Bachelorthesis

Evaluating Dual-Stack against NAT64 deployment schemes with DNS64 and CLAT

Wodke, Daniel Jin

21. Mai 2025

Gutachter:	Prof. Dr. Stefan Schmid
	Prof. Dr. Stefan Tai
Betreuerin:	Max Franke and Dr. Philipp Tiesel
Matrikelnr.:	456675

Technische Universität Berlin
Faklutät IV - Elektrotechnik und Informatik
Institut für Telekommunikationssysteme
Fachgebiet Intelligent Networks

Eidesstattliche Versicherung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den Date

____

Wodke, Daniel Jin

Zusammenfassung

Weit hinten, hinter den Wortbergen, fern der Länder Vokalien und Konsonantien leben die Blindtexte. Abgeschieden wohnen Sie in Buchstabhausen an der Küste des Semantik, eines großen Sprachozeans. Ein kleines Bächlein namens Duden fließt durch ihren Ort und versorgt sie mit den nötigen Regelialien. Es ist ein paradiesmatisches Land, in dem einem gebratene Satzteile in den Mund fliegen. Nicht einmal von der allmächtigen Interpunktion werden die Blindtexte beherrscht – ein geradezu unorthographisches Leben. Eines Tages aber beschloß eine kleine Zeile Blindtext, ihr Name war Lorem Ipsum, hinaus zu gehen in die weite Grammatik. Der große Oxmox riet ihr davon ab, da es dort wimmele von bösen Kommata, wilden Fragezeichen und hinterhältigen Semikoli, doch das Blindtextchen ließ sich nicht beirren. Es packte seine sieben Versalien, schob sich sein Initial in den Gürtel und machte sich auf den Weg. Als es die ersten Hügel des Kursivgebirges erklommen hatte, warf es einen letzten Blick zurück auf die Skyline seiner Heimatstadt Buchstabhausen, die Headline von Alphabetdorf und die Subline seiner eigenen Straße, der Zeilengasse. Wehmütig lief ihm eine rethorische Frage über die Wange, dann setzte es seinen Weg fort. Unterwegs traf es eine Copy. Die Copy warnte das Blindtextchen, da, wo sie herkäme wäre sie zigmal umgeschrieben worden und alles, was von ihrem Ursprung noch übrig wäre, sei das Wort "und" und das Blindtextchen solle umkehren und wieder in sein eigenes, sicheres Land zurückkehren.

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Contents

Chapter 1

Introduction

1.1 Motivation

The transition from IPv4 to IPv6 has been talked about for years, but it's finally becoming a real operational headache for companies running Kubernetes clusters. During my internship at SAP, I witnessed firsthand how teams struggle with this problem - IPv4 addresses are getting expensive (we're talking about 50 dollars per address in AWS), but you can't just flip a switch to IPv6 when half your enterprise customers still run legacy IPv4-only systems. What makes this particularly interesting is that Kubernetes adds another layer of complexity. Unlike traditional networks where you might have a few hundred servers, a single Kubernetes cluster can spawn thousands of pods dynamically, each potentially needing its own IP address. The standard approaches - either running both protocols simultaneously (Dual-Stack) or using translation mechanisms (NAT64/DNS64) - each come with their own problems that aren't well-documented in cloud-native environments. The real kicker is that most existing research treats this as a pure networking problem, ignoring how Kubernetes' orchestration model completely changes the game. When pods are ephemeral and can scale from 10 to 1000 instances in minutes, traditional IPv6 transition strategies start breaking down in ways nobody really talks about in the literature.

1.2 Problem Statement

The core problem is that Kubernetes operators are stuck between a rock and a hard place. They need to adopt IPv6 to avoid spiraling IPv4 costs and meet compliance requirements, but they also need to maintain compatibility with legacy systems that will probably outlive us all. The two main approaches available today force painful trade-offs that nobody has properly quantified in Kubernetes contexts.

Running Dual-Stack means managing two parallel network configurations - double the routing tables, double the firewall rules, and double the chances something goes wrong at 3 AM. I've seen deployments where a simple typo in an IPv6 CIDR range took down half a cluster because the debugging tools everyone knows are still IPv4-centric. On the flip side, NAT64/DNS64 promises a cleaner IPv6-only infrastructure, but it introduces translation overhead that could be catastrophic when you're dealing with latency-sensitive microservices making thousands of API calls per second. What's worse is that CLAT (Customer-side transLATor) deployments in Kubernetes are basically uncharted territory. Should you run one translator per node? Per pod? As a sidecar container? Nobody knows because nobody has actually tested this at scale. The few blog posts that exist are either vendor marketing or "it worked on my laptop" tutorials that fall apart under real workloads.

1.3 Objectives and Scope

This thesis aims to cut through the confusion by systematically comparing these approaches in real Kubernetes environments. I'm not interested in theoretical models or synthetic benchmarks - I want to know what actually happens when you deploy these solutions in production-like scenarios. Specifically, I'll evaluate three deployment strategies: pure Dual-Stack as our baseline, NAT64/DNS64 with per-node CLAT, and NAT64/DNS64 with per-container CLAT. For each approach, I'll measure the stuff that actually matters to operators: How much latency does translation add? At what point does the system fall over? How much CPU and memory overhead are we talking about? And perhaps most importantly - what breaks when things go wrong? The scope is deliberately focused on managed Kubernetes environments, particularly SAP Gardener, because that's where the rubber meets the road. I'm not trying to solve IPv6 transition for the entire internet - just trying to give Kubernetes operators actionable data to make informed decisions. This means testing with realistic workloads (think microservices, not ping floods), actual CNI plugins people use (Calico, Cilium), and failure scenarios that keep SREs up at night.

1.4 Cooperation with SAP SE

This thesis is being conducted in cooperation with SAP SE's Gardener team, who are dealing with these exact challenges across their managed Kubernetes offering. SAP provides not just the infrastructure for testing but also real-world use cases from their enterprise customers who are demanding IPv6 support while simultaneously running 20-year-old ERP systems that barely understand what IPv6 is. Working with SAP gives me access to production-grade Gardener clusters where I can test at scales impossible in a university lab. We're talking about clusters with hundreds of nodes, thousands of pods, and actual traffic patterns from enterprise workloads. The Gardener team has also been invaluable in identifying edge cases - like what happens when your NAT64 gateway needs to handle SAP's infamous "month-end processing" traffic spikes. The collaboration ensures that this research isn't just academically interesting but actually useful for operators. SAP has committed to integrating successful findings into Gardener's documentation and potentially its default configurations, which could impact thousands of Kubernetes clusters worldwide. Plus, having access to their engineering team means I can validate my results against their production experiences - if my tests show NAT64 adds 50ms of latency but they're seeing 5ms in production, I know something's wrong with my setup.

Chapter 2

Background

2.1 Overview of IPv4 Exhaustion and IPv6 Adoption

The exhaustion of IPv4 address space represents a fundamental constraint in modern network infrastructure deployment. The Internet Assigned Numbers Authority (IANA) allocated the final five /8 IPv4 address blocks to Regional Internet Registries (RIRs) in February 2011, marking the beginning of IPv4 scarcity. With approximately 4.3 billion available addresses in the 32-bit IPv4 space, the protocol cannot accommodate the exponential growth of internet-connected devices, particularly in cloud computing environments where elastic scaling demands dynamic address allocation. The economic implications of IPv4 exhaustion have created a secondary market for address blocks. Current market rates for IPv4 addresses range from \$35 to \$60 per address, with cloud providers implementing hourly billing models. Amazon Web Services charges \$0.005 per hour for elastic IP addresses, resulting in approximately \$43.80 per address annually. For organizations operating large-scale Kubernetes deployments with hundreds or thousands of public endpoints, these costs represent significant operational expenditure. Furthermore, IPv4 address leasing introduces administrative overhead through broker negotiations, transfer procedures adhering to RIR policies, and the management of increasingly fragmented address blocks. IPv6, utilizing 128-bit addresses, provides approximately 3.4×10^{38} unique addresses, effectively eliminating scarcity concerns. The protocol incorporates improvements over IPv4, including simplified header structures, mandatory IPsec support, and elimination of broadcast traffic through multicast and anycast addressing. However, global IPv6 adoption remains heterogeneous. According to Google's IPv6 statistics, worldwide adoption reached 45% in 2024, with significant regional variations: Belgium reports 70% adoption, Germany 55%, while other regions remain below 10%. Enterprise adoption lags behind consumer networks, primarily due to legacy application dependencies, training requirements, and the capital investment needed for infrastructure upgrades. The prolonged coexistence of IPv4 and IPv6 necessitates robust transition mechanisms. Industry analyses project this dual-protocol period will extend through 2035, requiring organizations to maintain interoperability between protocol versions. This creates particular challenges in containerized environments where dynamic orchestration, service discovery, and east-west traffic patterns amplify the complexity of multi-protocol networking.

2.2 IPv6 Transition Mechanisms

2.2.1 Dual-Stack Architecture

Dual-Stack architecture implements simultaneous IPv4 and IPv6 protocol stacks on network devices, allowing native communication in both protocols without translation overhead. RFC 4213 defines the basic transition mechanisms, establishing Dual-Stack as the IETF's preferred coexistence strategy. In this configuration, network interfaces possess both IPv4 and IPv6 addresses, applications can initiate connections using either protocol, and DNS returns both A and AAAA records for dual-stacked hosts. Implementation in Kubernetes environments requires comprehensive dual-protocol support across multiple networking layers. The Container Network Interface (CNI) must allocate and manage addresses from both IPv4 and IPv6 pools, typically requiring separate Pod CIDR ranges (e.g., 10.0.0.0/16 for IPv4 and fd00::/64 for IPv6). Service discovery mechanisms must handle dual addressing, with kube-dns or CoreDNS returning appropriate records based on client capabilities. The kube-proxy component maintains separate iptables or IPVS rules for each protocol, effectively doubling rule complexity. Kubernetes formally introduced Dual-Stack support as an alpha feature in version 1.16, promoting it to general availability in version 1.23. The implementation allows three configuration modes: IPv4-only, IPv6-only, and dual-stack. When enabled, pods receive addresses from both protocol families, services can expose IPv4, IPv6, or both endpoints, and ingress controllers must handle protocol selection based on client connectivity. Critical considerations include:

Address Assignment: Pods require addresses from both IPv4 and IPv6 CIDR ranges, potentially exhausting smaller IPv4 allocations in large clusters Service Exposure: LoadBalancer services may incur double costs when provisioning both IPv4 and IPv6 external addresses Network Policies: Security policies must account for both protocols, increasing configuration complexity and potential for misconfiguration Routing Tables: Nodes maintain separate routing tables and neighbor caches for each protocol, increasing memory consumption

Operational challenges emerge from managing parallel network configurations. Debugging tools and procedures must account for both protocols, monitoring systems require dual-protocol support, and troubleshooting often involves correlating issues across protocol boundaries. Performance implications include increased memory usage for dual routing tables, potential asymmetric routing when protocols traverse different paths, and the overhead of maintaining two sets of connection tracking states.

2.2.2 NAT64/DNS64 Principles

NAT64 (Network Address Translation IPv6-to-IPv4) and DNS64 (DNS Extensions for Network Address Translation) enable IPv6-only clients to communicate with IPv4-only servers through protocol translation. RFC 6146 specifies stateful NAT64 operation, while RFC 6147 defines DNS64 behavior. This mechanism allows organizations to deploy IPv6-only internal networks while maintaining connectivity to IPv4 internet resources. DNS64 operates by synthesizing AAAA records for hosts that only possess A records. When an IPv6-only client queries for a domain, the DNS64 server first attempts to retrieve native AAAA records. If none exist but A records are available, DNS64 constructs synthetic AAAA records by embedding the IPv4 address within a designated IPv6 prefix. The Well-Known Prefix (WKP) 64:ff9b::/96, defined in RFC 6052, serves as the default, though operators can configure Network-Specific Prefixes (NSP) for additional flexibility. For example, an IPv4 address 192.0.2.33 becomes 64:ff9b::192.0.2.33 (or 64:ff9b::c000:221 in hexadecimal notation). NAT64 performs bidirectional translation between IPv6 and IPv4 packets. The translation process involves several complex operations:

Header Translation: IPv6 headers (40 bytes) must be converted to IPv4 headers (20 bytes minimum), requiring field mapping and potential fragmentation handling Address Translation: The NAT64 gateway maintains stateful mappings between IPv6 source addresses and dynamically allocated IPv4 addresses/ports Protocol Adjustments: Differences in ICMP implementations, path MTU discovery mechanisms, and fragmentation strategies require careful handling Application Layer Gateway (ALG): Protocols embedding IP addresses in payloads (FTP, SIP) require deep packet inspection and modification

Stateful NAT64 implementations maintain session tables tracking active connections. Each table entry contains the IPv6 source address/port, translated IPv4 address/port, destination IPv4 address/port, and protocol-specific state information. Table exhaustion becomes critical in high-connection-rate environments typical of microservices architectures. Modern implementations like Jool support millions of concurrent sessions, but table lookups introduce latency, particularly under high load. Performance characteristics of NAT64/DNS64 deployments depend on multiple factors. Translation latency typically ranges from 0.1 to 5 milliseconds per packet, depending on table size and hardware acceleration. Throughput limitations arise from CPU-bound translation operations, with software implementations achieving 1-10 Gbps on commodity hardware. Memory requirements scale with connection count, requiring approximately 300 bytes per active session.

2.2.3 464XLAT: Combining NAT64 and CLAT

464XLAT (RFC 6877) extends NAT64/DNS64 by adding Customer-side transLATor (CLAT) functionality, enabling IPv4-only applications to operate in IPv6-only networks. This mechanism addresses NAT64's inability to handle IPv4 literal addresses and protocols that embed IPv4 addresses in their payloads. The architecture implements double translation: CLAT translates IPv4 to IPv6 at the client side, packets traverse the IPv6-only network, and Provider-side transLATor (PLAT, functionally equivalent to NAT64) translates back to IPv4 at the network border. CLAT provides a virtual IPv4 interface to applications, typically using the 192.0.0.0/29 prefix defined in RFC 7335. When applications send IPv4 packets to this interface, CLAT performs stateless translation to IPv6 using either the Well-Known Prefix or a Network-Specific Prefix. The translation follows RFC 6145 (IP/ICMP Translation Algorithm), mapping IPv4 headers to IPv6 equivalents while preserving transport-layer information. Critical aspects include:

Addressing Architecture: CLAT must coordinate prefix usage with PLAT to avoid translation loops MTU Handling: The 20-byte header size difference between IPv4 and IPv6 requires careful MSS clamping Fragmentation: IPv6's lack of in-transit fragmentation necessitates fragment handling at CLAT boundaries

In Kubernetes environments, CLAT deployment strategies significantly impact performance and operational complexity: Per-Node CLAT Deployment: Implementing CLAT as a DaemonSet provides one translator instance per node. This approach minimizes resource overhead and simplifies management but requires careful network namespace configuration to ensure pod traffic routes through the node-local CLAT. Implementation typically involves:

Configuring a virtual network interface (e.g., clat0) on each node Establishing IPv4 routes directing pod traffic to the CLAT interface Managing translation state sharing among pods on the same node

Per-Pod CLAT Deployment: Injecting CLAT as a sidecar container provides dedicated translation resources per pod. This strategy offers superior isolation and allows per-application translation policies but increases resource consumption. Implementation considerations include:

Init container configuration to establish networking before application startup Shared network namespace between CLAT and application containers Resource limits to prevent CLAT from consuming excessive CPU during translation

In-Container CLAT: Embedding CLAT directly within application containers provides the finest granularity but requires application modification or base image changes. This approach suits environments with strict isolation requirements or specialized translation needs. Performance implications vary significantly across deployment models. Per-node CLAT exhibits lower latency (typically 0.05-0.2ms additional RTT) due to kernel-level processing but may become a bottleneck under high pod density. Per-pod CLAT increases resource consumption by approximately 10-20MB RAM and 0.01-0.05 CPU cores per instance but provides predictable performance isolation. Translation throughput depends on implementation efficiency, with userspace CLAT achieving 100-500 Mbps and kernel implementations reaching 1-10 Gbps.

2.3 Kubernetes Networking

2.3.1 Networking Model

Kubernetes implements a flat networking model where every pod receives a unique IP address accessible from any other pod without Network Address Translation. This model, specified in the Kubernetes networking requirements, establishes three fundamental principles: pods can communicate with other pods without NAT, nodes can communicate with pods without NAT, and the IP address a pod sees itself as equals the IP address others use to reach it. The cluster network architecture comprises multiple interconnected components. The Pod network assigns IP addresses from a configured CIDR range (cluster-pod-cidr), typically using private RFC 1918 addresses for IPv4 or Unique Local Addresses (ULA) for IPv6. The Service network allocates virtual IPs from a separate CIDR range (service-cluster-ip-range), providing stable endpoints for pod groups. These virtual IPs exist only within iptables or IPVS rules, never appearing on physical interfaces. Service discovery and load balancing rely on kube-proxy, which programs packet forwarding rules based on Service and Endpoint resources. Three proxy modes offer different performance characteristics:

Userspace mode: Original implementation routing packets through kube-proxy process (deprecated due to performance limitations) iptables mode: Default configuration using netfilter rules for distributed packet forwarding IPVS mode: High-performance option utilizing Linux Virtual Server for connection scheduling

IPv6 integration introduces additional complexity to this model. Address allocation requires significantly larger CIDR ranges, with /64 being the minimum recommended prefix size per RFC 4291. Neighbor Discovery Protocol (NDP) replaces ARP for address resolution, requiring different proxy mechanisms. Router Advertisements (RA) may interfere with pod networking if not properly filtered.

Service discovery must handle both A and AAAA DNS records, with appropriate fallback mechanisms. Dual-Stack networking in Kubernetes extends the model to support simultaneous IPv4 and IPv6 operation. Pods receive addresses from both protocol families, services can expose single or dual-stack endpoints, and endpoints selection follows client protocol preference. Implementation requires:

Dual CIDR configuration for both pod and service networks CNI plugins capable of managing multiple address families Extended API objects supporting multiple IP fields Protocol-aware health checking and readiness probes

2.3.2 Container Network Interface (CNI) Plugins

Container Network Interface plugins implement the actual networking dataplane for Kubernetes clusters. The CNI specification defines how container runtimes interact with networking providers, enabling pluggable network implementations. IPv6 and Dual-Stack support varies significantly across CNI implementations, directly impacting transition mechanism viability. Calico implements a pure Layer 3 networking approach using BGP for route distribution. The architecture assigns /32 (IPv4) or /128 (IPv6) routes per pod, advertising them via BGP to ensure reachability. Key IPv6 capabilities include:

Full Dual-Stack support with separate IP pools for each protocol IPv6-only networking with NAT64 gateway integration Network policy enforcement for both protocols using iptables or eBPF BGP peering over IPv6 for route advertisement Support for IPv6 Router Advertisement daemon for SLAAC

Calico's implementation handles IPv6 through its IPAM (IP Address Management) system, allocating addresses from configured pools. The Felix agent programs forwarding rules, while BIRD handles BGP sessions. Performance characteristics show minimal overhead for IPv6, with routing table size being the primary scaling concern. Cilium leverages eBPF (extended Berkeley Packet Filter) for high-performance packet processing directly in the kernel. The architecture implements networking, load balancing, and security policies through eBPF programs. IPv6 support includes:

Native Dual-Stack operation with efficient packet handling Built-in NAT64 gateway functionality implemented in eBPF IPv6-aware network policies with CIDR matching Segment Routing v6 (SRv6) for advanced traffic engineering Transparent encryption using IPsec or WireGuard

Cilium's eBPF-based NAT64 implementation deserves particular attention for this research. The translation occurs entirely in kernel space, minimizing latency and CPU overhead. Performance testing shows sub-millisecond translation latency and line-rate throughput on modern hardware. However, eBPF program complexity limits advanced ALG functionality. Flannel provides simple overlay networking, primarily designed for ease of use. IPv6 support remains limited:

Basic IPv6 addressing through host-local IPAM VXLAN overlay supporting IPv6 encapsulation No native Dual-Stack support (requires manual configuration) Limited network policy capabilities

Weave Net creates a mesh overlay network with automatic discovery. IPv6 functionality includes:

Dual-Stack support with automatic address allocation IPv6 multicast for peer discovery Encryption support for IPv6 traffic Basic network policy implementation

Performance comparisons reveal significant variations. Calico demonstrates superior scalability for large clusters (>1000 nodes) but requires BGP expertise. Cilium achieves the lowest latency and highest throughput but demands kernel version compatibility. Flannel offers simplest deployment but lacks advanced features necessary for production IPv6 deployments.

2.4 SAP Gardener

SAP Gardener provides a Kubernetes-as-a-Service platform implementing hierarchical cluster management. The architecture utilizes Kubernetes principles to manage Kubernetes clusters, enabling multi-cloud deployments with consistent operational patterns. Understanding Gardener's networking architecture is essential for evaluating IPv6 transition mechanisms in production environments. Gardener's architecture comprises three cluster types with distinct roles. The Garden cluster hosts the Gardener control plane, managing the lifecycle of all managed clusters. Seed clusters run control plane components for managed clusters, providing isolation and scalability. Shoot clusters represent the actual workload clusters where applications run. This hierarchy enables efficient resource utilization while maintaining security boundaries. Network architecture in Gardener implements multiple isolation layers. Each shoot cluster receives dedicated network ranges for pods, services, and nodes. The Gardener networking extensions framework allows pluggable CNI implementations, with Calico as the default. Network isolation between shoots on the same seed uses either network namespaces or overlay networks depending on the infrastructure provider. IPv6 support in Gardener has evolved significantly. Current capabilities include:

Dual-Stack shoot clusters on supported infrastructure providers (AWS, Azure, GCP) IPv6-only shoots with NAT64 gateway provisioning Configurable IP families for different network components Integration with cloud provider IPv6 features (e.g., AWS Egress-Only Internet Gateway)

Chapter 3

Related Work

3.1 Existing Studies on IPv6 Transition

3.2 Identified Research Gap

Chapter 4

Implementation

4.1 Setup

4.2 AWS

4.3 Single Ubuntu Local

4.4 Double Ubuntu Local

4.5 Test Workloads and Traffic Patterns

4.6 Implementation Challenges and Solutions

Chapter 5

Results

5.1 Performance Metrics

Chapter 6

Evaluation

6.1 RTT

6.2 Throughput

6.3 TSC vs HPET Clocksource Difference

Chapter 7

Conclusion

7.1 Summary of Key findings

7.2 Limitations and Future Work

Chapter 8

References

Appendix

List of Figures

List of Listings