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

The transition from IPv4 to IPv6 has progressed from a long-term goal to a practical necessity. Global IPv4 exhaustion and the growing cost of public IPv4 create pressure to reduce IPv4 dependency while maintaining reachability to IPv4-only services[**7737362**, **LEVIN20141059**]. Dual-Stack, where both IPv4 and IPv6 run in parallel on the same devices and networks, remains the most widely deployed model because it keeps native behavior for both protocols, but it does not reduce demand for IPv4 addresses and increases operational surface area[**rfc4213**]. Translation-based approaches convert traffic between IPv6 and IPv4. A common variant is NAT64 (translation at the network layer), often paired with DNS64 (synthesizing IPv6 addresses from IPv4 DNS records) or with client-side translation via CLAT, as in 464XLAT (extending NAT64 functionality to end devices). These methods offer an IPv6-first architecture while still enabling access to IPv4-only endpoints and have seen broad adoption in mobile networks[**rfc6146**, **rfc6147**, **rfc6877**]. In enterprise and cloud contexts, however, the performance trade-offs between Dual-Stack and 464XLAT remain less well quantified.

Problem Statement Motivated by this setting, the thesis investigates how much latency a CLAT-based path introduces and what performance impacts it has. The study focuses on measuring throughput and round-trip time as fundamental metrics under realistic software implementations and topologies. Specifically, it compares a Dual-Stack baseline with three Linux translators that reflect different implementation strategies: Jool (version 4.1.7), a kernel-space translator supporting NAT64 and SIIT[**jool_introduction**], Tayga, a stateless user-space NAT64/NAT44[**palrd_tayga_readme**, **Repas_Farnadi_Lencse_2014**] and Tundra, a multi-threaded user-space SIIT/NAT64/CLAT implementation[**labuda_tundra_nat64**]. Measurements were conducted with iperf (version 3.16) and ping (version iputils 20240117) across three environments: an AWS deployment, a single-host Ubuntu setup with translators in isolated Linux network namespaces, and a two-host

Ubuntu setup connected via Ethernet (specific machine configurations can be found in the Appendix). By limiting the scope to TCP throughput and ICMP RTT, the experiments provide a reproducible view on translation overhead relative to native Dual-Stack, while leaving topics such as tail latency, CPU cost, DNS performance, and application-level behaviors to future work.

The thesis tries to quantify if the measured overhead of CLAT relative to Dual-Stack is acceptable, IPv6-only access with translation becomes an attractive option where IPv4 addresses are scarce or expensive. If not, Dual-Stack remains the safer choice. The following background chapter summarizes the transition mechanisms and Linux implementations that support the experimental design[**rfc6877**, **rfc4241**].

Industry collaboration This thesis was conducted in close cooperation with SAP SE and was supervised from the industry side by Dr.Philip Tiesel. This collaboration ensured that the research questions addressed are directly aligned with the real-world challenges faced by large-scale platform engineering teams. The primary motivation comes from the architectural decisions required by projects like SAP Gardener, an open-source initiative for managing Kubernetes clusters at scale[**gardener_docs**]. Within SAP's Gardeners ecosystem, operators decide between Dual-Stack clusters and IPv6-only clusters. The choice has implications for cost, operational complexity, and performance.

Thesis Structure Following the introduction, which frames the problem and outlines the industry motivation, Chapter 2 provides the technical background required to interpret the results. It revisits IPv4 exhaustion and the transition to IPv6, describes IPv6 transition mechanisms and introduces the principles of client-side translation. The chapter closes with a description of the Linux components used in the experiments: Jool, Tayga and Tundra. Chapter 3 presents related work and identifies the specific gap this study addresses. Chapter 4 then explains the experiment design by introducing the three test environments: an AWS cloud setup, a single Ubuntu host with translators running in separate network namespaces, and a dual-host Ubuntu setup connected via Ethernet with the iperf server on the second machine. Documenting how Tayga, Tundra, and Jool are set up, how namespaces and routing are configured, how measurements are taken for TCP throughput and RTT and discussing practical challenges encountered during setup. Chapter 5 presents the evaluation, analyzing the translators against the Dual-Stack baseline and synthesizing findings across environments. Finally, Chapter 6 concludes the thesis by summarizing the main findings with respect to the central question and suggesting directions for future work, while the document ends with references and an appendix containing configuration files, scripts and further plots to support reproducibility.

Chapter 2

Background

To understand the choices we're making for the internet today, we have to look back at a long-standing problem: we've run out of the original internet addresses (IPv4). This has forced us into a decades-long transition to a new system (IPv6), where parts of the internet will use only the new addresses. The core challenge is well-known: the old and new systems don't speak the same language, yet both have to work together on a global scale. At the same time, we're trying to balance running out of old addresses, increasing costs, and the constant risk of things breaking[**7737362**, **LEVIN20141059**].

2.1 IPv4 exhaustion and transition to IPv6

Internet use has expanded to critical infrastructure across consumer, enterprise, and public sectors. Services running at any time of the day and the increase of connected devices have driven steady growth in traffic and endpoints, making address management a central concern[**7737362**, **LEVIN20141059**]. IP addresses perform two fundamental roles: identifying endpoints and enabling packet delivery across networks, and uniqueness at global scale is essential[**LEVIN20141059**].

IPv4, standardized in 1981-1983, provides a 32-bit address space of roughly 4.3 billion addresses[**rfc791**].In practice, not all addresses are usable on the public Internet due to special-use and private allocations, and early design choices further reduced the effectively usable pool[**rfc1918**, **7737362**, **LEVIN20141059**]. Management techniques such as CIDR and NAT slowed the pace of consumption but could not eliminate the finite limit[**7737362**].

Exhaustion means that the pool of unused IPv4 addresses has run out, not that IPv4 connections themselves have stopped working. The process started at the global level and then moved downward: IANA allocated its final IPv4 blocks on 3 February 2011, after that, each Regional Internet Registry (RIR) moved into its final phase: APNIC in April 2011, RIPE NCC in September 2012, LACNIC in June 2014 while AFRINIC held on the longest[**LEVIN20141059**]. A global policy passed on 6 May 2012 established mechanisms for the recovery and redistribution of returned IPv4 address space, yet scarcity has continued to persist[**7737362**]. The impact has been uneven across regions because historical allocations left some operators with far fewer addresses per user than others[**LEVIN20141059**].

The shortage was unavoidable because demand kept rising: by 2014, about 2.9 billion people were online, with more than 200 million new users joining each year after 2010. Internet use jumped from less than 1% of the world's population in 1994 to over 40% by 2014, and surpassing 4 billion by 2020 and 5.5 billion in 2024[**7737362**, **itu_d_statistics**]. In short, exponential demand confronted a finite IPv4 pool.

IPv6 was standardized in 1995 as the long-term successor, expanding addressing to 128 bits—on the order of 3.4×10^{38} unique addresses—and introducing protocol-level improvements aimed at routing scalability, mobility support, and operational security[**rfc1883**, **7737362**, **LEVIN20141059**]. The allocation data highlights how abundant IPv6 is compared to IPv4, even when large IPv6 distributions are considered[**7737362**]. However, IPv4 and IPv6 don't work together on their own, so bridging mechanisms are needed while both remain in use[**LEVIN20141059**].

Even though the technical benefits of IPv6 were clear, its adoption lagged behind the urgency created by IPv4 shortages. IPv6 allocations reported by the RIRs lagged behind user growth, with especially low adoption in some regions, like Africa. At the same time, user-side measurements, such as those from Google, showed less than 10% IPv6 usage, though the trend was gradually increasing[**7737362**]. Operators had to cover costs for enabling IPv6, including new equipment and configuration, which extended the period of running both protocols, often well beyond 2020 and in some networks possibly past 2030[**7737362**].

Operators and policymakers have followed two main approaches: allocating the remaining IPv4 space more efficiently and transitioning to IPv6[**LEVIN20141059**]. On the technical side, NAT, especially carrier-grade NAT (CGN), conserves public IPv4 addresses by multiplexing many private hosts behind a smaller set of public addresses[**rfc2663**]. By around 2014, a measurable fraction of users were estimated to traverse CGN[**livadariu2018inferring**]. While NAT is effective at saving IP addresses, it can make end-to-end connectivity more complicated and add to operational complexity[**rfc2993**]. Dual stack (running IPv4 and IPv6 in parallel) is broadly available and will remain common for years, but it does not address the IPv4 address exhaustion[**LEVIN20141059**]. Policy measures like smaller allocations and efforts to reclaim unused addresses have helped ease IP address scarcity somewhat, but they don't eliminate the need for IPv6[**LEVIN20141059**].

Strategically this means that IPv4 and IPv6 will continue to coexist for a long time, connected through integration methods[**7737362**, **LEVIN20141059**]. This situation drives the focus of this thesis: as network operators decide whether to keep dual-stack setups or move to IPv6-only access with translation technologies (like NAT64/DNS64, often paired with CLAT on the client side), it's important to understand the performance trade-offs involved. The next section takes a look at the main transition mechanisms that form the basis of this comparison[**7737362**, **LEVIN20141059**].

2.2 IPv6 transition mechanisms

According to the IETF's classification of transition strategies, there are three main approaches: dual stack, tunneling, and translation[**rfc2893**]. Because this thesis focuses on comparing dual stack and translation, tunneling will not be covered.

Dual-Stack Architecture Dual stack allows networks and hosts to gradually migrate, letting IPv4 and IPv6 run side by side[**punithavathani2009ipv4**]. In this model, a node runs two IP protocol stacks side by side: one for IPv4 and one for IPv6. This applies to both end systems, clients and servers, and to intermediate devices such as routers and gateways, which can natively handle and forward both IPv4 and IPv6 traffic when they support dual stack [**punithavathani2009ipv4**].

Applications written for IPv4 use the IPv4 stack, and IPv6-capable applications use the IPv6 stack. When sending a packet, the destination address decides which stack to use[**rfc4213**]. In practice, this works through DNS: A records point to IPv4 addresses, while AAAA records point to IPv6 addresses. The host's resolver and socket APIs then automatically choose the appropriate protocol stack[**punithavathani2009ipv4**].

From a deployment standpoint, dual stack is practical because most modern operating systems come with mature support for both IPv4 and IPv6[**rfc7381**]. The wide support for both protocols, combined with the fact that most applications run over their native IP without any modifications, helps explain why dual-stack has become the common approach for running IPv4 and IPv6 together in real-world networks[**punithavathani2009ipv4**]. Its main goal is to ensure compatibility and enable a gradual transition: dual-stack lets operators roll out IPv6 while staying connected to the still-dominant IPv4 Internet, but it isn't meant to solve the issue of address shortages on its own [**LEVIN20141059**].

Dual stack also has clear limits. It only allows direct communication between the same protocol: IPv6 to IPv6 and IPv4 to IPv4, so it doesn't bridge the two by itself. When communication across the protocols is needed during the transition, extra mechanisms like tunneling or translation have to be used[**punithavathani2009ipv4**]. Moreover, dual-stack doesn't actually save IPv4 addresses. Seeing it as a way to conserve them is misleading, if anything, it maintains high demand for IPv4, since every dual-stacked device still requires IPv4 connectivity[**rfc4241**] . Global IPv4 exhaustion has continued even with widespread dual-stack deployments, highlighting that dual-stack alone cannot solve the shortage [**LEVIN20141059**].

Within this thesis, dual stack serves as the performance and behavior baseline when no translation is involved. As a result, any differences we observe compared to NAT64 with CLAT can be seen as the extra cost of using translation-based approaches. At the same time, dual stack can't reduce reliance on IPv4 without additional tools, which is why the translation-based mechanisms in the next section are important [**punithavathani2009ipv4**, **LEVIN20141059**].

NAT64/DNS64 Principles NAT64 with DNS64 has emerged as a practical response to the limits of dual stack in the current Internet[**rfc7269**]. Although dual stack was initially seen as the main strategy for transitioning to IPv6, the ongoing shortage of IPv4 addresses and the uneven adoption of IPv6 make it increasingly difficult to keep every endpoint and network fully dual-stacked[**rfc7269**]. Translation technologies help bridge the gap, letting IPv6-only devices talk to IPv4-only systems without needing upgrades on every device or huge pools of public IPv4 addresses. In many networks NAT64 and DNS64 can either support or even take the place of dual stack, especially when IPv4 addresses are scarce or updating software everywhere isn't practical[**6231295**].

NAT64 was designed as the next step in the evolution of IP translation technologies. Earlier approaches like Stateless IP/ICMP Translation (SIIT) handled translation on a per-packet basis but needed a strict one-to-one mapping between IPv4 and IPv6 addresses, which limited scalability and flexibility[**rfc6146**]. NAT64 and DNS64 were created to overcome these limitations, offering clearer design, well-defined behavior, and explicit support for DNSSEC, while keeping the effective header translation techniques from SIIT[**6231295**].

Stateful NAT64 enables two-way translation between IPv6 and IPv4, allowing IPv6-only clients to connect to IPv4-only servers using TCP, UDP, or ICMP. It's designed for the long transition period where new networks and devices are primarily IPv6, while many services are still IPv4-only. When used with DNS64, neither the IPv6 client nor the IPv4 server needs any changes to communicate across the translator. Typically, a NAT64 device sits at the boundary between an IPv6-only network and the IPv4 Internet, with one interface facing each side. Packets from an IPv6 host going to an IPv4 server are routed to the NAT64, translated into IPv4, and sent onward. Replies from the server are translated back into IPv6, using the translation state established for that session[**rfc6146**].

In practice, NAT64 works through three main components. First, it translates

IP and ICMP headers while making sure transport-layer checksums and communication rules are preserved across the IPv6-to-IPv4 boundary. Second, it uses algorithmic address embedding: an operator-assigned IPv6 prefix (Pref64::/n) is combined with the IPv4 address to create “IPv4-converted” IPv6 addresses that the translator can use. Third, its NAT behavior follows standard practices for TCP, UDP, and ICMP—like consistent endpoint mappings and typical filtering modes—so applications and network traversal techniques continue to work as expected. [**rfc6146**, **6231295**].

Address representation is a key part of how NAT64 works. It uses two address pools: an IPv6 pool, made up of a dedicated IPv6 prefix (Pref64::/n) that embeds IPv4 addresses, and an IPv4 pool—usually a small shared prefix—that represents IPv6 clients on the IPv4 Internet. The IPv6 addresses are formed by combining the Pref64::/n with the 32-bit IPv4 address, adding zeros if the prefix is shorter than 96 bits. The Well-Known Prefix 64:ff9b::/96 provides a globally recognizable format, but operators can also use local prefixes. This Well-Known Prefix is especially useful when DNS64 and NAT64 are managed by different parties, as it separates the resolver from the translator while still ensuring packets reach the NAT64 device correctly. On the IPv4 side, the NAT64 translator usually maintains a small pool of public IPv4 addresses that are shared among many IPv6 clients. To make the most of this limited resource, NAT64 often uses address-and-port translation (A+P), allowing multiple IPv6 flows to share a single IPv4 address by assigning each flow a distinct range of ports[**6231295**, **rfc6146**].

NAT64 is stateful. Every time an IPv6 client starts a new connection to an IPv4 server, the translator creates a binding that links the IPv6 address, port, and protocol to an IPv4 address and port from its pool. Inside the translator, the Binding Information Base keeps track of these mappings, connecting each internal IPv6 transport address to its assigned IPv4 address. These mappings can be reused across different destinations, enabling consistent endpoint-independent connections. A separate session table keeps track of each individual flow to allow more detailed filtering and maintain per-flow state when needed. Connection lifetimes follow BEHAVE guidelines (IETF behavioral requirements for NATs): UDP bindings typically last a few minutes (around two minutes), while TCP bindings can last hours for established connections. The translator also detects TCP connection closures so it can quickly free up resources. Without an existing state in the translator, only IPv6 clients can start new connections to IPv4 servers. For connections initiated from IPv4 to IPv6, the translator needs recent outbound traffic, explicit static mappings, or support from the application itself. Since NAT64 uses endpoint-independent mapping, standard NAT traversal techniques can still work across the translator[**6231295**, **rfc6146**].

DNS64 complements NAT64 by synthesizing AAAA records from A records.

When an IPv6-only client looks up a domain and the server only has an IPv4 (A) record, DNS64 steps in: it queries the A record, embeds the resulting IPv4 address into an IPv6 address using the operator's Pref64::/n, and returns that synthesized IPv6 address to the client[**rfc6147**]. DNS64 and NAT64 share no runtime state, they only need to agree on the Pref64::/n, using the Well-Known Prefix by default or a specific local prefix[**6231295**].

An end-to-end flow is therefore: an IPv6-only client receives a synthesized AAAA embedding the server's IPv4 address, sends IPv6 packets to that address (routed to the NAT64), the translator allocates or looks up the binding, translates headers, and forwards to the IPv4 server. Return traffic is reverse-translated using the session and Binding Information Base state until idle timers expire or TCP teardown is observed [**6231295**, **rfc6147**].

Choosing between the Well-Known Prefix and a local prefix can affect how well networks work together when DNS64 and NAT64 are run by different organizations, but it doesn't significantly impact the cost of translating individual packets[**6231295**]. From a deployment standpoint, setting up NAT64 with DNS64 at the network edge is relatively simple. It lets operators run IPv6-only access networks while still reaching IPv4-only services. This is different from running a full dual-stack network, where both IPv4 and IPv6 are enabled throughout. The main trade-offs of NAT64 with DNS64 are that session initiation tends to favor IPv6 to IPv4 connections, it relies on DNS64 for translating names, and some protocols aren't fully supported. In return, it allows for much more efficient sharing of IPv4 addresses[**rfc6146**].

From both an operational and cost perspective, NAT64/DNS64 allows operators to run IPv6-only networks at the edge, with IPv4 needed only at the gateway. This makes managing IPv4 addresses easier and avoids the complexity of running dual-stack everywhere. According to a 2024 study, large-scale measurements show that public deployment in DNS resolvers is still quite limited. Among public IPv4 resolvers, only about 0.1

Finally, NAT64/DNS64 forms the foundation of 464XLAT. In this setup, a client-side stateless translator (CLAT) handles local IPv4-only applications, while the provider's NAT64 performs the stateful IPv6-to-IPv4 translation. Both use the same Pref64::/n and DNS64 principles. This approach makes legacy applications work more seamlessly on IPv6-only networks, as discussed in the next section[**6231295**].

Client-Side Translation: The role of CLAT and 464XLAT Building on the NAT64/DNS64 principles, 464XLAT is a practical method to provide limited IPv4 connectivity across IPv6-only access networks without tunnels, keeping the network IPv6-first while allowing legacy IPv4-only applications to function. It is not a full IPv4 replacement: it supports outbound, client-to-server IPv4 use cases toward globally reachable IPv4 servers, does not provide inbound IPv4 reachability, and is not aimed at IPv4 peer-to-peer scenarios. The goal is to restore just enough IPv4 to keep legacy software usable while keeping native and modern traffic on IPv6 wherever possible [**rfc6877**].

The architecture defines two roles and introduces no new control protocols. On the customer side, the CLAT (Customer-side Translator) performs stateless IPv4/IPv6 translation (SIIT) as specified in the earlier section. On the provider side, the PLAT (Provider-side Translator) is a stateful NAT64 that allows many IPv6-only clients to share a pool of IPv4 addresses. Together, CLAT and PLAT build on existing SIIT and NAT64 technologies—no new protocols are needed[**rfc6877**].

464XLAT can operate with or without DNS64. It doesn't need synthesized AAAA records: an IPv4-only application can open IPv4 sockets or use IPv4 addresses directly, and the CLAT will translate those packets into IPv6 and send them to the PLAT, which then translates them back to IPv4. When DNS64 is used, name-based connections to IPv4-only destinations go through a single stateful translation at the PLAT. This avoids having both a stateless step at the CLAT and a stateful step at the PLAT, reducing per-connection processing and keeping the translation state centralized at the provider[**rfc6877**].

The resulting packet flows are straightforward. If the destination supports IPv6, traffic goes directly over IPv6 with no translation. If the destination is IPv4-only and DNS64 is used, the client receives a synthesized AAAA record, sends IPv6 packets, and the PLAT performs a single NAT64 translation. For applications that use IPv4 addresses directly or only support IPv4 sockets, the CLAT first translates the private IPv4 packets to IPv6 in a stateless manner, and the PLAT then applies stateful NAT64. This two-step translation ensures compatibility with applications that rely on hardcoded IPv4 addresses. [**rfc6877**].

This model works for both fixed and mobile networks. In wired networks, the CPE (customer premises equipment) acts as the CLAT: devices on the LAN keep using private IPv4, the CPE routes native IPv6 traffic, and it applies SIIT toward the PLAT for IPv4-only traffic. In 3GPP mobile networks, the user equipment usually runs the CLAT: it provides a private IPv4 stack for local apps, translates IPv4-only traffic into IPv6 for the PLAT, and sends IPv6-native traffic directly without involving the CLAT. When tethering, the user equipment can create a small private IPv4 LAN behind it (using NAT44) and still perform stateless translation before sending traffic over the IPv6 connection, staying compatible with mobile policies[**rfc6877**].

From an operational perspective, 464XLAT has several advantages. By keeping IPv4 state centralized in the PLAT, providers can efficiently share limited IPv4 resources among many subscribers. Deployments are also faster to roll out, since the access network can stay IPv6-only and existing standards are reused. Running an IPv6-only access network can be simpler than maintaining dual-stack, it involves fewer protocols to manage and troubleshoot. Native IPv6 traffic stays end-to-end, and translation only happens for IPv4 flows. The PLAT can even be provided by a third party, letting an access provider run an IPv6-only network while sending CLAT-translated traffic to an external NAT64 service. This approach works especially well in mobile networks, where maintaining separate PDP contexts for IPv4 and IPv6 adds complexity[**rfc6877**].

Some practical considerations apply. CLAT uses standard IPv4-embedded IPv6 formats. It usually reserves separate /64 blocks for the uplink, each downlink segment, and stateless translation traffic. If a dedicated translation prefix isn't provided, the CLAT can combine LAN addresses to a single IPv4 address and map that to an IPv6 address it controls. The CLAT also needs to know the PLAT's translation prefix, which can be discovered automatically or set manually. By design, the prefixes for CLAT and PLAT translations are kept separate[**rfc6877**].

2.3 Linux implementations of CLAT

Tayga Tayga is a free, stateless NAT64 implementation for Linux, positioned as a lightweight, production-quality translator for environments where deploying a dedicated hardware or full-blown software NAT64 device would be excessive[**Repas_Farnadi_Lencse_**
At the time of the cited study, the latest release referenced was 0.9.2, which the authors evaluate as representative for open-source NAT64 on Linux[**palrd_tayga_readme**].

Its design philosophy is to perform transparent IPv6–to–IPv4 address and header translation while explicitly leaving policy enforcement and state handling to the Linux packet filtering and NAT framework (iptables)[**Repas_Farnadi_Lencse_2014**]. The authors stress that Tayga does not aim to replicate the flexibility of Linux's packet filters; instead, it is built to integrate cleanly with them, keeping the translator itself simple and predictable [**Repas_Farnadi_Lencse_2014**].

Functionally, Tayga provides one-to-one IPv6↔IPv4 address mapping (stateless translation) and does not offer many-to-one address multiplexing. In a typical deployment, Tayga is paired with DNS64 and a stateful NAT44 configured in iptables [**Repas_Farnadi_Lencse_2014**]. For an IPv6 client reaching an IPv4 server, Tayga maps the client's IPv6 source to a private IPv4 from a configured dynamic pool (1:1), and then NAT44 performs SNAT from that private address to the NAT64 gateway's public IPv4 [**Repas_Farnadi_Lencse_2014**]. On the return path, NAT44 restores the private IPv4, after which Tayga reconstructs the corresponding IPv6 destination using its 1:1 mapping and forwards the IPv6 packet back to the client [**Repas_Farnadi_Lencse_2014**]. This design requires provisioning a sufficiently large private IPv4 pool to support concurrent mappings [**Repas_Farnadi_Lencse_2014**].

From a deployment perspective, Tayga's stateless core means there is no built-in session tracking or policy engine; scalability and IPv4 address conservation depend on the NAT44 stage and the size of the private IPv4 pool, making Tayga a good fit when a simple, transparent translator is needed[**Repas_Farnadi_Lencse_2014**].

Tundra Tundra is an open-source IPv6-to-IPv4 and IPv4-to-IPv6 translator for Linux that runs entirely in user space. It is written in C, implements SIIT[**rfc7915**], and is designed to take advantage of multicore CPUs with a multi-threaded architecture. Packet input and output can be handled through the Linux TUN driver or via inherited file descriptors[**labuda_tundra_nat64**].

Functionally, Tundra supports several stateless translation modes. In Stateless NAT64 mode it enables a single host to reach IPv4-only destinations and, when combined with NAT66, it can be used to serve multiple IPv6-only hosts behind it [**labuda_tundra_nat64**]. In Stateless CLAT mode it allows IPv4-only applications on an IPv6-only network with NAT64 to access IPv4-only hosts. Deployed on a router with an IPv6-only uplink and NAT64, it can produce a dual-stack internal network when paired with NAT44 [**labuda_tundra_nat64**]. Beyond these, a pure SIIT mode translates addresses that embed IPv4 within an IPv6 translation prefix and vice versa [**labuda_tundra_nat64**].

The design focuses on providing a minimal feature set for SIIT/NAT64/CLAT. It avoids unnecessary features and doesn't allocate any extra memory after initialization. [**labuda_tundra_nat64**]. The addressing model doesn't use a dynamic pool, it relies on a single fixed IP from the configuration. This means that on its own, the translator can only serve one host. However, it can scale to multiple hosts or networks when used together with NAT66 or NAT44[**labuda_tundra_nat64**].

Compared to similar user-space, stateless translators like Tayga, Tundra offers multi-threading, multiple configurable modes and the option to operate on inherited file descriptors, while it deliberately lacks a dynamic address pool[**labuda_tundra_nat64**].

Jool Jool is another well-known open-source implementation for IPv4/IPv6 translation, specifically designed for Linux systems[**jool_introduction**]. Unlike Tundra's focus on userspace implementation, Jool operates within the kernel space and provides support for both Stateful NAT64 and SIIT modes. The SIIT functionality, which was introduced starting with version 3.3.0, is particularly relevant for 464XLAT deployments as it enables CLAT-like functionality on Linux systems, while the NAT64 mode can handle the PLAT side of the translation process[**jool_introduction**].

From an architectural perspective, Jool offers two distinct integration modes for packet interception: Netfilter mode and iptables mode. Both approaches hook into the PREROUTING chain but differ in their operational characteristics[**jool_introduction**].

The Netfilter mode, which was the sole operation mode until Jool version 3.5, exhibits what can be described as "greedy" behavior. It intercepts all inbound packets within its network namespace without any matching conditions and attempts to translate everything, only leaving packets untouched when translation fails [**jool_introduction**]. This mode is limited to at most one Netfilter SIIT instance and one Netfilter NAT64 instance per network namespace and begins translating immediately upon creation.

In contrast, the iptables mode, available since Jool 4.0.0, implements a more selective approach by functioning as an iptables target that can be used within specific rules[**jool_introduction**]. This mode leverages iptables matching system, meaning only packets that match a particular rule are handed to Jool for processing, while other traffic proceeds normally through the network stack. This design allows for any number of iptables-based Jool instances and rules per namespace, with instances remaining idle until a matching iptables rule directs packets to them [**jool_introduction**].

An important characteristic shared by both modes is their handling of successfully translated packets. Rather than following the conventional path through the FORWARD chain, translated packets are sent directly to POSTROUTING, effectively bypassing the FORWARD chain entirely[**jool_introduction**].

Jool's packet handling logic follows a systematic approach for determining which packets can be translated. For packets that cannot be translated, Jool returns them to the kernel under various conditions, such as when an iptables rule references a non-existent instance, when translation succeeds but the resulting packet is unroutable, or when the translator is disabled by configuration[**jool_introduction**]. In SIIT mode, specific untranslatable conditions include scenarios where addresses cannot be translated due to local interface addresses or absence of applicable translation strategies[**jool_introduction**].

The fact that Jool supports the essential protocols used in our measurements makes it well-suited for the throughput and RTT evaluations conducted with iperf and ping respectively [**jool_introduction**].

Chapter 3

Related Work

Other researchers have already shown that using NAT64 is a good way to switch from IPv4 to IPv6. They found that NAT64 is just as fast as the old ones (NAT44), and proved it using cheap, standard software[**llanto2012performance**]. The methodology includes NAT64 using ping/ping6, complemented by a laboratory comparison of NAT44 and NAT64 that records RTT, CPU, and memory[**llanto2012performance**]. NAT64 is realized with Tayga. The results show that native IPv6 achieves the best RTT. NAT64 and NAT44 perform similarly, with only minor differences in throughput, though NAT64 has a slight edge[**llanto2012performance**]. A t-test finds no significant differences between NAT64 and NAT44 for RTT, total time, bytes transferred, successful keep-alives, requests per second, and time per request, but reports a positive difference for transfer rate favoring NAT64 [**llanto2012performance**].

Another study examined IPv4 and IPv6 performance across various operating systems and transport protocols, and evaluated tunneling methods. However, their insights into translation-based strategies were largely limited to specific implementations [**quintero2016performance**]. In the case of NAT64, evaluations mostly focused on individual implementations (e.g., Tayga), with little effort made toward cross-implementation comparisons[**quintero2016performance**].

However, research gaps remain. The first evaluation[**llanto2012performance**] centers on web workloads driven by ApacheBench without bulk TCP/UDP generators such as iperf and does not assess dual-stack versus NAT64 with CLAT, nor evaluate alternative NAT64 implementations such as Jool. Similarly the second study[**quintero2016performance**] does not evaluate client-side translation (CLAT/464XLAT) leaving the end-to-end impact of NAT64 versus dual-stack open[**llanto2012performance**, **quintero2016performance**].

Chapter 4

Evaluation

The experiments were executed in three environments to keep the topology constant while varying the platform: an AWS cloud instance, a single physical Ubuntu host, and two physical Ubuntu hosts connected by an Ethernet link. Across all environments, the setup isolated components using Linux network namespaces and instantiated the three translators under test (Jool, Tayga, Tundra) with identical addressing and routing. Configuration choices such as forwarding and addressing were held consistent across environments to support comparability.

4.1 Test environments

AWS In the AWS environment, a single EC2 instance within a dual-stack VPC hosted the entire virtual topology. Client, translator, and server roles were realized as separate namespaces interconnected by veth pairs[**veth4**]. DNS64 and CLAT ran locally on the instance to avoid external dependencies, and the dual-stack baseline followed a direct namespace path that bypassed translation.

Single Ubuntu Setup On the single Ubuntu host, the same namespace-based topology was reproduced on bare metal. Client and server namespaces were connected via veth, translators were deployed in dedicated namespaces with uniform routing, and CLAT was instantiated to mirror the cloud setup.

Dual Ubuntu Setup In the dual-host setup, two Ubuntu machines were connected via a dedicated Ethernet link. The client machine hosted the namespace topology and the translator variants. The second machine provided a standard IPv4/IPv6 stack reachable over the Ethernet link. CLAT ran on the client machine, the dual-stack baseline traversed the link natively without translation. Hardware details and all configuration artifacts for these environments are provided in the Appendix.

4.2 Networking Namespace Configuration

The experiments relied on Linux networking namespaces to isolate each translator and to keep the surrounding network conditions reproducible across local and the cloud setup. For Tayga and Tundra, a single namespace per translator was connected to the host through a veth pair[**veth4**]. Both ends of the veth received link-local IPv6 addresses and the namespace end additionally received a globally scoped IPv6 address. Inside the namespace, the default IPv6 route pointed to the host-side link-local address, while the host installed a route towards the namespace prefix via the namespace-side link-local address. IPv6 forwarding was enabled on the host and inside the namespaces to allow transit of translated traffic. Jool required a slightly different arrangement with two namespaces to create a single hop through the translator: an application namespace was linked to the translator namespace using a second veth pair. This common namespace wiring, addressing, and forwarding configuration was applied consistently on all machines so that the translator initialization in the next section could proceed without repeating network setup details.

4.3 Implementation of Translation Technologies

Tayga was implemented using a TUN device (nat64) inside a dedicated network namespace. Following Tayga's stateless design, the translator was configured with the well-known NAT64 prefix 64:ff9b::/96, one local IPv4 address for the CLAT side and one local IPv6 address for the NAT64 side, and a static one-to-one map to keep address selection deterministic. After creating and bringing up the nat64 device, the CLAT's IPv4 was assigned as a /32 and an explicit route for the mapped IPv6 was installed. The namespace's default IPv4 route was directed to the nat64 device so that IPv4-only applications would traverse the translator, and IPv6 forwarding was enabled to forward translated packets toward the host-side IPv6 path. This minimal setup produced a stable and reproducible CLAT path and integrated with the shared namespace wiring described previously. The configuration mirrors Tayga's intended use as a simple, stateless NAT64 that offloads policy and state handling to the host's packet filtering/NAT framework and is typically paired with DNS64 in practical deployments, which aligns with prior descriptions of Tayga's design [**Repas_Farnadi_Lencse_2014**, **palrd_tayga_readme**].

Tundra was deployed as a user-space, stateless CLAT translator built from source with CMake and gcc and operated via the Linux TUN driver, consistent with its SIIT design and multi-threaded architecture [**rfc7915**, **labuda_tundra_nat64**]. Address synthesis used the well-known 64:ff9b::/96 prefix, and translation of private IPv4 addresses was enabled to prevent corner cases during testing. The TUN interface was named "clat", configured with static local endpoints. The interface was brought up, an explicit route to the CLAT IPv6 address was added, and the namespace's default IPv4 route was set over the TUN interface. IPv6 forwarding was enabled to allow translated traffic to leave to the host's IPv6 domain. This configuration yielded a minimal user-space CLAT setup comparable in spirit to Tayga[**labuda_tundra_nat64**].

Jool was deployed as a kernel-space, stateless SIIT translator in its own network namespace and paired with a separate application namespace to enforce a single hop through the translator. The namespaces were connected by a veth pair configured as an IPv4 link with additional IPv6 addresses on the same link, while a second veth pair connected the Jool namespace to the host and carried an IPv6 address. IPv4 and IPv6 forwarding were enabled in both host and Jool namespaces, the application namespace used a default IPv4 gateway so that all IPv4 flows traversed the translator, and IPv6 followed the link-local default routing pattern established earlier. Inside jool-ns (networking namespaces where jool was configured, not the namespace for the application), the jool_siit module was loaded and a SIIT instance was created in Netfilter mode with pool6 set to `64:ff9b::/96`. An explicit Address Mapping Table entry bound the application's IPv4 address to the Jool-side IPv6 address to keep the translation deterministic for the measurements. This choice of Netfilter mode, where inbound packets are greedily intercepted within the namespace, kept the configuration minimal while ensuring a low-overhead kernel datapath comparable to the user-space translators[**jool_introduction**]. Given Jool's SIIT/CLAT capabilities the setup aligns with the stateless translation model defined by SIIT [**jool_introduction**, **rfc7915**].

4.4 Measurement Methodology

Throughput was measured with iperf3 using a dedicated namespace (iperf-ns). The namespace was connected to the host via a veth pair, both ends received IPv6 link-local addresses, and the namespace end was assigned an IPv6 address. The namespace installed a default IPv6 route toward the host's link-local address, while the host added routes to the translator prefixes over the same link. To enable NAT64-based reachability for an IPv4-only target within a stateless translation model, the well-known NAT64 prefix 64:ff9b::/96 was used to embed 192.0.0.171 as

64:ff9b::192.0.0.171, which was configured on iperf-ns so the server could accept both native IPv6 and synthesized IPv6 connections[**rfc7915**]. An iperf3 server ran inside iperf-ns, and clients were executed from tayga-ns, tundra-ns, and jool-app-ns as defined in Sections 4.2–4.3.

The measurement plan separated a native IPv6 baseline from the CLAT translation paths. The IPv6 target provided a no-translation baseline whose purpose was to show pure topology effects, in particular the extra namespace hop in the Jool setup compared to the single-hop topologies of Tayga and Tundra. In contrast, the IPv4 target is the focus of this evaluation: client traffic started as IPv4 inside each namespace and was translated by the respective CLAT implementation, enabling a direct comparison of Tayga, Tundra, and Jool and, secondarily, a comparison to the IPv6 baseline to measure how much of any difference came from translation versus hop count. All tests used iperf3 over TCP with two durations (30 s and 120 s) to capture short and sustained behavior. The same scripts and parameters were applied on all three environments.

RTT was measured with ICMP echo using a simple harness that iterates over namespaces and targets, executes ping inside each namespace, and stores raw outputs for later plotting. Two targets were probed: the IPv4 address (the CLAT case of interest) and the IPv6 address (baseline). The IPv6 baseline is included for completeness consistent with the TCP tests. For each run, the script selects ping or ping -6, uses a 1 s send interval and a deadline of 30s. The IPv4 measurements originate as IPv4 inside the namespace and are translated by the local CLAT toward the host-side IPv6 path. Replies are translated back by the same CLAT. The IPv6 target is reached natively without translation.

Chapter 5

Results

Chapter 5 reports the performance measurements and compares the three Linux translation implementations against a native Dual-Stack baseline across the three environments introduced in Chapter 4. The goal is to quantify the translation overhead of a CLAT-based path relative to Dual-Stack along two dimensions: TCP throughput and ICMP round-trip time (RTT). Sections 5.1 and 5.2 present the empirical results, Section 5.3 synthesizes and interprets the findings across environments and implementations and Section 5.4 summarizes practical challenges encountered during measurement and their mitigations.

Because the three environments differ in virtualization and CPU topology absolute numbers are not directly comparable across environments. Instead, the within-environment comparison to the IPv6 baseline is the primary focus.

5.1 Throughput

In this section all figures have the time in seconds on the x-Axis and the throughput in Mbits/s on the y-Axis. The first measurements were conducted in the AWS environment.

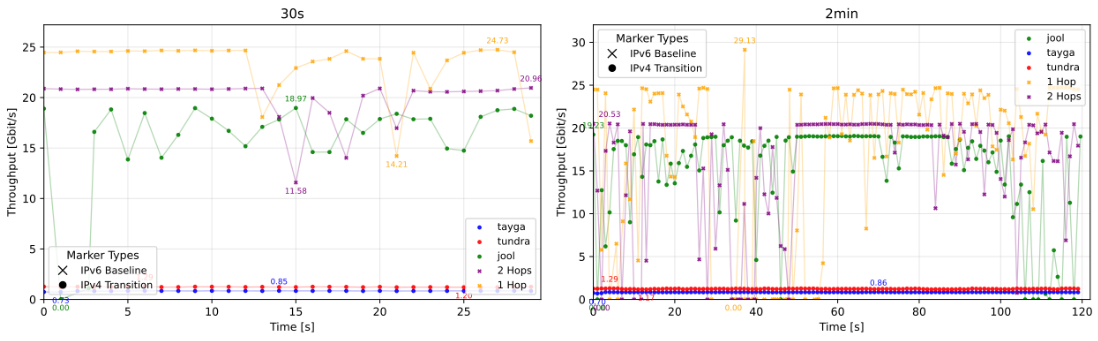

Figure 5.1: AWS cloud environment, KVM-clock, linear scale

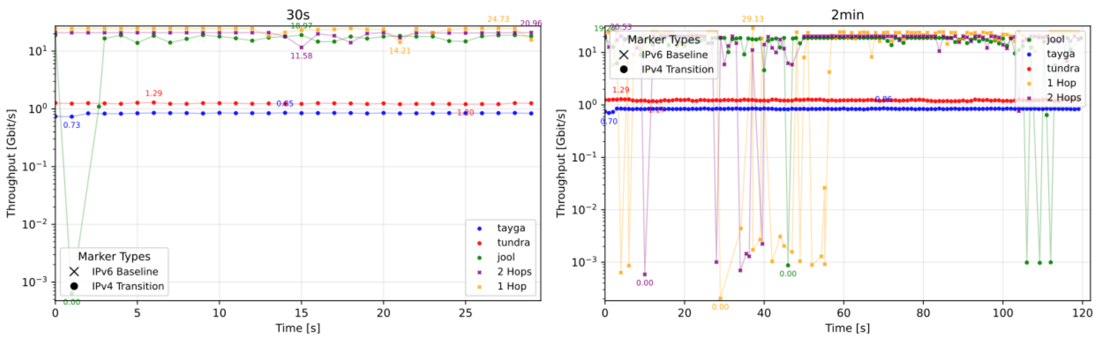

Figure 5.2: AWS cloud environment, kvm-clock, logarithmic scale

Figures 5.1 shows the initial TCP throughput results for the AWS environment in linear and logarithmic y-Axis scaling. Tayga and Tundra show stable, consistent performance across repeated runs. In contrast, both the native IPv6 baseline and Jool show large variance and no clear trend, suggesting measurement instability rather than a deterministic effect of the datapath.

Given the constant topology and parameters, this pattern pointed to a timing artifact on the host rather than a translator-specific issue. It was hypothesized that the guest clocksource used by iperf3 and the kernel (kvm-clock) might be introducing jitter under load. To test this, we repeated the measurements after switching the guest clocksource from kvm-clock to hpet. The results are presented in the following Figure 5.3.

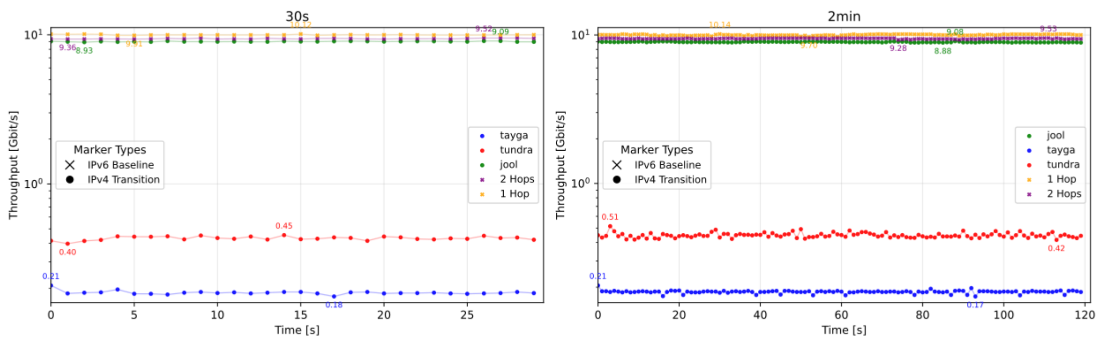

Figure 5.3: AWS cloud environment, hpet, log scale

After switching the clocksource to hpet, the single-axis plots still appeared compressed at the extremes: the IPv6 baseline and jool clustered near the top of the axis while the CLAT cases clustered near the bottom. To improve readability, a dual-y-axis figure was introduced: the left y-axis is scaled for the IPv6 baseline, and the right y-axis is scaled for the IPv4 translation methods. This separates the ranges and makes differences within each group visible.

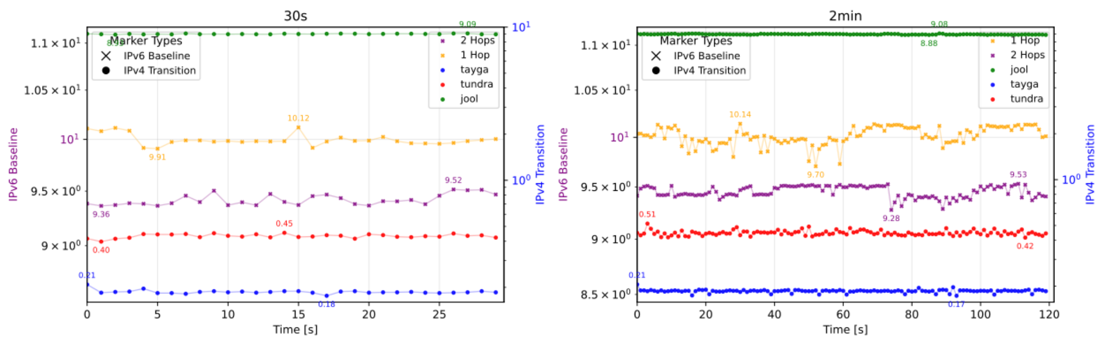

Figure 5.4: AWS Throughput Results, hpet, dual-y-axis, log scale

Even with HPET and the dual-y-axis view, the IPv6 baseline shows step-like jumps rather than a smooth distribution, indicating high amounts of jitter.

Because these irregularities persist despite the clocksource change and improved plotting, the experiments where moved to a local setup to reduce potential platform-induced artifacts. The next figures presents those results.

Local Machine Results The following figures show the results from the local machine environment. Starting with the results for the single local machine setup.

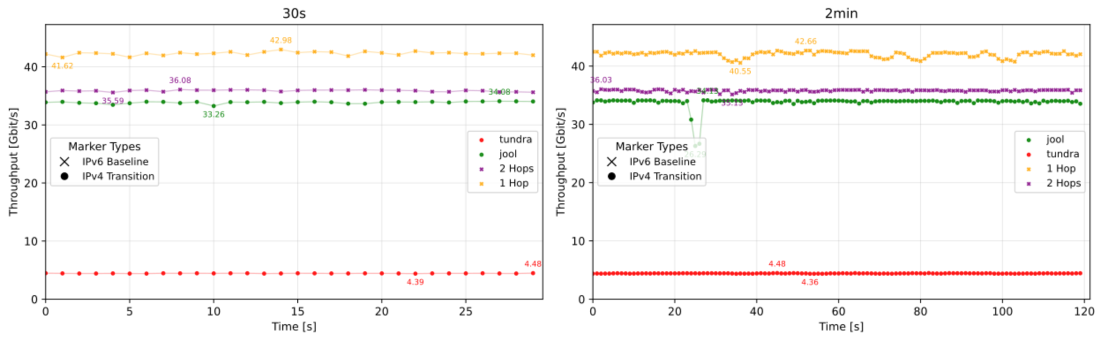

Figure 5.5: Single local machine, tsc, linear scale

For consistency we also switched the clocksource to hpet in the local machine environment.

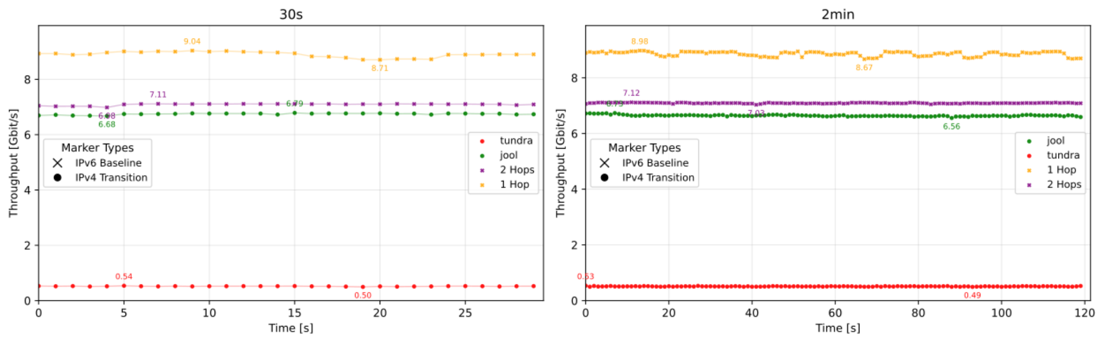

Figure 5.6: Single local machine, hpet, linear scale

Switching to the local setup resulted in noticably more stable TCP throughput for both the IPv6 baseline and the CLAT cases. Variability is substantially reduced compared to the AWS runs and repeated measurements cluster tightly around their central values.

Baseline behavior matches expectations: the one-hop IPv6 baseline sustains higher throughput than the two-hop baseline. The observed means are approximately 41.61 Gbit/s for the one-hop path versus 35.84 Gbit/s for the two-hop path, reflecting the additional namespace hop and overhead.

Within the local setup, Jool consistently delivers much higher throughput than Tundra. Under the TSC clocksource (Figure 5.5), mean throughput is about 33.7 Gbit/s for Jool versus 4.46 Gbit/s for Tundra (roughly 7 times higher). With HPET (Figure 5.6), the relative ordering is unchanged, with means of about 6.735

Gbit/s for Jool and 0.512 Gbit/s for Tundra (over 10 times higher). Because the absolute levels differ across clocksources, comparisons should be made within each clocksource. The discussion of timing effects and datapath mechanisms is delayed to Section 5.3. Tayga measurements are absent in this environment. The reasons are detailed in Section 5.3. Given that Tayga and Tundra are both user-space, stateless CLAT implementations with similar behavior, we use Tundra as a representative user-space translator here. The AWS results support this choice, showing comparable performance between the two. Figure 5.7 illustrates, in contrast to the earlier AWS dual-y-axis plot, a significantly more stable IPv6 baseline and a clear distinction between the two CLAT implementations.

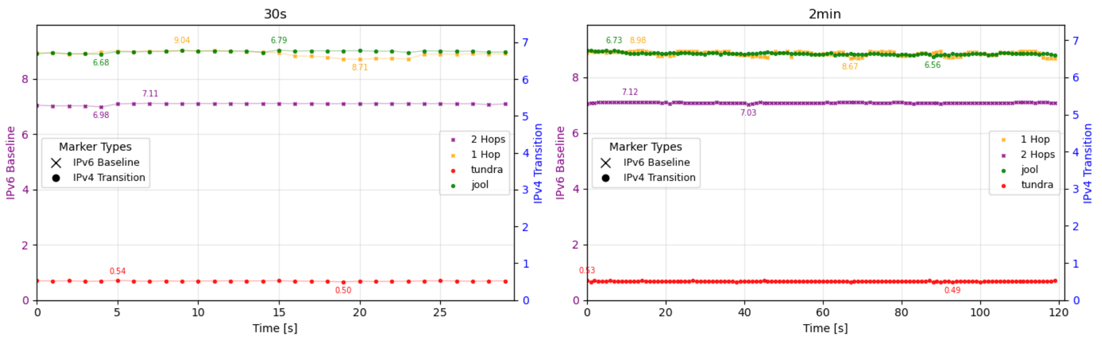

Figure 5.7: Single local machine, hpet, dual-y-axis, linear scale

For instance, the range of the IPv6 baseline for two hops has decreased from 0.25 Gbit/s in AWS to about 0.09 Gbit/s in the local setup, indicating a more stable measurement environment.

Dual Machine Local Setup Results The final environment for measuring TCP throughput performance is the dual-machine local setup. The following figures show the results for this environment.

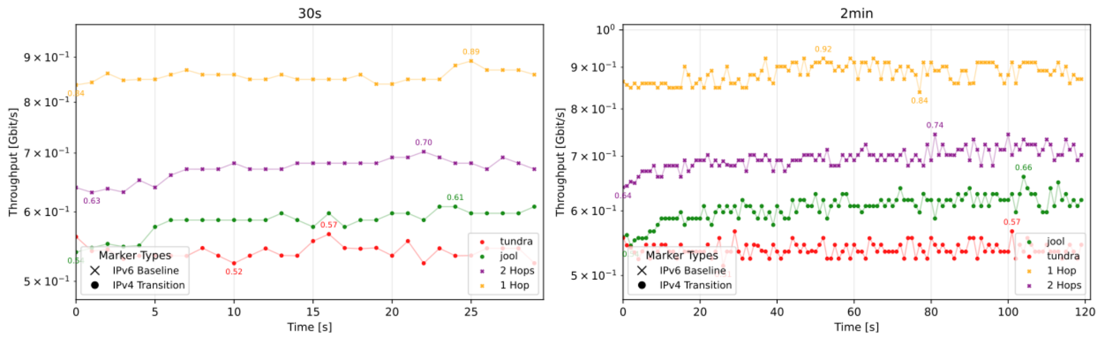

Figure 5.8: Dual local machines, hpet, log scale

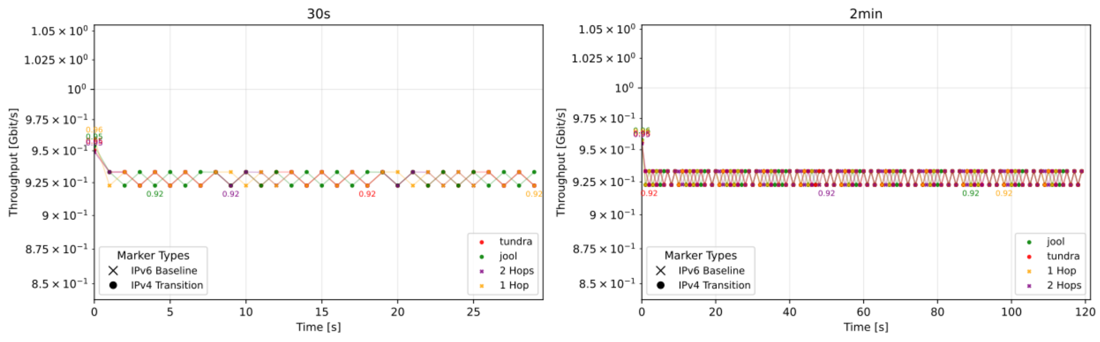

Figure 5.9: Dual local machines, tsc, log scale

Noticably the throughput values are significantly lower than for the previous environment. Now Jool peakes at 0.66 Gbit/s (Figure 5.8) and Tundra at 0.57 Gbit/s (Figure 5.8). The big difference between Jool and Tundra has almost disappeared. This outcome is expected, as the setup relies on Cat5e Ethernet links with a maximum capacity of 1 Gbit/s between machines, creating a bottleneck compared to the loopback interface used in the previous setup. The maximum achievable throughput is therefore limited by the link speed. This is reflected in Figure 5.9, where all datapoints are capped at 1 Gbit/s.

We observe here as well that Jool delivers the best performance, consistent with the single-machine setup. The slight upward trend is likely attributable to TCP congestion control mechanisms (with cubic being used in the dual-machine setup). Again, looking at the dual-y-axis plot in Figure **??** for this scenario, the IPv6 baseline shows a stable throughput compared to the AWS environment.

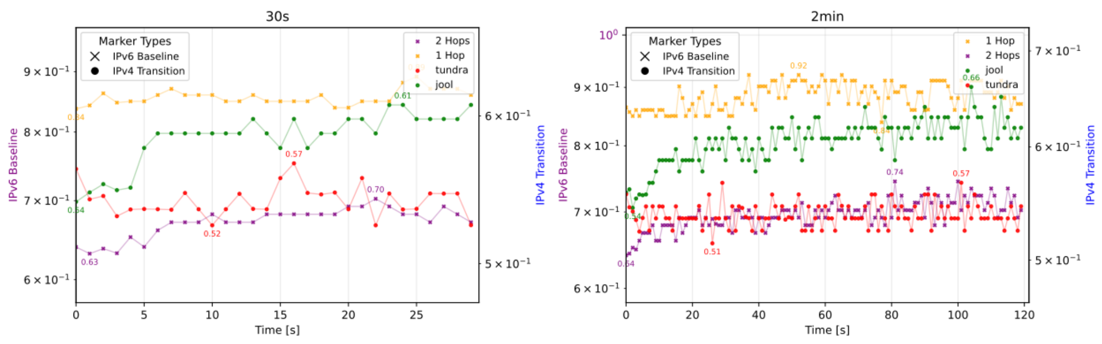

Figure 5.10: Single local machine, hpet, dual-y-axis, log scale

5.2 RTT

The following figures show the RTT measurements for the AWS environment first.

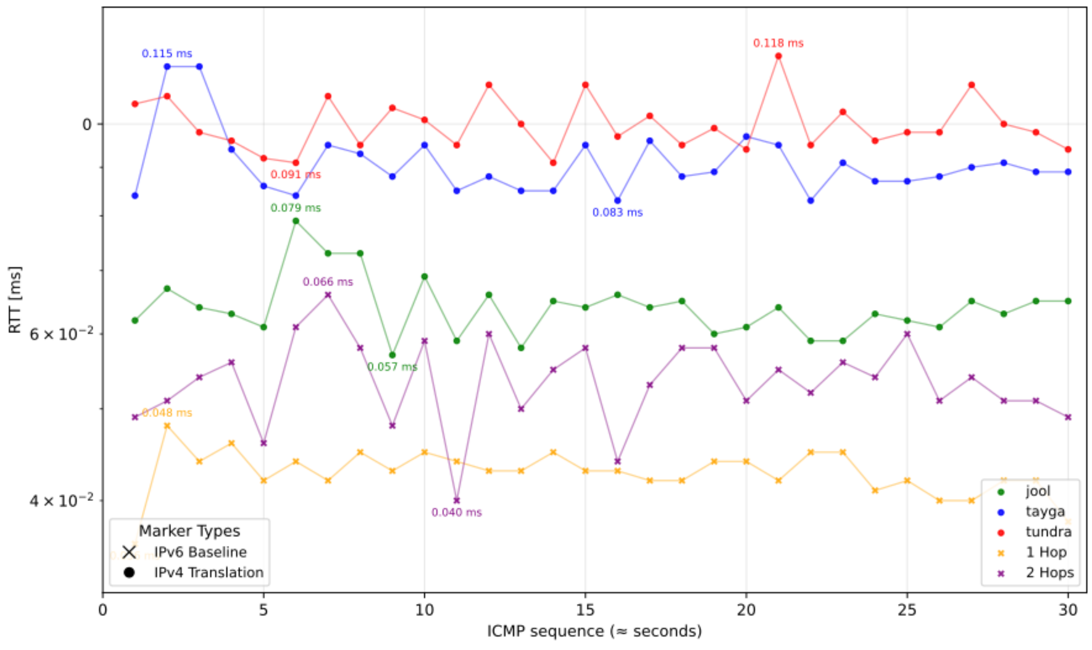

Figure 5.11: AWS cloud environment, hpet, log scale

The results from Figure 5.11 show a clear distinction between the IPv6 baseline and the CLAT implementations. This indicates that the translation process introduces additional latency, as expected. The baseline RTT for one hop is around

0.045 ms and for two hops around 0.052 ms. Jool again outperforms the user-space implementations, with a mean RTT of around 0.065 ms compared to around 0.086 ms for Tayga and 0.095 ms for Tundra.

The following figures show the results for the single local machine environment.

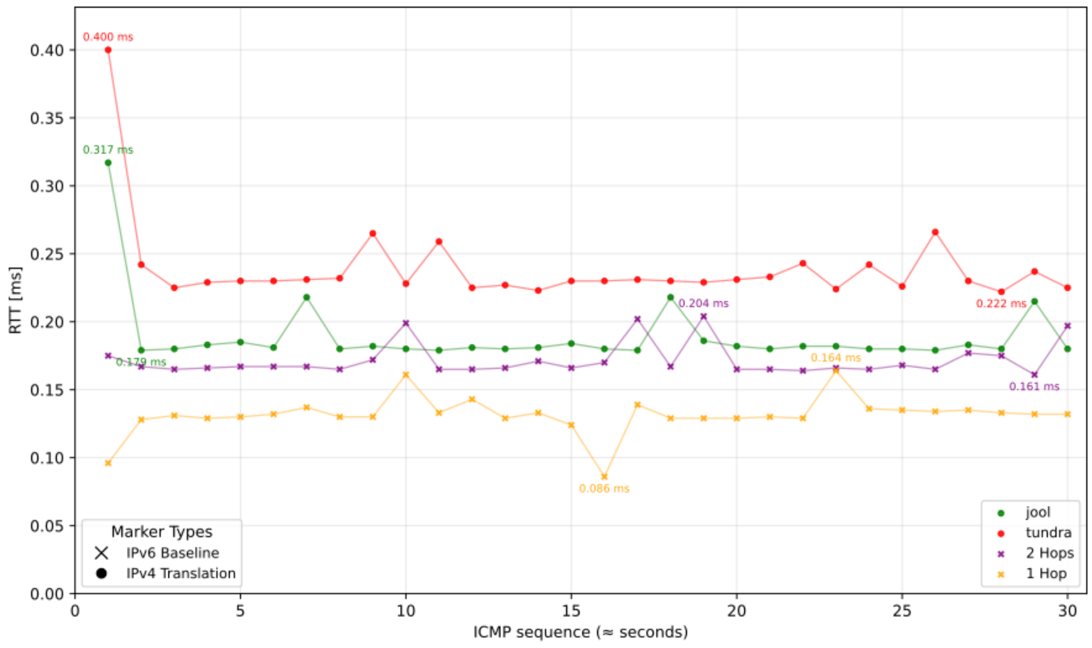

Figure 5.12: Single local machine environment, hpet, linear scale

The results from Figure 5.12 indicate that the single local machine environment exhibits higher RTT values across all implementations compared to the AWS environment. This is likely due to the weaker hardware of the local machine compared to the AWS instance. Jool again shows the best performance among the CLAT implementations, with a mean RTT of around 0.182 ms, while Tundra has a RTT mean of around 0.25 ms.

For the dual local machine environment this trend continues.

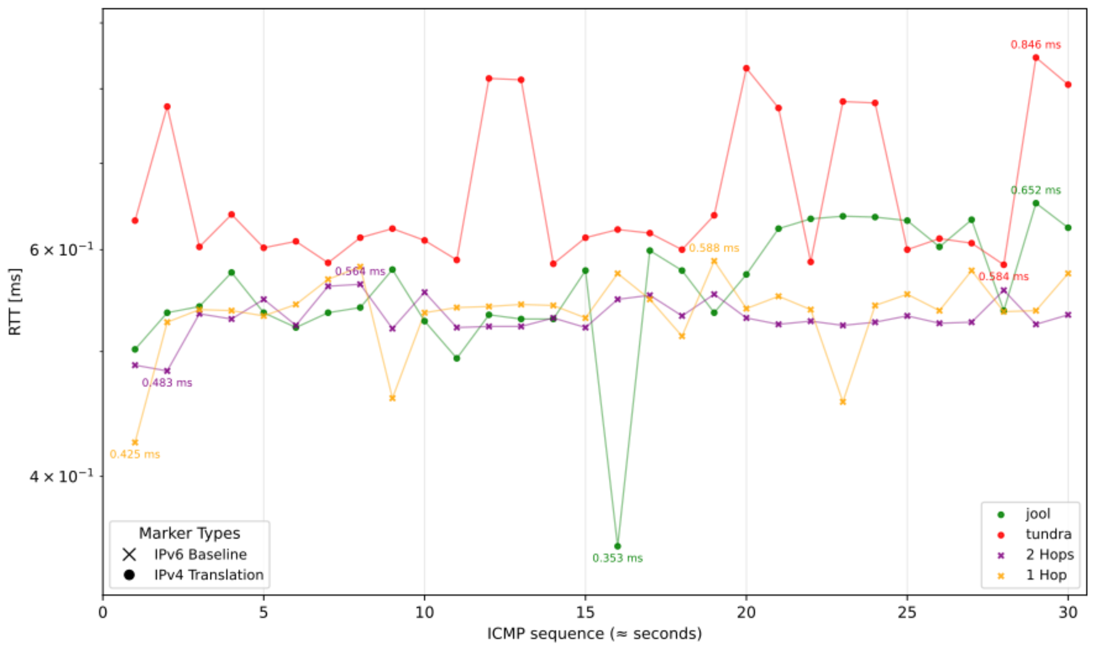

Figure 5.13: Dual local machines environment, hpet, log scale

Figure 5.13 shows that the dual local machine environment has the highest RTT values among all environments. The reason for this is the additional network hop between the two machines, which introduces extra latency. The additional hop between the two machines also adds more variability to the measurements, as seen in the wider spread of RTT values for all implementations. Nevertheless Jool continues to deliver the best performance among the CLAT implementations.

5.3 Discussion

5.4 Challenges and Solutions

Chapter 6

Conclusion

Summary of Key Findings

Future Work

Acknowledgments

Appendix

List of Figures

List of Listings