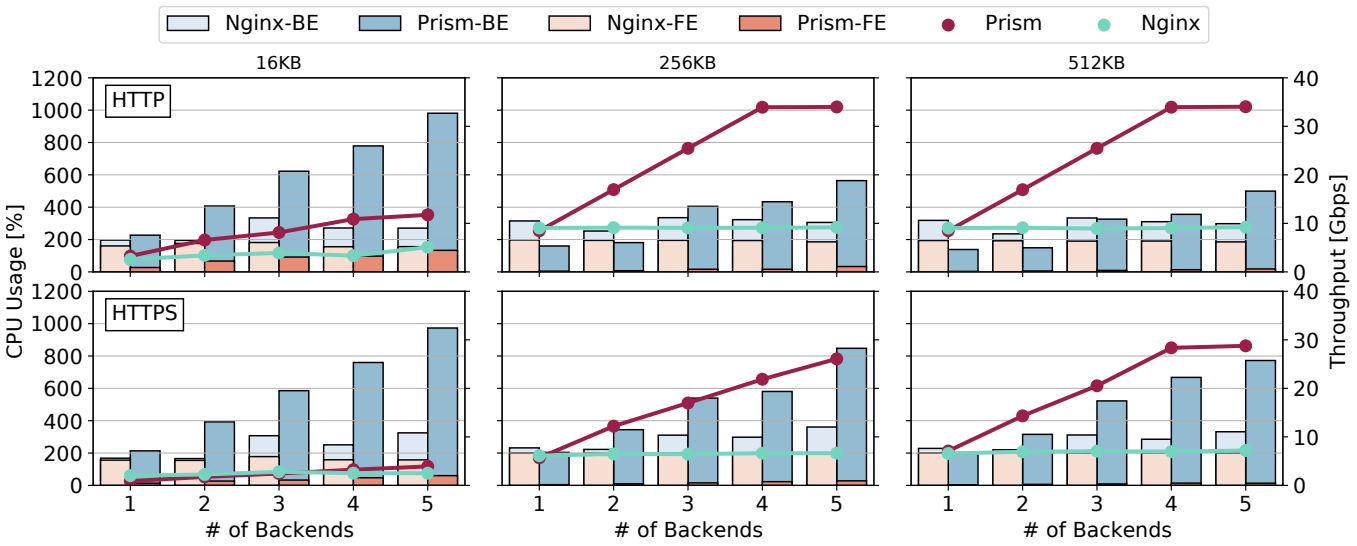

Figure 10: **Prism throughput with different numbers of backends.** The top and bottom row indicates HTTP and HTTPS, respectively. Prism linearly increases the throughput until the limit of client processing capacity.

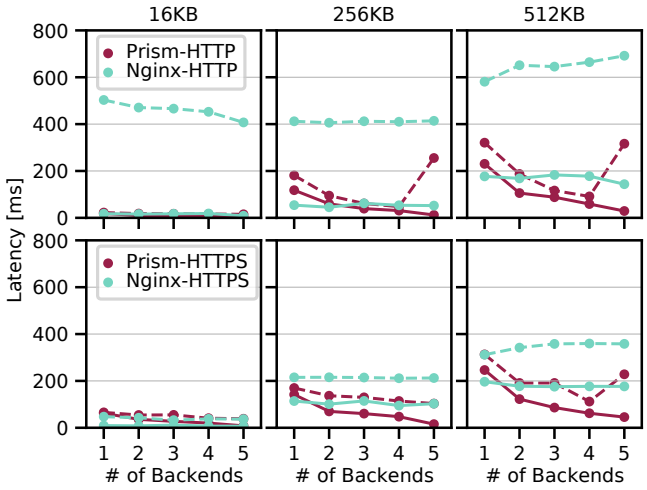

Figure 11: **Prism latency.** Solid and dashed lines indicate 50th and 90th percentile latencies, respectively.

performs the throughput of the traditional proxy architecture with 2 K B (HTTP) or 16 K B (HTTPS) of objects (Figure 9), we conclude that Prism outweighs the costs of the hand-off latency in various workloads.

5.5 Use Case

We implement two variants of an object storage system. The first *partitions* content across the backends, and the second *replicates* content at *all* backends. Thus, the partition variant writes a write request to one of the backends based on the key, but the replication variant does it to all the backends, before responding to the client. In contrast to the previous microbenchmarks in which a backend always serves the same in-memory content, these object storage variants use LevelDB where keys identify objects that are organized into a log structured merge tree. Thus, they include realistic storage stack overheads. We use a RAM disk for the storage medium, assuming faster-than-network NVM-style storage medium. The client protocol is again S3.

5.5.1 Partitioned Object Storage Backends

Figure 12 shows throughputs over two YCSB workloads, read-only and read-mostly that contains 5 % writes. We vary object sizes between 16 to 512 K B and request key skewness, which is 0.9 to 1.2 of Zipfian parameters, uniform distribution (least skewed) and requests that always ask for the same key (most "skewed").

We observe throughputs stay almost the same up to Zipfian 1.0 of skewness, and a slight, up to by 17 %, drop at Zipfian 1.2 that can be considered extremely skewed. We observe that even for an extremely skewed workload (Zipfian 1.2), throughput decays only by up to 17 % in comparison to the uniform distribution, demonstrating Prism's robustness against skewed workloads.

5.5.2 Replicated Object Storage Backends

Figure 13 shows throughputs with the same workloads but for the replicated backends, assuming fault tolerance and load balancing. We observe the same throughput regardless of skewness, as the frontend redirects requests in a round-robin fashion. Since every write is replicated to all the backends, throughput decreases with higher write rates, more so in comparison to the partitioned-backend cases. The lower overall rates compared to those cases can be attributed to larger active data that stress OS buffer caches.

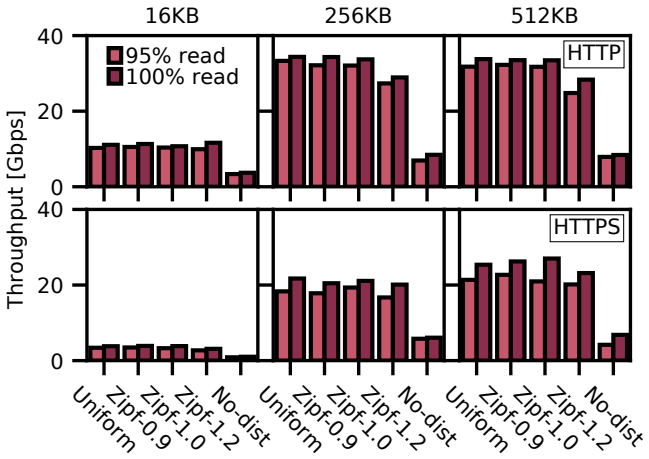

Figure 12: **Prism throughputs with partitioned backends.** Top and bottom rows plot results with HTTP and HTTPS, respectively. Prism preserves high throughputs even under skewed workloads.

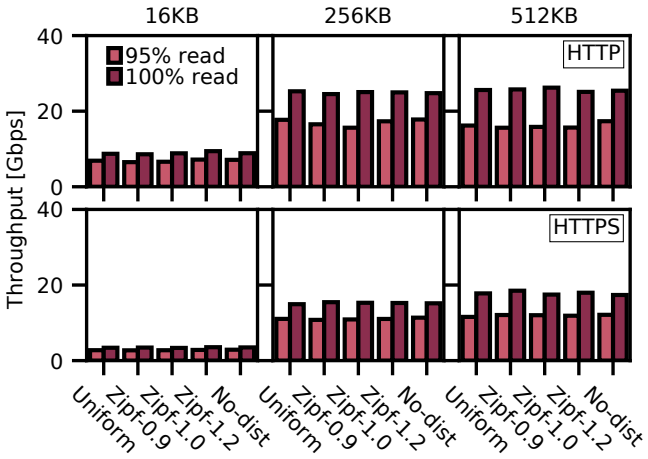

Figure 13: **Prism throughputs with replicated backends.** Throughputs are unaffected by key skewness due to evenly distributed requests.

Prism's ability to handle a partitioned key space and to balance loads across replicas indicates its feasibility for implementing sophisticated data layouts and replication algorithms, respectively.

6 Lessons Learned

Although TCP hand-off had been proposed multiple times at least since 1998, it has never been widely used. We initially attributed this to the lack of scalable flow-level programmable switches, but increasingly realized many other reasons. The mechanism to snapshot or instantiate TCP connections in any state, an essential feature, has been enabled in Linux in a rather inconvenient form, that is, packet transformations must be performed elsewhere, such as with a host firewall or at a switch. Moreover, the TCP stack needs to emit RST packets during the hand-off process to conform to the invariants that apply throughout the TCP implementation.

Alternative approaches that do not require programmable switches have been proposed by Snoeren et al. in 2000 [61, 62]. However, deploying new TCP options has been increasingly difficult over the last decade due to slow network stack evolution [30] and middlebox interferences [32].

These constraints resulted in difficulties for TCP hand-off overall, because it requires that many operations be performed atomically. This requirement prevents the use of a host firewall for source address rewriting, because otherwise manage flow state needs to be managed at both hosts and switches, requiring a complex coordination mechanism.

Building the end-system stack was also a great burden. However, since the TCP connection serialization has become available in the mainline kernel, systems like Prism can now be realized without modifications to the kernel; in fact, we still needed kernel modification, but the Linux community accepted the necessary changes, which are to implement a new API to access the in-kernel TLS state [26], to be included in the mainline kernel. We were able to implement the other kernel extension, which is the connection removal notification (Section 4.3), as a loadable kernel module. Moreover, fewer applications today call "low-level" socket APIs directly; many use higher-level networking libraries such as `libuv`, which we extend to enable the Prism stack (Section 4.3). These phenomena support the deployability of Prism.

Another problem is frequent switch rule updates. This has been problematic for older switches, because the rules must be updated via the switch control plane that runs slow CPUs. In fact, our initial prototype communicated with an RPC server running in the control plane, and it consequently suffered from high latencies of up to 828 µs, which requires 256 K B of transmitted data to amortize these hand-off costs. This led us to the use of a custom switch manipulation mechanism that updates the rules directly within the data plane. This becomes possible with newer hardware and software switches, such as P4 and eBPF, that have advanced programmability. Further, recent improvements of flow scalability in hardware switches [8, 51, 58] also supports the feasibility of this approach.

Further, Prism does not complicate layer 4 firewalls in the network or host, because Prism hosts never rely on host firewalls, nor "spoof" packet addresses, as the packet transformation happens in the switch. This means that host firewall rules can be configured based on not the address of the frontend but that of the individual host. Local firewall policies apply to the restored TCP connections, because Linux `netfilter` creates connection states when it sees any egress packets [11].

Last but not least, we believe our approach is feasible even in sharing switches, which has been a major concern in the use of programmable switches in many existing systems [50].

This is because we turn the feature of the content-based routing approaches into flow-level operations that can be easily isolated between tenants and machines; for example, the operator can limit address-modification rules inserted to the switch within the address range allocated to the tenant.

7 Related Work

Our previous short paper [27] introduced Prism's overall approach with a minimalistic proof-of-concept implementation. This paper significantly extends Prism by incorporating the ability to handle TLS-encrypted communication, a robust hand-off protocol, a much-improved software stack, a more mature implementation and an extensive evaluation.

TCP connection migration. The closest related work is a proposal by Aron et al. in 2000 [5] that proposes content-aware request distribution to the backend cluster, similar to Prism. However, their hand-off protocol can break client connections (Section 3.1) and does not support TLS (or SSL). TCP Migrate Options [61, 62] achieve TCP connection migration using TCP options instead of programmable switches, but have deployment problems (Section 6).

Proxy enhancements and L7 load balancers. TCP Splice (Section 2.2) has been improved by software [10, 59] or hardware-assisted [52, 72] approaches. Yoda [19] improves the fault tolerance of the proxy architecture. Squid [64], HAProxy [25] and Proxygen [60] are open source proxy implementations. Unlike Prism, these approaches do not eliminate the need of a frontend proxy to remain involved in bulk data relaying.

Content based routing. SwitchKV [45], Pegasus [44] and NetCache [37] as discussed in Section 3.1 eliminate frontends that mediate traffic between the client and the backend, by having programmable switches play the role of the frontend. They only support data that fits into a single packet over unencrypted custom UDP-based protocol. NICE [40] supports large data objects, but it relies on unencrypted UDP-based requests and TCP connections initiated by the server for reply. Therefore, it supports neither TLS nor industry-standard protocols such as S3. Strata [13] is a scale-out storage system with NFSv3. It breaks client connections and relies on reconnection to resume the NFS session after change of the storage backend. NetKV [70] is an application-level load balancer for memcached, but does not support TCP.

L4 load balancers. Maglev [14] and Ananta [56] are software load-balancers implemented in commodity servers. Duet [20], Rubik [18] and Faild [4] are similar, but partially leverage standard hardware switches for improved performance. Unlike Prism, none of these approaches support request-granularity redirection. L4 load balancers are used to distribute traffic between multiple frontends of the Prism or proxy architecture.

Flexible packet processing. The Prism frontend might resemble a packet forwarding system implemented as middleboxes, but it differs in that it does not forward packets but hands off established TCP connections. Our software implementation on a programmable soft-switch uses mSwitch [29] for performance and flexibility, and eBPF for protection, but it can be other flexible packet processing frameworks. Recent scalable hardware packet processing systems, such as SilkRoad [51] and FlowBlaze [8, 58], enable Prism to scale to a large number of flows and reduce the latency of switch rule updates. Some vendors are implementing eBPF hardware offloading [39], which accelerates our switch logic (Figure 5).

Caching, sharding and replication algorithms. Many object placement algorithms for storage systems have been proposed [7, 15, 40, 44]. Prism is a framework to implement object storage systems with these algorithms without worrying about communication with clients. In Section 5, we experimentally demonstrated that Prism can be used to implement replicated backends for load balancing and sharded ones for capacity scaling. More sophisticated algorithms, such as selective replication [44], will further improve the storage utilization and performance.

High performance host storage stack with TCP/IP. This class of work, such as Diskmap [49], ReFlex [42] and i10 [33] for NVMe, and Decibel [53] and PASTE [31] for persistent memory, could enhance Prism's backends, and benefit from Prism, because they encrypt or push data at higher rates than traditional host storage stack.

8 Conclusion

As faster storage devices push data to CPUs and networks at higher rates, it is important to scale-out these resources. We built Prism, which combines the performance and resilience of content-based routing approaches with the generality and flexibility of a conventional proxy architecture. We demonstrated that Prism can be used to build object storage systems for the industry-standard S3 protocol over TCP and TLS, and to implement partitioned and replicated backends for capacity scaling and load balancing. Prism is based on a connection hand-off technique that has been proposed in the past, but we redesigned it to address practical problems with the previous systems, taking into account modern technology and requirements across the network switches and today's OS kernels.

Future work will develop object storage systems with advanced data layout or load balancing algorithms, and further reduce the overheads of the TCP/TLS connection migration. Modern low-latency networking techniques based on efficient stacks [68] or RPC designs [38] are a perfect fit in this space.

Acknowledgments

We are grateful to anonymous NSDI reviewers and Z. Morley Mao, our shepherd, for their insightful comments.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data Center TCP (DCTCP)". *Proc. ACM SIGCOMM*. 2010.

[2] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. "Mos: Workload-aware elasticity for cloud object stores". *Proc. ACM HPDC*. 2016.

[3] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. "Improving Docker Registry Design Based on Production Workload Analysis". *Proc. USENIX FAST*. 2018.

[4] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa. "Balancing on the Edge: Transport Affinity without Network State". *Proc. USENIX NSDI*. 2018.

[5] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. "Scalable content-aware request distribution in cluster-based network servers". *Proc. USENIX ATC*. 2000.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. "Finding a Needle in Haystack: Facebook's Photo Storage." *Proc. USENIX OSDI*. 2010.

[7] N. Beckmann, H. Chen, and A. Cidon. "LHD: Improving Cache Hit Rate by Maximizing Hit Density". *Proc. USENIX NSDI*. Apr. 2018.

[8] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone. "Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing". *arXiv preprint arXiv:1605.01977*, 2016.

[9] P. Cheng, F. Ren, R. Shu, and C. Lin. "Catch the whole lot in an action: Rapid precise packet loss notification in data center". *Proc. USENIX NSDI*. 2014.

[10] A. Cohen, S. Rangarajan, and H. Slye. "On the Performance of TCP Splicing for URL-aware Redirection". *Proc. USENIX USITS*. Oct. 1999.

[11] *Conntrack tales - one thousand and one flows*. https://blog.cloudflare.com/conntrack-tales-one-thousand-and-one-flows/.

[12] J. Corbet. "TCP Connection Repair". *LWN.net*, May 2012.

[13] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. "Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory". *Proc. USENIX FAST*. 2014.

[14] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer". *Proc. USENIX NSDI*. Mar. 2016.

[15] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. "Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services". *Proc. ACM SoCC*. 2011.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat. "A Scalable, Commodity Data Center Network Architecture". *Proc. ACM SIGCOMM*. 2008.

[17] S. Floyd and V. Jacobson. "Traffic phase effects in packet-switched gateways". *SIGCOMM CCR*, 1991.

[18] R. Gandhi, Y. C. Hu, C.-k. Koh, H. Liu, and M. Zhang. "Rubik: Unlocking the Power of Locality and End-point Flexibility in Cloud Scale Load Balancing". *Proc. USENIX ATC*. Jul. 2015.

[19] R. Gandhi, Y. C. Hu, and M. Zhang. "Yoda: A Highly Available Layer-7 Load Balancer". *Proc. ACM EuroSys*. Apr. 2016.

[20] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. "Duet: Cloud Scale Load Balancing with Hardware and Software". *Proc. ACM SIGCOMM*. Aug. 2014.

[21] W. Glozer. *Modern HTTP benchmarking tool*. https://github.com/wg/wrk.

[22] R. Gracia-Tinedo, J. é, E. Zamora, M. Sánchez-Artigas, P. Garcıa-López, Y. Moatti, and E. Rom. "Crystal: Software-Defined Storage for Multi-Tenant Object Stores". *Proc. USENIX FAST*. 2017.

[23] R. Gracia-Tinedo, J. é Sampé, and G. Parıs. *Crystal: Software-Defined Storage for OpenStack Swift*. GitHub. 2017.

[24] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O." *Proc. USENIX OSDI*. 2012.

[25] HAProxy. *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer*. http://www.haproxy.org/.

[26] Y. Hayakawa. *Re: [PATCH net-next] net/tls: Implement getsockopt SOL_TLS TLS_RX*. https://www.mail-archive.com/netdev@vger.kernel.org/msg350523.html.

[27] Y. Hayakawa, L. Eggert, M. Honda, and D. Santry. "Prism: A Proxy Architecture for Datacenter Networks". *Proc. ACM SoCC*. 2017.

[28] T. Hoefler, R. B. Ross, and T. Roscoe. "Distributing the data plane for remote storage access". *Proc. ACM HotOS*. 2015.

[29] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. “mSwitch: A Highly-Scalable, Modular Software Switch”. *Proc. ACM SOSR*. Jun. 2015.

[30] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. “Rekindling Network Protocol Innovation with User-level Stacks”. *ACM SIGCOMM CCR*, Apr. 2014.

[31] M. Honda, G. Lettieri, L. Eggert, and D. Santry. “PASTE: A Network Programming Interface for Non-Volatile Main Memory”. *Proc. USENIX NSDI*. 2018.

[32] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. “Is It Still Possible to Extend TCP?”: *Proc. ACM IMC*. Nov. 2011.

[33] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. “TCP≈RDMA: CPU-efficient Remote Storage Access with i10”. *Proc. USENIX NSDI*. 2020.

[34] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module”. *CoRR*, 2019. arXiv: 1903.05714.

[35] Y. Japan. *Dragon: A Distributed Object Storage at Yahoo! JAPAN*. WebDB Forum 2017. 2017.

[36] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. “mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems”. *Proc. USENIX NSDI*. Apr. 2014.

[37] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”. *Proc. ACM SOSP*. 2017.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. “Datacenter RPCs can be General and Fast”. *Proc. USENIX NSDI*. 2019.

[39] J. Kicinski and N. Viljoen. “Making Linux TCP Fast”. *The Technical Conference on Linux Networking (NETDEV 1.2)*. Nov. 2016.

[40] S. Al-Kiswany, S. Yang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “Nice: Network-integrated cluster-efficient storage”. *Proc. ACM HPDC*. 2017.

[41] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. “Flash Storage Disaggregation”. *Proc. ACM EuroSys*. 2016.

[42] A. Klimovic, H. Litz, and C. Kozyrakis. “ReFlex: Remote Flash≈Local Flash”. *Proc. ACM ASPLOS*. 2017.

[43] M. Körner, T. M. Runge, A. Panda, S. Ratnasamy, and S. Shenker. “Open carrier interface: An open source edge computing framework”. *Proc. ACM NEAT*. 2018.

[44] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports. “Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories”. *Proc. USENIX OSDI*. Nov. 2020.

[45] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. “Be Fast, Cheap and in Control with SwitchKV”. *Proc. USENIX NSDI*. Mar. 2016.

[46] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. “Scalable Kernel TCP Design and Implementation for Short-Lived Connections”. *Proc. ACM ASPLOS*. 2016.

[47] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. “zUpdate: Updating data center networks with zero loss”. *ACM SIGCOMM*. 2013.

[48] D. A. Maltz and P. Bhagwat. “TCP Splicing for Application Layer Proxy Performance”. *J. High Speed Netw.* Jan. 2000.

[49] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart. “Disk| Crypt| Net: rethinking the stack for high-performance video streaming”. *Proc. ACM SIGCOMM*. 2017.

[50] J. McCauley, A. Panda, A. Krishnamurthy, and S. Shenker. “Thoughts on load distribution and the role of programmable switches”. *SIGCOMM CCR*, 2019.

[51] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. “SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs”. *Proc. ACM SIGCOMM*. 2017.

[52] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. “AccelTCP: Accelerating Network Applications with Stateful TCP Offloading”. *Proc. USENIX NSDI*. Feb. 2020.

[53] M. Nanavati, J. Wires, and A. Warfield. “Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage”. *Proc. USENIX NSDI*. 2017.

[54] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. “Flat Datacenter Storage”. *Proc. USENIX OSDI*. Oct. 2012.

[55] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. “Locality-aware Request Distribution in Cluster-based Network Servers”. *Proc. ACM ASPLOS*. 1998.

[56] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. “Ananta: Cloud Scale Load Balancing”. *Proc. ACM SIGCOMM*. Aug. 2013.

[57] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. “Improving network connection locality on multicore systems”. *Proc. ACM EuroSys*. 2012.

[58] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Bianchi. "FlowBlaze: Stateful Packet Processing in Hardware". *Proc. USENIX NSDI*. 2019.

[59] M. Rosu and D. Rosu. "Kernel Support for Faster Web Proxies". *Proc. USENIX ATC*. Jun. 2003.

[60] B. Schlinker, I. Cunha, Y.-C. Chiu, S. Sundaresan, and E. Katz-Bassett. "Internet Performance from Facebook's Edge". *Proc. ACM IMC*. 2019.

[61] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. "Fine-grained Failover Using Connection Migration". *Proc. USENIX USITS*. Mar. 2001.

[62] A. C. Snoeren and H. Balakrishnan. "An End-to-end Approach to Host Mobility". *Proc. ACM MobiCom*. Aug. 2000.

[63] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. "Optimizing TCP Forwarder Performance". *IEEE/ACM Trans. Netw.* Apr. 2000.

[64] Squid. *Squid: Optimising Web Delivery*. http://www.squid-cache.org/.

[65] O. Stack. *Swift Object Store*. Open Stack Software. 2016.

[66] Y. Takagawa and K. Matsubara. "Yet Another Container Migration on FreeBSD". *AsiaBSDCon*. 2019.

[67] P. Vajgel. *Needle in a haystack: efficient storage of billions of photos*. 2009.

[68] K. Yasukata, M. Honda, D. Santry, and L. Eggert. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". *Proc. USENIX ATC*. Jun. 2016.

[69] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. "Scalable Flow-Based Networking with DIFANE". *Proc. ACM SIGCOMM*. 2010.

[70] W. Zhang, T. Wood, and J. Hwang. "NetKV: Scalable, Self-Managing, Load Balancing as a Network Function". *Proc. IEEE ICAC*. Jul. 2016.

[71] W. Zhang, T. Wood, K. Ramakrishnan, and J. Hwang. "Smartswitch: Blurring the line between network infrastructure & cloud applications". *Proc. USENIX HotCloud*. 2014.

[72] L. Zhao, Y. Luo, L. Bhuyan, and R. Iyer. "SpliceNP: A TCP Splicer Using a Network Processor". *Proc. ACM/IEEE ANCS*. Oct. 2005.

[73] S. Zhao, R. Wang, J. Zhou, J. Ong, J. C. Mogul, and A. Vahdat. "Minimal rewiring: Efficient live expansion for clos data center networks". *Proc. USENIX NSDI*. 2019.