

Don't share, Don't lock: Large-scale Software Connection Tracking with Krononat

Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot, *Technicolor*

https://www.usenix.org/conference/atc18/presentation/andre

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Don't share, Don't lock: Large-scale Software Connection Tracking with Krononat

Fabien André *Technicolor*
Stéphane Gouache *Technicolor*
Nicolas Le Scouarnec *Technicolor*
Antoine Monsifrot *Technicolor*

Abstract

To simplify software updates and provide new services, ISPs are interested in migrating network functions implemented in residential gateways (such as DSL or Cable modems) to the cloud. Two key functions of residential gateways are Network Address Translation (NAT) and stateful firewalling, which both rely on connection tracking. To date, these functions cannot be efficiently implemented in the cloud: current OSes connection tracking is unable to meet the scale and reliability needs of ISPs, while hardware appliances are often too expensive. In this paper, we present Krononat, a distributed software NAT that runs on a cluster of commodity servers, providing a cost-efficient solution with an excellent reliability. To achieve this, Krononat relies on 3 key ideas: (i) sharding the connection tracking state across multiple servers, down to the core level; (ii) steering traffic exploiting the features of entry-level switches; and (iii) avoiding all locks and data sharing on the data path. Krononat supports a rate of 77 million packets per second on only 12 cores, tracking up to 60M connections. It is immune to single node failures, and supports elastic workloads by a fast reconfiguration mechanism ($<$ 500ms).

1 Introduction

Over the last years, network and datacenter operators have started *virtualizing network functions* such as routers, firewalls or load balancers. Network Function Virtualization (NFV) consists in replacing network functions implemented in hardware appliances by software implementations deployed on commodity servers. Thus, major companies such as Google or cloud providers such as Microsoft Azure rely on software implementations for their load balancing needs [9, 29]. In these cases, software implementations bring a number of benefits: (i) a better scalability than hardware devices, thanks to the use of a scale-out model, (ii) better redundancy properties and (iii) a higher flexibility, as new software can be easily deployed while hardware is hard to change.

Because of its benefits, Internet Service Providers (ISPs) are also embracing NFV. ISPs have expressed a growing interest in moving network functions implemented in residential gateways (also known as DSL, cable or fibre modems) to commodity servers in the core network. In addition to the physical layer, residential gateways usually implement tunneling, stateful firewalling, and Network Address Translation (NAT). While the physical layer and tunneling have to be implemented in the gateway, there is an opportunity for moving the firewall and NAT functions to commodity servers in the core network. For an ISP, this brings two main benefits: (i) simplifying updates by running software on a few servers rather than millions of gateways spread across a country, (ii) exposing the user's local network, creating opportunities for new services or troubleshooting.

If NFV brings many benefits to ISPs, it also comes with two critical challenges: cost efficiency and reliability. In current OSes, the performance of connection tracking and firewalls (such as netfilter in Linux) is such that deploying them at the scale of an ISP would require a prohibitive amount of servers. Moreover, they only provide limited options for fault tolerance, making them unable to meet the reliability requirements of large ISPs. In this paper, we tackle the problem of connection tracking at the scale of an ISP. We introduce Krononat, a distributed high-performance software stateful firewall and NAT that is able to meet the requirements of an ISP. This paper makes the following contributions:

- We highlight the features of modern CPUs and commodity server hardware architectures that enable the design of a resource-efficient connection-tracking system. We propose a hardware platform able to serve millions of users.
- We propose three software design principles that enable the construction of efficient network functions: (i) sharding the connection-tracking

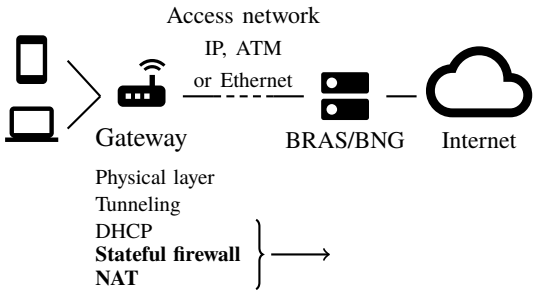

Figure 1: Simplified view of an ISP access network

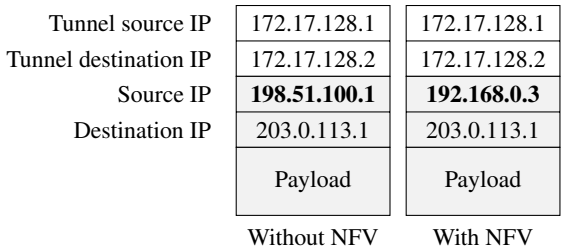

Figure 2: Tunneled packets on the access network

state down to the core level, (ii) using entry level switches to steer traffic to specific cores in multi-servers systems and (iii) avoiding all locks on the data path. We show how these principles are implemented in Krononat's software architecture.

- We show that Krononat can manage 60M flows, corresponding to an aggregated total throughput of 70 Mpps (equivalent to 155 Gbps Simple IMIX traffic), on just 12 cores (spread on 4 servers).

2 Background

In this section, we quickly review what NAT and stateful firewalls are, and explain their relationship with connection tracking. We then give a simplified overview of an ISP network to show where our software, Krononat, fits.

Connection Tracking is an essential building block for numerous network functions. In this paper, we focus on two of them (i) NAT and (ii) stateful firewall.

We are interested in port-restricted cone NAT [35], which is commonly implemented in residential gateways. NAT is used to address the scarcity of IPv4 addresses by making several devices (e.g., laptops, smartphones etc.) with local IP addresses (typically in the 192.168.0.0/24 range) appear as a single IP address on the internet, the public address of the gateway. Once a connection between (local address; local port) and (external address; external port) has been established, the NAT: (i) sends every packet from (local address; local port) through (public address; public port), and (ii) sends packets sent to (public address; public port) to (local address; local port) if and only if they go to/originate from (external address; external port). NAT thus requires tracking connections by storing entries in a table.

A stateful firewall allows incoming traffic that belongs to connections established by a local host and rejects all other incoming traffic. More formally, a stateful firewall allows an incoming packet for an external socket (external address; external port) if and only if a connection between a local socket (local address; local port) and this external socket (external address; external port) was previously established. Just like NAT, stateful firewalls require to track established connections in a connection-tracking table. Stateful firewalls are an essential security measure useful for both IPv4 and IPv6 Internet access.

Residential Access Networks We depict a simplified architecture of a residential access network on Figure 1.

The first component of an ISP network is the user's residential gateway. It is located at the user side and is known as the Customer Premises Equipment (CPE). Local devices (smartphones, laptops etc.) are connected to the gateway either through wired Ethernet or Wi-Fi. The gateway also connects to the ISP access network (xDSL, Cable or Fiber). In addition to implementing the physical layer, the residential gateway performs a number of network functions. First, the gateway attributes IP addresses to local devices through DHCP. The gateway also restrict incoming traffic to connections established by local devices (stateful firewall). Lastly, the gateway performs Network Address Translation (NAT), so that local devices appear as a single IP address on Internet (Fig. 1).

The access network carries customer traffic from the DSLAM (DSL Access Multiplexer) or OLT (Optical Line Termination) to the Broadband Remote Access Server (BRAS). Customer traffic is tunneled using PPP, L2TP or GRE and transported over IP, ATM or Ethernet. Krononat uses GRE but can be easily adapted to other tunneling protocols. For GRE, the original IP packet (inner, grayed on Figure 2) is encapsulated into another IP packet (outer, white) for routing on the access network.

The Broadband Remote Access Server (BRAS) also named Broadband Network Gateway (BNG) collects customer traffic in a central location. The BRAS decapsulates tunneled packets and forwards them to Internet routers. With NFV, ISPs are moving network functions from the gateway (DHCP, Firewall, NAT) to the BRAS. Low-traffic functions (e.g., DHCP which handles a few packets per hour) are easy to move to the BRAS. By contrast, for firewall and NAT, *every* packet must be checked against the connection-tracking table, requiring the infrastructure to handle millions of packets per second.

ISP Contraints are unique and challenge existing NAT solutions. We review the most decisive ones.
Scale: ISPs operate at a very large scale: they have millions of simultaneously connected customers, which

CPU	2x Intel Xeon E5-2698v4
	2x 20 cores, 2.2 Ghz
	2x 40 PCIe v3 lanes
Memory	128 GiB (>100 GB/s bandwidth)
NICs	10x Intel XL710 40Gbps
	400Gbps Total Throughput
Price	$30000

Table 1: Commodity server for high-performance NAT

translates into dozens of millions of connections. In addition, at peak hours, they forward traffic in excess of 100Gbps. Thus, any solution should be computationally and cost efficient, i.e., the cost per user should be low.

Reliability: NAT service is crucial for Internet connectivity, as a disruption of NAT service translates into a loss of Internet connectivity for users. An ISP NAT solution must therefore be highly reliable, and should continue working in the event of a node failure.

Elasticity: ISPs operate in a limited geographical area and have a load with strong diurnal patterns [21, 12, 32, 36]. Thus, there is an opportunity to reduce the operating cost by dynamically adapting the number of servers.

Software NAT solutions OSes offer NAT functionality, usually implemented in kernel-mode (e.g., netfilter on Linux). However, a single node is not able to handle the load generated by all users of an ISP, and these implementations do not offer easy ways to aggregate multiple servers (i.e., distribute load accross servers). Projects such as Residential Cord [2] have been started to allow the use of multiple servers but remain impaired by the low computational efficiency of OSes NAT implementations, which translates into a high consumption of computing resources and high costs. In Residential Cord [2], their Linux-based vSG (virtual Service Gateway) hits a memory limit at 2000-4000 users per server. In [25], Linux NAT achieves 200 kpps (kilo packets per second) using a single core and 1 Mpps with 8 cores while BSDs achieve 220 kpps using 1 core and 500 kpps using 8 cores of an Intel Atom C2758 processor.

Appliances	Vendor A	Vendor B	COTS server
Max. Throughput	130 Gbps	140 Gbps	400 Gbps
Max. Connections	76M	180M	1000M
List Price	$65000	$200000	$30000
Price / Gbps	**$500**	**$1400**	**$75**

Table 2: NAT Hardware Solutions

Hardware NAT solutions Major vendors offer NAT solutions that can accommodate the traffic generated by a high number of users. However, these solutions tend to have a high cost (see Table 2). Also, they lack elasticity: the addition of a device requires manual configuration. They offer limited reliability options (often limited to 1+1 redundancy). Lastly, these solutions rely on tightly coupled specialized processors and specialized software. They therefore do not offer the flexibility of a full software solution, slowing down the addition of new features and preventing independent sourcing of hardware.

3 Designing a Software NAT

In this paper, we present Krononat, a multi-user stateful firewall and NAT service. Krononat is distributed on multiple servers so that it can handle the load generated by millions of users. Krononat groups users into *shards* that are dynamically mapped on servers. Krononat ensures that a server failure does not cause service disruption by replicating the state for a shard on two servers (a slave and a master). Krononat sits at the BRAS level and thus receives IP/GRE tunneled traffic and forwards NATed packets to the Internet, and handles reverse traffic.

Our implementation builds on DPDK [1] and supports IPv4; yet our design generalizes to IPv6. We show that a careful software design and adequate hardware choice, allows achieving a high performance, and thus low-cost operations, without jeopardizing fault-tolerance.

3.1 Hardware Architecture

Current commodity servers and switches offer an opportunity for building NAT solutions that are competitive in performance with specialized solutions. Generally, specialized network appliances offer better performance than general-purpose servers through the use of content addressable memories (e.g., TCAM), that are notably used in routers and switches. However, maintaining connection-tracking tables requires much more memory than maintaining routing or switching tables: several gigabytes for connection tracking tables compared to tens of megabytes for routing tables. As TCAM are strongly limited in size ($<$ 100 Mb), network appliances such as those of Table 2 must store connection-tracking tables in DRAM. Thus, for connection tracking, network appliances do not have a decisive advantage over commodity servers, which also use DRAM. Moreover, modern CPUs compensate memory latency by caches and out-of-order execution.

Thus, to minimize the cost, we rely on commodity servers equipped with a large number of Network Interface Cards (NICs). Current dual Intel Xeon servers offer 40 PCIe lanes, enough to handle 10 40-Gbps NICs, for a total throughput of 400 Gbps. For a typical Internet workload (Simple IMIX), this corresponds to 180 Mpps. An optimal NAT solution must therefore process at least 4.5 Mpps *per core*, so that one server (Table 1) can forward 400 Gbps, thus saturating its NICs.

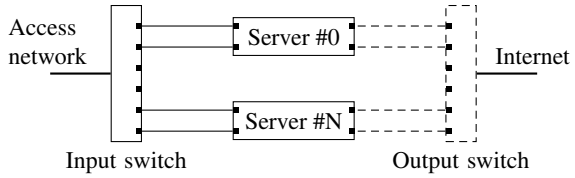

Figure 3: Hardware architecture

3.2 Software Design Principles

Achieving 4.5 Mpps per core or 400 Gbps per server requires an highly efficient implementation: each packet must be processed in less than 500 CPU cycles. We achieve this goal by relying on three design principles that allow an optimal exploitation of the hardware (servers, switches and NICs).

Sharding to the core To enable a high-performance implementation of a software NAT, we completely avoid cross-core data sharing. To this end, in Krononat, we use the CPU core as the unit of sharding in the overall system, thus departing from the traditional per-server or per-NIC sharding scheme. Hence, each CPU core can use its own dedicated connection-tracking table to process the traffic. Each customer is independent and we do not require a global shared connection-tracking table.

To implement this sharding scheme, each CPU core is associated with a NIC and exposed to the network as a distinct entity (i.e., each core has its own dedicated MAC addresses for load balancers to send traffic to it). The load balancers can thus forward the packets to a specific core. Our approach of having a dedicated network entity per core allows a greater control of the traffic-to-core mapping compared to commonly-used hashing-based methods, available on NICs (RSS) or routers (ECMP). This enables (i) a precise control of traffic steering whenever a failed master is replaced by its slave, (ii) sending upstream traffic and downstream traffic, which access the same connection-tracking table, to the same core. This cannot be achieved by Symmetrical RSS [40], because of rewritten IP headers.

To use the hardware as efficiently as possible, we do not dedicate one physical NIC for the input and one to the output for each core. Instead, we use the multi-queue capabilities of NICs (e.g., Intel VMDQ, MACVLAN filter, PCI SR-IOV) to have a dedicated queue for traffic from/to the input switch and a dedicated queue for traffic from/to the output switch[1]. Indeed, residential traffic is highly assymetrical, and dedicated NICs would not evenly use their RX and TX capabilities. Similarly, while a single core can handle the traffic of a 10 Gbps NIC, multiple cores are needed to handle the traffic of 40+ Gbps NICs. Thus, we also use the multi-queue capabilities of the NIC to expose one set of queues/identities per core to implement sharding to the core.

Sharding to the core is therefore highly beneficial for two main reasons: (i) it obviates the need for cross-core synchronization and (ii) it naturally provides NUMA-awareness, as a core never accesses data belonging to another core, and thus only accesses data on his socket.

Switch-based hardware load balancing To handle more than 400 Gbps of traffic, we use several servers. This requires balancing the traffic accross cores and across servers. To balance the load, we rely on the IP routing capabilities[2] of the input/output switches (Figure 3) which (i) remain more efficient than software for routing packets, especially when routing tables are small, (ii) are already present in the system for interconnection. Each shard has its own tunnel endpoint IP, and a dedicated subnet of public IPs. For each shard mapped to a core, a route to this core for the corresponding tunnel endpoint IP is declared to the input switch (to receive upstream traffic); and a route to this core for the corresponding subnet of public IPs is declared to the output switch (to receive downstream traffic). These routes are declared to the switches via BGP. Furthermore, all cores of Krononat thus act as IP routers by having their own IP/MAC addresses and implementing ARP protocol. Our sharding management is detailed in Section 3.4.

This approach avoids dedicating any CPU resources to traffic steering by leveraging existing switches. Also, it allows a more precise traffic steering than hash-based methods (e.g., ECMP). This precise control is needed by stateful network functions (e.g., NAT) that (i) require all packets of a connection to be handled by single core, (ii) have asymmetric headers for upstream and downstream, (iii) require controlling the route after a failover (transitioning from the master to the slave).

No locks on data path Our sharding approach ensures that threads running on different cores can forward or reject traffic without accessing data structures on other cores. This approach removes the need to lock the table to process traffic. Yet, maintenance or fault tolerance traditionally require locking data structures. For fault tolerance, connection-tracking tables need to be copied to other nodes. Traditionally, this is done by locking the table to ensure it is not modified while it is being copied. Because lock acquisition is costly and may block pro-

[1]For the sake of clarity, Figure 4 shows the NIC queues (dedicated MAC and VLAN) to which NAT thread of high-performance nodes are bound rather than the physical network interfaces. Despite the use of a shared hardware infrastructure, we still provide security isolation between the access network and the Internet. To this end, we rely on VLANs (layer 2) and VRF (layer 3) to provide isolated networks such that no packet can be switched/routed directly from the access network to the Internet without going through Krononat. Hence, input/output switch designate the input/output VRF on the physical switch.

[2]All modern entry-level 10/40 Gbps switches offer IP routing capabilities for routing tables of moderate size.

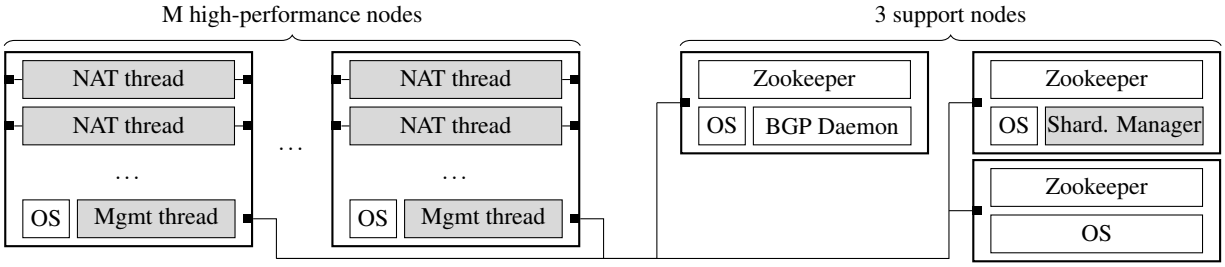

Figure 4: Software architecture

cessing on a given core, locking the table would strongly impair forwarding and is therefore not tractable in our use case. Therefore, we do not use a any lock on the data path. This requires a careful design of data structures and fault tolerance mechanisms, detailed in Section 3.5.

3.3 Software Architecture

We now present how these principles are applied to our system. Krononat comprises software components of its own (Figure 4, gray background) and uses external components (Figure 4, white background). The main functions of each component are detailed below.

NAT Thread The NAT Thread is the central component of Krononat and implements the core functionality of a NAT, as described in Section 2. Each NAT thread is pinned to a CPU core, and has the exclusive use of a set of NIC queues for output and input that it reads in poll-mode. The input switch forwards outbound traffic (GRE-encapsulated) to one input queue of a given NIC and server, depending on the Tunnel destination IP (Figure 2), which is used as the shard identifier (Section 3.2). The NAT thread associated with this input queue receives the traffic, and decapsulates GRE packets. It then forwards traffic to its output queue, and creates entries in its connection-tracking table for new connections. Conversely, each NAT thread receives inbound traffic forwarded by the output switch on its output queue. Inbound traffic is forwarded if and only if it belongs to a connection that has an entry in the table.

Management Thread The management thread communicates with Zookeeper, a strongly-consistent datastore and synchronization service that we use to coordinate all instances of Krononat. It fetches instructions (e.g., master/slave roles for NAT threads) and subscribes to asynchronous events sent by Zookeeper. This cannot be done directly by the NAT threads, because they cannot be interrupted for performance reasons. The management thread is also reponsible for most bookkeeping operations: initialization, statistics collection, etc. It synchronizes with the NAT thread using only non-blocking primitives (i.e., reads and writes to shared memory without locking nor spining).

Sharding Manager To scale to a large number of customers, we divide the load between multiple servers. The sharding manager allocates several shards on each core of each server based on the load of machines and on the traffic. The sharding manager does not directly communicate with the management thread. Instead, it writes the requested shard allocation in Zookeeper. The management thread reacts to updates in Zookeeper and propagates the allocation changes.

Zookeeper In Krononat, Zookeeper is used as central point for storing configuration (shard allocation, network configuration, addressing configuration, routes, etc.) and communicating configuration changes between servers. The use of Zookeeper for storing configuration data greatly simplifies the design of Krononat. For instance, the sharding manager does not need to persist shard allocation by itself. It can be easily restarted, or moved to another server and recover its state from Zookeeper. Similarly, when we start a new Krononat instance to handle more load it can load the global system configuration thanks to Zookeeper. We also use Zookeeper as a distributed lock service, and for detecting server failures.

3.4 Sharding and Fault tolerance

Shard In Krononat, a shard is a fixed-size group of users. In our experiments, we use shards of 256 users, but this number can be adapted. Users in the same shard share the same Tunnel destination IP, but have different Tunnel source IPs (Figure 2). In our experiments, we simulate 16384 users in 64 shards. Each user has a tunnel source IP in the range 172.17.0.0/18, and users belonging to the same shard share an address in the 172.16.0.0/26 range. Users of shard 0 have a source IP in 172.17.0.0-255 and share the tunnel destination IP 172.16.0.0; users of shard 1 have a source IP in 172.17.1.0-255 and share the tunnel destination IP 172.16.0.1; etc. One could also use the 10.0.0.0/8 range to support 4096 shards of 4096 users (i.e., a total of 2^{24} or 16M users).

Shard allocation Based on the traffic in each shard and on the load of each machine, the sharding manager allocates several shards to each NAT thread running on each CPU core of each host. The sharding manager then

writes the shard allocation in Zookeeper. In order to steer the traffic to the right core, we leverage IP routing. Each core has its own IP and MAC addresses. Whenever a given core is a master for a given shard, the routing tables are updated so that the given core becomes the default gateway for reaching all subnets associated to the given shard. Each Krononat core thus appear as an independent router in the network. This update is performed by another process that reads the routes in Zookeeper and announces them via BGP to the switches. For instance, if shard 1 has been allocated to a NAT thread (i.e., core) whose input NIC has the IP 172.27.0.1 and output NIC has the IP 127.28.0.1, the BGP will instruct the input switch to route traffic with destination IP 172.16.0.1/32 to 172.27.0.1, and the output switch to route traffic with destination 172.16.1.0/24 to 172.28.0.1. In this way, traffic for shard 1 will be received by the appropriate thread.

Fault tolerance NAT threads store their connection-tracking tables in RAM, which means they will be lost in case of hardware or software failures. One solution would be to persist the connection tracking table to a database. However, this solution would induce a high recovery time. In addition, the database would need to support a very high insertion rate, as each connection establishment results in an insertion. Database systems typically do not support such a high insertion rate because of the consistency and durability guarantees they offer. Instead, we choose to replicate the connection tracking tables in RAM, on another node. More precisely, each NAT thread is declared as *master* for a set of shards, and as *slave* for a distinct set of shards. The master NAT thread for a shard receives the traffic, updates its connection tracking table if necessary and forwards or rejects traffic. The master NAT thread for a given shard also forwards connection tracking table updates to the slave NAT thread of this shard. The slave NAT thread record these tables updates into its own connection tracking table so that entries are replicated. If the server on which the master NAT thread of the shard runs fails, the slave NAT thread becomes the master NAT thread for the shard, and a new slave NAT thread will be assigned for the shard. Server failures are detected using Zookeeper, and shard re-allocations are performed by the sharding manager. We design an ad-hoc replication protocol that allows incremental replication of the connection tracking table without locking it.

3.5 NAT Thread Implementation

The NAT thread continuously polls the NIC queues. To increase efficiency, it processes batches of packets using a run-to-completion model (i.e., packets are not queued except for sending on the network) [1, 33]. It also batches lookups in the connection-tracking table [43, 19].

Hash table Each NAT thread has a single connection-tracking table for all shards it manages either as a master or as a slave. The connection-tracking table is composed of *two* hash tables: one hash table for outgoing traffic and one hash table for incoming traffic. The outgoing traffic hash table maps 6-tuples identifying a connection (customer, protocol; private source ip; private source port; destination ip; destination port) to a 2-tuple (public source ip; public source port) used to rewrite outgoing packets. Symmetrically, the incoming traffic hash table maps 5-tuples (protocol; public source ip; public source port; destination ip; destination port) to a 3-tuple (customer, private source ip; private source port) used to rewrite incoming packets. We use Cuckoo++ hash tables [19] to store the incoming traffic table and the outgoing traffic table. Cuckoo hash tables store an entry at either one of two locations $h_1(k)$ or $h_2(k)$, where h_1 and h_2 are two distinct hash functions, and k the key of the entry. Therefore, a key lookup takes at most two memory accesses, allowing Cuckoo hash tables to support a very high lookup rate. This is key requirement in Krononat, as every packet triggers a table lookup. Cuckoo++ hash tables augment Cuckoo hash tables with a small cache-resident bloom filter that avoids checking the second location $h_2(k)$ in most cases including for negative lookups. This allows Cuckoo++ hash tables to maintain their high performance in presence of large volumes of invalid traffic or Denial-of-Service (DoS) attacks. Furthermore, to support replication without blocking traffic processing in the NAT thread, Cuckoo++ provide specific iterators, that support interleaving of updates and iteration steps by guaranteeing that all entries are iterated over at least once during a full hash-table scan.

Replication As ISPs need to provide uninterrupted Internet access, fault tolerance is a fundamental prerequisite for any network function deployed in their core network. Consequently, Krononat must support single server failures. We achieve this through *replication* of the connection-tracking entries. The master NAT thread for a shard receives all traffic belonging to the shard, and updates its hash tables accordingly. The slave NAT thread of a shard maintains a replica of all connection-tracking entries corresponding to that shard, so that it can take ownership of the shard if the master NAT thread fails. A naive approach for replication would be to lock the hash table on the master NAT thread, and dump the whole data structure on the network. This naive approach has two major defects that make it intractable: (i) locking the hash tables means stopping accepting new connections, which is obviously impossible for availability reasons, (ii) constantly dumping the whole data structure on the network would generate a high replication traffic, and also consume CPU cycles. To address both issues, we design a more elaborate replication proto-

col that has two modes: (i) *initial replication*, where we transfer the entire contents of the hash tables without locking them thanks to the aforementionned iterators in Cuckoo++, and (ii) *streaming replication*, where the master NAT thread sends incremental updates to the slave NAT thread, so as to reduce network traffic.

Initial replication When a NAT Thread becomes a slave for a shard, it has no knowledge of the connection-tracking entries the master for that shard has: the slave NAT thread needs to receive a full copy of the entries for that shard. This is achieved by the initial replication mode. In this mode, the master NAT thread iterates over hash tables entries, serializes them, and sends them over the network to the slave NAT thread. Note that this initial replication is not performed by an additional thread, it is performed by the NAT thread itself to avoid locking the table. After processing a batch of packets, the master NAT thread iterates over a few entries, and sends them over the network. It then processes the next batch of packets and iterates over the next group of entries. This procedure is repeated until the whole table has been replicated. This interleaving ensures that the initial replication does not preclude packet processing. When a packet is processed, the master NAT thread may update hash table entries (when creating new connections). Therefore, the hash table may be modified in the middle of the initial replication. To support this, Cuckoo++ iterators support modifications. More specifically, we associate a *modification bit* with each table entry. When the iterator sees a table entry, the modification bit is cleared. When a hash table entry is modified, the modification bit is set. The initial replication repeatedly iterates over entries as long as it sees a modified entry to guarantee that no unseen modified entries is left. To accelerate this process, multiple levels of modification bits are used to skip entire groups of unmodified entries.

Streaming replication When the initial replication is completed, the master NAT thread switches to streaming replication mode. In streaming replication mode, whenever it makes a change to the hash tables, the NAT thread inserts a description of this change into a *changelog*. After processing a batch of packets, the master NAT thread extracts remaining entries from the changelog, serializes them and sends them to the slave NAT thread. Because it does not constantly iterate over the hash tables, the streaming replication mode uses less CPU cycles and less network bandwidth than the initial replication mode. The replication protocol uses acknowledgments and retransmissions and in case of failure the slave is declared out of sync and must go through initial replication again.

Server CPU	Krononat (10G port)	Traffic gen.
1x E5-2695v3	1	2
1x E5-2695v3	3	2
1x E5-2643v3	4	0
2x E5-2690v4	4	4

Table 3: Allocation of servers in our testbed

4 Evaluation

Implementation Krononat (see Fig 4) is implemented in C (30K lines – gcc 5.4) on top of DPDK 17.08. The sharding manager is implemented in Scala (2K lines). The BGP part consists of a 500 lines wrapper between ZooKeeper 3.4.8 and GoBGP 1.18.

Hardware Our hardware testbed consists of four Dell R730 servers with varying CPU configurations (Table 3). They are configured in performance mode, with Turboboost disabled and equipped with Intel X540 10Gbps dual-port NICs. They are interconnected by an entry-level 10Gbps Alcatel OS6900-T20 switch, which is configured to operate as an IP router with multiple VLANs and VRFs (Virtual Routing Functions), so as to provide isolation between access, Internet, and management networks. Our testbed uses addresses in the range 172.16.0.0/12 so as not to conflict with the enterprise networks. This slightly limits the range of some parameters but our experiments show that those parameters have very little impact on performance anyway (Subsection 4.1). The hardware of the testbed is shared between Krononat and a traffic generator that we designed. Table 3 shows the mapping of 10G NIC ports.

Traffic Generator Testing the limits of Krononat requires generating a very large amount of traffic (tens of Mpps). We did not find a traffic generator that is able to generate such a load by utilizing several nodes, so we designed our own traffic generator to test Krononat. Our traffic generator builds upon DPDK, similarly to MoonGen [10] or pkt-gen [39], and borrows principles from Krononat such as (i) share-nothing, (ii) sharding to the core, (iii) switch-based hardware load balancing. It generates stateful traffic and keeps track of established connections. Each 10 Gbps NIC allocated is managed by a group of 4 threads (RX/TX Access/Internet). All instances (1 per server) emulate independant users and synchronize using Zookeeper (results, parameters, phase). This scale-out design allows a close to linear scalability and generating traffic beyond the scale of one server.

Metrics In our evaluation, we measure:

Rate for Connection initialization correspond to the rate at which packets (UDP) of new flows/connections are processed (100% upstream traffic). On actual Internet traffic, connection initialization packets represent 1-5% of the traffic [37, 18, 24, 5, 38, 31].

Rate for Established connections is the rate at which packets (UDP) belonging to existing flows are processed by the system. Our objective is to exceed 4.5 Mpps. We

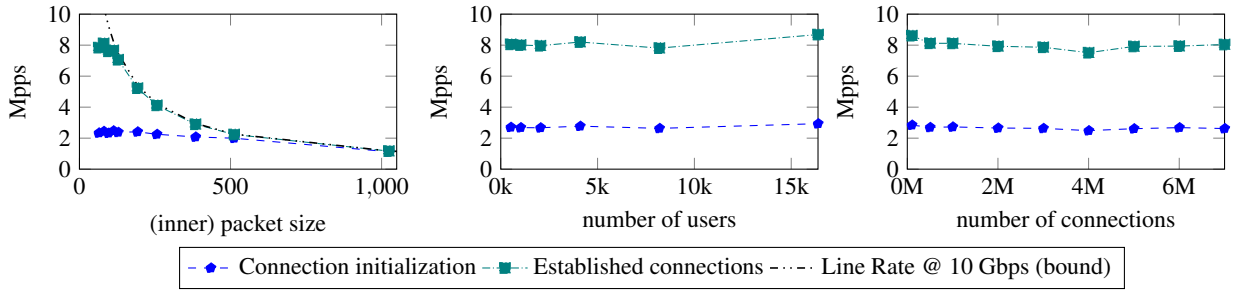

Figure 5: Performance impact of parameters in single-core case for Krononat (raw performance) (1 server)

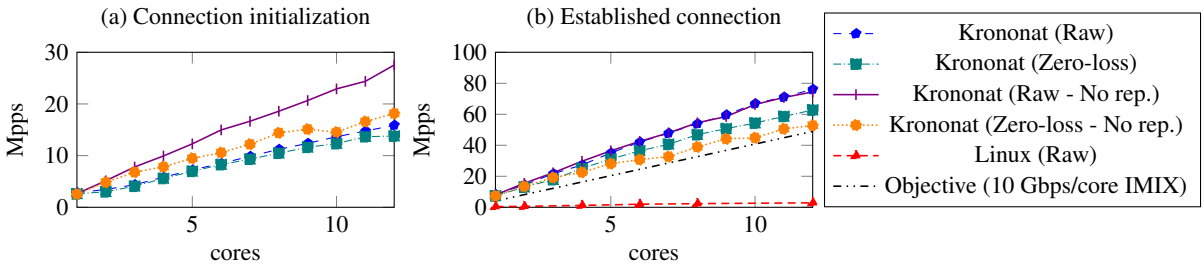

Figure 6: Performance (Mpps) in the multi-core case (4 servers).

measure for 50% upstream and 50% downstream traffic.

They are reported [7] as: (i) *raw* is the rate of traffic going through the system while flooding it; (ii) *zero-loss* is the rate which can be sustained without losing packets during 15 seconds. The *zero-loss* rate is only relevant for systems with real-time guarantees/non-blocking implementations (i.e., not for Linux-based NAT).

We also evaluate ***the duration of service interruption*** whenever a server leaves the system or crashes.

4.1 Influence of parameters

First, we evaluate the impact of a few parameters on Krononat performance. We report the raw rate for both connection initialization and established connections on Figure 5. Performance is stable at approximately 8 Mpps for established connections regardless of the number of users or connections. The performance is reduced as the packet size is increased: this is because the system achieves line rate (10 Gbps) and is therefore NIC-bound rather than CPU-bound. This shows the high efficiency of Krononat.

As we have shown that the packet size, number of users and number of connections have little to no impact on performance, we use fixed values in the remainder of this section: (i) 64-bytes packets, (ii) 16k users, (iii) 5M connections per core. We use the lowest packet-size (64 bytes without GRE encapsulation) to remain CPU-bound since we aim at evaluating the CPU-efficiency. This also allows our traffic generator, which is allocated only 8 NIC ports, to generate as many packets as necessary to evaluate Krononat on 12 cores without being limited by the speed of network interfaces.

4.2 Scalability

We benchmark Krononat with multiple cores, on all 4 testbed servers. Krononat is evaluated both with and without replication to show the overhead of replication. To give an idea of the achieved performance, we also plot performance of a trivial NAT system that uses the Linux kernel implementation and Linux namespaces[3]. We also display our performance objective: 400 Gbps/server or 4.5 Mpps per core, which enables the use of dense servers (i.e., fitted with as many NICs as CPUs support).

The results are reported on Figure 6 using 1 to 12 cores, averaged over 10 runs with random placement of NAT threads on our 4 servers. Krononat on 12 cores achieves 15 million connection initialization per second, with replication enabled (halved from Krononat without replication). For established connections, Krononat with replication is able to process packets at 77 Mpps on 12 cores[4]. The penalty when measuring performance with the zero-loss constraint is limited.

Krononat offers much higher performance than the Linux-based NAT. For established connections, Linux only achieves 0.6 Mpps on 1 core and scales to 2.9 Mpps on 12 cores. This is because Linux is a general-purpose system not dedicated to multi-tenant NAT; thus its design

[3]To maximize Linux performance, we take care to avoid extreme settings, and limit the experiment to 32 users (i.e., 32 namespaces) and 50000 flows. We distribute traffic accross cores of our Intel Xeon E5-2690v4 using RSS in a NUMA-aware way (i.e., on cores on the same socket as the NIC). Despite these advantageous settings, Linux performance remains low compared to Krononat.

[4]The performance for established connections without replication is lower as in this case, the sharding manager is not able to rebalance the load by swapping roles between master and slave cores.

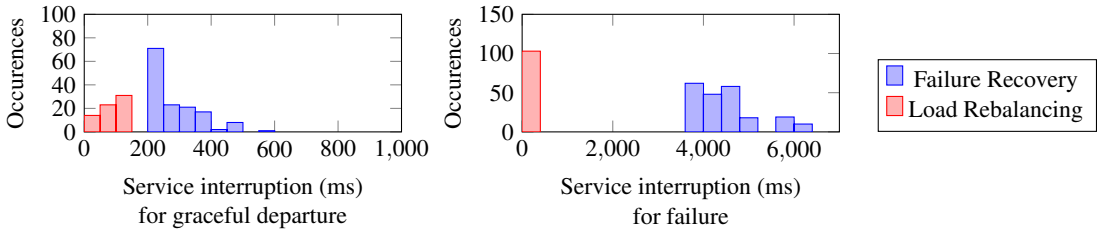

Figure 7: Service interruption due to failure or departure injection (4 servers)

favors configurability and generality over performance. This shows that Linux is unusable as an ISP-grade NAT solution. This is even more salient considering that Krononat also provides by default distribution accross servers and fault-tolerance. Krononat performance is well above our objective, reaching 6.3 Mpps/core when run on 12 cores, with replication enabled. These experiments show the excellent scalability obtained thanks to our scale-out architecture, consistently with the scalability of the underlying hash-table [19].

Running such experiments proved challenging: (i) network requirements inhibits the use of the cloud; (ii) commodity networking remains relatively expensive for large-scale experiments, (iii) simulation or virtualization overhead would make performance evaluation of such DPDK-based implementation irrelevant. We were thus limited by the networking equipment that we shared between traffic generation and Krononat. Nonetheless, our experiments involve four real servers and up to 12 cores, reaching up to 76 Mpps (6.3 Mpps/core), which is well above our target of 4.5 Mpps/core. The scale of these experiments also show the interest of using sharding to the core and hardware-based traffic steering to implement stateful multi-core multi-server traffic generators.

4.3 Service interruption duration

Beyond its high-performance and scalability, a major feature of Krononat is fault tolerance. As the state for each shard is continuously replicated on a master and a slave core, Krononat can recover from a server failure without disrupting service. Replication is also useful for dynamically adapting the number of servers (e.g., graceful departure) and rebalancing the load (i.e., swapping master and slave). We inject both graceful departures and hard failures and measure the durations of service interruptions in both cases. We report the durations of service interruptions due to recovery (i.e., the slave replaces the departed master – red) and to load-rebalancing (i.e., master and slave swap their roles – blue) on Figure 7.

Service interruptions due to graceful departure (i.e., the server disconnects gracefully from Zookeeper) remain below 500ms. This corresponds to the delay for the sharding manager to compute a new allocation, which is applied to NAT thread; and to announce routes via BGP. In the case of hard failures (i.e., the server does not disconnect gracefully from Zookeeper), the detection is left to ZooKeeper heartbeat. This increases the recovery delay to 4-7 seconds. This recovery is automatic, without any human intervention, ensuring that the system is highly available. Finally, interruptions due to load rebalancing last less than 100ms. Indeed, to rebalance the load, the sharding manager swaps roles between some masters and slaves. In this case, the interruption is mainly the delay for BGP to apply the new routes.

In all cases, these durations are low enough so that clients retransmit lost packets without declaring connections dead. This ensures that end users are not impacted. We performed real-life experiments by redirecting our own Internet traffic through Krononat for a few hours. The interruptions due to injected failures remained unnoticeable in web browsing and video streaming usages. Indeed, short interruptions are hidden by software buffering or retransmission, thus avoiding user-visible errors.

4.4 Recovery from failure

To further illustrate the system reaction to a crash, we report how Krononat (4 NAT threads on 4 servers) reacts step-by-step when one of the servers is electrically powered down. An electrical failure is triggered at the 7th second. For a short duration, approximately 16 shards have lost their master, and 13 have lost their slave. The service is thus interrupted for 16 shards. The system becomes fully available again within 200ms (i.e., no more shards in slave only mode). Shards that lost their slave or master are allocated a new slave that is being initialized (20th to 35th second). This initialization generates limited replication traffic ($<$ 65 Mbps) and has a very limited impact on the performance: traffic is still processed at approximately 24 Mpps. The performance drops slightly during recovery because the master must read and transmit its state to the new slave, and the new slave must record this state. The performance drop is limited thanks to the absence of locking of the connection-tracking table allowed by our replication protocol that allows initial replication to occur without freezing state. Without any human operator intervention, at the 35th second, the system becomes fully tolerant to failures again: all shards have a master and a synchronized slave.

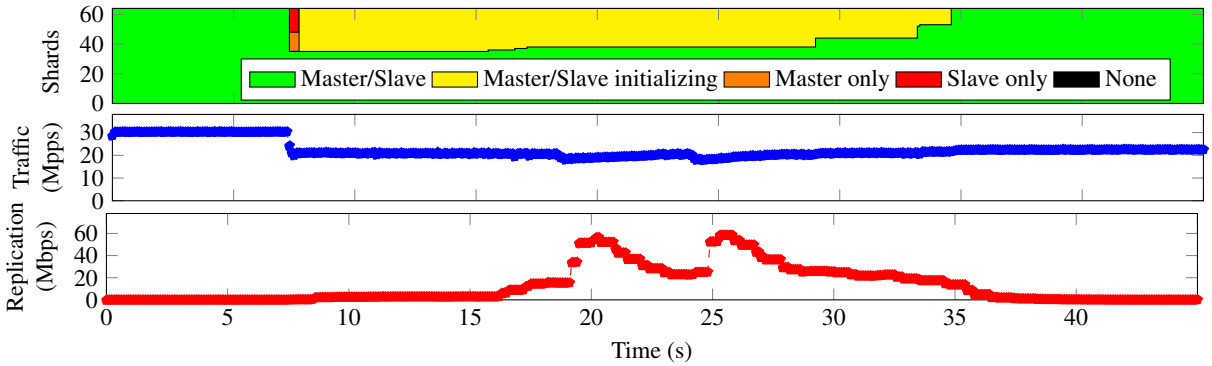

Figure 8: Execution on a system with 4 workers, on 4 servers. A server crash is introduced at time t=7s.

5 Discussion

SDN For load-balacing, Krononat relies on IP routing configured through BGP, and ARP/MAC learning for the discovery of Krononat instances. This greatly simplified the implementation of hardware-based load balancing. This choice also allows using VRF-based isolation, as well as easy inspection of routing table (e.g., using standard switch user interface). Furthermore, many production-ready BGP libraries are available.

Openflow [23] or P4 [6] could have been an alternative, but it requires specific models of switches. Also, the configuration or capabilities of switches for OpenFlow are not always well documented. Note that BGP requires routing based on IP addresses : OpenFlow could thus be advantageous if we want to do traffic steering based on other criteria (e.g., MAC addresses) or if non-IP protocols were used for tunneling on the access side.

Unavailability delay Krononat relies on Zookeeper for failure detection. This leads to uncompressible delays for recovery (4-7s) mainly due to Zookeeper failure detection. To improve this, one could rely on BFD [17] to monitor links at the switches and declare in BGP a primary route to the master and a secondary route to the slave. This way, in case of server or link failure, the switch could immediately react and route packets to the slave without waiting for Zookeeper to detect the failure. Yet, the unavailabilities we observe remain un-noticed in practice with typical web traffic including live streaming, mostly hidden by buffering and TCP retransmission.

6 Related Work

Krononat applies techniques such as kernel-bypass, run-to-completion, and core pinning [33, 1, 4, 8, 20] which are necessary to achieve high-performance on modern processors. Krononat shows how to put these in practice when dealing with a mutable state by relying on sharding-to-the-core to achieve share-nothing.

Switching or routing NFVs, targetting COTS servers, have already been studied and implemented [8, 30, 22]. ClickOS [22] also considers advanced middleboxes such as BRAS and CG-NAT but achieves only 2.3 Mpps with a 4-core processor. Switching and routing NFVs rely on a *small and static* state. They can thus share state between cores with limited performance penalty, which is not tractable in connection-tracking systems. Our papers extends the share-nothing principle beyond NIC queues and details how to distribute traffic to cores with finer control than classical hash-based traffic distribution.

NFV frameworks [28, 27, 42, 15, 41] consider the communication between NFV functions. They show that context switches have huge overhead and either (i) avoid containers/VMs by using other types of isolation [28, 42], (ii) optimize communication between containers/VMs [14, 15, 30]. They provide capabilities to share resources between multiple VMs running services consuming only a fraction of the resources. In Krononat, we do not need to isolate several chained components of our datapath, nor to share server resources between multiple small applications. Thus, these frameworks do not fit our use case and add complexity for little benefits.

Virtual switches [15, 30], can help in providing features missing from hardware switches or NICs. In our case, rather than providing software-based traffic distribution to address limitations of hardware NIC and switches, we choose to design our sharding scheme (e.g., using IP routing features) so that it can be supported by entry-level 10 Gbps switches and commodity 10 Gbps NICs. To avoid context switches, we use a single process and the run-to-completion model similarly to NetBricks [28] or BESS [15]. Yet, the modularity they bring comes at a cost: during prototyping we noticed that dynamic dispatch used in BESS or NetBricks can have a non-negligible overhead compared to static dispatch as it prevents some compiler optimizations.

The aforementionned frameworks focus on directing packets within a server, while Krononat provides a solution for directing packets accross servers through switch-

based load balancing. Also, these frameworks [15, 28] have limited features for directing packets for both flow directions to a single core (e.g., tunneled traffic, rewritten headers) as they rely on hash-based distribution (e.g., RSS), or attach a thread per NIC. By using MAC addresses to direct packet to per-core queues, we borrow from SoftBricks [8], an interesting paper that considers distribution accross server but for routing and VPN applications. It features an inspiring description of hardware capabilities and their impact on software router design.

Interestingly, designing for multi-tenancy can impact efficiency. CORD vSG [2] relies on namespace per user. This leads to one queue or datastructure per user. It prevents batching, which is yet key to high performance in large hash tables [19]. Indeed, a received batch is unlikely to contain only packets for a single user. On the contrary, Krononat relies heavily on batching to achieve its performance objective as commonly practiced in high-performance networking [1, 33].

Systems tracking connections such as load-balancers [9, 13, 29] or NAT/FW [16] also deal with the difficulty of preserving mutable state. A first approach is to rely on an external reliable database such as Memcached [11], RamCloud [26] or Adhoc [13]; while simplifying design, this comes with the cost of running the database (additional servers) and accessing it (dedicated NICs consuming PCIe lanes). A second approach is to rely on consistent-hashing, like the Maglev load balancer [9]. One enabler for this approach is that Maglev handles only unidirectionnal traffic as reverse traffic relies on DSR (Direct Server Return). MagLev achieves 2-3 Mpps/core. Krononat tackles a more challenging use case than Maglev (NAT versus load balancing), which requires handling bidirectional traffic. The NAT in [16] achieves 5 Mpps on 12 cores using RAMCloud. Krononat largely exceeds the performance of [16]. This is because Krononat underlying hash table is much faster (e.g., 10M insertions/second/core, 35M lookups/second/core and 350M lookups/second for 12 cores) than a remote RAMCloud (0.7M insertion/second and 4.7M lookups/second on 12 cores [16]). In addition, remote database accesses prevent run-to-completion.

Despite not relying on an external database, Krononat still offers reliability as it passively replicates of all connection entries. This design allows Krononat to offer a much higher performance than [16], strongly reducing costs for ISPs. An alternative design [34] to using a reliable databases is to snapshot the NFV periodically and log all packets to allow restarting the NFV from a snapshot and replaying traffic if needed. Interestingly, this approach allows adding reliability to any middlebox with little to no modification. Yet, this also comes at the cost of performance as FTMB is limited to 6 Mpps using 16 cores. Overall, Krononat favors liveness and performance over strong consistency. Indeed, for networking applications, strong consistency has a high performance impact, and may even be undesirable. A common choice in databases is to block or delay updates if they cannot be durably recorded, but for networking this means dropping any new connection thus interrupting the service. An alternative choice is thus to favor liveness: in the rather unlikely event of simultaneous failure of two servers, software clients will re-establish connections, causing little trouble.

Load-balancers [9, 29] often rely on IP routing as a first layer for traffic distribution from the Internet. Each load-balancer owns one or several of the VIP (virtual IPs) to capture traffic from the Internet. This is similar in design to our Tunnel end point IP addresses. Krononat further exploit this to also capture reverse traffic and use a different granularity by sharding down to the core-level so as to allow an highly efficient implementation.

Interesting concurrent work by, Araujo et al [3], describes a load-balancer design that shares a few key observation with Krononat : (i) commodity switches are incredibly efficient at distributing packets, both papers thus offload as much of their work as possible onto the switch, (ii) doing so requires to adapt the sharding and the network addressing scheme so that it is supported by commodity switches. Yet, as we target different applications (i.e., NAT and stateful firewall for us and load-balancing for them), other points of the design differ (e.g., permanent replication vs on-demand draining, handling bidirectional traffic, updating the routing table rather than updating the ARP table).

7 Conclusion

We presented Krononat, a high performance stateful networking service providing NAT and firewall for large-scale residential access network of ISPs. Krononat has a close to linear scalability thanks to its design relying on sharding to the core, and was shown to handle 77 Mpps on 12 cores, fully exploiting our testbed. It is designed for scale-out both accross cores and accross servers; it should scale linearly well beyond 12 cores and 4 servers.

Our design relies on sharding to the core, by exposing each core as an independent entity on the network. This allows traffic steering accross cores and servers to be performed by the switches freeing precious CPU resources. Traffic steering is based on IP routing as it allows a fine control, useful for stateful NFV functions for which RSS/ECMP offer insufficient control. Beyond Krononat, these principles proved useful for building the scale-out traffic generators that we use for the performance evaluation. We believe these principles can also apply widely to high-performance implementations of stateful NFV functions.

References

[1] DPDK: Data Plane Development Kit. `http://dpdk.org`.

[2] Residential CORD. `https://wiki.opencord.org/pages/viewpage.action?pageId=1278090`.

[3] ARAUJO, J. T., SAINO, L., BUYTENHEK, L., AND LANDA, R. Balancing on the edge: Transport affinity without network state. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, 2018), NSDI'18, USENIX Association, pp. 111–124.

[4] BARBETTE, T., SOLDANI, C., AND MATHY, L. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Washington, DC, USA, 2015), ANCS '15, IEEE Computer Society, pp. 5–16.

[5] BOCCHI, E., KHATOUNI, A. S., TRAVERSO, S., FINAMORE, A., MUNAFÒ, M., MELLIA, M., AND ROSSI, D. Statistical network monitoring: Methodology and application to carrier-grade nat. *Computer Networks 107* (2016), 20 – 35. Machine learning, data mining and Big Data frameworks for network monitoring and troubleshooting.

[6] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[7] BRADNER, S. Benchmarking terminology for network interconnection devices. IEETF RFC 1242, 1991.

[8] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 15–28.

[9] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 523–535.

[10] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., WOHLFART, F., AND CARLE, G. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference* (New York, NY, USA, 2015), IMC '15, ACM, pp. 275–287.

[11] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[12] FUKUDA, K., CHO, K., AND ESAKI, H. The impact of residential broadband traffic on japanese isp backbones. *SIGCOMM Comput. Commun. Rev. 35*, 1 (Jan. 2005), 15–22.

[13] GANDHI, R., HU, Y. C., AND ZHANG, M. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 21:1–21:16.

[14] GARZARELLA, S., LETTIERI, G., AND RIZZO, L. Virtual device passthrough for high speed vm networking. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Washington, DC, USA, 2015), ANCS '15, IEEE Computer Society, pp. 99–110.

[15] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[16] KABLAN, M., ALSUDAIS, A., KELLER, E., AND LE, F. Stateless network functions: Breaking the tight coupling of state and processing. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2017), NSDI'17, USENIX Association, pp. 97–112.

[17] KARTZ, D., AND WARD, D. Bidirectional forwarding detection (bfd). IETF RFC 5880, 2010.

[18] KIM, M.-S., WON, Y. J., AND HONG, J. W. Characteristic analysis of internet traffic from the perspective of flows. *Comput. Commun. 29*, 10 (June 2006), 1639–1652.

[19] LE SCOUARNEC, N. Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications. *ArXiv e-prints* (Dec. 2017).

[20] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 429–444.

[21] MAIER, G., FELDMANN, A., PAXSON, V., AND ALLMAN, M. On dominant characteristics of residential broadband internet traffic. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2009), IMC '09, ACM, pp. 90–102.

[22] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 459–473.

[23] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[24] MUSCARIELLO, L. *On Internet Traffic Measurements, Characterization and Modelling*. PhD thesis, Politecnico Di Torino, 2006.

[25] NEVILLE-NEIL, G., AND THOMPSON, J. Measure twice, code once: Network performance analysis for freebsd. In *ASIA BSD Conference* (2015).

[26] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The ramcloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[27] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136.

[28] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. Netbricks: Taking the v out of nfv. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 203–216.

[29] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 207–218.

[30] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E. J., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 117–130.

[31] QIAN, L., AND CARPENTER, B. E. A flow-based performance analysis of tcp and tcp applications. In *2012 18th IEEE International Conference on Networks (ICON)* (Dec 2012), pp. 41–45.

[32] QUAN, L., HEIDEMANN, J., AND PRADKIN, Y. When the internet sleeps: Correlating diurnal networks with external factors. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 87–100.

[33] RIZZO, L. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 9–9.

[34] SHERRY, J., GAO, P. X., BASU, S., PANDA, A., KRISHNAMURTHY, A., MACIOCCO, C., MANESH, M., MARTINS, J. A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 227–240.

[35] SRISURESH, P., AND EGEVANG, K. Traditional ip network address translator (traditional nat). IETF RFC 3022, 2001.

[36] STROWES, S. D. Diurnal and weekly cycles in ipv6 traffic. In *Proceedings of the 2016 Applied Networking Research Workshop* (New York, NY, USA, 2016), ANRW '16, ACM, pp. 65–67.

[37] THOMPSON, K., MILLER, G. J., AND WILDER, R. Wide-area internet traffic patterns and characteristics. *Netwrk. Mag. of Global Internetwkg. 11*, 6 (Nov. 1997), 10–23.

[38] VELAN, P., MEDKOVA, J., JIRSIK, T., AND CELEDA, P. Network traffic characterisation using flow-based statistics. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium* (April 2016), pp. 907–912.

[39] WILES, K. pktgen-dpdk. `http://pktgen-dpdk.readthedocs.io/`.

[40] WOO, S., AND PARK, K. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. Tech. rep., KAIST, 2012.

[41] WOO, S., SHERRY, J., HAN, S., MOON, S., RATNASAMY, S., AND SHENKER, S. Elastic scaling of stateful network functions. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, 2018), NSDI'18, USENIX Association, pp. 299–312.

[42] ZHANG, W., LIU, G., ZHANG, W., SHAH, N., LOPREIATO, P., TODESCHI, G., RAMAKRISHNAN, K., AND WOOD, T. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (New York, NY, USA, 2016), HotMIddlebox '16, ACM, pp. 26–31.

[43] ZHOU, D., FAN, B., LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 97–108.