BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **25**, No 3
Sofia • 2025 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2025-0027

Dynamic Load Balancing for Direct Server Return Networks Using eBPF for In-Band Metric Feedback

Hovhannes Sahakyan, Hrachya Astsatryan

Institute for Informatics and Automation Problems of NAS RA, Yerevan, Armenia
E-mails: hovhannes1417@gmail.com hrach@sci.am

***Abstract*: *Direct Server Return (DSR) enables backend servers to send responses directly to clients, bypassing the load balancer on the return path. Removing that extra hop trims end-to-end latency and prevents the balancer from becoming a bottleneck at high request rates. This paper introduces a backward-compatible DSR variant that encodes each server's load metric inside an Internet Protocol (IP) option, so the metric travels with ordinary data packets, and no polling traffic is needed. A Linux extended Berkeley Packet Filter (BPF) prototype adds only a small patch to the data path, yet yields up to 47% more requests per second than an explicit-polling baseline, requiring no changes to either the client or server. The proposed solution does not modify application logic and supports dynamic load balancing in heterogeneous and variable workloads, such as microservices, batch processing, or machine learning inference. It is fully deployable on commodity servers, runs entirely in kernel space, and eliminates separate metric-collection traffic. Performance evaluation demonstrates significant throughput and latency improvements needed for large-scale and low-overhead load balancing of real deployments.*

***Keywords*: *Direct Server Return (DSR), eBPF, Dynamic load balancing.*

1. Introduction

Load balancing is crucial for effective and equitable distribution of workloads and is extensively utilized in computing environments, web server farms, or content delivery networks [9]. Its primary function is to ensure the continual delivery of services despite the failure of a portion of the services through the intelligent handling of applications to utilize resources. It reduces task execution delays and enhances overall resource utilization, cost-effectively enhancing system performance. In practice, server load can be characterized in terms of CPU, memory, disk, I/O, or network utilization, all of which affect how jobs must be distributed. Various load-balancing methods and technologies, the layer of the network where they operate, and the setting where they are deployed, apply to different systems and workloads [16], e.g., latency-constrained web queries, throughput-constrained batch jobs, or compute-bound tasks. Various loads overload the system in multiple ways and require tailored balancing methods.

Static methods like round-robin, weighted round-robin, and Internet Protocol (IP) hash divide the workload according to fixed rules, irrespective of the system's state [10]. They are predictable, simple to build, and visually appealing for consistent and stable traffic flows. However, workloads are heterogeneous or time-varying, so static schemes cannot adapt, leading to loaded servers and low performance.

On the other hand, dynamic methods are more flexible and can be adjusted according to real-time performance indicators like the number of connections, response time, or utilization of resources [18]. As dynamic load balancers, reverse proxies route client requests judiciously based on real-time considerations like server load, response time, or active connections [13]. They run between client devices and backend servers and constantly check the health and performance of the backend servers. It enables adjusting to changing conditions to optimize resource utilization, reduce latency, and increase availability. Dynamic strategies are especially valuable in heterogeneous environments such as microservices, where services can degrade or scale asymmetrically, or exhibit bursty patterns in cloud environments. Unlike static approaches, they continuously adapt, prevent hotspots, and balance demand in near real time. The added smarts incur monitoring overhead and higher control complexity, which must be carefully maintained in check to avoid introducing new bottlenecks. Nonetheless, when server responses and client requests are passed through the load balancer, it can be a performance bottleneck [6, 7], especially under high-traffic conditions or resource-intensive applications. This is because the load balancer is a single mediation point, which adds latency, delay in packet processing, and lowered throughput, especially if the load balancer lacks resources. This is particularly problematic when responses are large or when many clients are active simultaneously.

Direct Server Return (DSR) can prevent this, where client requests pass through the load balancer, but responses are sent directly from the backend servers to the clients [3]. This bypasses the load balancer for the outgoing traffic, significantly reducing its load and improving system responsiveness and scalability. This cuts down on the traffic that the load balancer has to handle. While DSR requires special network settings and adjustments to IP addresses and routing, it can significantly enhance performance in busy environments. Nevertheless, while DSR effectively reduces the load on the load balancer, it fails to solve the problem of distributing the traffic over heterogeneous or dynamically changing backend workloads. Backend servers in contemporary distributed systems typically exhibit high variance in processing capacity and workload dynamics. For instance, batch processing environments can be exposed to jobs of different sizes, microservices can be exposed to sudden spikes in some endpoints, and machine learning inference workloads can experience broad swings based on input data. Such workload volatility demands more advanced load-balancing mechanisms than connection routing or traffic direction. However, DSR alone lacks the means to include real-time load metrics in routing, i.e., CPU, memory, or I/O usage. This makes it suboptimal in heterogeneous environments where backend servers exhibit extreme resource utilization variability and dynamic performance variation.

The article explores scaling network throughput in large installations by combining the merits of DSR and lightweight, in-band feedback mechanisms. The approach enables dynamic load balancing without additional request overhead, thus improving efficiency and scalability. To this purpose, a hardware-independent solution that does not rely on custom network interface cards or programmable switches is presented. The suggested environment can be integrated with commodity servers without altering application logic. The solution is tunable so that the balancing algorithm can be dynamically adjusted for varying workload characteristics, e.g., differentiating workloads with large requests and responses. Unlike existing DSR solutions that require additional specialized hardware or periodic inspection, ours includes in-band feedback into the data stream. Load-aware decisions can then be made with minimal overhead at the cost of losing DSR's scalability benefits. The remainder of this paper is organized as follows: Section 2 reviews related work, Section 3 introduces the proposed methodology, Section 4 presents the evaluation, and Section 5 concludes the paper.

2. Related work

Load-balancing studies have primarily evolved along the foundational axes of hardware and software implementations **using static or dynamic strategies** [11, 12]. Complementary directions have explored enhancements through hardware acceleration, protocol-level specialization, and adaptive feedback mechanisms, offering distinct performance, flexibility, and deployment complexity tradeoffs.

Hardware-accelerated systems exemplify the latency-performance tradeoff, inheriting limitations on memory and processor resources. Charon [17] demonstrates the potential of P4-programmable switches, achieving 10 million real-time and load-aware decisions per second through SYN-ACK (synchronize-acknowledge) packet instrumentation, avoiding additional control messages or instrumentation. Charon maintains a dynamic availability score table to guide server selection, reflecting each server's current load. This score is calculated based on metrics like queue length or response delay. Using the power-of-two-choices algorithm and these scores, Charon selects the less loaded server while ensuring per-connection consistency through a covert channel that encodes server identifiers in packets. Commercial implementations like Amazon Web Services' application load balancer [8] face similar limitations, as their reliance on centralized proxies creates throughput bottlenecks above 1 million connections per 1 s. Recent advances in data processing unit-based balancing, such as the NVIDIA Magnum input-output platform [4], show promise for homogeneous accelerator pools but cannot accommodate heterogeneous edge deployments mixing CPUs and GPUs.

Protocol-specialized approaches optimize for specific transport layers at the cost of generality. Google's QUIC (Quick UDP Internet Connections) load balancer [15] reduces connection establishment latency by 47% through HTTP/3 stream multiplexing. At the same time, QDSR (QUIC DSR) [21] achieves up to 12.2 times higher throughput than traditional proxies for QUIC traffic. QDSR splits QUIC connections into multiple independent streams, distributing them across real servers. These servers then send data directly to the client, avoiding the load balancer on the return path. These systems fundamentally depend on protocol semantics – QDSR's performance degrades to baseline HAProxy levels when handling legacy TCP traffic. The eBPF-based (extended Berkeley Packet Filter) solutions [19] like Istio Ambient Mesh bridge this gap through kernel-level L4 routing [5]. However, while solutions like Cilium can operate in DSR mode, they primarily rely on health checks or out-of-band metrics for load balancing, rather than the fine-grained, in-band, low-overhead feedback mechanism proposed here. By residing in kernel space, eBPF code can make real-time decisions without incurring the overhead of frequent context switches between user and kernel, potentially enhancing performance for specific workloads.

Cilium offers a similar solution to Istio; it is a networking platform that uses eBPF. It can be deployed in DSR mode, when it adds service IP and port info into an IPv4 option or IPv6 extension header. Servers then reply directly to the client, keeping the client's original IP. Only the first packet (SYN) includes this extra info for TCP, allowing a mix of DSR for TCP and source network address translation for UDP. Feedback-driven architectures represent another opportunity to address dynamic workload adaptation. Cloudflare's Unimog [22] achieves 100 μs decision latency through the eBPF-accelerated metric collection, though its inability to support DSR limits scalability for response-dominant workloads. More sophisticated systems like FaaSNet [20] employ remote direct memory access for cross-server state synchronization, which presupposes that InfiniBand infrastructure is unavailable in edge environments. A 2024 study by Zhang et al. [23] demonstrates the security challenges of these approaches, showing how in-band telemetry channels can be exploited for DDoS amplification without proper cryptographic authentication.

Functional load balancing suffers drawbacks despite these advances since most solutions support homogeneous server configurations or specialized hardware, with minimal application in edge-cloud or heterogeneous CPU/GPU environments. Traditional DSR solutions maximize throughput by bypassing the load balancer on return. Still, they cannot utilize real-time load metrics such as CPU, memory, or I/O utilization, which is crucial for dynamic workloads. Protocol-aware optimizations, like QUIC and HTTP/3 optimizations, provide excellent performance for the targeted traffic but drastically perform poorly when serving legacy TCP connections. Feedback-based solutions like eBPF-based routing reduce decision latency but usually assume ideal network conditions or, depending on high-end equipment, like Remote Direct Memory Access, curtailing deployment flexibility. Security is sometimes an afterthought, leaving in-band telemetry channels vulnerable to exploitation, like potential DDoS amplification. Most proposals also sacrifice low overhead and high-speed decision-making through increasing control traffic or limited scalability. Furthermore, existing approaches rarely integrate heterogeneous protocol support, dynamic load sensitivity, and low-latency decision-making into a single framework. These constraints make the need for in-band, lightweight signaling mechanisms more critical, as they offer real-time statistics without trading away DSR benefits. Unlike existing solutions, it supports TCP and QUIC traffic, operates entirely in commodity environments, and is secure by inserting metrics in a backward-compatible manner. This set of properties offers a scalable, low-overhead solution suitable for modern, heterogeneous workloads such as microservices, batch, and ML inference.

From this analysis, three main gaps are identified. First, existing systems cannot jointly optimize QUIC streams and TCP connections without performance penalties. Second, the hardware dependencies (e.g., requiring P4 switches or remote direct memory access-capable NICs) prohibit the possibility of flexible deployment over heterogeneous environments. Third, security is often provided as an add-on rather than a built-in property, notably in DSR deployments that avoid middlebox probing. A lightweight, in-band signalling scheme paired with software-defined metric aggregation is introduced to address these limitations. The design achieves sub-100 μs latency while supporting heterogeneous protocols and deployment scenarios without sacrificing security. The proposed method explicitly addresses these gaps, providing a practical, secure, and deployable DSR extension for real-world cloud and edge networks.

3. Methodology

The Layer 3 extension to the DSR method is proposed to manage routing decisions at the network level, allowing efficient handling of large-scale traffic without requiring changes to application-layer protocols or server configurations. The request rate, latency, and probe efficiency metrics are used to evaluate the effectiveness of this approach. Dynamic load balancing algorithms use server-side performance indicators based on a busyness score to make these decisions. These scores are commonly obtained through query/response probes or side-channel request/reply, causing redundant packet flows. To minimize the overhead, our approach embeds the load feedback directly in the standard return packets from Real Servers (RSs). These packets are then sent to the load balancer, which extracts and caches the embedded scores before relaying the response with the embedded scores to the client.

In addition, this design uses routing-level awareness of the DSR construct natively, in that decisions are not only load-aware but also resilient to packet path asymmetries typical in large-scale deployments. The system can avoid control-plane congestion and convergence delays by embedding the feedback in-band.

The methodology operates in four stages to facilitate dynamic load-aware routing within a modified DSR framework:

Step 1. Client tuple encoding. Upon receiving a client request, the load balancer encodes the client's IP address and port into an IPv4 option field or an IPv6 extension header. *This encoding is backward compatible with standard IP parsers.*

Step 2. Backend selection. The balancer selects a target RS based on a locally maintained map of busyness scores, which reflect each server's recent load conditions.

Step 3. Metric triggering. If a new or updated load metric is needed, the balancer sets the score-request flag in the packet header to signal the RS to provide its current busyness score.

Step 4. Feedback propagation. Upon receiving such a request, the RS caches the client tuple, processes the request, and – if the score-request flag is set – attaches its current busyness score to the next response packet. This response is routed back through the balancer, which extracts and updates the server's score before forwarding the reply to the client. Responses without the flag follow the standard DSR return path and bypass the balancer.

This four-step cycle is intentionally lightweight, with each step executed in-line without the necessity of additional synchronization between data and control planes. The design's modularity facilitates extension to other feedback metrics (e.g., GPU load, queue length) and the depicted busyness score. The platform is deployable and non-intrusive, with the following key properties:

Preserves application transparency by implementing all logic within the eXpress data path or traffic control eBPF programs, requiring no modifications to client or server applications.

Eliminates out-of-band polling for server load metrics by embedding feedback in in-band return traffic.

It runs on commodity infrastructure and requires only a standard Linux kernel without requiring specialized NICs or programmable switches.

Supports tunable feedback granularity, enabling operators to adjust the tradeoff between control-traffic overhead and metric freshness by configuring the sampling interval (e.g., every *n*-th packet).

Remains protocol-agnostic above Layer 3, ensuring compatibility with diverse transport and application protocols without additional customization.

Furthermore, using eBPF ensures packet-processing logic can be updated at run time, allowing operators to install evolving scheduling policies without kernel recompilation. This is a real-world advantage in experimental or multi-tenant environments where nimbleness is key.

For IPv4, the options segment of the header is used to embed metadata, with each field identified by a one-byte type code (called kind). This option is required once per connection (e.g., TCP SYN) or handshake message (e.g., QUIC initial); subsequent packets rely on a cached state, avoiding repeated header overhead. For IPv6, three new extension headers are defined to serve a similar purpose. The suggested prototype marks every *n*-th packet to request a metric update, where the sampling interval is configurable at runtime. The busyness score can represent individual system-level metrics, such as CPU utilization, memory pressure, number of active requests, or mean response time, depending on the monitoring and load balancing strategy. This sampling mechanism strikes a balance between responsiveness and efficiency: more frequent sampling improves freshness at the expense of bandwidth, and infrequent sampling reduces control overhead but may under-react to short-duration computations in load.

Custom IPv4 options (Table 1) are encoded using the standard 1-byte kind field, which includes a 1-bit copied flag (indicating fragment inheritance), a 2-bit class field (00: control, 10: monitoring/debug, 01 and 11: reserved), and a 5-bit option number. All options, except End-of-Options (0) and No-Operation (1), include a length byte and associated data. Each option must be aligned to a 32-bit boundary, with a minimum total size of 4 bytes. Such careful encoding structure ensures backward compatibility with existing IP parsers, with performance tradeoffs where choices direct slower packet-processing paths in hardware-accelerated routers.

Table 1. IPv4 options

IPv4 kind	Copied	Class	Number	Length (bytes)	Data	Purpose
155	1	00	27	8	4B client IP and 2B client port	Encodes client tuple for DSR
124	0	10	28	4	-	Tells the server to refresh its busyness score
125	0	10	29	4	2B busyness score (0-65535)	Sends the server’s load metric back to LB

IPv6 conveys similar data via extension headers. Unassigned types 27-29 (per IANA) are internally repurposed. Each header must be a multiple of 8 bytes and include fields for the next header, length, and the option type.

Table 2. IPv6 extension headers

IPv6 extension header type	Length (bytes)	Data	Purpose
27	24	16B source IP and 2B source port	Encodes client tuple for DSR
28	8	-	Tells the server to refresh its busyness score
29	8	2 B busyness score (0-65535)	Sends the server’s load metric back to LB

Each packet carries at most one custom option. Type 27 encodes the client tuple for the DSR path, 28 requests a metric update from the server, and 29 returns the metric. Packets that do not need any of these functions omit the option entirely. The proposed system is implemented using eBPF technology, using IPv4 at the network layer and TCP at the transport layer. However, the architecture is not inherently limited to these protocols and can be extended to support IPv6 and QUIC with minor modifications. The number of active connections on each RS is tracked in real time using eBPF maps to compute the busyness score. There are two possible packet-processing flows, illustrated in Fig. 1. For clarity, the diagram shows Flow 1 on RS 1 and Flow 2 on RS 2, but both flows apply to any server.

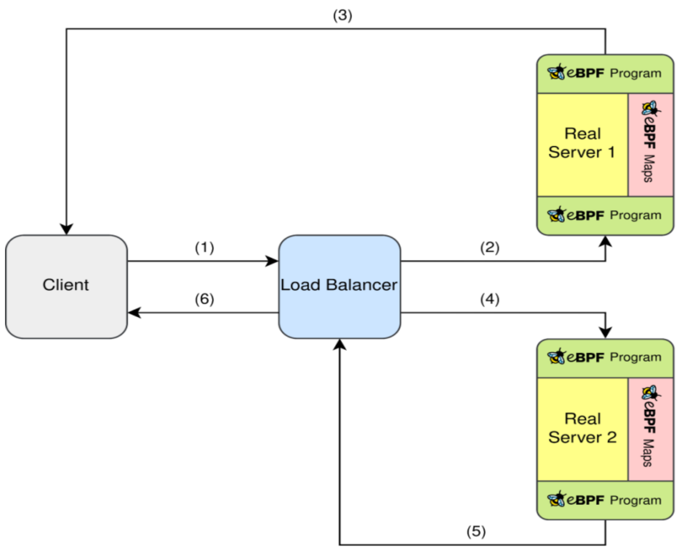

Fig. 1. Packet flow in the implementation

In the case of the first flow, the load balancer embeds the client's IP address and port into a custom IPv4 option field and forwards the packet to a selected RS based on the current load-balancing decision. After dispatching the packet, the balancer immediately releases any associated state without waiting for a response. Upon receiving the packet, the RS's ingress eBPF program parses the IPv4 options and stores the client tuple in an eBPF map using the key [source IP, destination IP, source port, destination port]. Later, when the server generates a response, the egress eBPF hook checks for a matching entry in the map using the reversed 4-tuple key. If found, it rewrites the packet's destination fields to match the original client, enabling DSR. In the case of the second flow, the load balancer marks a packet with a metric request flag using the same IP option field. It forwards the packet to the selected RS, retaining a temporary state for a single return response. The RS's ingress eBPF logic stores a "metrics requested" flag under the same 4-tuple key. When the response is sent, the egress eBPF logic checks for this flag and, if set, appends the server's current busyness score, derived from the number of active connections, into the IPv4 options of the response packet. This packet is then routed back through the load balancer. Upon reception, the balancer extracts and records the metric, strips the custom option, and finally forwards the cleaned packet to the client. The load balancer controls the flow of packets. Applying the second flow to every *n*th packet gathers metrics for dynamic load balancing.

Using eBPF on real servers enables dynamic, high-performance packet processing without modifying the existing service application code. Each TCP connection state is stored in an eBPF map. Each time a connection is established (SYN flag in TCP) in ingress, the program increments the busyness score, and when a connection is closed or reset (FIN or RST flags) on egress, it decrements it. Alternatively, the application can manage the busyness score via the bpf_map_update_elem system call. The load balancer selects the server with the lowest busyness score; if two servers have the same score, it selects randomly.

This architecture avoids separate metric-collection messages by carrying busyness data in the IP header. Therefore, no extra packets are exchanged. Merging DSR with on-path feedback achieves dynamic load balancing while keeping processing inside the kernel: eBPF parses the custom IP option, eXpress data path handles ingress for minimal overhead, and TC handles egress where XDP is unavailable. Because the load balancer chooses when to request an update, operators can tune the balance between control-traffic overhead and metric freshness. Despite its benefits, the scheme carries several limitations, such as adding custom IP options, which enlarges each packet and can lower the maximum transmission unit. This effect is more pronounced for IPv6. The prototype also requires eBPF support and runs only on recent Linux kernels, although early efforts exist to bring eBPF to Windows and to user-space VMs on other platforms. Compatibility is not guaranteed across the network path: certain middleboxes may strip or mis-handle non-standard options, and the extra header fields can complicate packet tracing and diagnosis. This approach is best suited for environments with variable server workloads. Such workloads include workflow or batch processing systems with uneven job sizes, microservices architectures where traffic can spike unpredictably for certain services, and machine learning deployments where processing time depends on input size or model complexity.

In contrast, it is less effective in environments with predictable or static traffic patterns, such as content delivery networks serving cached static content or simple request-response services with uniform load. Only the load balancer is modified; real-server applications stay untouched, running an attached eBPF helper. The developed library [24] is compiled into eBPF byte-code with the target IP and port as constants, then loaded with xdp-loader (ingress) and tc (egress) tools.

4. Evaluation

All experiments were conducted using the Armenian national cloud infrastructure resource, providing IaaS and SaaS services to academia and universities [1, 2, 14]. In the test environment (Fig. 2), the load balancer runs on a Broadcom BCM2711 SoC (1.5 GHz 64-bit quad-core ARM Cortex-A72) with 4 GB RAM, Debian 13, and Linux kernel 6.12. The RSs use Intel Core i7-9750H CPUs (2.60 GHz), 16 GB RAM, Ubuntu 22.04, and kernel 6.5. Each experiment was repeated multiple times, and averages were computed to minimize the effect of transient fluctuations. Standard deviation was also measured to quantify variance in response time and throughput.

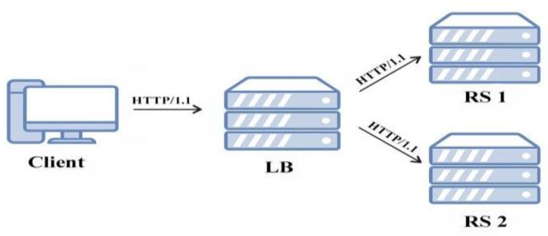

Fig. 2. Benchmark's network topology

To isolate the impact of DSR and the probing strategy, the following four configurations are evaluated (see Table 3):

Pass Through. Standard round-robin load balancing without DSR.

DSR Round-Robin. DSR enabled with round-robin distribution.

DSR Request Busyness. Load balancer queries server load before forwarding; with only two servers, power-of-two choices are unnecessary.

DSR Dynamic. Proposed method; every 100th packet triggers a busyness probe by disabling DSR. The interval was empirically determined.

Locust is an open-source, Python load-testing framework that models user workflows in code and spawns thousands of concurrent virtual clients to approximate production traffic. It scales transparently from a single node to a distributed cluster and reports live throughput, latency, and error metrics through a web interface. Script-defined scenarios remain reproducible, easily version-controlled, and quick to refine, enabling performance studies with minimal instrumentation. Version 2.33.2 is used.

These scenarios were selected to provide both a baseline (pass-through), a simple stateless policy (round-robin), a stateful but static policy (request busyness), and finally our adaptive dynamic policy. This layering of experiments allows for clear attribution of observed performance improvements. For the experiments, Locust runs a synthetic HTTP workload of 150 concurrent virtual users issuing back-to-back GET requests for 10 minutes (no inter-request delay) without a warm-up or discard phase. This lets us stress the load-balancer and compare setups under a controlled, repeatable workload. The 90th, 95th, 98th, and 99th percentiles were computed over the complete set of requests for the run, with no binning applied. Dynamic load balancing is only valid when the processing time of requests is unpredictable, so servers can take from 500 ms to a few seconds to process one request when the server is not busy. The request is, on average, 1 KB, and the response is, on average, 46 KB. Each one is run for 10 min.

Table 3. Benchmark results

Name	Median response time	Average response time	RPS	90%	95%	98%	99%
Pass Through	3.8 s	5184 ms	23.24	8.6 s	12 s	18 s	27 s
DSR Round-Robin	1.9 s	2693 ms	40.55	4.4 s	5.9 s	8.9 s	13 s
DSR Request Busyness	1.7 s	1945 ms	44.33	2.8 s	3.3 s	4.1 s	5 s
Dynamic DSR	1.2 s	1611 ms	65.45	2.7 s	3.4 s	4.5 s	5.5 s

Dynamic DSR improves average response time by 3.2 times over the baseline (Pass Through) and 20% over DSR with explicit busyness queries. Dynamic DSR increases throughput by +47.6% RPS versus DSR Request Busyness (65.45 vs 44.33). Mean latency drops −17% (1.61 s vs 1.95 s) and median −29% (1.2 s vs 1.7 s). At the tail, P95 is similar (+3%, 3.4 s vs 3.3 s), while P98–P99 are modestly higher (+9.8% and +10%). In short, Dynamic DSR mainly raises capacity while keeping P95 comparable and slightly increasing deepest tails.

As shown in Fig. 3, both DSR variants maintain at least 1.5 times higher and more stable RPS than the baseline. The proposed method combines these gains to deliver 2.8 times higher RPS than Pass-Through.

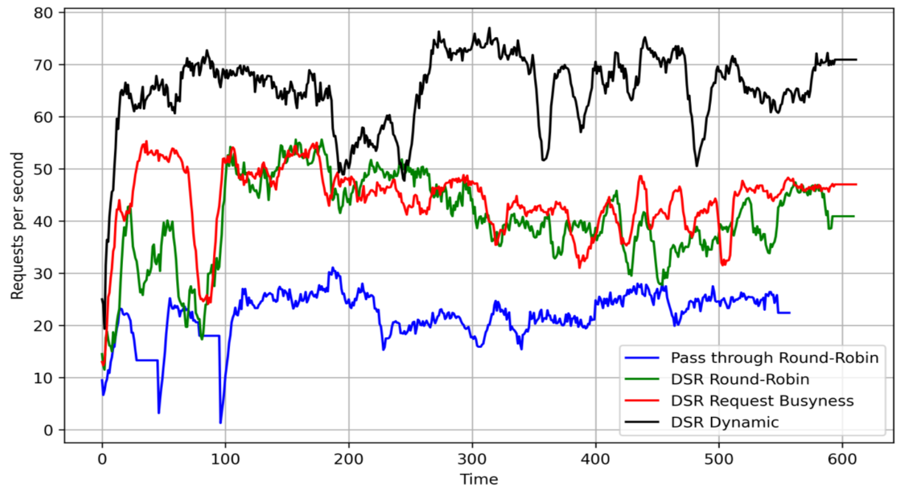

Fig. 3. Request-rate trace (10 min) for baseline and three DSR variants

The RPS ordering appears immediately. Within 10 s, Dynamic DSR, DSR Request, DSR Round-Robin, and Pass-Through reach ~34.8, 22.1, 15.8, and 10.7 RPS, about 53%, 50%, 39%, and 46% of their averages. By 30 s, they reach ~52.9, 36.1, 26.8, and 16.6 RPS (81%, 81%, 66%, 72%). This shows that Dynamic DSR's advantage is present from the start and persists throughout the run.

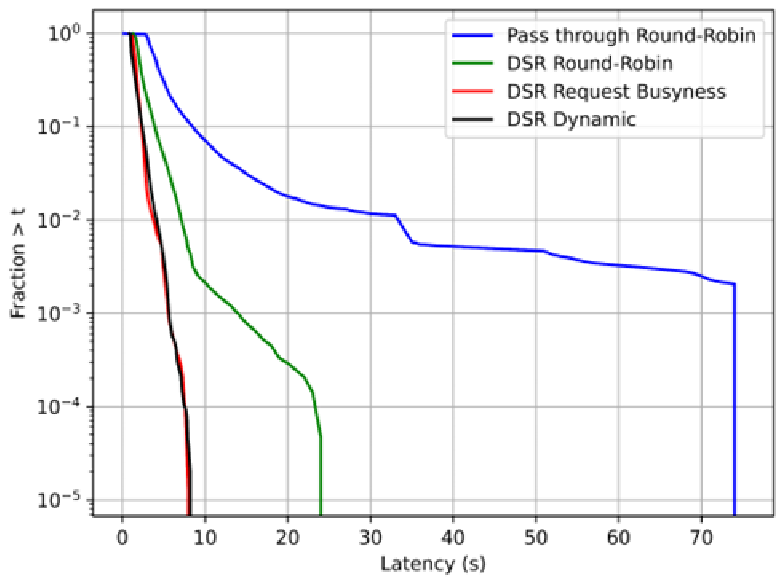

Fig. 4. Cumulative distribution of tail latency

Fig. 4 is the Complementary CDF (CCDF) of latency: for each latency t on the *x*-axis, the *y*-axis shows the fraction of requests slower than *t* ("tail" mass). The pass-through baseline (blue) has a heavy tail, stretching far to the right. Enabling DSR with round-robin (green) cuts that tail by orders of magnitude. Most importantly, the two adaptive methods: DSR Request Busyness (red) and Dynamic DSR (black), track each other almost exactly across the full range, showing no meaningful loss in latency, including the tail. Combined with the higher throughput (RPS) of Dynamic DSR, this figure shows the key result: we keep the latency profile while gaining a lot of capacity.

Our testbed uses symmetric links – the client-to-LB and LB-to-server paths have the same hop count and RTT – so all nodes sit close in the network. This symmetry is uncommon in real deployments, so the latency gap between pass-through and DSR is expected to shrink. The workload uses synthetic GETs on a small, symmetric testbed (low RTT, identical servers). Real deployments with WAN links or heterogeneous servers may shift absolute values; relative trends should hold. The examination demonstrates that incorporating busyness scores into in-band traffic allows the resulting DSR extension to outperform static and stateless baselines in terms of throughput, efficiency, and responsiveness without degrading latency properties. This offers more justification for deployment in actual heterogeneous and dynamic settings.

5. Conclusion

After examining several approaches and conducting benchmarks, this paper shows that feedback-driven architecture offers substantial benefits over standard setups. The presented method avoids separate metric-collection requests by allowing servers to return responses directly while carrying real-time load information. This feature proves especially useful for workloads that vary widely, such as microservices or machine learning tasks; in the test, it shows up to 47% improved throughput. Future work will focus on prototyping IPv6 extension headers and QUIC streams and will quantify the parsing overhead and header size impact on latency and throughput. The paper sketches the option layout but does not evaluate it in practice. Comparative wire size and response time measurements under both transports will reveal whether the added flexibility outweighs potential parsing overhead.

Another open question is survivability on real paths. Many middleboxes still drop packets that carry unknown IP options. A systematic traverse of common enterprise firewalls, CGNATs, and cloud observability taps can quantify these losses. Fallback encodings – such as wrapping the flow in GRE/UDP, or co-locating the score in a TCP option – can be benchmarked to identify the least intrusive workaround.

Currently, the method only works with a Layer-4 balancer; adapting it to an upstream Layer-7 LB could create more adaptation advantages.

Finally, sampling frequency is fixed in the current code. Allowing the balancer to expand or shrink the "every *n*th packet" window based on short-term variance in server load would reduce overhead when traffic is stable and tighten reaction time during bursts. An adaptive controller – driven either by heuristic thresholds or a simple PID loop – could be implemented entirely in eBPF, making it lightweight and predictable. Overall, the novel dynamic DSR mechanism presents an effective, deployable, and useful load-aware packet routing mechanism, and the extensions presented here will further enhance its robustness, flexibility, and utility in highly variable and heterogeneous environments.

***Acknowledgements*:** The research was supported by the Science Committee of the Republic of Armenia, by the Base, and the project entitled ”Self-organized Swarm of UAVs Smart Cloud Platform Equipped with Multi-agent Algorithms and Systems” (No 21AG-1B052).

References

1. A s t s a t r y a n, H., W. N a r s i s i a n, A. M i r z o y a n, V. S a h a k y a n. Research Cloud Computing Ecosystem in Armenia. – In: Proc. of 9th Int. Conf. “Distributed Computing and Grid Technologies in Science and Education”, 2021, pp. 117-121.
2. A s t s a t r y a n, H., Y. S h o u k o u r i a n, V. S a h a k y a n. The Armcluster Project: Brief Introduction. – In: Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’04), 2004, pp. 1291-1295.
3. B o u r k e, T. Server Load Balancing. O’Reilly Media, Inc., 2001. 175 p.
4. B r u c k n e r, F., S. K o r a l t a n, C. A b e r t, D. S u e s s. Magnum.NP: A PyTorch-Based GPU-Enhanced Finite Difference Micromagnetic Simulation Framework for High-Level Development and Inverse Design. – Scientific Reports, Vol. **13**, 2023, No 1, 12054.
5. C a l c o t e, L., Z. B u t c h e r. Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe. O’Reilly Media, 2019.
6. D y m o r a, P., M. M a z u r e k, B. S u d e k. Comparative Analysis of Selected Open-Source Solutions for Traffic Balancing in Server Infrastructures Providing Web Service. – Energies, Vol. **14**, 2021, No 22, 7719.
7. GitHub Repository.
http://github.com/hov1417/dynamic-lb-dsr-ebpf
8. H o t a, N., B. K. P a t t a n a y a k. Cloud Computing Load Balancing Using Amazon Web Service Technology. – In: Progress in Advanced Computing and Intelligent Engineering. Springer, 2020, pp. 661-669.
9. I v a n i s e n k o, I. Methods and Algorithms of Load Balancing. – International Journal of Information Technologies & Knowledge, Vol. **9**, 2015, No 4, pp. 340-375.
10. L i, T., D. B a u m b e r g e r, S. H a h n. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin. – ACM Sigplan Notices, Vol. **44**, 2009, No 4, pp. 65-74.
11. M a, C., Y. C h i. Evaluation Test and Improvement of Load Balancing Algorithms of Nginx. – IEEE Access, Vol. **10**, 2022, pp. 14311-14324.
12. M a l l i k a r j u n a, B., P. V. K r i s h n a. OLB: A Nature-Inspired Approach for Load Balancing in Cloud Computing. – Cybernetics and Information Technologies, Vol. **15**, 2015, No 4, pp. 138-148.
13. M i l a n i, A. S., N. J. N a v i m i p o u r. Load Balancing Mechanisms and Techniques in the Cloud Environments: Systematic Literature Review and Future Trends. – Journal of Network and Computer Applications, Vol. **71**, 2016, pp. 86-98.
14. P e t r o s y a n, D., H. A s t s a t r y a n. Serverless High-Performance Computing over Cloud. – Cybernetics and Information Technologies, Vol. **22**, 2022, No 3, pp. 82-92.
15. K u m a r, P., B. D e z f o u l i. Implementation and Analysis of QUIC for MQTT. – Computer Networks, Vol. **150**, 2019, pp. 28-45.
16. R a t h o r e, N. A Review Towards: Load Balancing Techniques. – i-Manager’s Journal on Power Systems Engineering, Vol. **4**, 2016, No 4, pp. 47-60.

17. R i z z i, C., Z. Y a o, Y. D e s m o u c e a u x, M. T o w n s l e y, T. C l a u s e n. Charon: Load-Aware Load-Balancing in P4. – In: Proc. of 17th International Conference on Network and Service Management (CNSM'21), 2021, pp. 91-97.
18. S h a f i q, D. A., N. Z. J h a n j h i, A. A b d u l l a h, M. A. A l z a i n. A Load Balancing Algorithm for the Data Centres to Optimize Cloud Computing Applications. – IEEE Access, Vol. **9**, 2021, pp. 41731-41744.
19. S h a r a f, H., I. A h m a d, T. D i m i t r i o u. Extended Berkeley Packet Filter: An Application Perspective. – IEEE Access, Vol. **10**, 2022, pp. 126370-126393.
20. W a n g, A., S. C h a n g, H. T i a n, H. W a n g, H. Y a n g, H. L i, Y. C h e n g. {FaaSNet}: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. – In: Proc. of USENIX Annual Technical Conference (USENIX ATC'21), 2021, pp. 443-457.
21. W e i, Z., et al. QDSR: Accelerating Layer-7 Load Balancing by Direct Server Return with QUIC. – In: Proc. of USENIX Annual Technical Conference (USENIX ATC'24), Santa Clara, CA: USENIX Association, 2024, pp. 715-730.
22. W r a g g, D. Unimog – Cloudflare's Edge Load Balancer, 2020.
https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer
23. Z h a n g, X., et al. Security Implications of In-Band Telemetry in Feedback-Driven Systems: A Study of DDoS Amplification Risks. – IEEE Transactions on Cloud Computing, Vol. **12**, 2024, No 3, pp. 456-470.
24. Dynamic Load Balancing for DSR Networks Using eBPF for In-Band Metric Feedback. 2025, GitHub Repository.
http://github.com/hov1417/dynamic-lb-dsr-ebp

Received: 03.06.2025. First Revision: 23.08.2025. Accepted: 28.08.2025