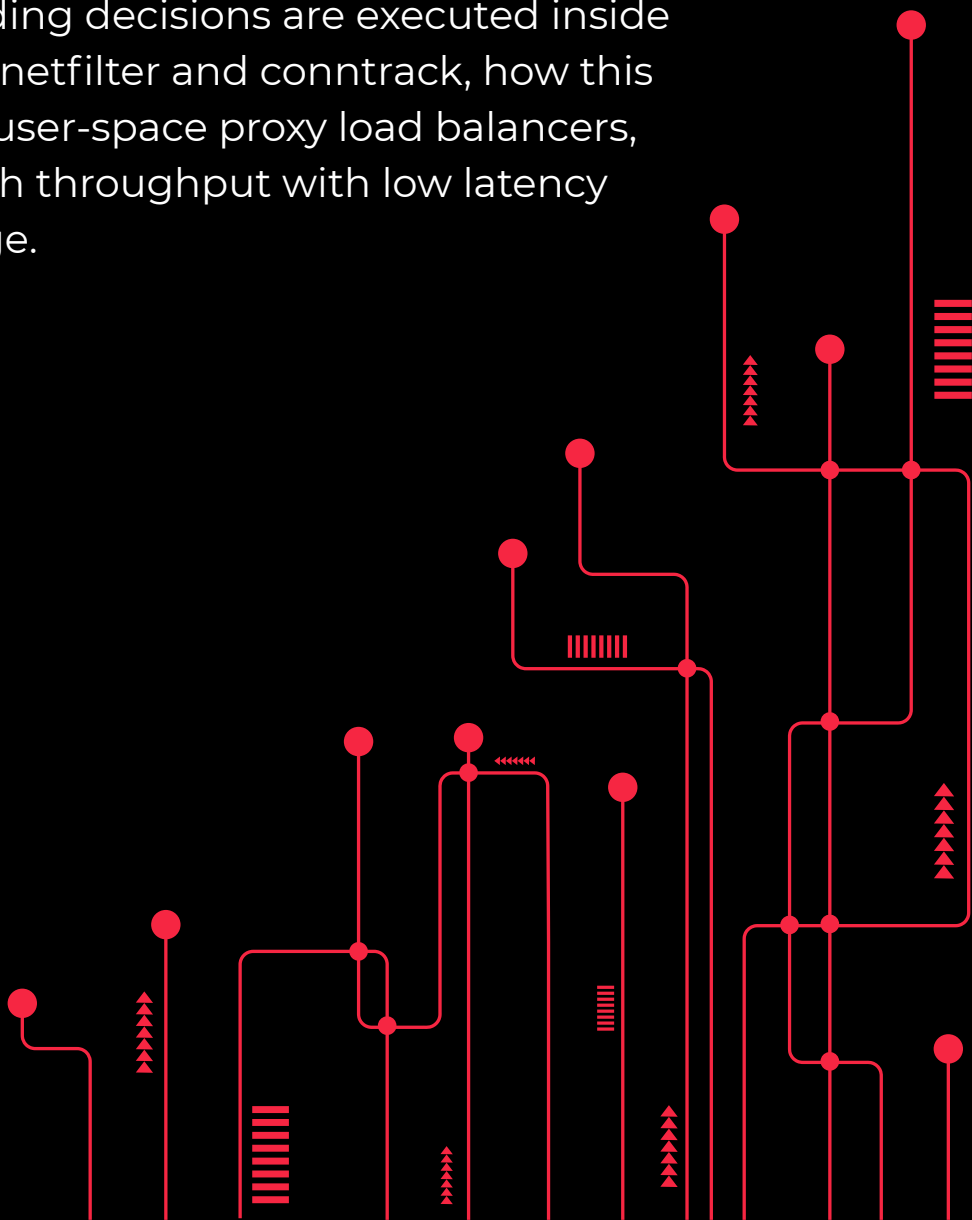
SKUDONET

SKUDONET DATA PLANE

TCP and UDP Load Balancing from the Kernel

This document describes the internal architecture and performance characteristics of SKUDONET's kernel-based data plane for TCP and UDP load balancing.

It explains how forwarding decisions are executed inside the Linux kernel using netfilter and conntrack, how this approach differs from user-space proxy load balancers, and how it enables high throughput with low latency and efficient CPU usage.



Introduction

When it comes to extreme performance, the distance between the kernel and user space is an invisible yet decisive boundary.

At that boundary, SKUDONET has built a bridge — an architecture that harnesses the raw power of the Linux kernel — the TCP/IP stack, netfilter, and conntrack — and exposes it through a clean, understandable, and consistent control path.

1. Kernel Space: Where the Magic Happens

In Linux, the **kernel** is the only place where **packets truly exist.**

Everything that happens in **user space** — sockets, proxies, processes — is an abstraction. Each transition between these two worlds involves memory copies, interrupts, scheduling, and the inevitable **"context switch"** that kills latency and consumes CPU cycles.

SKUDONET takes a direct approach: it moves traffic where it belongs — **to the kernel's data plane** — while keeping orchestration and control in user space.

The result is a system capable of balancing **millions of TCP and UDP flows per second**, with minimal CPU usage and no loss of visibility or governance.

How SKUDONET Tames the Kernel

At the heart of the system, the **SKUDONET core** works directly with **netfilter** structures — the Linux subsystem responsible for **routing, NAT, and packet filtering.**

Instead of re-implementing a load balancer in **user space,** SKUDONET injects rules and tables directly into the kernel, where every forwarding decision is made in microseconds, with no intermediate steps.

This philosophy achieves a perfect balance:

- The **data plane** resides in the **kernel**, where traffic flows at line rate.
- The **control plane** remains in **user space**, where rules are defined, managed, and dynamically applied.

2. TCP and UDP Load Balancing from the Kernel

TCP and UDP load balancing in the kernel is not just a matter of elegant design — it is proof that the right code, in the right place, can turn modest hardware into a precision instrument.

SKUDONET has put this into practice: layer 4 performance that is high, stable, and verifiable.

2.1 Test Environment

The tests were performed on an SNA hardware appliance running **SKUDONET Enterprise Edition 10, optimized for the SNA 7108 model.**

The environment consisted of two directly connected 1 GbE networks — one for services (172.16.1.0/24) and one for backends (172.16.2.0/24).

The operating system and kernel were standard, with netfilter, conntrack, and NAT active — no bypass, no DPDK, no fast paths outside the TCP/IP stack.

Component	Configuration
CPU	Intel Xeon E3-1245 v5 (4C / 8T, 3.5 GHz)
RAM	8 GB DDR4
NICs	2 × 1 GbE
Client Tool	wrk (10 threads, 360 connections)
Backend	nginx serving empty responses

2.2 L4xNAT Profile: Simplicity and Speed

The L4xNAT profile with SNAT is the core of transport-layer load balancing in SKUDONET.

Here, there are no user-space processes managing sockets or proxies relaying bytes: packets arrive, the kernel makes the decision, and traffic leaves — all within netfilter's domain.

During a 30-second sustained test (wrk → 172.16.1.1:80), the system achieved:

- **475,983 requests per second,** equivalent to over 3.33 GB read.
- **Average latency:** 1.63 ms.
- **Standard deviation:** 0.95 ms.
- **Average total CPU usage:** 27.7%.

In other words, the system handled nearly half a million HTTP connections per second on mid-range hardware, with a full kernel stack and real netfilter rules.

2.3 Performance Interpretation

In a traditional user-space load balancer — such as an L7 proxy — each request crosses the kernel↔user boundary multiple times: receive, read, write, and forward.

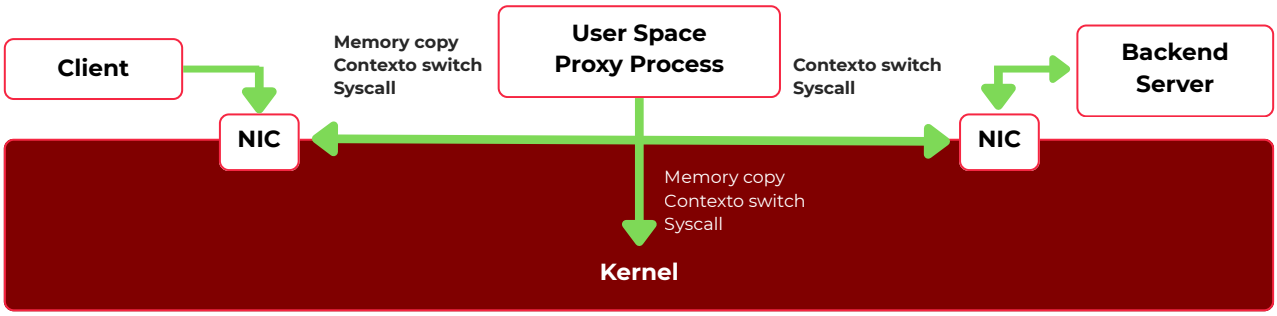


Figure 1. In user-space proxy architectures, traffic repeatedly transitions between kernel space and user space, increasing latency and CPU consumption due to memory copies and context switches.

Each transition involves memory copies, context switches, and queue waits — processes that increase latency and CPU consumption.

SKUDONET, on the other hand, delegates all of these operations to the kernel, where netfilter applies its rules directly to packets.

Meanwhile, the orchestration logic — the part visible to the operator — remains outside the data path, governing system behavior through a readable, consistent, and predictable control plane.

The results are measurable:

- **Latency remains stable** even with hundreds of thousands of active connections.
- **CPU usage scales smoothly**, with no abrupt spikes or IRQ saturation.
- The **TCP/IP stack remains intact**, allowing inspection, tracing, and security rules without performance loss.

2.4 UDP Load Balancing: Stateless Forwarding

Although the previous tests focus on **TCP**, the same **SKUDONET L4xNAT engine** extends to **stateless UDP**, where there is **no session or state** to maintain.

Lightweight protocols such as **Syslog** or **UDP metrics** benefit from direct kernel-level forwarding capable of handling **millions of datagrams per second per core,** with only fractions of a millisecond of additional latency.

Here, the limits are set by the **PCIe** bus or the **NIC** itself — not by the software.

To understand why these figures are possible, it is necessary to **compare the packet path in the kernel** with that of a traditional user-space load balancer.

3. TCP in the Kernel vs. User Space

Kernel Space (SKUDONET L4xNAT)

In SKUDONET, forwarding decisions take place entirely within the kernel's data plane. User space is only used for control and configuration, not for packet handling.

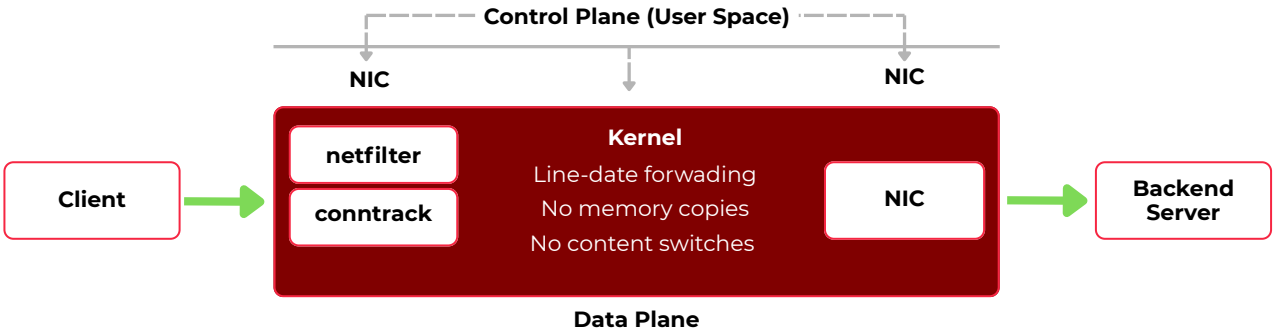


Figure 2. SKUDONET executes all forwarding logic inside the kernel (netfilter + conntrack), eliminating kernel↔user-space transitions and enabling low latency and efficient CPU utilization.

- The entire flow occurs in **kernel mode.**
- **0 buffer copies** between kernel and user space.
- **NAT, checksums, tracking**, and forwarding all happen within the same stack.
- Decisions made in **microseconds; minimal latency (10–20 µs).**
- Memory per connection: **~3–5 KB.**
- **CPU remains available** for the control path and auxiliary services.

In simple terms: the packet enters through the **NIC**, the **kernel** makes the decision, and it exits through the other interface — **never leaving privileged space.**

The control plane (the SKUDONET console) simply manages **netfilter rules — it never touches the traffic.**

User Space

- 2 kernel↔user transitions per packet (read/write).
- 2 memory copies (RX→user, user→TX).

- Each connection requires 2 TCP sockets (client and backend).
- Duplicated buffers (kernel and process).
- Minimum latency: 100–300 µs even when idle.
- Limited scalability: ~200–300k connections per core before saturation.

Parameter	Kernel (SKUDONET L4xNAT)	User Space
Memory copies per packet	0	2
Context switches (syscalls)	0	2
TCP buffers per connection	1	2
Average memory per connection	~3–5 KB	**~8–12 KB**
Typical latency	10–20 µs	100–300 µs
Concurrent connections per core	**> 1M**	200–300k
CPU per million connections	**~25–35%**	~70–90%

In summary, kernel space is an environment without intermediaries — each packet moves at the speed of copper.

User space offers flexibility but pays the cost of each copy and each syscall.

In SKUDONET, the L4xNAT core takes the shortest possible path within the kernel: it doesn't parse headers or protocols — it simply decides and forwards.

That's the difference between a load balancer that processes traffic and one that lets it flow.

4. Load Balancing Modes in the Kernel

SNAT — Source Network Address Translation

Flow: Client → Load Balancer

The client sends packets to the VIP. SKUDONET intercepts them in PREROUTING and creates an entry in conntrack.

Before forwarding, the kernel rewrites the source IP.

The backend responds to the balancer, which reverses the translation.

Technical Impact:

- Fully bidirectional flow inside the kernel.
- CPU and memory: ≈ 3–5 KB per connection.
- Full stability even in complex NAT environments.

Ideal for production environments requiring control, logging, and security.

DNAT — Destination Network Address Translation

Flow: The balancer receives the packet addressed to the VIP.

It rewrites the destination IP to the real backend (POSTROUTING).

The backend sees the client's real IP.

Technical Impact:

- Less header manipulation than SNAT.
- Preserves client IP (useful for ACLs and tracing).

- Requires controlled routing.
- Excellent balance between transparency and performance.

DSR — Direct Server Return

Flow: The balancer forwards the packet at Layer 2 (MAC rewrite or IPIP/GRE encapsulation).

The backend has the VIP configured on its loopback interface (arp_ignore=1, arp_announce=2).

It responds directly to the client, bypassing the balancer on the return path.

Technical Impact:

- Eliminates the return path through the balancer.
- Minimal latency (<0.5 ms).
- Massive CPU savings.
- Ideal for read-heavy, UDP, or CDN traffic.

Mode	Translation	Response Path	Client IP Visible	Typical Latency	Complexity	Best Use Case
SNAT	Source IP	Through LB	No	1–2 ms	Low	NAT deployments
DNAT	Destination IP	Through LB (optional)	Yes	0.8–1.5 ms	Medium	Internal routed farms
DSR	None (L2)	Direct to client	Yes	<0.5 ms	High	Read-heavy / CDN / UDP

5. The Control Plane: Where the Kernel Becomes Understandable

Working with the kernel is not simple. The **netfilter APIs, hook chains**, and **priority levels** form a complex environment filled with flags, tables, and structures.

SKUDONET simplifies this environment through its **control path,** translating kernel configuration into a coherent, documented, and secure syntax. What for a kernel engineer would be a collection of **nftables rules** or a chain tree, becomes — for the operator — a set of clear actions: create a service, add a backend, adjust weights, and let the core kernel execute.

Every byte that ascends to **user space** consumes CPU cycles, and every connection handled outside the kernel adds latency. That is why SKUDONET chooses to **work with the kernel, not against it**.

The result is a load balancer capable of handling **millions of TCP and UDP connections per second**, maintaining **consistency, performance**, and **operational transparency** — all without auxiliary processes, heavy libraries, or interpreter layers, relying solely on the **Linux kernel stack.**

This integration between the control plane and the kernel defines SKUDONET's architecture: it doesn't hide complexity but transforms it into a **predictable, stable, and manageable tool.**

Conclusion

SKUDONET does not seek to reinvent the kernel, but rather to **harness its potential** and make it operationally accessible. Its architecture allows traffic to be processed **at line rate**, provides operators with **clear visibility** into the flow, and ensures **stable, predictable system behavior.**

This philosophy defines its design: applying **kernel engineering** in a practical, controlled manner — without adding unnecessary complexity.

Kernel-level load balancing is not a theory or an experimental demonstration; it is **the foundation** on which SKUDONET builds its performance. By transforming the complexity of **netfilter** into an efficient and predictable architecture, the system proves that **performance and control can coexist.** The results speak for themselves: **measurable performance, full visibility**, and a network layer optimized to operate with **precision and consistency.**

Further Technical Resources

Documentation, configuration guides, and deployment examples are available in our → Knowledge Base.

Technical Inquiry

If you would like to discuss specific performance requirements or architectural considerations, our engineering team can provide technical guidance → info@skudonet.com