Capybara: Dynamic Load Balancing with Microsecond-Scale TCP Migration

Inho Choi
National University of Singapore
inhochoi@comp.nus.edu.sg

Nimish Wadekar
ETH Zürich
nwadekar@student.ethz.ch

Guangda Sun
National University of Singapore
sung@comp.nus.edu.sg

Raj Joshi
Red Hat / Harvard University
rajjoshi@g.harvard.edu

Joshua Fried
University of Pennsylvania
josh.fried@cis.upenn.edu

Omar S. Navarro Leija
University of Pennsylvania
leija.this@gmail.com

Dan R. K. Ports
Microsoft Research
dan@drkp.net

Irene Zhang
Microsoft Research
irene.zhang@microsoft.com

Jialin Li
National University of Singapore
lijl@comp.nus.edu.sg

Abstract

Layer-4 load balancers are a popular solution to high tail latencies but perform poorly under unpredictable skewed workloads because they statically assign connections to servers. We present Capybara, a new load balancer architecture that enables dynamic rebalancing of established connections. Capybara divides load balancing responsibility into a fast L4 load balancer, a host-switch co-designed connection migration protocol, and a transport interface for application-level connection state migration. Capybara leverages two trends – programmable switches and kernel-bypass – to efficiently implement connection migration without disruption, while maintaining transparency to clients. Under realistic workloads, Capybara achieves up to 149× lower tail latency and more than 2× higher throughput for scale-out services compared to state-of-the-art load balancing approaches.

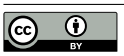
CCS Concepts

• **Networks** → **Network design principles**; **Transport protocols**; *Data center networks*; *Bridges and switches*; Network performance analysis.

Keywords

Datacenter Systems, Load Balancing, TCP Migration, Programmable Switches, Kernel-Bypass

ACM Reference Format:
Inho Choi, Nimish Wadekar, Guangda Sun, Raj Joshi, Joshua Fried, Omar S. Navarro Leija, Dan R. K. Ports, Irene Zhang, and Jialin Li. 2026. Capybara: Dynamic Load Balancing with Microsecond-Scale TCP Migration. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3789240.3829197



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/2026/08
https://doi.org/10.1145/3789240.3829197

1 Introduction

Over the last decade, response times in cloud applications have dramatically improved. Modern datacenter networks can deliver a packet with single-digit microsecond latencies, and recent kernel-bypassing network stacks [7, 27, 31, 41, 53] provide sub-microsecond TCP processing. Yet predictable, μs-scale tail latency [12, 34, 48, 55] remains elusive due to unpredictable workload skews, traffic bursts, and dynamic resource contention.

The standard approach to building scalable services, as shown in Figure 1, is to use a layer 4 load balancer to distribute incoming connections among a pool of frontend servers. These application servers rely on shared backend services that maintain all persistent state, allowing any application server to handle requests from any client. Layer-4 (L4) load balancers, provided by major cloud providers [2, 3, 19], distribute incoming network traffic – usually TCP – across frontend servers. This architecture ensures prompt responses while no single server is a failure point or bottleneck.

Current L4 load balancers can pose a barrier to predictable performance because they cannot rebalance skewed workloads. Once the load balancer assigns a client connection to a server, it must enforce *per-connection consistency* (PCC), which requires all packets in the connection to arrive at the assigned server. PCC is necessary for TCP because it is a stateful protocol that requires servers to update control state on every packet. Even when any frontend server can handle any client's request – as it must, since if the TCP connection is lost, the load balancer will likely reassign the client's next connection to another server – the semantics of existing L4 load balancers [5, 13, 33, 37, 39] prevent finer-grained load balancing. As we show later, this can lead to server load imbalance and poor tail latency in the presence of skewed workloads.

There are alternative approaches that provide more flexible load balancing. L7 frontend proxies [21, 36] can distribute load at request granularity. However, they are not general because they only work with specific applications, and can become a scalability bottleneck. Prism [22] migrates connections from the proxy to backends along with each request, allowing backends to respond directly to clients. This mitigates but does not eliminate the bottleneck, because the proxy still parses every request and the migration is slow. Switch-based research systems [25, 28] direct requests to servers using

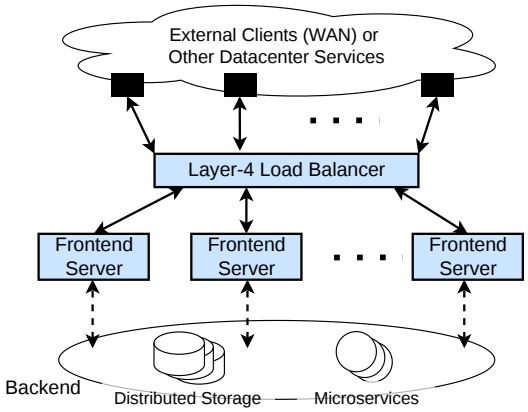


Figure 1: The typical datacenter architecture places the L4 load balancer as a middlebox between the application servers and clients. Capybara replaces the load balancer with a programmable switch and the server-side TCP stack with a kernel-bypassing stack that supports low latency TCP migration. As a result, Capybara can improve tail latency by dynamic load balancing while still supporting all TCP client endpoints.

switch hardware, but only support UDP; TCP remains the dominant datacenter protocol.

This paper addresses this problem with Capybara, a new load balancer architecture. Capybara divides the load balancing responsibilities into three components. First, a high-performance load balancer switch resembling a traditional L4 load balancer that distributes new connections across servers. Second, a host network stack that is co-designed with the switch to support μs-scale connection migration. Live migration enables *dynamic* load distribution even for established connections. Lastly, a transport interface that allows developers to define how application-level connection state is migrated together with the TCP state for correct application semantics.

The Capybara approach is compatible with two datacenter trends – programmable network devices and kernel-bypass NICs – which allows it to be used in high-performance, μs-scale environments. The switch requires minimal additional logic in the data plane, allowing it to be implemented either in a software middlebox or a hardware-accelerated platform (we demonstrate it with a Tofino [23] switch implementation). This accommodates the trend towards hardware offload for load balancing [42, 52]. The switch provides migration-aware packet routing and monitors server load distribution for load-aware connection migrations. Capybara's host network stack implements a two-phase migration protocol, co-designed with the switch data plane, to migrate connections without dropping packets or breaking connections. This keeps migration transparent to clients and avoids tail latency spikes from retransmission timeouts. By implementing connection state migration on a state-of-the-art kernel-bypassing network stack used in production at a major cloud provider [53], Capybara achieves μs-scale migration overhead.

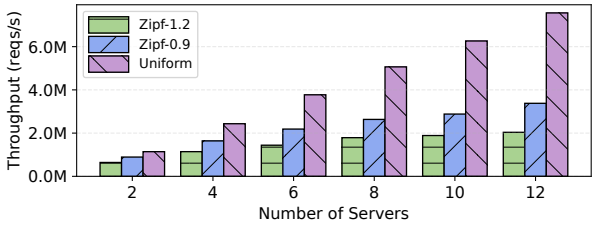


Figure 2: Peak throughput of an HTTP server pool with an existing L4 load balancer, Load-aware Weighted Round Robin (LWRR), under a p99 latency SLO of 100 μs. Skewed workloads (Zipfian [43]) across 720 long-lived connections limit server scalability, as existing L4 LBs cannot dynamically alleviate bottleneck servers. Setup details are in §8.

To evaluate Capybara's performance and generality, we implement three use cases: 1) an HTTP server, representing a stateless, scale-out application, 2) a read-only Redis [1] workload, representing a stateful, scale-out application with interchangeable replicas, and 3) TLS, representing a general-purpose, higher-level network protocol. Under realistic workloads for these applications, Capybara achieves up to 149× lower p99 latency and more than 2× higher throughput compared to state-of-the-art L4 load balancers. This is enabled by Capybara's efficient migration, which incurs less than 3 μs of host CPU overhead and completes in under 15 μs including network round-trips.

Ethics statement. This work does not raise any ethical issues.

2 System Model and Design Goals

Capybara is a new load balancer architecture for stateful network protocols. In this section, we define its design objectives, the system model, and the guarantees provided by Capybara.

2.1 Motivation & Design Goals

To motivate the need for a new architecture, Figure 2 shows the scalability of an HTTP server pool with existing L4 load balancers. The connections are long-lived, so the LB can only distribute them to servers at connection establishment time. With a uniform traffic distribution, the throughput of HTTP servers increases linearly; however, the skewed workloads do not scale well because some servers become bottlenecks.

This limitation is fundamental to the L4 load balancer design, where load balancing is done statically at connection setup. An L7 load balancer allows dynamic load balancing at the request level. However, the load balancer needs to process and understand application-level semantics. The requirement raises two limitations: (1) Each application requires a dedicated load balancer implementation, and (2) the implementation is not amenable to in-network processing, missing the opportunity of efficient, scalable hardware offloading.

To address these limitations of L4 and L7 load balancers, Capybara has the following design requirements:

Dynamic Load Balancing. Capybara should balance highly dynamic and skewed workloads even when connections are long-lived.
Compatibility. Capybara should be compatible with existing datacenter applications. A single Capybara deployment can serve multiple applications simultaneously.
Amenable to In-Network Processing. The Capybara design should be efficiently implemented on commodity programmable network hardware, including P4-based reconfigurable switches.
Generality. Capybara should work with existing network protocols over the wide-area internet. It should support TCP and other higher-level protocols like TLS and HTTP.
Fault Tolerance. Capybara should not introduce any additional failure points to L4/L7 load balancing. It should be able to recover from both failures of the switch and various network failures (e.g., dropped packets).

2.2 System Model and Guarantees

Capybara is designed to run in a datacenter setting with a pool of servers which communicate with clients either running in the datacenter or over the wide-area. Clients and servers communicate using a stateful protocol (i.e., TCP) with potentially more stateful protocols layered on top (e.g., TLS and HTTP). Each load balancer assigns client connections across the pool of *interchangeable* server end hosts.

Capybara provides a connection-oriented transport layer protocol which is fully compatible with TCP. Clients establish connections with the servers without a connection-terminating proxy. Like existing L4 load balancers, Capybara provides PCC; however, the PCC endpoint can *change* over the course of the connection. For simplicity, we assume only one endpoint of the TCP connection pair changes, which we generally refer to as the *server*. The server endpoint is *decoupled* from the physical server, *i.e.*, throughout the lifetime of a Capybara connection, the physical server responsible for the server endpoint may change. This new form of PCC is necessary to satisfy the dynamic load balancing requirement.

Because a connection can move between hosts, Capybara provides an additional **endpoint disjointness** guarantee: For a server endpoint, each physical server is assigned a *non-overlapping* stream segment. Connection packets within that segment are delivered to the assigned physical server. The segments combined provide PCC for the entire connection.

Requirements and application scope. Capybara assumes all servers in the pool are interchangeable, *i.e.*, any server can process any client request. This targets a broad class of applications with interchangeable frontends, including HTTP servers, proxies, caches, and stateless frontends. Interchangeability does not require read-only workloads or stateless applications. Mutating requests are supported when persistent state is externalized or replicated. A server may still hold application-level connection-local state, such as a TLS session. Because such state must move with the connection, Capybara requires some modifications to the application, unlike a traditional L4 load balancer. Therefore, Capybara is an opt-in extension to an L4 load balancer, with a clear division of responsibility. The load balancer data plane stays application-agnostic and only handles traffic distribution, while applications that opt in provide state-transfer callbacks for their connection-local state. Applications that do not opt in can use an ordinary L4 load balancer on the same infrastructure.

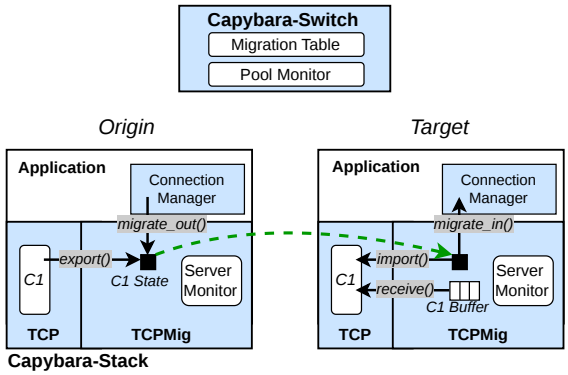


Figure 3: Capybara consists of three main components: (1) a programmable switch that provides migration-aware packet redirection and pool monitoring, (2) a host network stack that supports connection migration between servers (from *origin* to *target*), and (3) a connection manager for migrating application-level connection state.

3 Design Overview

In this work, we propose a new load balancer architecture. To achieve our design goals in §2.1, the architecture divides the responsibility of load balancing into three components:

In-network load balancer. Similar to traditional designs, the load balancer in our architecture distributes new connections to servers in the pool. It maintains a forwarding table for established connections, ensuring PCC. The LB processes headers up to layer 4, is agnostic to application semantics, and can be implemented on a programmable switch.

Load balancer-host co-designed connection migration. Our architecture enables dynamic load balancing by live migrating connections across servers. We design a migration module in the server TCP stack, and use the on-path load balancer as a passive coordinator for the migration. The load balancer enforces endpoint disjointness by managing an *atomic* migration switch-over point. The hosts and the load balancer also coordinate to implement a migration policy for balancing load.

Application-level state migration. Application state tied to a connection, such as HTTP and TLS sessions, needs to move alongside the connection for proper application semantics. We push this responsibility to the application developer, by exposing a migration API in our TCP stack. When migrating a connection, the stack makes an upcall to transfer and restore the user-defined connection state to the target.

3.1 System Overview and Components

Capybara is a concrete instance of our new load balancer architecture. Figure 3 shows the high-level system components. It co-designs a programmable network data plane with a host networking

stack. Similar to prior hardware-offloaded LBs [16, 33], Capybara deploys a programmable switch reachable from all servers in the service pool. The host networking stack is augmented with the TCP migration modules. Each server in the pool runs an instance of the application. Capybara identifies a server by its (IP, port) endpoint address. Clients establish TCP connections either directly using physical server addresses, or to a common service address which Capybara translates to physical server addresses. Capybara can migrate established TCP connections across servers for dynamic load balancing. However, connection migration is completely transparent to the clients. Capybara exposes a state migration API to the applications, which serializes and deserializes application-level connection state as part of the migration.

3.2 Switch Design

`Capybara-Switch` provides two key functionalities by leveraging its on-path position between clients and servers. First, it supports *migration-aware packet redirection* by maintaining a *migration table* that tracks migrated connections in the service pool. For each migrated connection, a table entry maps the client address to the current *target* server address. For incoming request packets matching an entry, the switch rewrites the destination to the *target* and forwards the packet. Second, a *pool monitor* enables *workload-aware* migration decisions by tracking workload distribution across servers. It periodically counts request packets destined to each server and coordinates with servers to trigger migrations that balance load across the pool. We discuss details of these migration policies in §5. All packet processing and monitoring are done in the switch data plane, achieving line-rate performance.

3.3 Host Network Stack

The Capybara host architecture consists of a migration-enabled network stack (TCP and TCPMig) and an application-level *connection manager*. This section focuses on the network stack; we discuss the application-level connection manager in §6.

TCPMig. The Capybara network stack adds a new *TCPMig* module and integrates it with the TCP stack. The TCPMig stack includes a *server monitor* that tracks local workload state, such as per-connection TCP queue lengths and traffic rates. It also implements a lightweight TCP migration protocol between two servers: the *origin*, which migrates out a connection (C1 in Figure 3), and the *target*, which migrates it in. The *target*'s TCPMig stack includes a temporary buffer to store transient packets while the connection state is in-flight (*i.e.*, when neither server holds the state), preventing TCP RSTs which can break the connection. After migration completes, the *target*'s TCP stack processes the buffered packets and resumes normal operation of the connection. We discuss details of the migration protocol in §4.

TCP. The TCP stack interacts with the TCPMig module via `export` and `import` interfaces for state migration. When sending responses on a migrated connection, the TCP stack rewrites the source address to the original server address (or Capybara's common service IP) rather than its own, ensuring that migration remains transparent to clients. This address rewriting could alternatively be performed on the Capybara switch.

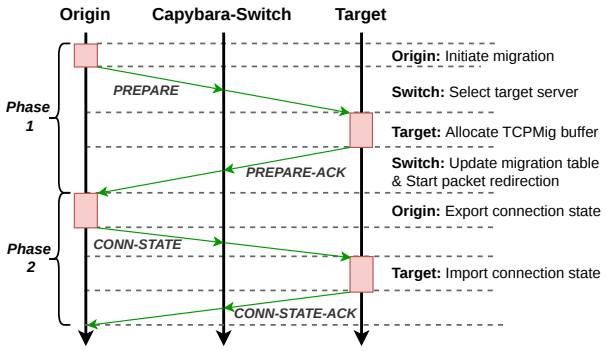


Figure 4: Capybara two-phase migration protocol in action.

4 Migration Protocol

To achieve dynamic load balancing, Capybara needs to migrate a connection at any point in its TCP stream. A key challenge in TCP migration is preventing packets from being delivered to a server that does not hold the connection state, as this triggers a TCP RST (reset). Prism [22] addresses this by blocking the connection (*i.e.*, dropping all incoming packets) during migration. Because Prism only migrates at request boundaries, no request data is in flight during migration, so blocking does not cause data loss. However, Capybara needs to migrate at any point in the stream where in-flight packets may exist, so this blocking approach cannot be directly applied.

To address this, Capybara uses a two-phase protocol. During migration, the target temporarily buffers in-flight packets; once the connection state is fully transferred, it processes the buffered packets and resumes normal operation. This section describes the two-phase protocol in detail and discusses how Capybara handles various failure scenarios.

4.1 Two-Phase Migration Protocol

Figure 4 illustrates the protocol steps. Capybara migrates connections from an *origin* server to a *target* server, following a policy discussed in §5. While a connection state is being transferred, neither server holds the state; in-flight packets must be buffered to prevent TCP resets. To this end, Capybara employs a two-phase protocol: the first phase prepares a buffer at the *target* and begins redirecting packets, while the second phase transfers the connection state.

4.1.1 Phase 1: Prepare. The *origin* sends a `PREPARE` message to the *target*; upon receiving it, the *target*'s TCPMig stack prepares a buffer to hold incoming packets until the connection state arrives. The *target* then responds with a `PREPARE-ACK` through the switch. The switch first installs a migration table entry mapping the client's address to the *target*, then forwards the `PREPARE-ACK` to the *origin*. From this point, subsequent requests from the connection are redirected to the *target* and buffered until the connection state arrives.

4.1.2 Phase 2: State Transfer. Upon receiving PREPARE-ACK, no new packets will arrive at the *origin* since the switch is already redirecting them to the *target*. The *origin* then exports the connection state to the *target* as a CONN-STATE message, including all pending packets in the TCP RX/TX queues and any session state from upper-layer protocols (§6). Upon receiving CONN-STATE, the *target* imports the connection state and processes any buffered packets. Then, the *target* completes the migration by sending a CONN-STATE-ACK to the *origin*, and resumes processing the connection. When sending response packets, the *target* uses the connection's original server address as the source, making the migration transparent to the client.

Our buffering solution effectively addresses the previous challenge. Clients can freely send packets during migration without risking breaking connections. These packets will either be processed by *origin* or be buffered by *target* and later be processed by *target* after it has imported the connection state, depending on whether they are sent before or after the switch receives PREPARE-ACK. Each migrating connection has its own buffer, which grows dynamically for packets arriving after redirection and before migration completes, and is bounded by the TCP window. In the end, Capybara is capable of initiating migration at any packet of a TCP connection and can efficiently migrate connections regardless of the sending pattern of clients.

4.2 Failure Handling

Capybara is robust to various failure cases during the migration protocol. We now elaborate on how Capybara handles each failure scenario.

TCP packet drop. Capybara migration is transparent to clients. TCP packet drops are handled in the same way as regular TCP—through client timeout and retransmission. The migration protocol properly handles redirection of retransmitted messages.

Migration protocol (TCPMig) packet drop. Capybara uses acknowledgement timeout and retransmission for reliable packet delivery during the migration protocol. Similar to TCP, we divide state transfer into segments, and use sequence numbers for retransmission and duplicate detection. Network congestion or packet loss may delay a migration but does not affect its correctness, since the connection state is delivered reliably and in-flight packets remain buffered at the *target* until the state arrives.

Switch failure. Capybara provides the same fault tolerance guarantees as existing L4 load balancers. When establishing a connection, Capybara writes the flow-to-server mapping to a fault-tolerant data store. Prior L4 load balancers also maintain this mapping reliably for PCC. Capybara additionally updates the mapping in the data store when migrating a TCP connection. The Capybara switch only stores soft state. When the switch fails, we fail over to a backup Capybara switch, and install the mapping table from the data store to the switch. The design ensures transparent failure recovery, even for migrated connections.

Host failure. Host failures before or after a migration are handled in the same way as existing TCP protocols, *i.e.*, connection termination. For host failures that happen during a migration, the client receives no further acknowledgement; it will eventually timeout and terminate the connection. The switch control plane periodically runs a liveness check and removes any stale entries from the migration table.

5 Migration Policies

The previous section presented Capybara's connection migration protocol. This section introduces policies for utilizing migration to effectively rebalance skewed workloads across servers. While we present representative policies here and demonstrate their load balancing benefits in §8.1, Capybara can accommodate other policies tailored to specific deployment requirements.

5.1 Proactive Migration

Proactive migration aims to keep workload consistently balanced across servers. To this end, Capybara continuously monitors workload distribution across servers and triggers migrations as soon as an imbalance is detected – even if no server is currently overloaded.

When to migrate? During initialization, each server registers with the Capybara-Switch. The *Pool Monitor* module in Capybara-Switch tracks a per-server workload estimate measured in PPS, and periodically (e.g., every 1 ms) sends a LOAD-NOTIFICATION message to each server, containing both the server's own PPS and the total PPS of the pool. Upon receiving this, each server computes the average load, and if its local load exceeds the average by more than a threshold, it migrates connections to shed the excess load.

Which connections to migrate? To reduce migration overhead, Capybara aims to minimize the number of migrations. To this end, the *server monitor* in Capybara-Stack tracks two connection-level metrics: 1) packets per second (PPS) and 2) RX and TX queue lengths (RQLEN and TQLEN). When the excess load is X PPS, the server selects the minimum number of connections whose combined load closely matches X PPS, prioritizing those with smaller TQLEN and RQLEN to further reduce migration overhead.

5.2 Reactive Migration

In-network load monitoring can still miss factors that lead to server hotspots, such as requests with dispersed service time, host resource contention, and scheduling issues. To deal with such cases, we implement a server-local policy that reacts quickly to sudden load spikes.

When to migrate? The total RQLEN is an accurate indicator of server workload [50], as growing queues signal insufficient capacity and directly increase latency. When the total RQLEN exceeds a threshold, the Capybara-Stack triggers migration to shed the excess load.

Which connections to migrate? When a server needs to reduce its receive queue by Y, it selects connections whose combined RQLEN closely matches Y. Similar to the proactive policy, the selection algorithm is optimized to minimize migration overhead.

5.3 Target Selection Policy

For each migration, Capybara needs to select a *target* server as the destination. Leveraging the Capybara-Switch's centralized view of the pool, we implement two policies: round-robin, and load-aware,

where the Pool Monitor selects the server with the lowest current PPS. Alternatively, servers can share load information among themselves and select targets directly.

6 Application Connection State

Capybara migrates TCP connections to rebalance workloads across servers. We extend the runtime interface (e.g., `accept()`) so applications can accept migrated connections in the same way as newly established ones, and we extend socket I/O operations to return a custom status code indicating that a connection has migrated out. These extensions preserve application backward compatibility.

Capybara preserves TCP byte-stream semantics, so applications handle message boundaries (*e.g.*, HTTP or RPC requests) exactly as they would without migration. If a request is still arriving when migration occurs, the bytes not yet delivered to the application either move with the TCP state or continue to arrive on the *target*. The application may also buffer a partially received request and have it serialized with the TCP state so that the *target* can reconstruct the request before decoding it. More generally, applications often keep state that is tightly coupled with a connection, such as TLS session data. Such state needs to be moved alongside the TCP state to avoid disrupting upper-layer semantics. This section presents Capybara's API for migrating application-level connection state. We also show two concrete applications as use cases.

6.1 Manager Registration API

We expose a *connection manager* interface that maintains per-connection state and facilitates migration to the application. The connection manager exposes `export` and `import` *upcalls* and is registered at application startup via a Capybara API. Specifically, it handles migrate-in and migrate-out events by 1) exporting application-level state for a migrating-out connection as a serialized byte array and, 2) importing serialized state for a migrating-in connection. A practical approach is to implement protocols as serializable state machines, a technique widely used in existing protocol libraries [46].

When a connection migrates out, the *origin* stack upcalls into the manager's `export` method to obtain serialized application-level state. This state is transferred to the *target* together with the TCP state in phase 2. The *target* stack then upcalls into the manager's `import` method with this state so the *target* can prepare the migrated connection before accepting it (via the extended runtime interface).

Capybara also supports nested application-level protocol stacks, such as HTTPS (i.e., TLS atop HTTP). In this case, the connection manager orchestrates all protocol layers, concatenates their states, and returns a single byte array. This design keeps Capybara agnostic to how upper-layer protocols are composed.

6.2 Use Cases

We use two concrete cases, HTTP and TLS, to demonstrate how applications use the connection manager API. These two protocols are also used in our evaluations. First, we implement a minimal HTTP server framework that supports migration. The framework buffers incoming request data, which may span multiple packets, until a complete request is received. If migration occurs in the middle of a request, this buffer is `exported` as the connection state. If migration occurs after the request has been fully received and delivered to the application, no HTTP state needs to be migrated, and the migrating-in server delivers nothing to the application.

Second, we modify the TLS-related code in the Redis [1] codebase. With the backward-compatible runtime extensions, unmodified Redis can serve client requests over plain TCP. However, Redis performs a TLS handshake when accepting a TLS connection, which would disrupt clients after migration. We develop a lightweight connection manager for Redis based on TLSe [46], a TLS implementation that supports serialization of connection state (i.e., `TLSContext` objects). Upon migrate-in upcalls, the manager initializes the `TLSContext` from the `imported` state and suppresses the TLS handshake when the connection is accepted. Because the serialized `TLSContext` captures the record-layer state, including read/write sequence numbers and any partially received record, Capybara can migrate a TLS connection at any point in the data stream, not only at record boundaries. We show our code to port Redis in §A.

Mid-stream migration. Migrating in the middle of a stream requires the application to produce a consistent snapshot of its connection-local state at the migration point. Because Capybara migrates the TCP byte stream and is agnostic to application semantics, a connection may be interrupted at any packet boundary, so this snapshot must be available at any point in the stream, not only at request boundaries. Beyond a partially received request, migration may occur while the application is in some intermediate state, such as in the middle of updating its connection-local state. In our prototype, the application buffers data that does not yet form a complete request so that the buffer migrates with the connection state and processing resumes consistently on the *target*; Capybara could alternatively expose an interface for the application to signal a consistent point and migrate only then. This is a clean division of responsibility: Capybara provides the migration mechanism and the state-transfer interface, while the application provides a serializable representation of its connection-local state. The HTTP and TLS use cases demonstrate this approach.

7 Capybara Implementation

Targeting μs-scale, high-performance datacenter services, we implement Capybara using a kernel-bypass network stack and programmable switch hardware to achieve low-overhead and scalable load balancing. We evaluate the performance benefits of different implementation choices in §8.3.

Our implementation has two main components: the host stack (`Capybara-Stack`) and the switch (`Capybara-Switch`). The host stack is written in Rust (7,500 lines of code) and integrated into Demikernel [53], a production-grade kernel-bypass library OS. Despite using a userspace stack, Capybara's transport layer is fully compatible with standard TCP, ensuring interoperability with unmodified clients. The switch is implemented in P4 [9] (1,800 LoC) and Python (500 LoC) for a Tofino-based [23] programmable switch.

Our host stack uses Demikernel as a representative kernel-bypass TCP stack for development ease and memory safety, but Capybara is not specific to Demikernel or single-threaded architecture. High-performance multi-threaded servers typically separate state across threads, avoiding shared state and eliminating inter-core locking

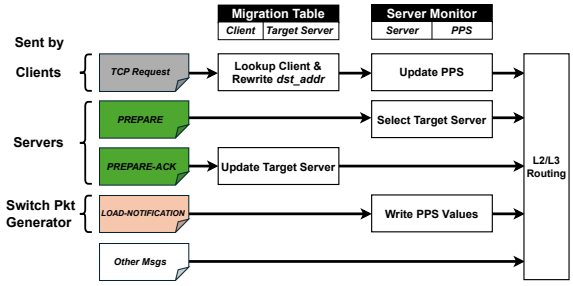


Figure 5: Capybara switch data-plane design for migration-aware packet redirection and monitoring of workload distribution.

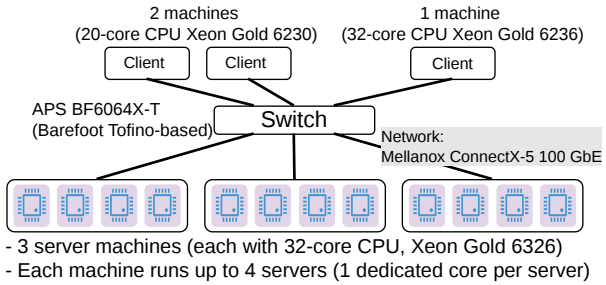


Figure 6: Evaluation cluster setup.

overhead [6, 14, 24, 40]. Capybara's connection migration operates within the existing per-thread concurrency model and does not introduce additional locking requirements. For applications that register connection managers (§6), upcalls to the application-level managers occur while the application is blocked in the library OS. Due to Demikernel's single-threaded architecture, these upcalls are never concurrent with application logic. For multi-threaded applications requiring concurrent access to connection state, existing application-level locks already control such access; these locks suffice to protect state during migration.

7.1 Switch Data Plane

Figure 5 shows the `Capybara-Switch` data-plane design. The switch has two main components: migration table for packet redirection and pool monitor. We implement both entirely in the data plane on a Tofino-based programmable switch, achieving line-rate packet processing.

Migration Table. The migration table maps each migrated connection's client address (key) to its current *target* address (value), stored in SRAM registers. Each entry's register index is computed by hashing the key; collisions are handled using backup registers. Following the two-phase protocol (§4.1), the switch creates an entry upon receiving a `PREPARE-ACK`. For every incoming TCP packet whose source address matches a key in the table, the switch rewrites the destination address to the corresponding *target* and forwards the packet. If a connection migrates multiple times, the entry is updated with the latest *target*; when the connection returns to its original server or terminates, the entry is deleted.

Pool Monitor. The Pool Monitor maintains a per-server PPS counter in SRAM registers for workload monitoring. For each TCP packet destined to a server (after any rewrite by the migration table), the switch increments the corresponding counter. Then, using the switch's on-chip packet generator [47], the pool monitor periodically (e.g., every 1 ms) sends `LOAD-NOTIFICATION` messages to each server with individual and total PPS values, records the server with the minimum PPS, and resets the counters for the next period. Upon receiving a `PREPARE` message, the pool monitor selects the *target* based on the configured policy (§5.3) and forwards the message accordingly.

Other Switch Platforms. While our prototype uses a Tofino-based programmable switch, Capybara's design can be implemented on any switch that supports header rewriting and basic counter statistics—including software switches, other data-plane programmable switches, or fixed-function hardware switches. On fixed-function switches, the migration table would use match-action tables updated via the control plane, increasing migration latency due to control-plane update costs [10]. Similarly, counter statistics would only be accessible via the control plane, slowing the system's reaction to load changes. Software switches would additionally suffer reduced throughput, as software-based packet processing scales poorly compared to hardware (see §8.2.2).

Cloud-scale deployment. Capybara does not require servers to be directly attached to a single switch; packets of a managed connection only need to traverse a common Capybara load-balancing data plane. Capybara can thus fit into hardware-accelerated, cloud-scale L4 load balancing architectures [18, 38], as long as they either maintain per-flow affinity or replicate and recover migration state, as cluster-scale load balancers already do for their flow mappings [39].

8 Evaluation

We evaluate Capybara to answer the following questions:

(1) Does Capybara's dynamic load balancing (LB) improve performance over existing approaches (§8.1)?
(2) Does Capybara's migration scale with respect to the number of connections and servers, connection state size, migration frequency, and switch resource utilization (§8.2)?
(3) How do Capybara's design choices (*e.g.* kernel-bypass, lossless, target selection policy) impact its performance (§8.3)?
(4) Can Capybara improve performance for other use cases (§8.4)?

We also evaluate the effort to modify a real-world application (Redis) for an upper-layer protocol (TLS) migration in Appendix A.

Experimental setup. Figure 6 illustrates our evaluation cluster setup. It consists of three client machines, three server machines, and one programmable switch, all connected via 100 GbE links. We deploy up to 12 server instances (4 per machine), each pinned to a dedicated CPU core. Our prototype is built on Demikernel, so each instance is single-threaded, and instances on the same machine share the IP and are distinguished by port; Capybara addresses each as an independent (IP, port) endpoint. This per-core structure is not a requirement of Capybara, but an artifact of Demikernel. Capybara applies two complementary migration policies: proactive and reactive (§5). The switch generates `LOAD-NOTIFICATION` messages every 1 ms with per-server and total PPS values. Migrations

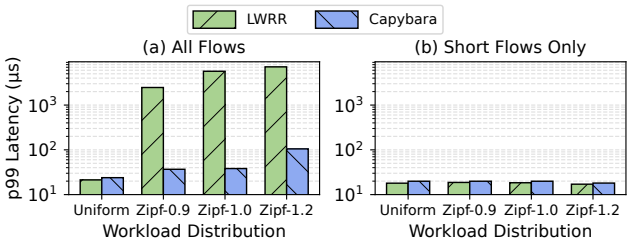


Figure 7: Skewed workloads increase LWRR's p99 latency due to long-lived connections pinned to initial servers, while Capybara mitigates this through dynamic connection migration.

are triggered when a server's load exceeds the pool average by more than 10% (proactive) or when its total TCP receive queue length exceeds 20 (reactive).

Evaluated applications. We evaluate Capybara using three applications. First, we run HTTP servers as a stateless scale-out service with a default response size of 256 bytes. Second, we use Redis [1], a widely deployed in-memory key-value store, with read-only workloads (1 million keys, 64-byte values) as a stateful scale-out service with interchangeable replicas. Lastly, we evaluate TLS-enabled servers to validate support for upper-layer protocol migration.

Workloads. We generate realistic workloads using concurrent open-loop clients with inter-arrival times following a Poisson distribution [32, 44, 54]. The load distribution across connections follows either a uniform or skewed (Zipf [43]) distribution, modeling realistic traffic patterns in large-scale datacenters.

Baseline. Following state-of-the-art load-aware LB [5, 17], we compare Capybara against Load-aware Weighted Round Robin (LWRR), which monitors per-server workload at the switch and dynamically adjusts weights to assign new connections to less-loaded servers. We also implemented other policies such as hash-based [13], least-connection [35], and simple round robin [5], and confirmed that LWRR outperforms all of them in our experiments.

8.1 Load Balancing Benefits

We evaluate Capybara's LB benefits under various workload patterns and deployment scenarios.

8.1.1 Skewed Workloads. To evaluate Capybara LB under skewed workloads, we measure p99 latency of 4 servers at 60% utilization. We use 100 long-lived and 900 short-lived connections (1:9 ratio [8, 26]). Figure 7 shows p99 latency for all flows and short flows only, under load distributions across long-lived connections ranging from uniform to Zipf (α = 0.9-1.2). Our baseline, LWRR, effectively maintains low p99 latency for short-lived flows by always assigning new connections to less-loaded servers. However, since long-lived connections remain pinned to their initially assigned servers, skewed workloads cause load imbalance and push overall p99 latency over 7 ms. Unlike LWRR, Capybara dynamically migrates connections to maintain balanced load, achieving 149× lower p99 latency under skewed workloads. On the short-flow-only workload, Capybara's p99 latency stays within 1–2 μs of LWRR, the small gap reflecting the overhead of its load monitoring and migration support. Capybara is not meant to beat LWRR on short flows alone, since an L4 load balancer already balances these at connection arrival. Capybara's advantage instead comes from rebalancing long-lived connections, which an L4 load balancer cannot move once established.

8.1.2 Step Function Workload. To illustrate how Capybara reacts to load imbalance in a controlled setting, we test a step function workload with 2 servers and 100 long-lived connections. Figure 8 shows per-server and total workload over time (top row) and p99 latency (bottom row). Each server initially handles 20% load; the load from connections on Server 0 then increases every 20 ms. With LWRR, Server 0 becomes overloaded at only 60% global utilization (after 40 ms), causing p99 latency to spike. Capybara addresses this with two complementary migration policies. The reactive policy (Capybara-Reactive) detects server-local overload (*e.g.*, via queue buildup) and promptly migrates connections, keeping p99 latency below 1 ms despite fluctuations during migrations. The proactive policy further reduces these fluctuations by migrating connections as soon as imbalance is detected, even before any server becomes overloaded. Using both policies, Capybara maintains stable p99 latency below 100 μs even under near-100% global loads.

8.1.3 Redis Benchmark. We also evaluate Capybara with a real-world application, Redis [1]. We run two Redis servers on Capybara via a shim layer that provides standard POSIX APIs. Clients generate read-only workloads with Zipf-1.2 distribution across 100 long-lived connections. Figure 9 shows the top 10% tail latency CDF at 60% utilization. LWRR suffers from p99 latency exceeding 8 ms, while Capybara maintains it around 100 μs (74× lower). We tested both with and without TLS and observed equivalent latency benefits (see Appendix A for TLS migration details).

8.1.4 Large Scale Deployments. To evaluate Capybara at larger scale, we measure peak throughput with 12 servers, 720 long-lived connections, and response sizes of 1–20 KB. Figure 10 compares Capybara and LWRR under Zipf-1.2 workloads against the ideal uniformly balanced case. LWRR achieves only 35–50% of the ideal throughput, as the most heavily loaded server becomes a bottleneck. In contrast, Capybara dynamically migrates connections away from the overloaded server, mitigating the bottleneck and achieving about 2× higher throughput than LWRR.

8.1.5 Layer 7. The preceding evaluations focused on Capybara's primary goal: dynamically rebalancing skewed workloads via connection migration at L4. However, our design choices, hardware-based LB and lossless migration, also benefit L7 LB. Figure 11 compares the peak throughput of L7 LB solutions under both closed- and open-loop client models. For fair comparison, all solutions are implemented on the same DPDK-based network stack and programmable switch. Traditional L7 proxies relay all traffic through the proxy server, creating a bottleneck that limits server utilization and overall scalability. Prism [22] addresses this by migrating connections to backend servers, allowing them to respond directly to clients via the switch hardware. However, Prism assumes closed-loop workloads where clients wait for a response at request boundaries without sending new requests, creating safe migration points with no in-flight data. Under open-loop workloads where clients

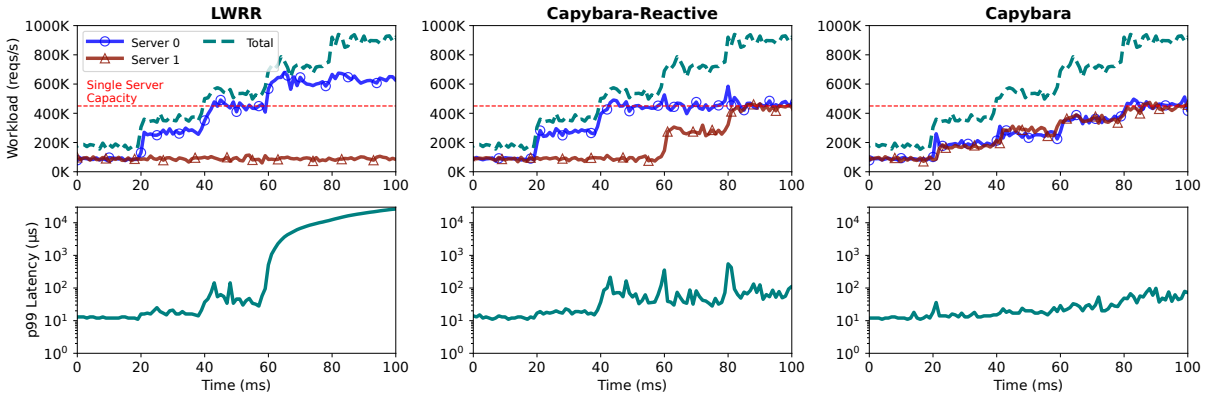


Figure 8: Capybara enables dynamic workload rebalancing across servers: the reactive policy quickly resolves server overload when it occurs, while the proactive policy prevents overload by continuously maintaining balanced workloads.

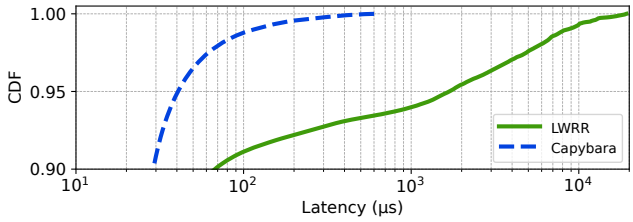


Figure 9: Tail latency CDF of Redis servers at 60% utilization with Zipf-1.2 workloads. Capybara reduces p99 latency by 74× compared to LWRR.

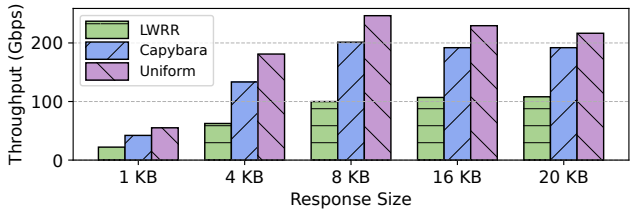


Figure 10: Even in large-scale deployments, Capybara effectively rebalances skewed (Zipf-1.2) workloads and achieves peak throughput close to the uniformly balanced case.

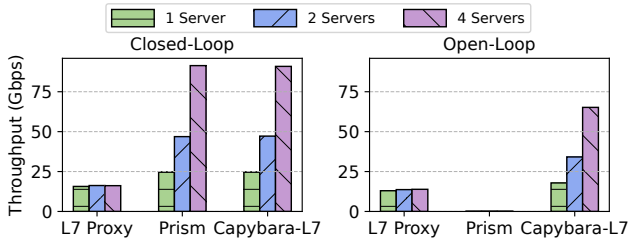


Figure 11: Even for L7 LB, Capybara matches Prism's throughput in closed-loop workloads and, unlike Prism, effectively handles open-loop workloads.

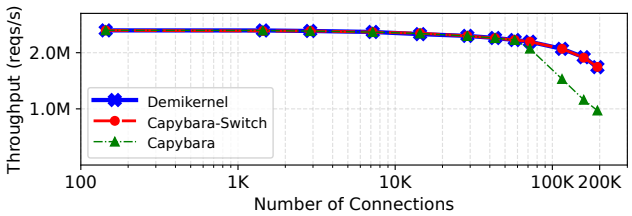


Figure 12: Connection scalability of Capybara.

issue concurrent requests, Prism drops in-flight requests during every migration, rendering the service effectively unavailable. Like Prism, Capybara-L7 is a frontend proxy that migrates connections to backends, with migration triggered by the application at request boundaries rather than by the policies of §5. Unlike Prism, however, Capybara-L7 uses Capybara's two-phase protocol to migrate connections without dropping in-flight requests. It achieves high and linearly scaling throughput for both closed- and open-loop clients. The slightly lower throughput in the open-loop case compared to closed-loop stems from the additional overhead of migrating concurrent requests buffered within each connection.

8.2 Scalability

8.2.1 Number of Connections. We evaluate the scalability of Capybara with a large number of concurrent connections. We use the `wrk` HTTP benchmarking tool [51] on 72 client cores to generate 30% load across 12 servers. Figure 12 shows throughput as a function of the number of concurrent connections up to 200K—the upper limit supported by the available client machines. The Demikernel baseline, which uses conventional switch forwarding, shows gradually decreasing throughput as the number of concurrent connections increases, due to connection multiplexing overhead. With Capybara-Switch, all connections are migrated from the initial server endpoint, so every request traverses the migration table and header-rewrite logic at the switch. Despite this additional logic, throughput matches Demikernel, indicating that Capybara switch introduces no datapath overhead. Lastly, we test Capybara with

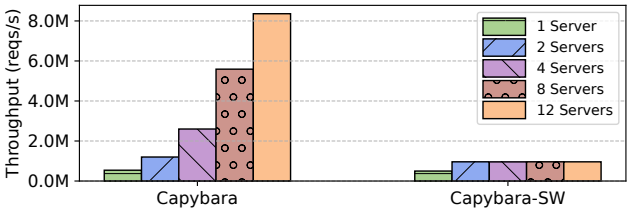


Figure 13: Capybara scales linearly with server count, whereas a software-based switch limits scalability.

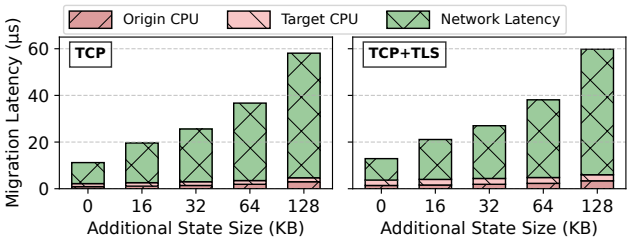


Figure 14: Total migration latency with varying connection state sizes. Large states incur only a minor latency increase for both TCP and TLS.

the server monitor updating per-connection statistics at maximum frequency (once per I/O poll). Under this setting, throughput drops more steeply beyond 80K connections due to the server monitor overhead. This overhead can be reduced by lowering the update frequency of the server monitor.

8.2.2 Number of Servers. Figure 13 shows Capybara's scalability with server count. Peak throughput scales linearly with the number of servers. As discussed in §7.1, one possible deployment option without programmable switches is a software-based Capybara switch on an endhost server (Capybara-SW); however, its scalability is constrained by the switch server's capacity.

8.2.3 Connection State Size. Connection state size varies depending on factors such as TCP buffer size and upper-layer protocol states. Figure 14 shows total migration latency with additional state sizes from 0 to 128 KB, on top of the default 75-byte TCP state (server overhead < 3 μs, total latency < 15 μs). Migration latency increases with state size, yet Capybara keeps total latency under 60 μs (including only 6 μs server CPU overhead) even with a 128 KB TLS connection state. Note that server CPUs are available for other processing during network transfers.

8.2.4 Migration Frequency. Figure 15 shows throughput impact on a TCP connection repeatedly migrated between two servers at varying frequencies. As response size increases, each migration incurs greater overhead due to larger TCP buffer size. For responses smaller than 16 KB, migration has negligible throughput impact, even at frequencies as high as 1,000 migrations per second. Even for larger responses (*e.g.*, 64 KB), Capybara supports up to 100 migrations per second with minimal performance impact.

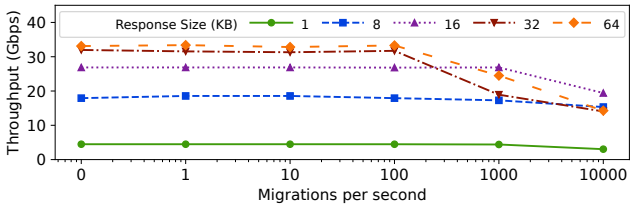


Figure 15: Capybara migration incurs low overhead even at high frequencies.

Table 1: Capybara's switch resource utilization.

Stages	Hash Bit	Hash Unit	SRAM
9	5.5%	16.7%	9.9%

Table 2: Connection state export/import latency. Capybara enables faster migration than Linux via kernel bypass.

	Avg. Operation Latency (μs)					
	Export		Import		**Total**	
	TCP	TLS	TCP	TLS	**TCP**	**TCP+TLS**
Linux	3.4 *	0.2	6.0 *	0.8	**9.4**	**10.4**
Capybara	0.1		0.2		**0.3**	**1.3**

* Requires system calls; Observed max. latency > 200 μs.

8.2.5 Switch Resource Utilization. Table 1 reports the resource utilization of our switch data plane implementation for tracking up to 200K migrated connections. Hash resources are used to index entries in the migration table, while SRAM stores the key-value pairs of the migration table and pool monitor. Importantly, the SRAM only stores minimal metadata per entry (specifically, the client and *target* addresses, and PPS per server) without maintaining any other connection state. This low resource overhead allows Capybara to scale effectively with a large number of migrated connections.

8.3 Microbenchmarks

We microbenchmark the benefits of Capybara's design choices.

8.3.1 Kernel Bypass. Table 2 compares connection state export/import latency between the Linux kernel-based approach (as in Prism [22]) and Capybara. Export saves the connection state and closes the socket at the origin; import restores it into the target stack. Over 10,000 sampled migrations, the combined latency of export and import averages about 1 μs with Capybara, compared to 10 μs with Linux. TLS operations are handled in userspace and incur the same latency in both cases. While Capybara could potentially integrate with kernel-based stacks when ultra-low latency is not critical, this approach has drawbacks. Linux's multiple system calls result in unstable latency, with maximum latencies exceeding 200 μs in our

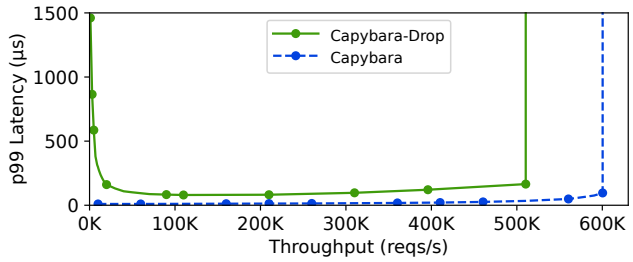


Figure 16: Capybara's buffered migration enables migrations with minimal performance impact.

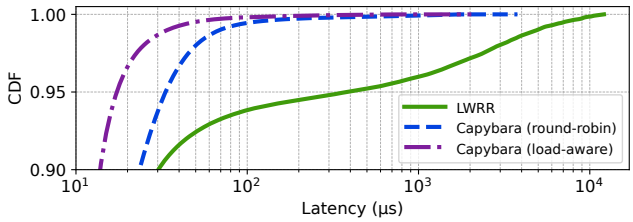


Figure 17: While round-robin target selection provides good load balancing, load-aware further improves tail latency through more precise balancing.

benchmarks and potentially higher under heavy load. Moreover, implementing the Capybara protocol in the kernel requires non-trivial kernel modifications.

8.3.2 Buffered Migration. To prevent TCP RST during migration, Prism [22] drops packets from migrating connections, while Capybara buffers these packets at the target (§4). Figure 16 compares the performance of Capybara (buffered migration) against Capybara-Drop that drops packets during migration like Prism. We migrate a connection between two servers every 1 ms while varying the request rate. At lower request rates, the client takes longer to detect and retransmit lost packets due to slower generation of the server's duplicate ACKs for subsequent requests [45], resulting in higher p99 latency. At higher request rates, although packet loss is detected more quickly, more packets are dropped during each migration, reducing peak throughput. Consequently, Capybara achieves about 18% higher peak throughput and at least 86% lower p99 latency than Capybara-Drop, demonstrating the performance benefit of buffered migration.

8.3.3 Target Selection Policy. Figure 17 compares two target selection policies, round-robin and load-aware (§5), using four servers and 100 long-lived connections. Although both policies maintain good load balance, round-robin may occasionally select an already overloaded server as the *target*, which the load-aware policy avoids by considering per-server workload. As a result, load-aware achieves p99 latency 141 × lower than LWRR and 2 × lower than round-robin.

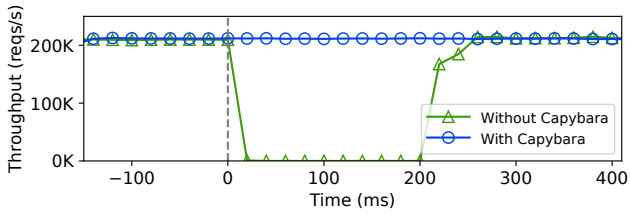


Figure 18: Throughput over time during server maintenance (starting at time 0). Capybara enables maintenance without service downtime.

8.4 Server Maintenance Use Case

Scheduled server maintenance is a constant concern for scale-out applications. With a large number of servers running, datacenter operators continuously take servers down for updates, especially in today's security-conscious environments. When a Redis server (or VM) goes down for maintenance (e.g., OS updates), existing clients are disconnected and must reestablish connections, leading to service downtime. After discussions with a large-scale cloud provider that runs Redis as a service, we found that Capybara can mitigate this. We set up an experiment with a primary and a backup Redis server on separate machines, both with TLS enabled, and run `redis-benchmark` to generate GET requests from 16 concurrent connections. When the primary server shuts down, clients detect the disconnection after the TCP retransmission timeout and reconnect to the backup. As shown in Figure 18, it takes about 200 ms to fully recover throughput. With Capybara, we proactively migrate connections from the primary to the backup before shutdown, preserving throughput throughout the maintenance.

9 Related Work

Load balancing and connection handoff have a long history in systems research. While Capybara proposes a new architecture that addresses the limitations of existing load balancers, it builds on previous work in both areas.

L4 load balancer. Layer-4 (L4) load balancers [5, 13, 15–17, 33, 37, 39] are widely deployed for TCP-based services. Providers advertise a virtual IP (VIP) for each service and allocate a pool of servers with direct IPs (DIPs). When a client initiates a TCP connection to the VIP, the load balancer selects a server and rewrites the destination to the corresponding DIP. For example, KnapsackLB [17] adapts where new connections are placed based on performance. Once a server is selected, per-connection consistency (PCC) requires all subsequent packets of the connection to be routed to that server. As a result, a connection cannot be moved after its initial placement, leaving load imbalances due to workload skew across connections unresolved (see Figure 2).

L7 load balancer. Frontend proxies like Nginx [36] and HAProxy [21] perform per-request load balancing. The proxy maintains TCP connections with both clients and backend servers, forwarding each request to a selected backend and relaying the response. This enables effective load balancing even under skewed and dynamic workloads. However, since all traffic passes through the proxy, it becomes a scalability bottleneck.

Connection handoff. Prism [22] uses the Linux TCP_REPAIR [11] feature and programmable switches to migrate TCP connections from a frontend proxy to backend servers at request granularity, allowing direct server return (DSR). XO [29] enables the same DSR without relying on programmable switches, performing the redirection in the commodity Linux network stack instead. QDSR [49] applies a similar approach to QUIC, where the frontend splits a connection into independent streams and distributes them across multiple backends. These approaches offload response traffic to the backends and relieve the frontend bottleneck of L7 load balancers. However, because the load balancer needs to interpret application-level semantics, it requires a dedicated implementation for each application and cannot benefit from the efficient, scalable offloading that in-network hardware provides. On the other hand, HA/TCP [20] provides TCP server failover and migration through state synchronization between servers, but its migration protocol is much simpler than Capybara's, because it does not support address changes and all connections must move together due to network-level failover. Capybara instead enables per-connection migration to any server in the pool by handling the address change in the switch, while keeping it transparent to the client.

Content-based routing. To achieve scalability, prior work [25, 28, 30] proposes using programmable switches that parse message payloads to make per-request routing decisions. These systems leverage the Tbps processing capability of switches to provide strong load balancing for high-throughput, μs-scale applications. However, they are limited to connectionless UDP transport, as implementing a complete TCP stack on resource-constrained switches is impractical, while TCP remains the dominant protocol in datacenters [4].

10 Conclusion

Existing L4 load balancers struggle with skewed workloads because they statically assign connections to servers. Capybara addresses this limitation with a new load balancing architecture that enables connection migration for dynamic load balancing. By co-designing a programmable switch with a kernel-bypass host network stack, Capybara achieves μs-scale migration overhead while keeping migration transparent to clients. Under realistic workloads, Capybara achieves up to 149× lower tail latency and more than 2× higher throughput compared to state-of-the-art load balancing approaches.

Acknowledgments

We sincerely thank the anonymous reviewers and our shepherd for their invaluable feedback. This work was supported partly by the Singapore Ministry of Education Academic Research Fund Tier 3 grant MOE-MOET32024-0003.

References

[1] [n. d.]. Redis - The Real-time Data Platform. https://redis.io/.
[2] 2024. What is Azure Load Balancer? https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview.
[3] 2025. Amazon Elastic Load Balancer. https://aws.amazon.com/elasticloadbalancing/.
[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.
[5] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire, Panagiotis Papadimitratos, and Marco Chiesa. 2020. A high-speed load-balancer design with guaranteed per-connection-consistency. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI '20)*. USENIX Association, USA, 667–684.
[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 29–44.
[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Broomfield, CO, 49–65.
[8] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 267–280.
[9] 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. doi:10.1145/2656877.2656890
[10] Huan Chen and Theophilus Benson. 2017. The case for making tight control plane latency guarantees in SDN switches. In *Proceedings of SOSR*.
[11] Jonathan Corbet. 2012. TCP connection repair. *LWN. net* (2012).
[12] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013).
[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 *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI '16)*. USENIX Association, USA, 523–535.
[14] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.
[15] Rohan Gandhi, Y. Charlie Hu, Cheng kok Koh, Hongqiang (Harry) 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)*.
[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* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/2619239.2626317
[17] Rohan Gandhi and Srinivas Narayana. 2025. KnapsackLB: Enabling Performance-Aware Layer-4 Load Balancing. *Proceedings of the ACM on Networking* 3, CoNEXT1 (2025), 1–20.
[18] Jiaqi Gao, Jiamin Cao, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, and Ennan Zhai. 2024. Sirius: Composing Network Function Chains into P4-Capable Edge Gateways. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, USA, 477–490.
[19] Google. 2025. Passthrough Network Load Balancer overview. https://cloud.google.com/load-balancing/docs/passthrough-network-load-balancer.
[20] Haoyu Gu, Ali José Mashtizadeh, and Bernard Wong. 2025. HA/TCP: A Reliable and Scalable Framework for TCP Network Functions. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 899–914.
[21] haproxy [n. d.]. HAProxy The Reliable, High Performance TCP/HTTP Load Balancer. https://www.haproxy.org/.
[22] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. 2021. Prism: Proxies without the Pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 535–549. https://www.usenix.org/conference/nsdi21/presentation/hayakawa
[23] Intel. [n. d.]. Intel® Tofino™. https://web.archive.org/web/20200928054540/https://barefootnetworks.com/products/brief-tofino/.
[24] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, Seattle, WA, 489–502.
[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 121–136. doi:10.1145/3132747.3132764
[26] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The nature of data center traffic: measurements & analysis. In

Proceedings of the 9th ACM SIGCOMM conference on Internet measurement. 202–208.

[27] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, Dresden, Germany. doi:10.1145/3302424.3303985

[28] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 387–406. https://www.usenix.org/conference/osdi20/presentation/li-jialin

[29] Shuo Li, Steven Chien, Tianyi Gao, and Michio Honda. 2026. Remote TCP Connection Offload and Applications. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 527–541. https://www.usenix.org/conference/nsdi26/presentation/li-shuo

[30] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI '16)*. USENIX Association, USA, 31–44.

[31] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, Huntsville, Ontario, Canada, 399–413. doi:10.1145/3341301.3359657

[32] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*. 319–330.

[33] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, Los Angeles, CA, USA, 15–28. doi:10.1145/3098822.3098824

[34] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. 2022. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of SIGCOMM*.

[35] Mustafa ElGili Mustafa. 2017. LOAD BALANCING ALGORITHMS ROUND-ROBIN (RR), LEASTCONNECTION, AND LEAST LOADED EFFICIENCY. *Computer Science & Telecommunications* 51, 1 (2017).

[36] nginx [n. d.]. Nginx HTTP and reverse proxy server. https://nginx.org/en/.

[37] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-Balancing with Beamer. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI '18)*. USENIX Association, USA, 125–139.

[38] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, Jun Zeng, Biao Lyu, Ennan Zhai, Jiao Zhang, Tao Huang, Dennis Cai, and Shunmin Zhu. 2024. LuoShen: A Hyper-Converged Programmable Gateway for Multi-Tenant Multi-Service Edge Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, USA, 877–892.

[39] 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* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 207–218. doi:10.1145/2486001.2486026

[40] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*. 337–350.

[41] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Broomfield, CO, 1–16.

[42] Dan Ports. 2024. The Future of Cloud Networking Is Systems. APSys Keynote. https://drkp.net/papers/cloudnet-keynote-apsys24-slides.pdf.

[43] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. 2012. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review* 42, 1 (2012), 16–22.

[44] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open versus closed: A cautionary tale. (2006).

[45] Wright Stevens. 1997. RFC2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms.

[46] Eduard Sui. 2018. TLSe: Single C file TLS 1.2/1.3 implementation. https://github.com/eduardsui/tlse. GitHub repository.

[47] Francisco Germano Vogt, Sergio Rossi Brito Da Silva, Fabricio Eduardo Rodriguez Cesen, Filipo Gabert Costa, Marcelo Caggiani Luizelli, and Christian Esteve Rothenberg. 2025. TFTG: Time Fidelity Traffic Generation Through P4/Tofino Programmable Hardware. *IEEE Network* (2025).

[48] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, Dai Zhang, Binzhang Fu, Frank Kelly, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. 2022. Predictable vFabric on Informative Data Plane. In *Proceedings of SIGCOMM*.

[49] Ziqi Wei, Zhiqiang Wang, Qing Li, Yuan Yang, Cheng Luo, Fuyu Wang, Yong Jiang, Sijie Yang, and Zhenhui Yuan. 2024. QDSR: Accelerating Layer-7 Load Balancing by Direct Server Return with QUIC. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 715–730.

[50] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS operating systems review* 35, 5 (2001), 230–243.

[51] wrk2 [n. d.]. wrk2 A HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2.

[52] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. 2022. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*. USENIX, Renton, WA, USA.

[53] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, Virtual Event, Germany, 195–211. doi:10.1145/3477132.3483569

[54] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). *IEEE, Seoul, South Korea* (2016), 456–468.

[55] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. 2020. Flow event telemetry on programmable data plane. In *Proceedings of SIGCOMM*.

Appendices are supporting material that has not been peer-reviewed.

A Porting Applications

```c
struct ContextEntry { // global state for TLS contexts
    int fd;
    struct TLSContext *context;
};
// context maintenance routines
struct ContextEntry *allocate_entry() { /* ... */ }
struct ContextEntry *find_entry(int fd) { /* ... */ }
void clear_entry(struct ContextEntry *) { /* ... */ }

// Redis callback that requires modification to integrate
     migration
static connection *connCreateAcceptedSocket(int fd, void *priv) {
    // Redis create and initialize connection; this is unmodified
    connection *conn = connCreateSocket();
    ...
    // modification starts here
    struct ContextEntry *entry = NULL;
    if (entry = find_entry(fd)) {
        // existing entry found for accepted connection; this
             connection is being migrated in
        return conn;
    }
    // otherwise, this is a new connection
    entry = allocate_entry();
    entry->fd = fd; // context is initialized elsewhere
    return conn;
}

// for migrated connections, this is called before
     connCreateAcceptedSocket
void migrate_in(int fd, const uint8_t *data, size_t data_len) {
    struct TLSContext *context = tls_import_context(data, data_len
         );
    struct ContextEntry *entry = allocate_entry();
    entry->fd = fd;
    entry->context = context;
}

void *migrate_out(int fd) {
    struct ContextEntry *entry = find_entry(fd);
    struct TLSContext *context = entry->context;
    clear_entry(entry);
    return context;
}

struct connection_manager_ffi capybara_connection_manager = {
    .migrate_in = migrate_in,
    .migrate_out = migrate_out,
    // omit serialization implementations
    .serialized_size = /* ... */,
    .serialize = /* ... */,
};
```

Listing 1: Required modifications to Redis for TLS migration with elaborated comments. The total LoC required is less than 50.

In this section, we evaluate the experience of porting applications that utilize upper-layer protocols. As discussed in the paper, Capybara applications register connection manager function tables that Capybara uses to obtain and provide upper-layer protocol states. An application must provide four callback functions: (1) `migrate_in`, which is called to install the session state for a migrated-in connection; (2) `migrate_out`, which cleans up for a migrated-out connection and returns session state; (3) `serialized_size`, which returns the buffer size needed to contain the serialized session state; and (4) `serialize`, which does the actual serialization of the state into the provided buffer. We found that porting applications to use Capybara was quite straightforward. For the Redis server, our modifications to support migrating Redis connections with TLS resulted in fewer than 50 lines of code changed; Listing 1 shows a simplified snippet of the code changes.