Capybara: μSecond-Scale Live TCP Migration

Inho Choi[†], Nimish Wadekar[†], Raj Joshi[†], Joshua Fried[‡],
Dan R. K. Ports[★], Irene Zhang[★], Jialin Li[†]
[†]National University of Singapore [‡]MIT CSAIL [★]Microsoft Research

ABSTRACT

Latency-critical μs-scale data center applications are susceptible to server load spikes. The issue is particularly challenging for services using long-lived TCP connections. This paper introduces Capybara, a highly efficient and versatile live TCP migration system. Capybara builds atop a deterministic, kernel-bypassed TCP stack running in a library OS to realize its μs-scale TCP migration mechanism. Using modern programmable switches, Capybara implements migration-aware dynamic packet forwarding and transient packet buffering, further reducing system interference during live TCP migration. Capybara can transparently migrate a running TCP connection in 4 μs on average. It improves the average migration host latency by about 12 times compared to a Linux kernel-based solution.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; • **Networks** → **In-network processing**; *Programmable networks.*

KEYWORDS

operating system, kernel bypass, datacenter, distributed systems, load balancing, programmable switches

ACM Reference Format:
Inho Choi, Nimish Wadekar, Raj Joshi, Joshua Fried, Dan R. K. Ports, Irene Zhang, Jialin Li. 2023. Capybara: μSecond-Scale Live TCP Migration. In *14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '23), August 24–25, 2023, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609510.3609813

1 INTRODUCTION

Modern data center networks have made dramatic improvements in network latency. The latest Ethernet fabrics can deliver a packet with single-digit microsecond latencies, while modern kernel-bypassing network stacks [2, 12, 15, 22, 26] provide sub-microsecond TCP processing. For data center applications, such advances should herald a new era of ultra-low median response time and predictable, μs-scale tail latency [4, 17, 24, 27]. Yet these ambitious performance goals remain elusive. Unpredictable workload skews, traffic bursts, and dynamic resource contention prevent applications from reaping the benefits of microsecond-scale data center hardware and networking stacks.

In principle, load balancers offer a way for scale-out services to mitigate the effects of workload skew on tail latency. Such load balancers [5, 16, 21] and frontend proxies [8, 9, 18] are near ubiquitous in data centers. Concurrently, recent research [10, 13, 14] has made great strides in achieving predictable load balancing for microsecond-scale workloads by dynamically routing individual requests. However, a key mismatch prevents their benefits from being realized in production systems: load balancers for connection-oriented protocols like TCP – which are nearly universally used in the data center – can only statically assign entire connections to backend servers, which is insufficient to ensure consistent microsecond-scale latency.

The ability to efficiently migrate TCP connections would permit load balancers to change server assignments between each request, unshackling them from the assignment they made at setup time. Unfortunately, while live migration of TCP connections has been explored in the past [9, 23], existing systems impose substantial latency overhead. These systems run a complex hand-off between servers to correctly transfer connection state without breaking an active TCP connection. The resulting latency overhead was tolerable for the original TCP migration use case – allowing mobile devices to seamlessly move between networks – but not for μs-scale data center applications. Even more recent work, aimed at load balancing, has enough overhead that migration itself will *worsen* system latency. For example, Prism [9] requires more than 45 μs to migrate a TCP connection, which is about 3 times the average network latency between servers in modern data centers.

Can we migrate live TCP connections *at microsecond scale*? We answer this question with Capybara, the first TCP migration system capable of microsecond migration latencies in the modern data center environment. Capybara, leverages three key design insights to achieve our goal of reducing

APSys '23, August 24–25, 2023, Seoul, Republic of Korea
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0305-8/23/08.
https://doi.org/10.1145/3609510.3609813

tail latencies for microsecond-scale data center applications. First, we co-design a microsecond-scale TCP stack [26] with high-performance programmable switches to perform dynamic load balancing of live TCP connection between servers, while maintaining microsecond tail latencies. Next, Capybara's custom TCP stack leverages *determinism* – a common distributed systems property, but one that is unusual for TCP implementations – to efficiently migrate TCP connection state. This determinism lets Capybara treat its TCP stack as a deterministic state machine, which can be easily moved or replicated, while reasoning about its correctness. Finally, Capybara uses programmable switches as a centralized component capable of monitoring server load and providing transient state management and migration-aware packet routing during migration. This in-network approach minimizes the impact of live migration (e.g., packet drops and retransmissions) and ensures correctness of TCP processing.

Our preliminary evaluation shows that Capybara can migrate a TCP connection in 4 μs on average. This is a 12x improvement over existing Linux-based solutions. With these latencies, we are able to migrate TCP connections to reduce tail latencies for modern data center applications with minimal overhead.

2 THE CASE FOR μS-SCALE TCP MIGRATION

Challenging datacenter workloads. Workload skew is a universal issue faced by data center applications: web servers [20], data storage [13], video streaming, caches [19, 25], and the list goes on. The issue is challenging due to the unpredictable nature of real-world traffic patterns, both spatially and temporally. If not handled properly, skewed workloads easily cause load imbalance across servers, leading to degraded service response time. The challenge is particularly daunting for modern applications with μs-scale latency service level objectives (SLOs). Any load balancing issue not resolved promptly can result in queuing delay that exceeds the latency agreement.

Switch-based load distribution. Recent work [10, 13, 14] leverages programmable switches to perform dynamic load balancing on the data path. The approach achieves strong load balancing guarantees even for highly skewed and dynamic workloads. One caveat of existing solutions is that they only work for single packet-sized requests over UDP protocol. However, most deployed data center applications rely on TCP for reliable message delivery, efficient bandwidth utilization, and congestion control. Can we apply similar data path load balancing techniques to TCP-based applications?

Stateful load balancers. Unfortunately, conventional wisdom implies a negative answer. The most widely deployed load balancing solution for connection-based applications are stateful L4/L7 load balancers. These solutions, however, require per-connection consistency (PCC): once a TCP connection is established on a backend server, the load balancer must map the connection flow consistently to the same server. While effective for flows with uniform load distribution, they fall short when flow size for each connection fluctuates over time. Other statistical load distribution solutions [1, 11] share similar shortcomings.

TCP migration to the rescue? Since load balancing during connection setup time is not sufficient for μs-scale applications, can we migrate established TCP connections for more dynamic load distribution? In fact, Linux already implements a feature called TCP connection repair [3], which prior work [9] exploits to implement live TCP migration. However, the high overhead of the kernel TCP stack, coupled with a blocking-based migration protocol, results in long migration latency. A slow migration is particularly detrimental in deployments where network round-trip times (RTTs) are within a couple tens of μs. Not able to shed load quickly also results in immediate queue build-ups for our target high-throughput applications.

Measurement study. To study the impact of migration overhead on application tail-latency guarantees, we run the following experiments: We deploy two Capybara servers and 150 clients. Each client establishes a TCP connection to one of the servers and generates HTTP GET requests. For the client, we use an open-loop and kernel-bypassing HTTP load generator based on Caladan [6]. During each experiment, we trigger 10,000 TCP migrations between the two servers. We simulate different migration overheads by generating additional delays ranging from 0 to 500 μs during each migration, and measure the latency of HTTP requests under two different workloads (150K and 170K requests per second). The experiment results are shown in Figure 1.

When the overhead of TCP migration is below 200 μs, the 99% tail latency of HTTP requests remains below a healthy 500 μs, which is desirable for μs-scale applications. However, above 200 μs overhead, tail latency increases significantly. Especially, this negative impact on the tail latency caused by migration overhead becomes even worse when the workload is higher due to more requests queuing up during the migration. The experiment demonstrates that slow migration can cause significant spikes in tail latency during periods of high server load, which is exactly the situation in which migrations are needed.

Live migration of TCP connections also faces other challenges not addressed by previous solutions. One well-known challenge in TCP migration is ensuring that packets are not delivered to a server that does not have the required TCP state. For example, if the switch starts to forward packets to

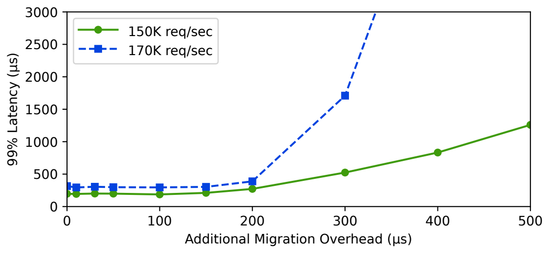


Figure 1: Tail latency as a function of increasing migration overhead.

a new owner before the state is migrated, the new owner will generate a TCP RST message, causing the connection to be aborted. Similarly, if the switch forwards any packets to the original server after a migration is completed, the original server may also send a RST. Essentially, during migration there is a small time-window during which neither of the servers has the TCP state to serve requests: from the time the original server migrates out the state, until the new owner migrates in the state.

3 DESIGN

Capybara aims to transparently migrate TCP connections with μs-scale overheads, enabling fine-grained load balancing to decrease tail latencies. Capybara includes a kernel-bypass μs-scale TCP stack and a load balancer implemented on a high-performance programmable switch. The Capybara network stack has a deterministic TCP implementation, based on Demikernel [26], and a novel two-phase commit protocol to efficiently and atomically migrate TCP connections while ensuring the correctness of the TCP protocol. The Capybara load balancer efficiently routes packets to the correct end-host before and after migration and monitors load across servers to determine when migration is needed.

Scope and assumptions. Capybara is transparent to the clients and does not require any client-side modifications. The current design and implementation of Capybara works for a single-rack-scale system and supports standard TCP connections.

3.1 Capybara Overview

The system setup consists of a top-of-rack (ToR) programmable switch that connects all servers and implements migration support such as migration-aware dynamic packet forwarding. Also, each server uses a kernel-bypassed network stack supporting standard TCP [26]. The TCP stack enforces determinism by capturing both incoming packets and time and feeding them into the TCP stack as a state machine. Capybara consists of two main components – (i) in-network load monitoring and migration management, implemented in a switch-based load balancer; and (ii) a two-phase TCP migration protocol, implemented in the kernel-bypassing TCP stack.

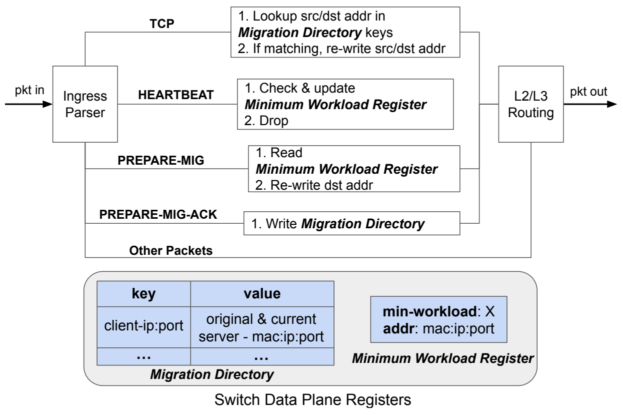

pkt in
Ingress Parser
TCP
1. Lookup src/dst addr in Migration Directory keys
2. If matching, re-write src/dst addr
HEARTBEAT
1. Check & update Minimum Workload Register
2. Drop
PREPARE-MIG
1. Read Minimum Workload Register
2. Re-write dst addr
PREPARE-MIG-ACK
1. Write Migration Directory
Other Packets
L2/L3 Routing
pkt out
key
value
client-ip:port
original & current server - mac:ip:port
…
…
Migration Directory
min-workload: X
addr: mac:ip:port
Minimum Workload Register
Switch Data Plane Registers

Figure 2: Capybara switch design. The switch maintains two data structures in the data plane: 1) migration directory to manage mapping of each migrated connection and the address of original and current server of the connection; 2) minimum workload register to track the server with current minimum workload.

3.2 Capybara Load Balancer

To support dynamic TCP migrations across servers, the switch needs to handle two main tasks: (1) maintaining original and current server address information of each migrated connection, and (2) migration-aware dynamic packet forwarding. For this purpose, we implement a migration directory inside the ToR switch as shown in Figure 2. Each entry of the migration directory is a mapping between a migrated connection and its original and current server addresses, which is added or modified in response to requests from servers that are migrating the connection (§3.3). The migration directory is implemented as a lookup table where the lookup key is an identifier of each migrated connection (e.g., a pair of the IP address and port of the client).

Migration-aware packet forwarding: For every TCP packet, the switch checks whether the source or destination address and port pair match any of the keys in the migration directory. If a packet's source address and port match a key, it indicates that the packet is a request for a migrated connection. Then, the switch reads the address of current server for this connection from the migration directory, and re-writes the destination addresses before forwarding. Similarly, if the destination address and port match a key, then the packet is a *response* for a migrated connection, so the switch re-writes the source address and port.

The migration directory is implemented as a two-level hash-based lookup table which essentially consists of two sets of data plane registers. We perform the lookup as follows. For each packet, we get a 16-bit hash value of the client address and port, and this value forms the index in the first level of the migration directory. At each index, the directory has a key-value pair. If the key matches the client address and port, the value (original and current server addresses) is returned. However, if the key does not match, it means that we have a hash collision, and we perform the same lookup in the second level of the directory. While our current implementation uses two levels, additional levels could be implemented as permitted by the available data plane resources of the switch being used[1]. Also, once the switch receives the RST packet from a connection, the switch deletes the corresponding entry from the migration directory.

Tracking the server with the minimum workload: Additionally, Capybara enables server-load-aware migration via in-network server-workload tracking. Ideally, for the best load balancing, the least loaded server should be selected as the new server when a server tries to migrate out a connection. As the central point of all servers, the switch is in a good place to track the server with the minimum workload. Capybara implements this using a switch data plane register, named minimum workload register (see Figure 2).

Each server periodically sends a HEARTBEAT message to the switch with the current workload metric value. On receiving a HEARTBEAT message, the switch checks the minimum workload register, and updates it if the received workload value is smaller than the current one in the register.

3.3 Capybara TCP stack

The Capybara TCP stack builds a TCP migration mechanism on top of the Demikernel TCP stack. To ensure correctness (i.e., no reordering or loss of packets in the TCP stream), TCP migration must be *atomic*: either the connection is owned by the *originating* server or by the *destination* server. To achieve correct TCP migration, we use a two-phase migration protocol which performs a handshake between the originating and destination server.

Once a server decides to initiate a TCP migration (e.g., it detects that the current workload is higher than a certain threshold), it selects a connection to migrate out, and sends a PREPARE-MIG message to the switch.

Phase 1. PREPARE MIGRATION The PREPARE-MIG message carries an identifier of the selected connection (i.e., client address) and the original server's address. Then, the switch reads the address of current least-loaded server from the minimum workload register, re-writes the destination address of the PREPARE-MIG to be the new server, and forwards it.

When the new server receives the PREPARE-MIG message, it allocates a temporary buffer for the connection, and replies back with a PREPARE-MIG-ACK. The temporary buffer holds messages received from the connection, until it completes migrating in the connection state.

Once the switch receives the PREPARE-MIG-ACK, it knows that the new server is prepared to buffer messages for this connection, so it can safely start to forward messages from this connection to the new server. It adds a new entry in the migration directory with the client address of the connection as the key and the original and new server addresses as the value. After this point, the switch forwards packets from this connection to the new server, following the switch design in Figure 2. Then, it forwards the PREPARE-MIG-ACK to the original server.

Using the temporary packet buffer on the new server and migration-aware dynamic packet forwarding at the switch, Capybara prevents connections from being mistakenly reset by forwarding packets to the new server only once it is aware of the migrated connection.

Phase 2. CONNECTION STATE TRANSFER Upon receiving the PREPARE-MIG-ACK, the original server knows that the new server is ready to migrate in the connection state, and that the switch is forwarding subsequent messages to the new server. Finally, before migrating the connection state, the origin server ensures that there are no outstanding NIC packet transmissions that depend on the connection's TX buffers being present in memory.

Once the original server confirms that all SEND operations submitted on the connection have been completed, it proceeds to migrate the connection state to the new server using a CONN-STATE message. The state includes all data pending in both the receive and send buffers. When the new server receives the CONN-STATE message, it processes any messages in the temporary buffer through the TCP stack, and starts to serve the connection.

3.4 Capybara architecture

Capybara integrates its TCP migration protocol with a recent kernel-bypassing library OS architecture, Demikernel [26]. Demikernel's library OS architecture provides two important benefits to Capybara. First, TCP migration, which involves multiple network stack operations, is supported entirely in userspace, resulting in significantly lower overheads and more predictable latencies. Second, userspace libraries are easier to develop, allowing us to implement more sophisticated customized protocols.

[1]Our current implementation of two-level migration directory consumes 32.5% of SRAM on an APS BF6064X-T (Intel Tofino-based) programmable switch.

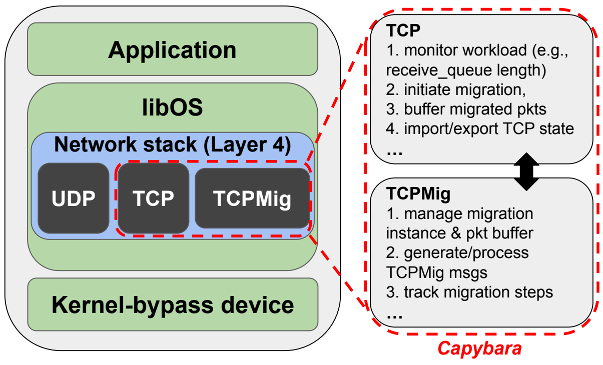


Figure 3: Capybara network stack architecture.

Figure 3 illustrates the implementation of Capybara inside the library OS. Capybara adds TCPMig as an additional Layer 4 protocol to support TCP migration. Capybara also modifies some of the standard TCP stack (e.g. to support connection serialization/deserialization). The TCP and TCPMig stacks closely interact with each other to drive and handle migrations, following the two-phase protocol discussed in §3.3.

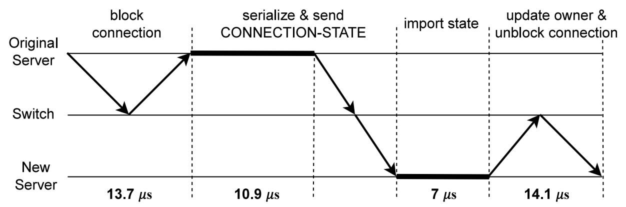
4 PRELIMINARY EVALUATION

Capybara's host-side stack is implemented with an addition of ~1100 lines of Rust code to Demikernel [26], while the switch implementation consists of ~1400 lines of P4 code. We evaluate Capybara on a hardware testbed consisting of an APS BF6064X-T (Intel Tofino-based) programmable switch and 3 servers. Each server is equipped with two Intel Xeon Gold 6226R (32 cores, 2.90 GHz) CPUs, 256 GB memory and a Mellanox ConnectX-5 100 GbE NIC.

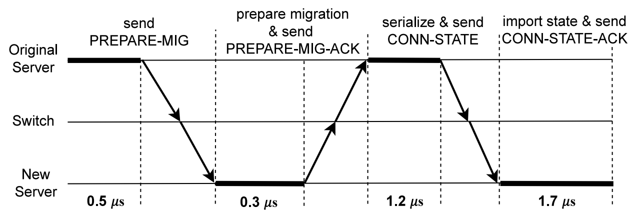
4.1 Migration Overhead

For a preliminary evaluation of our Capybara system, we compare the migration overhead of Capybara to an existing Linux-based approach, Prism. The open-source version of Prism implements the switch functionality in Linux and eBPF based software switches. To make a fair comparison, we implement the Prism switch logic in P4, and run it on the same Tofino switch. Then, we run a closed-loop client sending TCP requests to the servers to trigger migrations.

Figure 4a breaks down the average server overhead in each step of a TCP migration in Prism. Using the Linux kernel TCP repair API, a single instance of TCP migration takes more than 40 μs on average. In Figure 4b, we show the same latency break-down for a TCP migration in Capybara. Capybara takes full advantage of its kernel-bypass network stack and its switch co-designed migration protocol. It only incurs less than 4 μs of average server overhead during a migration, 10× lower than that of the state-of-the-art system.



(a) Linux-based solution (Prism). Total latency overhead = 45.7 μs.


send PREPARE-MIG
prepare migration & send PREPARE-MIG-ACK
serialize & send CONN-STATE
import state & send CONN-STATE-ACK
Original Server
Switch
New Server
0.5 µs
0.3 µs
1.2 µs
1.7 µs

(b) Capybara. Total latency overhead = 3.7 μs.

Figure 4: Average server overhead of a TCP migration.

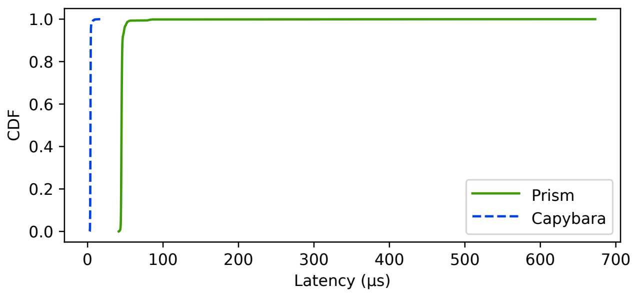


Figure 5: Distribution of aggregated server overhead for a migration. We measured the latency of each migration step for 10,000 times.

Moreover, we repeatedly migrate TCP connections for both Capybara and Prism, and plot the migration latency distribution in Figure 5. The result shows that Capybara achieves significantly more stable migration latency than Prism. Linux-based approaches are vulnerable to scheduling delays and other forms of interference, which contribute to high tail-latencies for migration. Capybara avoids this by incorporating all network stack operations such as TCP state serialization, into its userspace TCP stack. These results demonstrate that Capybara provides a promising solution for TCP migration for μs-scale data center systems.

5 RELATED WORK

Capybara is closely related to two lines of research: The first set are the recent proposals that leverage programmable network hardware to perform dynamic load balancing. The second line of research is in stateful load balancers in various combinations of software and switch hardware.

Content-based routing for load balancing. Many prior works have focused on load balancing issues in distributed storage systems. Consistent hashing [11] is a classic approach which relies on statistical load balancing. Systems such as Slicer [1] remap data shards dynamically based on load information but at coarser granularity. SwitchKV [14] and NetCache [10] leverage in-memory caching to reduce load hotspots on backend servers. SwitchKV stores caching metadata in programmable switches to perform content-based routing, while NetCache directly stores popular keys in the switch data plane. Harmonia [28] implements read-write conflict detection in switches to enable linearly scalable replica reads without violating linearizability. Pegasus [13] further improves storage load balancing by implementing a coherence directory in the switches. The approach enables dynamic, load-aware routing for both read and write requests. Existing switch-based solutions, however, only work for the UDP protocol with storage requests that fit in a single packet. Capybara addresses both of the constraints.

Stateful load balancers. There exists a long line of research on stateful L4 load balancers. Ananta [21] and Maglev [5] are distributed software-based load balancers targeting data center-scale deployments. Ananta divides traditional load balancer responsibilities into a reliable control plane and a scalable data plane, while Maglev uses a fully distributed architecture with optimized packet processing. Duet [7] leverages line-rate processing capability on programmable switches to accelerate virtual IP (VIP) to direct IP (DIP) translation. SilkRoad [16] goes a step further by moving the connection table to the switch, eliminating all software components while ensuring PCC. As we discussed, stateful load balancers can only perform load balancing at connection setup time, a limitation Capybara addresses.

6 CONCLUSION

In this paper, we explore the implication of fast live TCP migration on μs-scale data center applications. We design Capybara, a novel co-designed solution combining a kernel-bypass TCP stack and programmable data plane to enable transparent TCP connection migration in $4\,\mu$s on average, which is about 12 times faster than a Linux-based solution.

We plan to further extend the design of Capybara, besides completing the current implementation. One direction we are considering is adding TLS support to our system. Capybara currently also assumes a single ToR switch for its design. To make our approach more practical, we are looking into multi-rack and other general scalable designs. Lastly, our current switch load balancing policies are simple. We plan to explore more sophisticated scheduling algorithms and control plane designs in the future.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. We also thank Omar S. Navarro Leija for his inputs on our system design, and Fabrício B. Carvalho and Guangda Sun for their helpful feedback on the implementation. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship. Jialin Li is supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (T1 251RES2104) and Tier 2 (MOE-T2EP20222-0016).

REFERENCES

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 739–753, USA, 2016. USENIX Association.

[2] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 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, pages 49–65, Broomfield, CO, 2014. USENIX Association.

[3] Linux TCP connection repair. https://lwn.net/Articles/495304/.

[4] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2), 2013.

[5] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 523–535, USA, 2016. USENIX Association.

[6] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

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

[8] HAProxy the reliable, high performance tcp/http load balancer. https://www.haproxy.org/.

[9] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 535–549. USENIX Association, April 2021.

[10] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, Shanghai, China, 2017. Association for Computing Machinery.

[11] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, El Paso, Texas, USA, 1997. Association for Computing Machinery.

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

[13] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 387–406. USENIX Association, November 2020.

[14] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 31–44, USA, 2016. USENIX Association.

[15] 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. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 399–413, Huntsville, Ontario, Canada, 2019. Association for Computing Machinery.

[16] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 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, pages 15–28, Los Angeles, CA, USA, 2017. Association for Computing Machinery.

[17] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of SIGCOMM*, 2022.

[18] Nginx http and reverse proxy server. https://nginx.org/en/.

[19] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 385–398, Lombard, IL, 2013. USENIX Association.

[20] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 125–139, USA, 2018. USENIX Association.

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

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

[23] A session-based approach to internet mobility. http://nms.lcs.mit.edu/~snoeren/papers/thesis/.

[24] 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. Predictable vFabric on informative data plane. In *Proceedings of SIGCOMM*, 2022.

[25] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Trans. Storage*, 17(3), aug 2021.

[26] 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. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 195–211, Virtual Event, Germany, 2021. Association for Computing Machinery.

[27] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of SIGCOMM*, 2020.

[28] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, nov 2019.