Highly-Scalable Transparent Performance Enhancing Proxy

Jae Won Chung, Xiaoxiao Jiang, Jamal Hadi Salim*, Roman Mashak*, Sriram Sridhar, Manish Kurup

Verizon Labs, *Mojatatu Networks
jae.won.chung, xiaoxiao.jiang, sriram.sridhar, manish.kurup@verizon.com, hadi,mrv@mojatatu.com

Abstract

Performance Enhancing Proxies (PEPs) are often employed to improve degraded TCP performance caused by characteristics of specific link environments, for example, in satellite, wireless WAN, and wireless LAN environments. A classical PEP use case is to bridge two different TCP congestion avoidance algorithms; one suitable for wireless and another for wired network. By terminating an end-to-end TCP flow at the boundary of the networks, PEP allows to use TCP congestion avoidance algorithms designed/tuned for the environment irrespective of TCP versions used by the end-systems. One of the important traits of such PEP includes IP address transparency to avoid masking the clients from the Internet content servers. In this paper, we introduce a simple but effective way to build a highly scalable transparent PEP on Linux using HAProxy, an open-source TCP proxy, tc, containers and TProxy module. This configuration does not require TProxy Netfilter hooks to NAT and plumb TCP traffic to the proxy. The transparent PEP achieved closed to 100K TCP proxy transactions per second with 8KB object download per connection on a single server with 14 core Sandy Bridge CPUs and a RSS enabled NIC. This is a significant milestone in transparent PEP tuning. This talk presents performance analysis of the chosen and alternative design options we considered.

Introduction

Transparent L4/L7 proxies are often used to implement security gateway services or to enhance end-to-end TCP performance over satellite and wireless networks. The proxies used for the latter purpose are referred to as Performance Enhancing Proxies (PEPs) [4, 5]. Recently, PEPs are getting attentions in mobile wireless industry as a way to improve user-perceived network speed, i.e., object download time and TCP throughput. Most Internet content servers use a version of TCP designed for wired networking environment. TCP congestion control algorithms suited for wirelines may not work well on wireless environments characterized by high bit-error-rate, larger L1/L2 queues and a wide range of frequently changing available bandwidth, since assumptions made for congestion detection and recovery may not hold.

Recently, new generation TCP congestion control algorithms [3, 2, 1] have been proposed to adapt to evolving communication network environments. These algorithms use RTT-based congestion feedback loop to control TCP transmission rate, and thus have potential to perform better on mobile wireless environments. The new generation algorithms are being actively evaluated [7, 9, 8], although wide adoption as a de facto Internet standard will take a little more time and effort as their performance and fairness on various networking environments and conditions need to be evaluated.

A practical way to adopt the new TCP algorithms into mobile environment progressively before a wide adaptation is to deploy transparent L4 PEPs in the wireless access network borders. This paper shows a simple but effective way to build a highly scalable transparent PEP on Linux using HAProxy [6], an open source TCP proxy, in an attempt to fulfill the following requirements: fast time-to-market, low deployment and maintenance cost, and ease of adaptation to emerging technology.

Two assumptions behind our proof of concept (PoC) transparent TCP proxy service implementation were the following. First, the proxy servers are front-ended by L3 load balancers that forward packets from either end of a proxy flow to the same instance. This paper does not focus on the routing or the load balancer design. Second, the proxy would merely serve as PEP, and would not inspective or modify the bearer packets. This allows us to optimally splice the two sockets minimizing kernel/user cross-space memory copies and context switches.

We adopted the following three design principles: First, maximize parallel processing by pining packets from a pair of proxy flows to the same CPU core using receive side scaling (RSS). This maximizes the chance of a core processing incoming packet to completion, from an input direct memory access (DMA) ring to local TCP sockets to an outbound DMA ring. Second, minimize memory access across the non-uniform memory access (NUMA) boundary to reduce interrupts and context switches. Although not a new concept in Linux packet switching and routing domain, we show that this design principle can be extended to socket access and management for TCP termination service to further reduce software interrupts. We ended up running all HAProxy instances on the same NUMA node responsible for the NIC PCI management to get the best performance.

Third, use containers to simplify transparent proxy routing and the service orchestration. The proxy service orchestration and policy enforcement was performed at the host using Linux Traffic Control (TC) subsystem, where service eligible

flows are mirrored to proxy container via virtual interfaces. Within the service container, all incoming packets are routed to local sockets in IP_TRANSPARENT mode without using TPROXY Netfilter hooks to NAT or check for local socket. Localizing transparent proxy routing within a container helps flexible proxy service orchestration.

Our PoC transparent proxy on a single x86 server with 14 core Sandy Bridge CPUs and a RSS enabled NIC achieved closed to 100K TCP proxy transactions per second with 8KB object download per connection. This paper presents performance analysis of the chosen and alternative design options we considered, and organized as follow. In Section Methodology, we explain all the experiment scenarios and test cases. Section Bottleneck Removal describes the efforts on eliminating the bottleneck for improving the performance. Experiment results are demonstrated and analyzed in Section Performance Results and Section Conclusions and Future Work concludes our work.

Methodology

In this section, we describe our environment setup and transparent PEP performance tuning options we considered in the experiments.

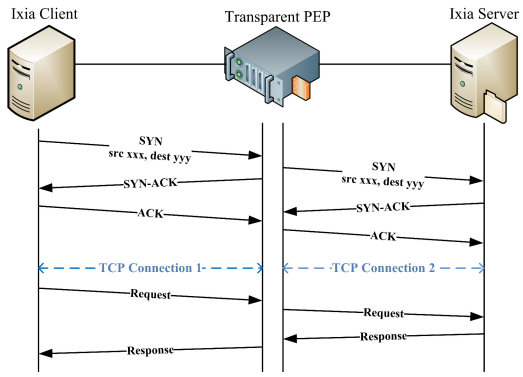

Figure 1: End-to-End Experiment Setup

Table 1: Hardware and System Information

Architecture	x86_64
Number of Sockets	2
Cores per Socket	14
Threads per Core	2
Model Name	Intel Xeon E5-2648L@1.80GHz
NIC	Intel XL710 40GbE
Kernel Version	Linux 4.11.0

Experiment Setup

Figure 1 demonstrates the end-to-end experiment setup. In our experiment, Ixia is employed to generate HTTP (port 80) traffic. While Ixia runs both client and server, the x86 blade in the middle provides the proxy service. Table 1 shows the hardware and system information of the x86 blade.

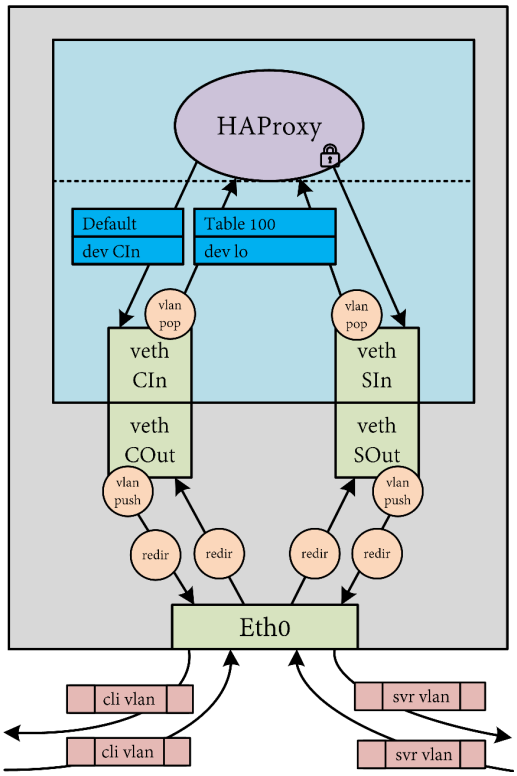

Figure 2: PEP Overview

Table 2: Summery of Scenario Configurations

Scenario	Netfilter	Proxy Listen Port
NO-NF	Disable	80
NF-NO-IPT	Enable	80
NF-IPT	Enable	1234

HAProxy is an open-source TCP/HTTP proxy offering transparent mode of operation. We used HAProxy version 1.4 for this project. It uses IP_TRANSPARENT socket option (part of tproxy kernel feature) to terminate redirected TCP connections as long as the destination port matches the local listening port, and spoof the client IP when establishing a TCP connection to the server. In addition to HAProxy configuration, Linux networking is configured to route the service eligible packets to the loop-back device to which the local proxy listen sockets are attached. This is typically done using Netfilter TPROXY hooks and routing rules.

To localize transparent proxy routing, we created a network container for HAProxy and isolated the proxy networking environment. Two pairs of virtual Ethernet interfaces (veth) are configured to connect the host and container for client-side and server-side respectively. Figure 2 describes the transparent PEP configuration. Inside the container, default route is set to the client-side veth, and the HAProxy is configured to bind the server-side connection to the server-side veth. Using this approach, packets sent from HAProxy to the client take the default route while packets from HAProxy to the server

go to the bound server-side veth interface. In addition, routing table 100 and a rule for each veth is added within the container to route all incoming packets to the loopback device.

To differentiate traffic coming from client side and Internet/server side, two VLANs are used. When Ixia sends traffic from client to server, client-side VLAN tag is attached while traffic from server to client is using the server-side VLAN. When a packet arrives at the host network interface, Traffic Control (TC) Mirred Action redirects packets based on their VLAN tag to the corresponding container virtual interfaces. When traffic landing into the container, TC VLAN Action strips off the VLAN tag and put the VLAN back when traffic sent out from the container.

Three system scenarios were considered for the performance analysis: two scenarios without using IPTables and the third using IPTables for traffic routing within the container. In the first two scenarios, we configure the HAProxy to listen on port 80, the same as Ixia generated traffic destination port. The difference between first two system scenarios is that the first scenario (NO-NF) uses Linux kernel without Netfilter support and the second (NF-NO-IPT) uses kernel with Netfilter support but without any IPTables rule. The third scenario uses IPTables TPROXY target to NAT incoming port 80 packets to a local proxy service port. This gives flexibility of mapping a wide range of destination service ports to a single local proxy port. Table 2 summarizes configurations for the three system scenarios. In all scenarios, 20 HAProxy processes were running in the container.

Performance Tuning Options

Table 3: Summery of All Test Cases

Scenario	Proxy	Splice	RSS Mode	NUMA
NO-NF	TCP	Yes	Symmetric	No Bind
NO-NF	TCP	Yes	Symmetric	Bind
NO-NF	TCP	Yes	Asymmetric	Bind
NO-NF	TCP	No	Symmetric	Bind
NO-NF	HTTP	Yes	Symmetric	Bind
NO-NF	HTTP	No	Symmetric	Bind
NF-NO-IPT	TCP	Yes	Symmetric	No Bind
NF-NO-IPT	TCP	Yes	Symmetric	Bind
NF-NO-IPT	TCP	Yes	Asymmetric	Bind
NF-IPT	TCP	Yes	Symmetric	No Bind
NF-IPT	TCP	Yes	Symmetric	Bind
NF-IPT	TCP	Yes	ASymmetric	Bind

Four variables considered for our proxy service performance tuning are RSS mode, splicing, proxy mode (TCP/HTTP) and proxy-NUMA binding.

RSS (Receive Side Scaling): RSS is a mechanism provided by NIC to support multiple receiving queues that distributes traffic among multiple CPUs. By configuring the hash key in NIC, symmetrical RSS can be achieved such that the same CPU will handle both sides of the connection. For results shown in Section Performance Results, the NIC was configured to RSS with 28 queues, one for each hyper-thread.

Splicing: Splicing is another technique which potentially boosts up the proxy performance, by which two sockets can be spliced inside kernel instead of sending traffic to the user-space proxy.

Proxy Mode: HAProxy can be configured as either HTTP or TCP proxy. While TCP proxy runs on layer 4 doesn't involve layer 7 examination, HTTP proxy needs to parse the HTTP request/response in order to proxy the traffic.

Proxy-NUMA Binding: Instead of letting CPU cores from both the NUMA nodes run HAProxy processes, we bind those processes to the cores within the same NUMA that manages the NIC PCI.

Totally 12 combinations were tested in this paper for showing the effect of each factor. Table 3 lists of all the tests we run in this paper.

Bottleneck Elimination

Before jumping to the transparent proxy performance evaluation, we base-lined the performance of containerized service orchestration using Linux TC subsystem. 128K TC (CLSFW rules) graphs were created and used in the baseline setup. The qualifying metrics are packets per second (PPS) and average latency. These metrics shall effectively assess the capability of the NIC+TC pipeline to sustain TCP/IP encapsulated HTTP session traffic at some pre-determined TPS. All the metrics are calculated by pushing traffic through our x86 blade using the Ixia traffic generator. The latency is calculated by inserting a timestamp into the packet during Tx and calculating the differential when its received at the other end. This requires us to cap the minimum packet size at 78 byte packets. We used single-VLAN-tagged UDP/IP packets for this test, since the objective was to tune raw L2/L3 bandwidth through the NIC+TC subsystem in preparation for actual TCP+HTTP traffic.

For this effort, the NIC was configured to do the following: RSS with 12 queues, use the multi queue priority (mqprio) qdisc, tuned netdev_budget_usecs = 4000 and set the CLSFW hash bucket size as 128K. Different packet sizes (78-1500 bytes) were used through the baseline tests. The mqprio Linux queuing discipline provides a method to map multiple traffic classes to specific hardware queues on the NIC. The general observation was that larger packet sizes yield higher throughput (close to 20Gbps), but this only means that PPS is the real bottleneck, and all further attempts we made were only to maximize PPS.

Further analysis made along this path revealed a couple of kernel locks that were causing bottlenecks: 1) A transmit lock in the prio qdisc causes poor performance when contended by multiple cores. The prio qdisc uses a single lock to guard accessing to NIC hardware queues (as opposed to the mqprio qdisc). 2) A TC action context and statistics update lock in the VLAN action was causing performance issues as well. Both bottlenecks were resolved by using the mqprio qdisc and modifying the TC VLAN action which is a kernel module to use the Read Copy Update (RCU)) mechanism, instead of a spinlock. The linux RCU provides users (kernel threads/functions) a method to protect shared structures against concurrent read/write access. Table 4 presents the results after removing the mentioned bottlenecks.

Table 4: Results after Bottleneck Removal

Pkt Size(B)	Rx PPS	Tx PPS	Rx Mbps	Tx Mbps	Tx Avg. Latency (us)	Rx Avg. Latency (us)
78	10M	6.8M	6240	4261	873	873
256	8.9M	7.7M	18285	15736	883	883
800	3M	3M	19417	19417	34	34
900	2.7M	2.7M	19483	19483	190	97
1500	1.6M	1.6M	19680	19680	106	105

Performance Results

This section shows the performance results of all test scenarios, and analyzes the impacts of the four tuning options mentioned in Section Methodology.

Figure 3 and Figure 4 depict the performance results of the three system scenarios: NO-NF, NF-NO-IPT and NF-IPT. Overall, the NO-NF with all the tuning options enabled: Symmetric RSS, Proxy-NUMA Bounding, Splicing, TCP Proxy, produces the best performance as 97K proxy transactions per second (TPS) and 2.8ms of time-to-first-byte latency. Scenario NF-NO-IPT has 3K less TPS and 0.2ms more latency comparing to NO-NF, which indicates, even IPTables is not used, the Netfilter still introduces some overheads. Similarly, scenario NF-NO-IPT performs slightly better than NF-IPT, however the difference is insignificant (1% for TPS and 3% for latency) as NF-IPT uses only a single stateless TPROXY NAT rule.

Initially, in order to validate NUMA overhead for packet processing, we tried RSS on all the cores across both NUMA nodes. As expected, we got a poor performance low as 20K TPS. Thus, we decided to RSS on the first NUMA node responsible for NIC PCI management for the rest of our tests. As shown in the results, symmetric RSS having the same CPU core to handle data and acknowledge packets from both end of TCP proxy connection pairs improves the performance. Comparing to asymmetric RSS, symmetric RSS supports 2% more TPS and 3% to 6% less time-to-first-byte latency.

We found that not binding HAProxy processes to the same NUMA node where packets ingress severely hurts the proxy performance even when splicing is used. This is because NUMA overhead applies to socket structure access. For all the three system scenarios, the TPS drops around 10% to 14% when HAProxy processes are not bounded to the first NUMA node. The time-to-first-byte latency also decreases by 14%, 20% and 32% for NO-NF, NF-NO-IPT and NF-IPT system scenarios respectively as the traffic load on the systems is decreased.

Although TCP proxy does fulfill our requirements, we are still interested in understanding the performance of HTTP proxy and seeing how much overhead it can introduce. As expected, Figure 5 and Figure 6 show that due to extra load of HTTP request and response header processing, the proxy performance reduces dramatically when HTTP proxy mode is used. TPS of HTTP proxy drops by 23K (23.7%) and time-to-first-byte latency increases by 1.6ms (57%) comparing to the TCP proxy. Due to the huge performance impact, it is recommended to use HTTP proxy mode only when necessary.

Next, we investigate the impact of socket splicing option

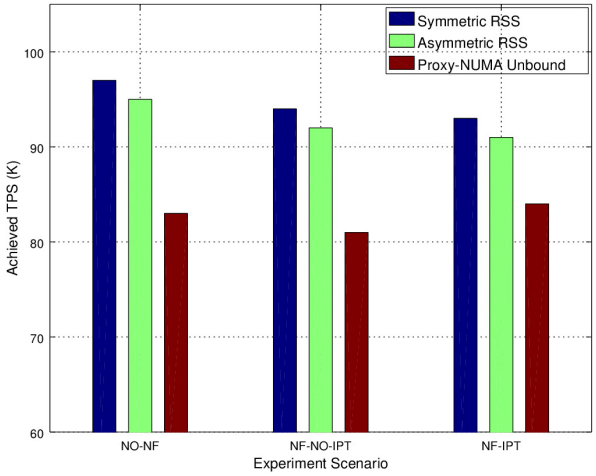

Figure 3: TPS Impact of RSS Mode

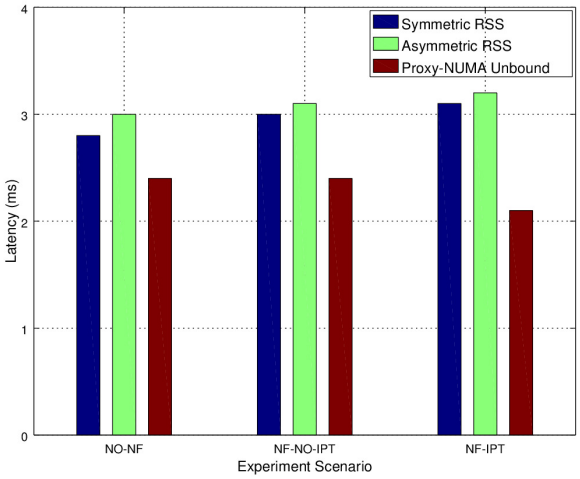

Figure 4: Latency Impact of RSS Mode

provided by Linux kernel. By turning the splicing feature off, the TPS decreases about 3K (3%) and latency increases by 0.3ms (10%) when HAProxy is running on TCP mode. Since we are transmitting objects with very small size (8KB), the advantage of splicing is insignificant. We are expecting more

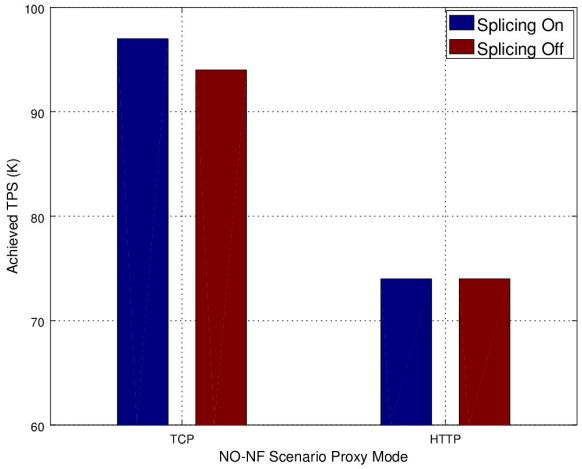

Figure 5: TPS Impact of TCP Splicing

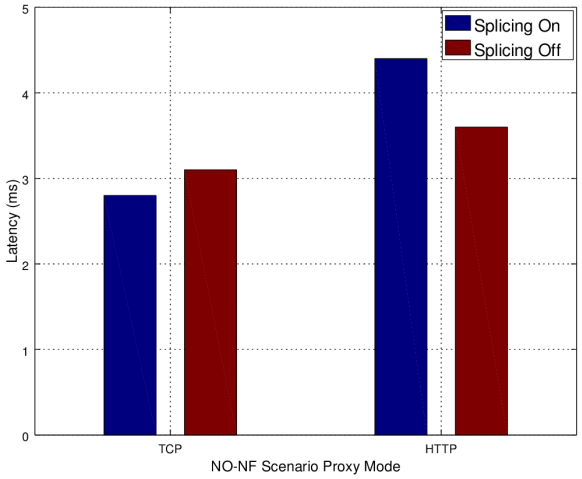

Figure 6: Latency Impact of TCP Splicing

performance gain from splicing for larger object transfers. On the other side, for HTTP proxy, the splicing little impacts the performance. TPS stays the same when splicing is turned off, and time-to-first-byte latency is even improved by 18%. This is because, although HTTP response header is given to the client-side socket, the splice system call delays the initial move of the remaining response data to the client-side socket which waits for enough data to generate the first packet for efficiency reason.

Conclusions and Future Work

In this paper, we introduce a simple but effective way to build a highly scalable transparent PEP on Linux using open-sourced HAProxy. When Netfilter is completely disabled, our transparent PEP with 14 core Sandy Bridge CPUs and symmetric RSS enabled NIC achieved closed to 100K proxy transactions per second for 8KB download object sizes. This paper also identifies and evaluates major proxy performance tuning design considerations on NUMA architecture.

In the future, we would like to continue our work to evaluate transparent PEP performance on more realistic traffic models. We are also interested in investigating TC scaling as a tool for containerized service orchestration, and XDP scaling as a mean to enforce service bypass rules.

Acknowledgment

We want to thank our colleagues, Damascene Joachimpillai, Mark Richardson, Anh Quach and Rekha Sundararajan, who provided insight and expertise that greatly assisted this project.

References

[1] Alizadeh, M.; Greenberg, A.; Maltz, D. A.; Padhye, J.; Patel, P.; Prabhakar, B.; Sengupta, S.; and Sridharan, M. 2010. Data Center TCP (dctcp). In *ACM SIGCOMM Computer Communication Review*, volume 40, 63–74. ACM.

[2] Brakmo, L. 2010. TCP-NV Congestion Avoidance for Data Centers. In *Proceedings of the Linux Plumbers Conference 2010*.

[3] Cardwell, N.; Cheng, Y.; Gunn, C. S.; Yeganeh, S. H.; and Jacobson, V. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October.

[4] Dukkipati, N.; Mathis, M.; Cheng, Y.; and Ghobadi, M. 2011. Proportional Rate Reduction for TCP. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11.

[5] Flach, T.; Papageorge, P.; Terzis, A.; Pedrosa, L.; Cheng, Y.; Karim, T.; Katz-Bassett, E.; and Govindan, R. 2016. An Internet-Wide Analysis of Traffic Policing. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, 468–482.

[6] http://www.haproxy.org/. HAProxy, The Reliable, High Performance TCP/HTTP Load Balancer.

[7] Li, F.; Chuang, J. W.; and Jiang, X. 2017. Driving TCP Congestion Control Algorithms on Highway. In *Proceedings of Netdev 2.1*.

[8] Nguyen, B.; Banerjee, A.; Gopalakrishnan, V.; Kasera, S.; Lee, S.; Shaikh, A.; and Van der Merwe, J. 2014. Towards Understanding TCP Performance on LTE/EPC Mobile Networks. In *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges*, 41–46.

[9] Snellman, J. 2015. Mobile TCP Optimization: Lessons Learned in Production. Technical report, Telco. Networks.