HA/TCP: A Reliable and Scalable Framework for TCP Network Functions

Haoyu Gu
University of Waterloo

Ali José Mashtizadeh
University of Waterloo

Bernard Wong
University of Waterloo

Abstract

Layer 7 network functions (NFs) are a critical piece of modern network infrastructure. As a result, the scalability and reliability of these NFs are important but challenging because of the complexity of layer 7 NFs. This paper presents HA/TCP, a framework that enables migration and failover of layer 7 NFs. HA/TCP uses a novel replication mechanism to synchronize the state between replicas with low overhead, enabling seamless migration and failover of TCP connections. HA/TCP encapsulates the implementation details into our replicated socket interface to allow developers to easily add high availability to their layer 7 NFs such as WAN accelerators, load balancers, and proxies. Our benchmarks show that HA/TCP provides reliability for a 100 Gbps NF with as little as 0.2% decrease in client throughput. HA/TCP transparently migrates a connection between replicas in 38 µs, including the network latency. We provide reliability to a SOCKS proxy and a WAN accelerator with less than 2% decrease in throughput and a modest increase in CPU usage.

1 Introduction

Middleboxes are ubiquitous in modern networks. Providers often deploy middleboxes or network functions through network function virtualization (NFV) to simplify their management and deployment by freeing them from physical appliances. Layer 7 NFs are a growing area in industry and academia that includes caching proxies [3, 30], TLS termination [40], TCP splicers [14, 36, 37], and WAN accelerators/optimizers [44, 48]. These NFs provide complex network services that process and transform traffic in non-trivial ways at critical points in the network. For example, Cloudflare uses layer 7 proxies to improve performance and security for web services. As a central point for incoming traffic, every outage translates to a financial loss and a reputational loss for both customers and service providers [19, 52].

Layer 7 NFs compound the well known reliability and scalability concerns of users of layer 2–3 NFs [45]. Many layer 7 NFs perform transformations, e.g., traffic encapsulation and TLS termination, that prevent using fail-to-wire to temporarily bypass the NF to avoid service outages. Disruptions of these NFs are problematic as connections are terminated inside the NF [36], making failures visible to end nodes.

To support migration and failover, these NFs must migrate all states including, the TCP stack and any data currently being processed by the NF. Layer 7 NFs are unique because they read-modify-write many state variables throughout their software stack (e.g., TCP, TLS, and NF state) multiple times during traffic processing. State changes must be replicated immediately to allow a replica to take over connections seamlessly when failures occur.

Prior approaches to NF reliability, including those of S6 [47], Stateless-NF [31], and FTC [21], cannot be applied to layer 7 NFs. These prior systems commonly assume that layer 2–3 NFs modify at most a few state variables. Most of these systems store the state remotely for migration and failover, and it is practical because most NFs only access or modify 1–2 state variables per flow or per packet. Layer 7 NFs have complex state that is accessed multiple times throughout the packet processing pipeline, necessitating multiple remote accesses to the state that is more costly than existing systems are designed to handle.

This paper introduces HA/TCP, the first framework to support the migration and failover of TCP-based layer 7 NFs for reliability and multi-node scalability. HA/TCP actively replicates traffic from primary to replica to support seamless failover and migration. We build a high performance IP-based protocol for low overhead traffic replication between primary and replica. HA/TCP introduces *replicated sockets*, a compatible socket API that hides the complexity of replication and eliminates the non-determinism between primary and replica. Replicated sockets simplify reliable NF development because we observed that most NF state derives from the TCP flow.

HA/TCP is highly optimized for modern high speed networks. Our benchmarks show that HA/TCP can saturate a 100 Gbps link while providing reliability. We made many critical optimizations, including eliminating all deep and shallow

copies of packets inside the critical path, relying on TCP itself to correct network errors between the primary and replica, supporting receive side scaling (RSS) and process concurrency, and supporting TCP options important to NF performance. Together these optimizations improve the performance of HA/TCP by more than 10×.

HA/TCP is built on a production grade TCP stack based on FreeBSD 13.1 and runs under DPDK using F-Stack. Our system consists of a TCP/IP stack extension to replicate the connection state, the replicated sockets API that eliminates non-determinism in sockets, and the IP Clustering system negotiates with network switches for load balancing. HA/TCP uses the common address redundancy protocol (CARP [9]) for failure detection and leader election. We implement many optimizations for high throughput and low latency. To evaluate HA/TCP, we built three layer 7 NFs including a WAN accelerator, a SOCKS proxy and a distributed load balancer. Each NF supports HA/TCP in less than 100 lines of code.

Our system makes the following contributions:

- HA/TCP is the first system to provide reliability and scalability for TCP-based layer 7 NFs. We introduce the *replicated sockets* interface that simplifies the development of reliable NFs.
- HA/TCP migrates TCP connections 2.4× faster than Prism [24] and 1.7× faster than Capybara [12] when including network latency.
- The implementation of several systems in addition to HA/TCP including: the first open-source WAN accelerator, our distributed load balancer, and our novel IP clustering system that works with commodity switches.
- Our experience building and optimizing HA/TCP for 100 Gbps networks.

Our evaluation results show that HA/TCP saturates 100 Gbps with just four connections. HA/TCP adds on average 11 µs to the round-trip time and reduces client throughput by as little as 0.2%. Our NF benchmark shows HA/TCP migrates transparently in 38 µs, which is faster than the state-of-the-art TCP migration systems. HA/TCP scales linearly across multiple nodes. The load-balancing benchmark shows HA/TCP evenly distributes load among nodes.

2 Background

The reliability and scalability challenges of layer 7 NFs are unique because their complexity far exceeds other types of NFs. Layer 7 NFs contain a complete TCP stack that terminates connections. A few recent works, Microboxes [36] and mOS [29], introduce frameworks that provide a custom TCP stack with better APIs for developing layer 7 NFs. Neither system provides reliability or multi-node scalability. Our work is complementary and can be used with either system.

Types of Layer 7 NFs. For our purposes, we classify layer 7 NFs into two categories: *pass-through* and *endpointing* NFs. Pass-through NFs, such as an Intrusion Detection System (IDS), are transparent to network connections as they only monitor traffic. They only read the packets that flow through connections and reconstruct the TCP state of each connection locally, but do not alter the packet contents.

Endpointing NFs, such as a TCP splicer, SOCKS proxy and WAN accelerator, terminate the TCP connection. The NF splices the traffic from one connection to another while optionally transforming the payload.

Endpointing NFs contain a complete TCP stack and the NF application. The TCP stack provides full TCP packet processing for the NF application. The TCP connections on either end of the NF terminate at the NF. The NF application works directly on the data that it reads from the TCP stack and sends over another connection.

The failure of pass-through NFs does not affect the connections that they are monitoring. This is because they tap each connection and reconstruct a copy of the minimal TCP state for monitoring purposes. Pass-through NFs use fail-to-wire to allow traffic to continue to flow after a failure.

However, if an endpointing NF fails, all connections through the NF break, because endpointing NFs terminate all TCP connections. A TCP connection is stateful, and the NF relies on having the correct state of both the TCP stack and the NF application. The connections to nodes on either end of the NF cannot be re-established without the NF state, which is lost during a failure.

The Need of Seamless Failover. Prior works have cited that vendors are reluctant to reset network connections because of user visible disruptions [45]. FTMB [45] shows that a number of popular applications come with defaults that cause user visible disruptions and suffer poor recovery times in the range of minutes. A recent Microsoft study stated that NF failures accounted for 42% of severe incidents, and that some failures lasted over a day [41].

NFs such as WAN accelerators are often deployed to efficiently make use of a company's network infrastructure to connect all of their remote sites together. Often they are deployed using a hub and spoke model [13, 43] which means a few NFs in the primary data centers can disrupt traffic nationally or globally for these companies.

There is no equivalent to fail-to-wire for endpointing NFs, because these NFs transform traffic. For example, a WAN accelerator may compress, encrypt and encapsulate a TCP flow over UDP to traverse the internet. Sending the traffic without traversing the NF may leave the destination unable to process the incoming connection that is incompatible, or the intermediate network is unable to forward the traffic.

Existing Replication Frameworks. Existing NF state management systems [20, 21, 31, 32, 42, 47] address the major problems of scalability and reliability for layer 2–3 NFs. These systems either replicate the state among NF instances or store the state on another node. These NFs can scale elastically by migrating flows and seamlessly handling failures.

Existing layer 2–3 NF frameworks show that their state is relatively simple, consisting of a few state variables [21, 31, 47]. These frameworks use this fact to efficiently replicate state updates by coalescing or piggybacking state updates into a smaller set of packets.

The state in layer 2–3 NFs does not update frequently. StatelessNF [31] summarized the states and the access patterns for common layer 2–3 NFs. For most NFs, such as load balancers, firewalls and NATs, the states are updated at the beginning or the ending of the connection, and the state is only read during steady state processing. Other NFs may read and write the state simultaneously, but at most once per packet.

The Problem with State Replication. Existing state replication systems are ill suited for layer 7 NFs for two reasons. First, the TCP NFs have a large complex state. TCP NFs terminate the TCP connection which necessitates replicating a non-trivial state. These NFs terminate both endpoints of the TCP connection to rewrite the TCP traffic. Both the NF application state and TCP stack state are part of the NF state that must be made reliable. Furthermore, all packets inside the NF application are thus part of the state. The resulting state update of a single packet is larger than the packet itself.

Second, the states are updated multiple times throughout the NF pipeline, and attempting to batch state updates leads to correctness or performance issues. All packet data, including the payloads, is part of the NF state. When sending the data in TCP, the TCP stack holds the data in the socket buffer until it receives the acknowledgment from the peer. However, the acknowledgment packet only indicates that the peer TCP stack has processed the packet, but the application may not have started processing. If the peer, say an endpoint NF, fails before the NF application finishes the processing, even if the NF gets failed over, the sender will never retransmit the previously sent packet.

Delaying the TCP acknowledgment until the NF application completes processing and updates its state is problematic. While simplifying replication, it introduces a performance problem for TCP, because the acknowledgment will be delayed. The delay increases the perceived round-trip time (RTT), which increases the TCP window size and the memory requirements of all nodes. More importantly, any variability in the RTT reduces the TCP window depending on the congestion control algorithm. The result is that the end nodes perceive worse network conditions, which hurts performance.

Prior TCP Failover Systems. Unfortunately, prior approaches to TCP failover [2, 49, 50] are not transparent, do not support common TCP extensions, and suffer from high overheads. These systems change the TCP protocol and require nodes to participate in migration or failover events. Several systems use another machine (i.e., a NF) to store traffic to ensure the replica node can resume communications after a failure [4, 16]. Their NF is a single point of failure that must be made reliable, yet they still incur overheads of 30% or more. These systems are unmaintained for over 20 years.

Our Approach. Our approach is to use active replication inside the network stack. We intercept incoming packets just before the TCP stack, transmit them to the replica, and resume TCP processing once the replica has acknowledged reception. Naïvely implementing this approach results in low performance. One of the main challenges of HA/TCP is how to achieve good performance.

One benefit of our approach is that we simplify the development requirement for providing high-availability for NFs by introducing *replicated sockets*. Rather than replicate the NF application state, we maintain output determinism through our replicated socket interface, which means the output from the network stack is identical on all nodes. NF applications need to ensure that they do not generate any externally visible non-determinism. This is similar to the programming discipline that game engines [11] use.

Providing output determinism does not preclude applications internally diverging, for example, the WAN accelerator may take a cache hit on the primary and a miss on the replica because of timing differences. NFs with unavoidable internal non-determinism must ensure that the output remains identical. In this example, the replica that misses, would fetch the value from its peer, resulting in identical output. Alternatively, one could use record/replay [38, 45] to synchronize state.

3 Design

TCP NFs using HA/TCP are deployed similarly to other NFs. In an NFV environment, we use an SDN controller (e.g., ONOS [7]) to manage the deployment of NF instances and coordinate traffic steering. The SDN controller coordinates with the network function's controller, which uses our novel IP Clustering system to manage the network function and traffic routing. Our main contribution is a TCP stack extension that supports the migration and failover of individual flows between NFs.

HA/TCP has three main goals:

1. **Transparent to Remote Nodes:** We cannot make protocol changes that require modifying existing TCP stacks in client nodes.

2. **Low Overhead:** While network reliability is important, users are not willing to sacrifice performance. The end-to-end impact on throughput and latency must be low to make HA/TCP practical.

3. **Fast Failover and Migration:** Failover and migrations must complete quickly to avoid user visible disruptions.

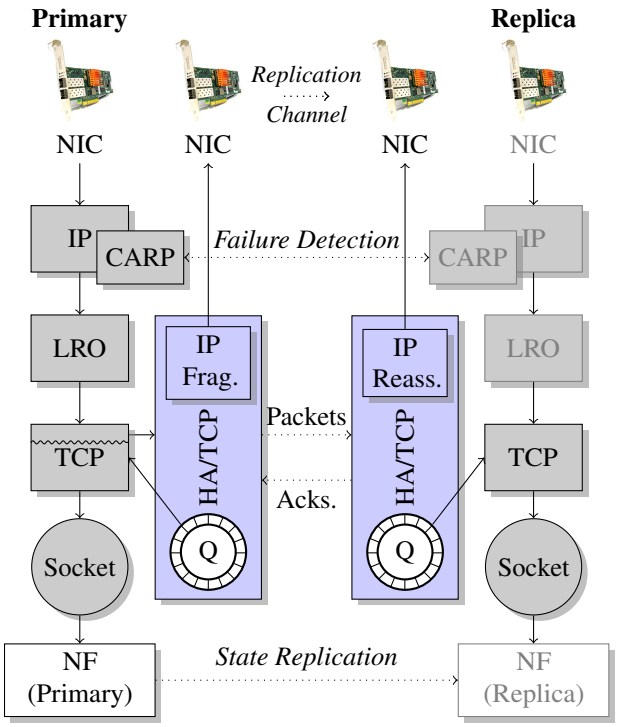

Figure 1: HA/TCP system architecture.

NFs using the HA/TCP framework, support migration and failover using our *replicated socket* interface. Our replicated socket provides a socket object that behaves identically on both primary and replica. In the rest of the design, we explain the operation of our replicated sockets, which we use to create reliable NFs. The most difficult challenge is to replicate the TCP stack to ensure that it behaves identically on both the primary and replica. In § 3.8 we discuss our extensions to CARP and our IP clustering protocol that provides additional functionality to the NFs.

3.1 Overview

Figure 1 shows the HA/TCP system architecture. HA/TCP intercepts and queues incoming packets from the TCP stack after *large receive offload* (LRO) and checksum validation. HA/TCP sends the TCP traffic and non-deterministic state (e.g., congestion control variables) over the replication channel. Both primary and replica have copies of the same incoming packet in their respective HA/TCP queue. After the replica acknowledges receiving the packet, both sides dequeue the packet and begin TCP processing. If the primary fails, CARP notifies HA/TCP to fail over the connection.

To achieve the full duplex throughput of our primary network interface, we dedicate a network interface for the replication channel between nodes. We use dual ported network interface cards (NICs), which are marginally more expensive than a single ported NICs. Our design uses active replication to a single replica. When a node fails, a new replica is deployed through NFV. Alternatively, HA/TCP supports multiple active replicas, see § 3.8.

Our performance goal is to support 100 Gbps network cards which requires examining many performance issues throughout the network stack. We can classify our optimizations into four categories: First, reducing overhead by processing larger packets. Second, eliminating memory allocations and copies throughout our code. Third, hiding replication latency. Fourth, supporting RSS for multithreaded application scalability.

HA/TCP creates listening and connected sockets that behave identically on the primary and replica, including all TCP states. The NF only needs to ensure that it identically transforms the data on the primary and replica.

Applications use `setsockopt` and `getsockopt` to enable and configure HA/TCP. A typical workflow is to enable HA/TCP on a listening socket, and any accepted connections will inherit the configuration. The listening socket may accept spoofed connections where the NF uses `getsockname` and `getpeername` to retrieve the endpoint addresses.

When the primary fails, the connection seamlessly fails over to the replica. The replica takes over the client connection and becomes the new primary. The new primary exchanges the traffic directly over the local NIC rather than the replication channel. In addition, HA/TCP provides the ability to migrate sockets to other nodes as needed.

3.2 Replicated Sockets

HA/TCP's replicated socket interface extends network sockets to provide a socket object that behaves identically on the primary and replica. Both listening and connected sockets are replicated between nodes. A common workflow for our NFs is to enable HA/TCP on a listening socket, which each accepted socket inherits.

Connected Sockets: After enabling HA/TCP, the primary continues to process traffic until a replica joins. Once a replica joins, the primary temporarily pauses the traffic and synchronizes all TCP state with the replica. Once complete, the primary resumes the client traffic.

HA/TCP Replication Channel: HA/TCP exchanges state and packet data over the replication channel, which is an IP protocol that encapsulates these packets with a small header. Our system implements and registers an IP protocol with the network stack that allows us to process packets in-line. For example, on the replica we decapsulate packets and forward them to the usual TCP input processing path as if they were received from the client directly.

The IP protocol is simpler and more efficient than using layer 4 protocols such as UDP or TCP for replication. The IP layer does not look up control blocks, update the internal state, acquire any locks nor apply congestion control, which lowers overhead and simplifies our design. Furthermore, modern NICs support IP checksum offloading.

The IP layer offers several other benefits over layer 2 protocols such as Ethernet. The IP layer provides IP fragmentation and reassembly to exchange large packets in MTU sized chunks. IP protocols are routable and can traverse many network devices including over metropolitan area networks.

As a performance optimization (see § 3.5), we do not handle packet loss in our replication channel. Instead, we rely on the clients to time out and retransmit to the primary and replica. Without an acknowledgment from the replica, the primary will not acknowledge the TCP packet and thus the client will retransmit the packets.

In an earlier design, we used a TCP connection for the replication channel, which led to performance issues from high CPU usage and the TCP meltdown problem [27]. This doubles most of the network stack overhead as every incoming packet traverses the primary's NIC driver through the TCP stack and out the replication interface's NIC driver to the replica. Similarly, the replica processes TCP twice before forwarding data to the application.

TCP meltdown is a problem that occurs when running TCP over TCP. It arises because HA/TCP holds packets on the primary until the replica acknowledges reception. If a packet is held longer than the client's retransmission timeout, the client treats this as packet loss and retransmits the packet. For example, the replication channel may delay delivery because of congestion control, which triggers a cascade of retransmissions by the client that exacerbates the issue.

Avoiding IP Reassembly Collisions: The challenge of using IP for the replication channel is that IP fragmentation suffers from reassembly collisions [26]. The IP layer breaks up larger packets into MTU sized chunks. At high throughput, packets may be incorrectly reassembled because of collisions with the fragment identifier. HA/TCP discards incorrectly assembled packets, which results in the client observing packet loss. The packet loss causes client retransmissions, which reduces application performance.

IP reassembly uses the 16-bit *IP identification* field and the 13-bit *IP offset* field to identify fragments belonging to the same packet and reassemble them into the original packet. Packets that are retransmitted because of loss or timeouts pollute the IP reassembly queue with old fragments. Usually, this is handled by clearing the IP reassembly buffer every 30 seconds. At high throughput, this is insufficient and we see periodic collisions.

To solve this we added an IP option containing a 32-bit unique ID and 32-bit timestamp to eliminate collisions. Each replicated TCP connection is assigned its own unique ID to avoid collisions between different connections.

Listening Sockets: HA/TCP replicates listening sockets to the replica. Upon establishing the replication channel on the replica, all states in the socket object and TCP stack are synchronized. The replica then copies and applies all states to the corresponding object.

HA/TCP ensures that new connections are created identically on both primary and replica. When a new connection arrives, the primary forwards the 3-way handshake packets to the replica.

First, the SYN packet, the primary's initial sequence number, and a time offset for TCP timestamps are forwarded to the replica. This packet is processed normally by the TCP input path except that it uses the initial sequence number from the primary.

Second, after receiving an acknowledgment from the replica, the primary generates and sends a SYN-ACK packet back to the client.

Third, the primary replicates the ACK message from the client and the initial window size. We replicate the initial congestion window size in case TCP fast open or the host cache provides initial values. The replica processes the TCP ACK normally.

HA/TCP inherits the replication configuration for newly accepted sockets. Before returning from `accept`, HA/TCP initializes and activates replication on the newly created socket, before the application uses the socket.

3.3 Steady State Processing on Primary

In normal operation, packet processing on the primary undergoes five steps.

First, HA/TCP intercepts incoming packets inside of the TCP input path. After the TCP input performs the checksum validation and looks up the TCP control block, which filters out any corrupt packets or packets not belonging to established connections, HA/TCP intercepts the packet.

Second, HA/TCP places the original packets in a queue. The packet mbufs and the TCP control block are stored to allow the resumption of the TCP processing without repeating the work completed in the first step.

Some time-based state is modified by buffering. The round trip time increases by the additional latency of waiting for the replica's acknowledgment. For some congestion control algorithms, we need the replica to respond with near constant latency otherwise the algorithm may believe there is congestion in the network. To avoid accidental duplicate packet detections, we deliver packets in order so the packet timestamps [8] are monotonic (see § 3.7).

Third, HA/TCP duplicates and transmits the packet to each replica. The duplication is necessary because when the packet is sent through the replication channel, the network stack frees the packet mbufs. As an optimization, we merge packet duplication with IP fragmentation to reduce overhead (see § 3.5). HA/TCP prepends a small header containing the packet size and, critically, any congestion control updates including window size changes (see § 3.4). The packets are then transmitted over the replication channel to the replica nodes.

Fourth, HA/TCP waits for an acknowledgment from the replica. While waiting for the replica to reply, HA/TCP continues to process incoming packets from the client.

Fifth, HA/TCP releases the queued packet to the TCP stack once the replica acknowledges the packet. The packet resumes processing in the TCP stack, where it left off in the first step.

3.4 Steady State Processing on Replica

The replica socket replicates the state and traffic from the primary. HA/TCP simulates the interface and TCP behaviors until a failover or migration event occurs. Critically, for good performance, the replica acknowledges packets back to the primary in constant time to avoid triggering client retransmissions and congestion control. In normal operation, packet processing on the replica also undergoes five steps.

First, HA/TCP receives a packet from the replication channel and parses the HA/TCP header. If the packet contains a replicated packet, HA/TCP removes the header and acknowledges back to the primary. All processing in this step occurs on the NIC thread associated with the replication channel. By acknowledging immediately, we eliminate any timing jitter from queuing or TCP processing. Otherwise, TCP congestion control may reduce the window because of timing variance.

After the replica acknowledges the packet, the primary may send a TCP acknowledgment to the client. The replica is required to keep all packets it has acknowledged regardless of whether the TCP stack is ready to process them. HA/TCP uses a queue to store any packets that are not ready to be processed by the TCP stack. For example, the client may acknowledge a packet from the primary that the replica has yet to transmit. In this case, the acknowledgment number is in the future and the TCP stack would discard it.

Second, HA/TCP enqueues the packet until it is ready to be processed by the replica TCP stack. The queue masks performance differences between primary and replica. When a replicated packet arrives, HA/TCP checks if the acknowledgment number is within the maximum sequence number sent, and only delivers the packet when this condition is met.

The queue length trades off tolerance for the performance differences between nodes and the failover time. Large queues better mask performance variability and tail latency in the replica application. However, the large queue increases the switchover time during the failover process because the queue must be drained before the replica assumes the role of primary. The queue size can be tuned for a specific workload and we limit the queue size to the receive socket buffer size.

HA/TCP monitors the queuing level on the replica and throttles the incoming traffic if the replica queue is saturated. The replica acknowledgment contains the queue size that the primary uses to check whether the replica is keeping up. When the replica falls behind and the queue fills up, which should not happen in a properly configured environment, the primary throttles to avoid overflowing the replica queue.

Third, HA/TCP checks the packet header to determine if any time-based state must be synchronized. The RTT estimates and window sizes are synchronized between nodes as needed to prevent inadvertently rejecting packets.

Fourth, HA/TCP iterates through all packets queued for delivery. If the condition mentioned in the second step is met, HA/TCP delivers the packet to the TCP stack; otherwise it moves to the next packet in the queue. For example, if packet loss occurs between the primary and client, the packets at the front of the queue will not match the TCP state. Without iterating through the queue, the retransmitted packets at the tail will block further processing.

Fifth, for outgoing traffic, HA/TCP inspects and discards the packets after all TCP processing is complete. Although HA/TCP frees the packet, the content of this packet resides in the socket buffer until the replica receives the TCP acknowledgment from the client. The packet can be retransmitted from the socket buffer. HA/TCP also attempts to deliver packets from the queue, after the TCP output code path transmits a packet. This delivers any packets that did not previously meet the condition (from the second step).

3.5 Performance Optimizations

HA/TCP is optimized for high performance network interfaces. In this section, we discuss each optimization and show the performance breakdowns. Table 1 shows the performance as we enable optimizations one at a time.

HA/TCP has three major data processing paths.

1. INPUT: The input path includes enqueuing the packets and transmitting them through the replication channel.

2. DELIVERY: Receives the replica acknowledgments and releases packets to the TCP stack.

3. REPLICA: The input path on the replica.

In HA/TCP, the INPUT path contains most of the high overhead operations. The DELIVERY path and REPLICA paths are similar to the baseline TCP network path. The total system overhead from HA/TCP consists of all three paths from all nodes. The overhead of each path is shown to make the effect of the optimization clear.

Recall from § 3.2, our early design uses TCP as the transport for our replication channel. Columns 2–4 in Table 1 use the TCP-based replication channel, while the remaining columns use the IP-based replication channel.

Use TCP LRO to Reduce Overhead. HA/TCP is optimized for the standard MTU of 1500 bytes to support user facing services in the cloud or data centers. In our experience, the FreeBSD TCP stack consumes CPU proportional to the packets per second (PPS). Thus, reducing the number of packets reduces the processing cost. LRO merges multiple TCP packets into a single packet earlier in the network stack to eliminate the cost of processing many TCP headers. For large flows, LRO can combine packets into 64 KiB packets.

		Unreplicated	No Opt.	w/LRO	w/LRO+NoCopy	IP+LRO	w/NoCopy+ReAsm
	Throughput	93.60 Gbps	8.60 Gbps	53.39 Gbps	73.13 Gbps	54.23 Gbps	90.50 Gbps
	Pkt/s	7.80 Mpps	0.72 Mpps	4.45 Mpps	6.09 Mpps	4.52 Mpps	7.54 Mpps
Primary	CPU INPUT	79 %	94 %	91 %	94 %	90 %	89 %
Primary	CPU DELIVERY	-	53 %	66 %	63 %	28 %	28 %
Primary	CPU APP	61 %	35 %	52 %	59 %	54 %	71 %
Repl.	CPU REPLICA	-	67 %	39 %	52 %	43 %	71 %
Repl.	CPU APP	-	47 %	42 %	56 %	38 %	60 %

Table 1: Shows HA/TCP with optimizations turned on/off. The first column shows the performance without replication. Columns two through four uses our TCP-based replication channel, and the last two columns uses our IP-based replication channel.

Without any optimization, the INPUT path is the performance bottleneck with two sources of overhead.[1] First, the INPUT path spends 9.7% of CPU time processing the incoming packets. This is roughly 0.72 M PPS with the default MTU of 1500 bytes. Without optimizations, CPU overhead scales linearly with the packets per second.

Second, the INPUT path consumes 5.3% of CPU time duplicating network packets using `m_dup`. The CPU time scales with the number of packets and the size of the payload, because `m_dup` deep copies the packet headers and data.

Enabling LRO chains consecutive packets into larger packets of up to 64 KiB in size through mbuf chaining, i.e., without deep copies. LRO trades off the processing overhead of identifying consecutive packets in a stream with the per-packet processing costs throughout the TCP stack. Furthermore, LRO piggybacks on interrupt coalescing to chain packets that are delivered in the same interrupt without increasing latency. Potentially LRO can use hardware offloads, but in the NICs we tested this performed worse.

The average packet traversing the network stack with LRO is 61400 bytes in size, resulting in a 40× reduction in effective PPS. While HA/TCP saves the per-packet CPU processing and packet duplication overheads with LRO, LRO itself adds overhead. However, the performance gain more than offsets the cost of LRO. LRO improves HA/TCP throughput by 6.2×. In our measurements, LRO slightly decreased latency because of the substantial increase in throughput.

Avoid Deep Copies of mbufs on the Primary. Our replication protocol prepends its headers to the packet for transmission. HA/TCP duplicates every incoming packet to retain an unmodified copy for the local TCP stack.

Even after enabling LRO, the INPUT path remains the performance bottleneck. The deep copying of mbufs using `m_dup` consumes 12% of CPU. To address this, we use `m_copypacket` to perform a shallow copy. The function duplicates mbuf headers and reference counts the payload. Shallow copying reduces memory copies and cuts the number of allocation/deallocations in half. Switching to `m_copypacket` improves throughput and reduces CPU usage.

Use IP-based Protocol for the Replication Channel. In § 3.2, we mentioned HA/TCP uses an IP-based protocol for the replication channel. Compared to a TCP-based transport, the IP protocol is simpler and more efficient.

With the optimizations we mentioned above, we use our original TCP-based replication channel. We found that the INPUT path consumes 8% of the CPU time on TCP encapsulation and processing. Our TCP-based replication channel reduces the client throughput by 28% to 73 Gbps as compared to an unreplicated setup. Furthermore, TCP over TCP suffers from TCP meltdown problem (see § 3.2).

Next, we implement an IP-based replication protocol. We compare the IP-based and TCP-based protocols with LRO enabled. This rules out effects from other dependent optimizations (see next section).

The IP-based protocol simplifies the locking design. Sending and receiving traffic in the TCP-based replication channel goes through the TCP stack, which locks the TCP control block. In addition, reading or writing the socket buffer for the replication channel requires an additional lock. By switching to an IP-based protocol, we eliminate these locks; moreover, this simplifies the design and avoids lock ranking problems that may lead to deadlocks.

Both versions are bottlenecked on the INPUT path because of the packet duplication function. With the IP-based protocol, the throughput only increases marginally. Turning on the shallow copying optimization significantly improves the performance but it requires the next optimization.

Create Replica Copies with IP Fragmentation. Our IP-based replication protocol depends on IP fragmentation to split large packets into smaller MTU sized packets, which replaces TSO (TCP segmentation offloading) in our TCP-based protocol. IP fragmentation is required as LRO merges small packets up to 64 KiB in size and this size far exceeds the jumbo MTU size of the replication channel.

Our next major source of overhead arises from the deep packet copies caused by IP fragmentation. When transferring

[1] Adaptive polling is disabled in our configuration. The CPU spends 5–10% on CPU scheduling and waiting for interrupts. We confirmed by monitoring the interrupt rate and using CPU profiling.

the replicated packet through the replication channel, IP fragmentation breaks the packet to fit the MTU of the replication channel. The IP fragmentation process calls the packet duplication function, which allocates and copies the mbuf. The MTU between the primary and the client is by default 1500 bytes, while the replication channel uses a 9k MTU. LRO merges small packets into a 64 KiB packet, and subsequently these large packets are broken into 9k jumbo packets.

To optimize IP fragmentation, HA/TCP unifies packet duplication and IP fragmentation. The unified IP fragmentation path only copies the packet once to simultaneously fragment the large packet into the replication channel and retain a reference to the original mbuf chain for retries and delivery to the TCP stack. Merging these two functions with shallow copying reduces the overhead. Combining these functions improves client throughput by 67% over the LRO optimization alone.

3.6 Additional Optimizations

We also added a few optimizations that we do not include in our performance breakdown. These optimizations reduce CPU usage on non-bottlenecked paths, avoid unpredictable behavior or support performance features of TCP/IP.

Allow Client to Repair Packet Loss. We expect packet loss to be more common between the client and the primary than over the replication channel. We rely on the client to retransmit any lost packets. Recall that the primary only processes the packet after receiving the acknowledgment from the replica. Any packet loss on the client-primary connection or replication channel causes the primary to not acknowledge the packet, resulting in a client retransmission. Packet loss in a data center, or even across a metro area network, is rare when compared to the wide area network, and thus having the client retransmit packets negligibly impacts performance.

Eliminate Delayed Processing on the Replica. We disable Naggle's algorithm and TCP delayed acknowledgments on the replica to reduce buffering. Both optimizations delay and batch network packets to reduce the overhead in the network stack. However, the delay results in more packet buffering and slows down the TCP state updates. The falling behind TCP state on replica leads to more buffering in the queue, and thus affects the performance of failover and migration.

Support Receive Side Scaling. RSS is a NIC feature that load balances traffic across CPU cores for performance. Connections are routed to a specific core based on a hash of the 5-tuple. Unfortunately, IP protocols are hashed based on their 3-tuple of IPs and protocol numbers resulting in a miss match between flows and application threads. We use Intel Flow Director [15] or Mellanox Flow Steering to maintain a one-to-one match between cores on the primary and replica. We use a range of IP protocol numbers so that the flow rules can forward the replication channel traffic to the correct core.

3.7 Challenges with TCP options

HA/TCP supports modern TCP options. Failing to support these features leads to performance or correctness issues.

Concurrent Listening Ports (`SO_REUSEPORT`). Multi-process or multi-threaded applications often create multiple listening sockets on the same port with `SO_REUSEPORT(_LB)`. To ensure deterministic behavior we establish a one-to-one mapping of listening sockets on the primary and replica.

SYN Cache. HA/TCP handles randomness from the TCP handshake. When the primary receives the SYN packet, the SYN cache generates random states for security purposes. These random states are sent to the replica along with the SYN packet. After the replica acknowledges the replication of this SYN packet, the primary delivers the packet again to the SYN cache for the client to respond. However, the primary needs to reapply the same random states to make sure the SYN cache generates the identical states.

Syncookie and TCP Fast Open (TFO). The syncookie and TFO use a cryptographic secret and a MAC function to allow the client to store cached values from the server. To ensure both TCP options function correctly, we synchronize these two secrets between the primary and replica.

SACK on the Replica. HA/TCP guarantees the state of SACK is identical on the primary and replica after a failover or migration. The replica delivers queued packets only when the acknowledgment number refers to packets that have already been "sent" by the replica (§ 3.4). During the migration, HA/TCP drains all remaining packets into the TCP stack, and missing packets will result in the same SACK state.

Synchronizing Clocks for PAWS. TCP PAWS [8] uses TCP timestamps to detect retransmitted packets and discards the packet. To support PAWS, HA/TCP tracks the time difference between primary and replica (`ts_offset`) and delivers the packet with an updated TCP timestamp adjusted by the time difference. The PAWS check is completed on the primary and for valid packets, the timestamp offset will ensure that the PAWS check will pass on the replica. We periodically synchronize the clocks to avoid drift.

Congestion Control Algorithms. We tested HA/TCP with both loss-based and RTT-based congestion control algorithms and found no significant performance differences. For loss-based algorithms, such as NewReno [22], DCTCP [6], and H-TCP [35], HA/TCP does not alter the retransmission mechanism in TCP, nor the congestion detection that the algorithms rely on. For RTT-based congestion control algorithms, such as HD [10] and CHD [25] that monitor network latency and latency variance to detect congestion, both of which are relatively stable. HA/TCP, and in particular the IP replication channel, is optimized to reduce the processing overhead, which results in a low and constant increase in latency (§ 5.4).

3.8 Load Balancing, Failover and Migration

HA/TCP uses the Common Address Redundancy Protocol (CARP) [9] protocol (based on VRRP [39]) to detect node failures and perform leader election, and our novel IP Clustering system to share a single IP address across nodes for scalability. We also discuss migration and having multiple replicas at the end of this section.

Failover (CARP). Each CARP node sends a periodic multicast advertisement. If three consecutive advertisements are not received from the primary, CARP declares the primary as failed. CARP then promotes a replica to be the new primary, and transmits an ARP announcement to notify the switch of the migration of the IP and MAC address to the new port.

In this paper, we set the advertisement interval to 100 ms, which results in a 300 ms detection and recovery time that is less than most application and TCP timeouts. Identifying the proper trade-off between detection time and false positives is beyond the scope of this paper.

HA/TCP fails over with minimal disruption. During steady state, HA/TCP replicates the incoming traffic to all replicas. Although replicas may lag behind the primary, our measurements show that the delay is only in the tens of milliseconds. The CARP failure detection time masks this lag and allows the replica to drain the queue before the failover.

IP Clustering. HA/TCP's IP clustering allows multiple nodes to share the same IP address for scalability. Our IP clustering system uses the link aggregation control protocol (LACP) to negotiate load balancing with the switch [1]. LACP allows one node to have multiple network interfaces to a switch, but we developed a distributed version of LACP that allows multiple nodes to share the same IP and MAC address.

Our distributed LACP protocol communicates between nodes in a cluster to load balance traffic between the switch and the nodes. The challenge is that HA/TCP's distributed LACP needs to synchronize the LACP state and ARP state between nodes continually. All the nodes must present themselves to the switch as a single node with several state fields being synchronized. ARP messages from the switch are only routed to a single node, leaving other nodes unaware of the ARP traffic. We use the replication interface to synchronize the LACP state and the ARP entries among all nodes.

Currently, IP clustering is supported without replication, because it requires additional application orchestration to manage the placement of per-connection primary-replica pairs. With complete orchestration, our IP clustering system will subsume CARP's functionality.

Migration. Migrations are driven by the primary, which stops processing the client traffic, sends a migrate command to the new primary, and demotes itself to a replica. The new primary sends the promotion message to the replica, i.e., the previous primary, to confirm the migration completion. Once complete, the replica updates its state and the new primary begins processing traffic.

Implementation	Count
HA/TCP Kernel Code	10K SLOC
HA/TCP IP Clustering	1.4K SLOC
SOCKS Proxy	3.3K SLOC
WAN Accelerator	8.7K SLOC
Distributed Load-balancer	1.2K SLOC
Latency Prober	1.9K SLOC
Multi-protocol Load Generator	1.9K SLOC

Table 2: Approximate source lines of code (SLOC) for HA/TCP and our NF applications.

Multiple Replicas. HA/TCP supports multiple replicas. Given the resource requirements of active replication between the primary and replica, it is recommended to use NFV to deploy a new replica if a node goes down. This eliminates the resource consumption of having another full replica, and the extra demands on the primary to keep them synchronized.

HA/TCP supports multiple active replicas with unicast connections for the replication channel. The primary requires additional dedicated network interfaces if it wants to serve the full available bandwidth of its primary interface. Alternatively, multicast can be used to eliminate the need for additional network interfaces with low latency and low packet loss, but it has not been implemented.

4 Implementation

HA/TCP is implemented in the FreeBSD 13.1 network stack. This is a popular operating system for network appliances and the basis of F-Stack, a TCP/IP stack for DPDK. Table 2 shows the breakdown of lines of code for HA/TCP and the NFs. The HA/TCP implementation consists of 8.8K source lines of code (SLOC) that extend the TCP stack, 100 SLOC in CARP, and 1.6K SLOC in the socket API. The IP Clustering system consists of 1.4K SLOC. The SOCKS proxy, WAN accelerator and load balancer consist of 13.2K SLOC. In addition, the SOCKS proxy and multiprotocol client conform to RFC1928 [34], RFC7230 [17] and RFC7231 [18].

5 Evaluation

We evaluate HA/TCP with micro-benchmarks and three TCP NF applications. The micro-benchmarks measure the scalability, migration/failover time, replication overhead, and latency of HA/TCP. For the three TCP NF applications we show the throughput and CPU usage.

In all benchmarks, the throughput is measured from the client application. HA/TCP replicates and forwards the traffic to the replica, and thus generates traffic in the replication channel equal to the client traffic. The replication channel uses

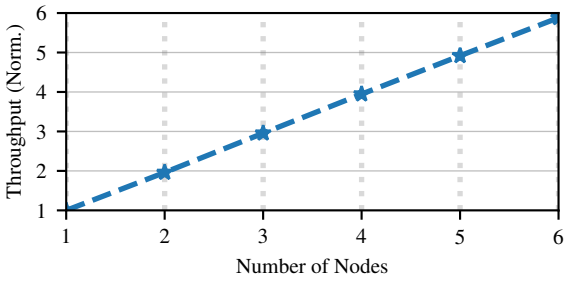

Figure 2: HA/TCP scales linearly with number of nodes while sharing a single IP and MAC address. HA/TCP load balances connections through our IP Clustering system. The throughput with 6 nodes is 2% lower than ideal.

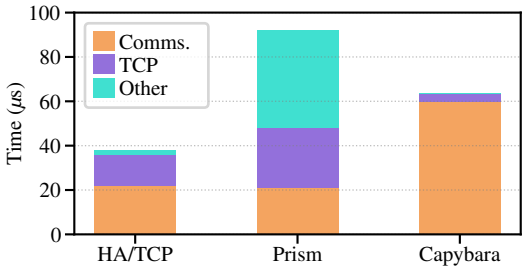

Figure 3: Migration time breakdown including network latency. HA/TCP takes 16 µs to complete the migration process, while the network latency adds the remaining 22 µs. HA/TCP is faster than the state-of-the-art systems.

a dedicated network interface to support the full bandwidth of the primary interface for client traffic.

We measure the CPU usage in the INPUT, DELIVERY and REPLICA paths (see § 3.5), along with the application CPU usage including both user and kernel time. We present a breakdown of the CPU usage for the baseline, primary and replica. The total CPU overhead is the percentage increase in CPU from both the primary and replica.

To minimize variance in our measurements, we pinned each context to a separate CPU and presented the total CPU usage. We used multiple tools to monitor system performance and hardware performance counters to breakdown the CPU usage. This is the same process we used during development.

All measurements were done on a local area network with low latency between all clients, primary and replica nodes. This represents the worst-case, as the latency between the client and primary is the same as the latency between the primary and replica. Over wide area networks, the extra latency induced by replication is negligible when compared to WAN latencies. In all benchmarks we used a single replica.

The primary and replica ran on two dual socket Intel Xeon Gold 6342 processor machines. All servers and clients use a dual-ported 100 Gbps Mellanox ConnectX-6 Ethernet NICs connected to a 100 Gbps Mellanox switch. The MTU between client and primary is set to the standard 1500 bytes, while the MTU between nodes is 9000 bytes. The TCP stack uses the New Reno congestion control algorithm. For failover tests, we set the advertisement interval (`adv_base`) to 100 ms to allow CARP to detect failures on average in 300 ms (§ 3.8).

5.1 Scalability

Figure 2 shows the scalability of HA/TCP's IP clustering. Our IP clustering allows NFs to share a single IP address across any number of nodes. The scalability is only limited by the switch's ability to load balance traffic. In this benchmark, we had the clients send traffic to six Intel Xeon Silver 4116 nodes each with a 25 Gbps interface. We use these nodes because we have a limited number of nodes with 100 Gbps NICs.

5.2 Migration and Failover

Figure 3 shows the breakdown of the migration time for HA/TCP, Prism [24] and Capybara [12]. To provide a fair comparison, we breakdown the migration time for the TCP/IP state, and omit system specific features such as application level migration that not all systems support. For each system, we breakdown the migration into three main costs: *Comms.* that includes any synchronization between nodes, *TCP* that includes the serialization and reconfiguration costs, and *Other* that contains other system specific costs.

Migrations are initiated by the source. The migrate message is sent through the replication channel to the destination in 22 µs. The communication latency dominates the total migration time. Afterwards, HA/TCP updates the TCP control block with the new IP address in 5 µs. The control block is stored in a hash table indexed by the connection's 5-tuple. HA/TCP removes and re-inserts the control block into the hash table using the new 5-tuple value. After updating the control block, HA/TCP flushes any buffered packets in 3 µs. Finally, HA/TCP re-enables the TCP processing in 2 µs.

Prism takes 92 µs if we consider only TCP operations. Prism serializes and reconfigures TCP in 27 µs. Prism takes 21 µs to communicate once to exchange the TCP state. The *Other* in Prism includes the traffic blocking and rule rewriting, which takes 44 µs to complete.

Capybara transfers the TCP state in 3.7 µs, which is much faster than HA/TCP and Prism because it uses a library OS with a custom TCP stack. HA/TCP and Prism are based on commercially used networking stacks (FreeBSD and Linux), which are substantially more complex to reconfigure. While Capybara's state transfer is fast, it communicates over the network three times. Capybara completes the migration in 64 µs including the three round trips over the network.

The failover process is identical to the migration process except that it is triggered by CARP. The failover time depends on the CARP detection time. As mentioned in § 3.8, we configured CARP to detect failures on average in 300 ms. HA/TCP finishes the migration 13 µs after failure detection.

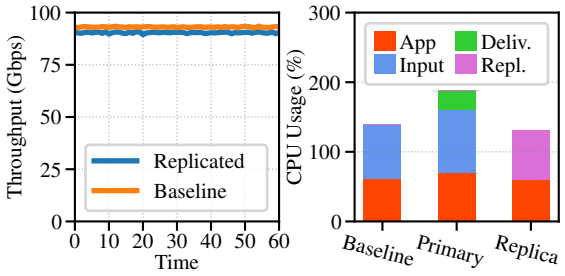

Figure 4: Replication overheads for iPerf3 with the receive-bound traffic configuration. The increase in latency results in a decrease of 3.4% in throughput.

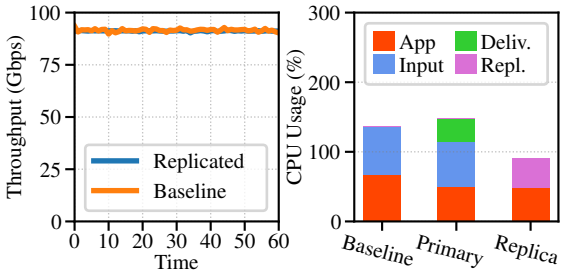

Figure 5: Replication overheads for iPerf3 with the transmit-bound traffic configuration. The throughput and CPU overheads are low because HA/TCP is mainly doing bookkeeping and replicating acknowledgments.

5.3 Replication Overhead

We ran iPerf3 in transmit-bound and receive-bound configurations to measure the overhead of the transmit and receive paths. The transmit-bound configuration has the client send data to the server, while the receive-bound configuration has the server send the data to the client.

Figures 4 and 5 show the results for the receive-bound and transmit-bound configurations. The replication channel adds 116 bytes of metadata to every packet. For iPerf3, LRO reconstructs mostly 64 KiB packets that are fragmented by our IP fragmentation code. The replication channel header should result in a 0.2% decrease in peak throughput if the workload is saturating the 100 Gbps link.

The receive-bound configuration reduces throughput by 3.4%, because of the higher input path processing costs and the additional latency of waiting for the acknowledgment from the replica, before completing TCP processing. In our measurements, we found that the input path is the bottleneck with over 90% of a single core used per connection.

The transmit-bound configuration reduces throughput by 0.3%, because the primary mostly performs bookkeeping to track which outgoing packets are ready to be sent to the client. Application CPU usage is lower than the baseline, because the longer latency taken for application acknowledgments results in more buffering in the outgoing socket buffer. The increase in latency causes the application to block more often. In response, the application increases the amount of data sent in subsequent calls, which amortizes the system call overhead.

HA/TCP uses additional memory for replication. During the steady state, both the primary and replica hold the replicated packet in the packet queue until the conditions are met. The memory overhead is small because HA/TCP reduces memory consumption through shallow copying.

In our setup, under the maximum throughput of 100 Gbps the RTT of the replication channel acknowledgments was approximately 70 µs resulting in a peak memory usage of 875 KiB. The peak memory usage on the primary is derived from the bandwidth delay product of the replication channel. We expect the replica to use a similar amount of memory.

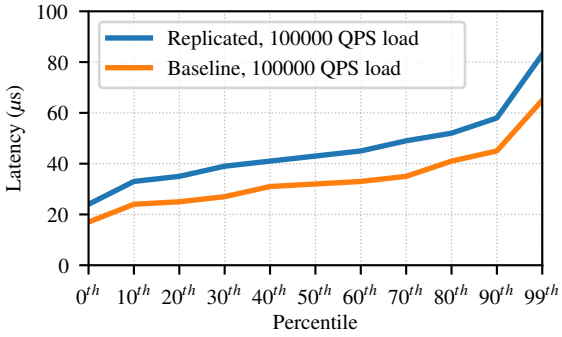

Figure 6: Latency overheads for replication. The client probes the latency by sending requests at 1000 QPS while the server is serving a 100k QPS workload. The latency increase reflects the delay in replicating packets to the replica. We achieve consistent low latency by avoiding the latency of waiting for TCP processing on the replica.

5.4 Latency Overhead

We evaluated the latency response using a distributed workload simulator for key-value stores [56]. In this benchmark, two clients connect to the server simultaneously. One client generates 100k queries per second (QPS) while another client measures the server response time by issuing 1k QPS. The results are an average of ten runs for each configuration.

Figure 6 shows the application latency with and without replication. The latency increases 11 µs on average.

The latency is lower than twice the RTT because the RTT in the replication channel is lower than the RTT between the client and the primary. We achieve this low replication channel RTT by having the replica send acknowledgments inline in the network stack, and our optimizations, including using our IP-based replication protocol, which reduce the CPU usage. Furthermore, the replica buffer hides processing delays to avoid increasing tail latency.

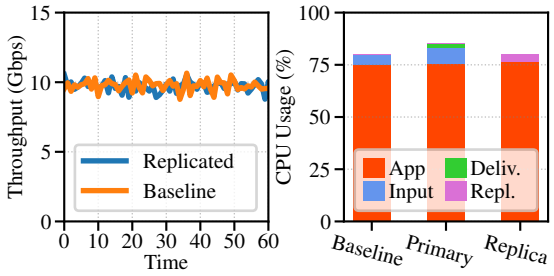

Figure 7: Replication overheads for the WAN accelerator. This workload is CPU and latency bound.

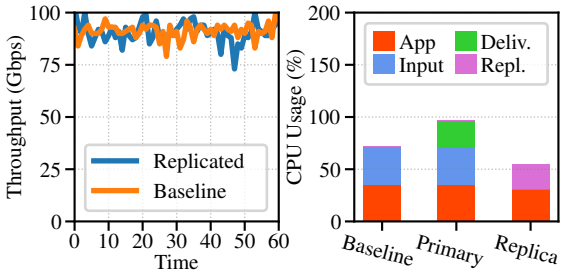

Figure 8: Replication overhead for the SOCKS proxy that shows a decrease in throughput of 2%. This workload is transmit-bound with small incoming requests and large responses. The higher variance in throughput comes from the replica's queuing mechanism delaying a substantial amount of response traffic from the server.

5.5 WAN Accelerator

The WAN accelerator is our most complex NF. WAN accelerators often serve as permanent VPNs connecting two or more data centers, but with the addition of compressing and deduplicating traffic to increase the effective bandwidth. They also aggregate traffic and send them through fewer connections between instances.

We found no open-source WAN accelerators and developed our own that will be available for others to use. WAN accelerators have very high CPU requirements because of the compression and deduplication, but commercial ones also provide protocol specific optimizations that require more CPU. Our accelerator can support 9.7 Gbps, which is faster than commercially available accelerators [48].

The WAN accelerator is configured to deduplicate and compress traffic before encapsulating and sending the traffic to the remote accelerator. This benchmark is unique because we replicate both sides of the WAN accelerator. During the test, the client creates eight long-live connections and issues requests to fetch a 10 MiB size file from the HTTP server through the accelerator. The accelerator uses eight cores for traffic deduplication and eight cores for traffic compression.

Throughput Overheads. Figure 7 shows the throughput. We apply HA/TCP to both the connection between accelerator peers and the connection between the accelerator and the server. We measured no statistically significant change in client throughput. In this benchmark, the application is not throughput bound or CPU bound, therefore we expect no change in client throughput.

CPU Overheads. The application is CPU and latency bound when passing data through the WAN accelerator's deduplication and compression pipeline on the side closest to the server. HA/TCP consumes 6.4% more CPU on the primary compared to the baseline CPU usage. Including the total replica's CPU usage, the replication consumes 106% more CPU across both the primary and the replica.

Failover. We tested failover using a multi-path setup with three appliances. There is a primary and replica on each site. This setup bridges two networks and allows for per-connection load-balancing. CARP's failure detection accounts for most of the failover time, and HA/TCP fails over all connections to the replica in 132 µs after failure detection.

5.6 SOCKS Proxy

A SOCKS proxy is an NF that forwards multiple protocols through a server to restrict Internet access from inside a secure network. It is often used to block phishing and other undesirable sites. We built an event-driven SOCKS proxy that conforms to the SOCKS5 protocol as defined in RFC1928 [34].

We used our multi-protocol load generator with 16 connections to fetch a 1 MiB file from the HTTP server through the SOCKS proxy. The long-lived connection option is enabled on the server and client. The connections between the SOCKS proxy server and clients are replicated for the benchmark.

Throughput Overheads. Figure 8 shows the client throughput with HA/TCP enabled. Replication reduces throughput by 2%. The throughput is bounded by the remote server response latency.

CPU Overheads. Figure 8 shows the CPU usage in the application and the input path does not change significantly from the baseline. HA/TCP consumes 29% CPU more on primary compared to the baseline. Including the total replica's CPU usage, the replication consumes 102% more CPU in total across both the primary and the replica.

Failover. Each buffered client request generates a 1 MiB response from the server. Even a modest amount of queuing on the replica leads to a large failover time. As part of the steady state processing on the replica, the queue is drained concurrently with failure detection. The replica is behind the primary by roughly 44 requests. The failover completes in 84 ms after failure detection.

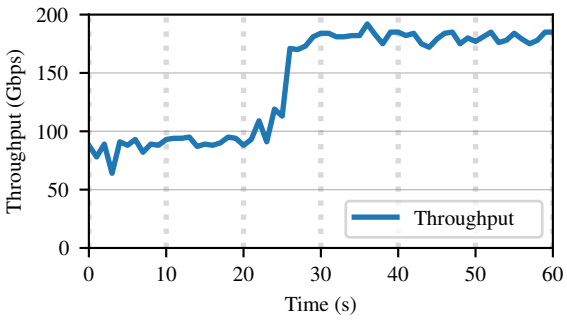

Figure 9: Connection level load-balancing. The load-balancer is deployed on two nodes. Two clients initially connect to a single server, and then load balancing is enabled to migrate half the connections to the second node.

5.7 Load Balancer

HA/TCP allows transparent connection-level load balancing among nodes. We implemented a distributed load-balancer that uses HA/TCP to balance connections for an HTTP service. The load-balancer works across multiple nodes to direct traffic to an HTTP server instance.

In this experiment, we run the load balancer on two servers and migrate half the connections while leaving the other connections alone. We used two identical servers with 100 Gbps NICs and two clients also with 100 Gbps NICs. Initially we create 64 connections to one server, and we enable the load balancing operation at around 22 s. HA/TCP then migrates 32 connections to the second server.

Figure 9 shows the aggregate throughput of all servers. One server is processing 90.6 Gbps and after the migration both are processing an aggregate of 181.2 Gbps.

6 Related Work

In addition to the works described in the background, there are other works on migration and replication of transport protocols. Most of these approaches deviate substantially from our criteria for HA/TCP.

Record-and-replay Based TCP Failover. FT-TCP [53,54] replicates TCP using record and replay. During a connection's lifetime, FT-TCP records all client network traffic and system calls on the server. When a failure occurs, the replica replays the recorded traffic and the system calls to reconstruct the primary's state. This approach suffers from high overheads and a slower failover.

Application Specific TCP Failover. Snoeren et al. [49] proposed an approach that supports TCP failover and migration. This approach uses a non-transparent state replication extension to facilitate the migration. The migration process is based on a proposed TCP migration extension [5], which requires the client to respond to a new *MigrateSYN* packet. However, this proposal has since been abandoned.

Surton et al. [51] introduced a middleware system TCPR that masks TCP failures from the border gateway protocol (BGP). TCPR implements a protocol specific *graceful recovery* technique, which avoids the cost of replaying the entire session for long lived connections, to recover the BGP session state quickly and resume communications. This proposal does not include the transparent failover and the migration features present in HA/TCP.

Other Protocols. QUIC [28, 33] is a transport protocol initially made to improve performance for HTTPS traffic. The protocol is based on UDP and allows multiple streams per connection. QUIC replaces the IP-Port tuple with a connection-ID to allow for connection migration. QUIC requires client application support. Our techniques from HA/TCP can be applied to support the failover of QUIC NFs.

Trickles [46] is a TCP-like transport protocol that is stateless on the server. The server-side protocol state and the application state are moved to the client and sent along with the user payload every time. This allows the server to migrate because all states are stored on the client. Trickles requires kernel and application changes on both servers and clients. Moreover, the state information that is included with every request, is prohibitively expensive for small packet workloads.

Zandy et al. [55] proposed a reliable network connection library. The library addresses issues for mobile clients. The library is not transparent since it requires integration on both server and client applications. In addition, the library only supports server side connection migration using an external checkpointing mechanism.

Other NF FT Techniques. FTvNF [23] combines replay of state access, similar to FTMB [45], with in-chain replication, similar to FTC [21], to provide low overhead reliability. None of these systems provide support for TCP NFs.

7 Conclusion

We present HA/TCP, a scalable framework that enables migration and failover for TCP NFs. HA/TCP provides a replicated socket interface to simplify the implementation of migration and high availability in NFs. HA/TCP fails over in milliseconds after detecting the failure. The migration is faster than the state-of-the-art systems. HA/TCP can also be used to extend these benefits to server applications.

Our code is available at https://github.com/rcslab/hatcp/

Acknowledgments

The authors would like to thank Amilios Tsalapatis, Xinan Yan, and Oscar Zhao for fruitful discussions during HA/TCP 's development. We would also like to thank the NSDI reviewers for their valuable feedback. This research is supported by the following grant: NSERC Discovery, CFI JELF, WHJIL, and NSERC CRD.

References

[1] Ieee standard for information technology - local and metropolitan area networks - part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications-aggregation of multiple link segments. *IEEE Std 802.3ad-2000*, pages 1–184, 2000.

[2] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 1, pages 329–337 vol.1, 2001.

[3] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xomb: Extensible open middleboxes with commodity servers. In *2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 49–60, 2012.

[4] Narjess Ayari, Denis Barbaron, Laurent Lefevre, and Pascale Primet. T2CP-AR: A system for Transparent TCP Active Replication. In *21st International Conference on Advanced Information Networking and Applications (AINA '07)*, pages 648–655, May 2007.

[5] Hari Balakrishnan and A Snoeren. TCP Connection Migration. Internet-Draft draft-snoeren-tcp-migrate-00, Internet Engineering Task Force, November 2000. Work in Progress.

[6] Stephen Bensley, Dave Thaler, Praveen Balasubramanian, Lars Eggert, and Glenn Judd. Data Center TCP (DCTCP): TCP Congestion Control for Data Centers. RFC 8257, October 2017.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[8] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. TCP Extensions for High Performance. RFC 7323, RFC Editor, September 2014.

[9] BSD Kernel Interfaces Manual. Common Address Redundancy Protocol, Feb 2022.

[10] Lukasz Budzisz, Rade Stanojevic, Robert Shorten, and Fred Baker. A strategy for fair coexistence of loss and delay-based congestion control algorithms. *IEEE Communications Letters*, 13(7):555–557, 2009.

[11] John Carmack. .plan File. unpublished, October 1998.

[12] Inho Choi, Nimish Wadekar, Raj Joshi, Joshua Fried, Dan R. K. Ports, Irene Zhang, and Jialin Li. Capybara: μSecond-Scale Live TCP Migration. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, page 30–36, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Cisco Systems, Inc. Best Practice Design - MX Security and SD-WAN . https://documentation.meraki.com/Architectures_and_Best_Practices/Cisco_Meraki_Best_Practice_Design/Best_Practice_Design_-_MX_Security_and_SD-WAN, 2020.

[14] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. On the performance of TCP splicing for URL-Aware redirection. In *Second USENIX Symposium on Internet Technologies & Systems (USITS 99)*, Boulder, CO, October 1999. USENIX Association.

[15] Intel Corporation. Introduction to Intel(R) Ethernet Flow Director and Memcached Performance. White paper, Intel Corporation, 2014.

[16] C. Fetzer, M. Marwah, and S. Mishra. TCP Server Fault Tolerance Using Connection Migration to a Backup Server. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, page 373, Los Alamitos, CA, USA, jun 2003. IEEE Computer Society.

[17] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, RFC Editor, June 2014.

[18] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, RFC Editor, June 2014.

[19] Edward Gately. Cloudflare outage knocks out "significant portion" of global traffic, Dec 2023.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[21] Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali José Mashtizadeh. Fault tolerant service function chaining. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies,*

Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 198–210, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Andrei Gurtov, Tom Henderson, and Sally Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, April 2004.

[23] Yotam Harchol, David Hay, and Tal Orenstein. Ftvnf: fault tolerant virtual network functions. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 141–147, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the Pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 535–549. USENIX Association, April 2021.

[25] David A. Hayes and Grenville Armitage. Improved coexistence and loss tolerance for delay based tcp congestion control. In *IEEE Local Computer Network Conference*, pages 24–31, 2010.

[26] J. Heffner, M. Mathis, and B. Chandler. IPv4 Reassembly Errors at High Data Rates. RFC 4963, RFC Editor, July 2007.

[27] Osamu Honda, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, and Junichi Murayama. Understanding tcp over tcp: effects of tcp tunneling on end-to-end throughput and latency. In *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, volume 6011, pages 138–146. SPIE, 2005.

[28] J. Iyengar and M. Thomson. Quic: A udp-based multiplexed and secure transport. RFC 9000, RFC Editor, May 2021.

[29] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, March 2017. USENIX Association.

[30] K. Poul-Henning. Varnish HTTP Cache, Mar 2023.

[31] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association.

[32] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 501–516, Boston, MA, 2019. USENIX Association.

[33] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[34] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928, RFC Editor, March 1996.

[35] Doug Leith. H-TCP: TCP Congestion Control for High Bandwidth-Delay Product Paths. Internet-Draft draft-leith-tcp-htcp-06, Internet Engineering Task Force, April 2008. Work in Progress.

[36] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 504–517, New York, NY, USA, 2018. Association for Computing Machinery.

[37] D.A. Maltz and P. Bhagwat. Msocks: an architecture for transport layer mobility. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, volume 3, pages 1037–1045 vol.3, 1998.

[38] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. *SIGPLAN Not.*, 52(4):693–708, apr 2017.

[39] Stephen Nadas. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798, March 2010.

[40] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context tls (mctls): Enabling

secure in-network functionality in tls. 45(4):199–212, aug 2015.

[41] Rahul Potharaju and Navendu Jain. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, page 9–22, New York, NY, USA, 2013. Association for Computing Machinery.

[42] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: a high availability framework for middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[43] Riverbed Technology, Inc. SD-WAN Deployment Guide. https://support.riverbed.com/bin/support/static/bk3e4nsvev67aokj3qg5gtcfv4/html/ulhrppo7aoojclf7jbgjnlji07/scm_dg_html/index.html, 2022.

[44] Riverbed Technology, Inc. SteelHead CX. https://www.riverbed.com/sites/default/files/file/2021-10/steelhead-cx-data-sheet.pdf, 2022.

[45] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 227–240, New York, NY, USA, 2015. Association for Computing Machinery.

[46] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, page 175–188, USA, 2005. USENIX Association.

[47] Shinae Woo and Justine Sherry and Sangjin Han and Sue Moon and Sylvia Ratnasamy and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, 2018. USENIX Association.

[48] Silver Peak Systems, Inc. VX Virtual WAN Optimization Software. https://www.silver-peak.com/sites/default/files/infoctr/silver-peak_ds_vx-virtual-wan-optimization.pdf, 2022.

[49] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-Grained failover using connection migration. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)*, San Francisco, CA, March 2001. USENIX Association.

[50] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: connection migration for service continuity in the Internet. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 469–470, 2002.

[51] Robert Surton, Ken Birman, and Robbert van Renesse. Application-driven tcp recovery and non-stop bgp. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[52] Paul Walsh. Cloudflare outage to cause $16M of business interruption losses, Jul 2019.

[53] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 393–402, 2003.

[54] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. Practical and Low-Overhead Masking of Failures of TCP-Based Servers. *ACM Trans. Comput. Syst.*, 27(2), may 2009.

[55] Victor C. Zandy and Barton P. Miller. Reliable Network Connections. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, page 95–106, New York, NY, USA, 2002. Association for Computing Machinery.

[56] Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh. SKQ: Event scheduling for optimizing tail latency in a traditional OS kernel. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 759–772. USENIX Association, July 2021.