Fault Tolerant Service Function Chaining

Milad Ghaznavi
University of Waterloo
eghaznav@uwaterloo.ca

Elaheh Jalalpour
University of Waterloo
ejalalpo@uwaterloo.ca

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

Raouf Boutaba
University of Waterloo
rboutaba@uwaterloo.ca

Ali José Mashtizadeh
University of Waterloo
mashti@uwaterloo.ca

ABSTRACT

Network traffic typically traverses a sequence of middleboxes forming a service function chain, or simply a chain. Tolerating failures when they occur along chains is imperative to the availability and reliability of enterprise applications. Making a chain fault-tolerant is challenging since, in the event of failures, the state of faulty middleboxes must be correctly and quickly recovered while providing high throughput and low latency.

In this paper, we introduce FTC, a system design and protocol for fault-tolerant service function chaining. FTC provides strong consistency with up to f middlebox failures for chains of length $f + 1$ or longer without requiring dedicated replica nodes. In FTC, state updates caused by packet processing at a middlebox are collected, piggybacked onto the packet, and sent along the chain to be replicated. Our evaluation shows that compared with the state of art [51], FTC improves throughput by 2–3.5× for a chain of two to five middleboxes.

CCS CONCEPTS

• **Computer systems organization → Fault-tolerant network topologies**; • **Networks → Middle boxes / network appliances**;

KEYWORDS

Service Function Chain Fault Tolerance; Middlebox Reliability

ACM Reference Format:
Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali José Mashtizadeh. 2020. Fault Tolerant Service Function Chaining. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3387514.3405863

1 INTRODUCTION

Middleboxes are widely deployed in enterprise networks, with each providing a specific data plane function. These functions can be composed to meet high-level service requirements by passing traffic through an ordered sequence of middleboxes, forming a service function chain [45, 46]. For instance, data center traffic commonly passes through an intrusion detection system, a firewall, and a network address translator before reaching the Internet [59].

Providing fault tolerance for middleboxes is critical as their failures have led to large network outages, significant financial losses, and left networks vulnerable to attacks [11, 44, 55, 56]. Existing middlebox frameworks [29, 32, 35, 47, 51] have focused on providing fault tolerance for individual middleboxes. For a chain, they consider individual middleboxes as fault tolerant units that together form a fault tolerant chain. This design introduces redundancies and overheads that can limit a chain's performance.

Independently replicating the state of each middlebox in a chain requires a large number replica servers, which can increase cost. Part of that cost can be mitigated by having middleboxes share the same replica servers, although oversharing can affect performance. More importantly, replication causes packets to experience more than twice its normal delay, since each middlebox synchronously replicates state updates before releasing a packet to the next middlebox [29, 32, 35, 47].

Current state-of-the-art middlebox frameworks also stall as they capture a consistent snapshot of their state leading to lower throughput and higher latency [35, 47, 51]. These stalls significantly increase latency with packets experiencing latencies from 400 μs to 9 ms per middlebox compared to 10–100 μs without fault tolerance [35, 47]. When these frameworks are used in a chain, the stalls cause processing delays across the entire chain, similar to a *pipeline stall* in a processor. As a result, we observed a ~40% drop in throughput for a chain of five middleboxes as compared to a single middlebox (see § 7.4).

In this paper, we introduce a system called *fault tolerant chaining* (FTC) that provides fault tolerance to an entire chain. FTC is inspired by *chain replication* [58] to efficiently provide fault tolerance. At each middlebox, FTC collects state updates due to packet processing and piggybacks them onto the packet. As the packet passes through the chain, FTC replicates piggybacked state updates in servers hosting middleboxes. This allows each server hosting a middlebox to act as a replica for its predecessor middleboxes. If a middlebox fails, FTC can recover the lost state from its successor servers. For middleboxes at the end of the chain, FTC transfers and replicates their state updates in servers hosting middleboxes at the beginning of the chain. FTC does not need any dedicated replica servers to tolerate f number of middlebox failures for chains with more than $f + 1$ middleboxes.

We extend chain replication [58] to address challenges unique to a service function chain. Unlike the original protocol where all nodes run an identical process, FTC must support a chain comprised of different middleboxes processing traffic in the service function chain order. Accordingly, FTC allows all servers to process traffic and replicate state. Moreover, FTC's failure recovery instantiates a new middlebox at the failure position to maintain the service function chain order, rather than the traditional protocol that appends a new node at the end of a chain.

Furthermore, FTC improves the performance of fault tolerant multicore middleboxes. We introduce *packet transactions* to provide a simple programming model to develop multithreaded middleboxes that can effectively make use of multiple cores. Concurrent state updates to middlebox state result in non-deterministic behavior that is hard to restore. A transactional model for state updates allows serializing concurrent state accesses that simplifies reasoning about both middlebox and FTC correctness. The state-of-the-art [51] relies on complex static analysis that supports unmodified applications, but can have worse performance when its analysis falls short.

FTC also tracks dependencies among transactions using *data dependency vectors* that define a partial ordering of transactions. The partial ordering allows a replica to concurrently apply state updates from non-dependent transactions to improve replication performance. This approach has two major benefits compared to *thread-based* approaches that allow concurrent state replication by replaying the operations of threads [51]. First, FTC can support *vertical scaling* by replacing a running middlebox with a new instance with more CPU cores or failing over to a server with fewer CPU cores when resources are scarce during a major outage. Second, it enables a middlebox and its replicas to run with a different number of threads.

FTC is implemented on Click [34] and uses the ONOS SDN controller [7]. We compare its performance with the state-of-the-art [51]. Our results for a chain of two to five middleboxes show that FTC improves the throughput of the state of art [51] by 2× to 3.5× with lower latency per middlebox.

2 BACKGROUND

A service function chain is an ordered sequence of middleboxes. Following the *network function virtualization* (NFV) vision [1], an increasing number of middleboxes are implemented as software running on commodity hardware.

In an NFV environment, as shown in Figure 1, an orchestrator deploys, manages, and steers traffic through a chain of middleboxes. Each middlebox runs multiple threads and is equipped with a multi-queue network interface card (NIC) [15, 42, 50]. A thread receives packets from a NIC's input queue and sends packets to a NIC's output queue. Figure 1 shows two threaded middleboxes processing two traffic flows.

Stateful middleboxes keep dynamic state for packets that they process [24, 52]. For instance, a stateful firewall filters packets based on statistics that it collects for network flows [6], and a network address translator maps internal and external addresses using a flow table [25, 53].

Middlebox state can be *partitionable* or *shared* [6, 20, 23, 47]. Partitionable state variables describe the state of a single traffic flow (e.g., MTU size and timeouts in stateful firewalls [6, 20]) and are only accessed by a single middlebox thread. Shared state variables are for a collection of flows, and multiple middlebox threads query and update them (e.g., port-counts in an intrusion detection system).

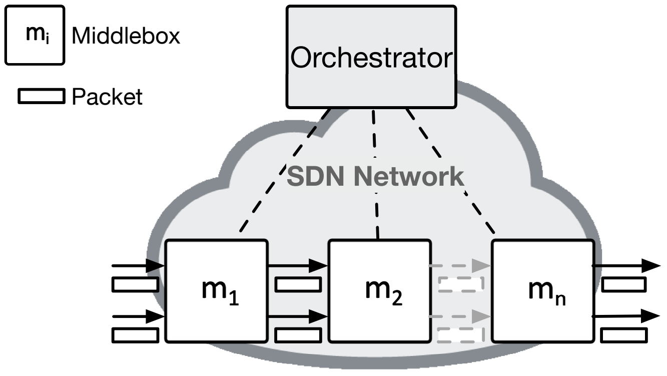


Figure 1: Service function chain model in NFV

A stateful middlebox is subject to both hardware and software failures that can cause the loss of its state [44, 51]. The root causes of these failures include bit corruptions, cable problems, software bugs, and server failures due to maintenance operations and power failures [22, 44]. We model these failures as *fail-stop* in which failures are detectable, and failed components are not restored.

2.1 Challenges

To recover from a middlebox failure, traffic must be rerouted to a redundant middlebox where the state of the failed middlebox is restored. State replication has two challenges that affect middlebox performance.

First, most middleboxes are multithreaded [15, 26, 50, 51], and the order in which interleaving threads access shared state is non-deterministic. Parallel updates can lead to observable states that are hard-to-restore. The difficulty in achieving high performance multithreaded middleboxes is how we capture this state for recovery. One approach to accommodate non-determinism is to log any state read and write, which allows restoring any observable state from the logs [51]. However, this complicates the failure recovery procedure because of record/replay, and leads to high performance overheads during normal operation.

Second, to tolerate f failures, a packet is released *only* when at least $f+1$ replicas acknowledge that state updates due to processing of this packet are replicated. In addition to increasing latency, synchronous replication reduces throughput since expensive coordinations between packet processing and state replication are required for consistency (e.g., pausing packet processing until replication is acknowledged [29, 32, 35, 47]). The overhead of this synchrony for a middlebox depends on where its replicas are located, and how state updates are transferred to these locations. For a solution designed for individual middleboxes, the overheads can accumulate for each middlebox of a chain.

2.2 Limitations of Existing Approaches

Existing middlebox frameworks provide fault tolerance for individual middleboxes. These frameworks provide fault tolerance for a chain with middleboxes deployed over multiple servers; however, their high overheads impact the chain’s performance.

Existing frameworks use one of two approaches. The first approach takes snapshots of middlebox state for state replication [35, 47, 51]. While taking snapshot, middlebox operations are stalled for consistency. These frameworks take snapshots at different rates. They take snapshots per packet or packet-batch introducing 400 μs to 8–9 ms of per packet latency overhead [35, 47]. Periodic snapshots (e.g., at every 20–200 ms intervals) can cause periodic latency spikes up to 6 ms [51]. We measure that per middlebox snapshots cause 40% throughput drop going from a single middlebox to a chain of five middleboxes (see § 7.4).

The second approach [29, 32] redesigns middleboxes to separate and push state into a fault tolerant backend data store. This separation incurs high performance penalties. Accessing state takes at least a round trip delay. Moreover, a middlebox can release a packet only when it receives an acknowledgement from the data store that relevant state updates are replicated. Due to such overheads, the middlebox throughput can drop by ~60% [29] and reduce to 0.5 Gbps (for packets with 1434 B median size) [32].

3 SYSTEM DESIGN OVERVIEW

The limitations of existing work lead us to design fault tolerant chaining (FTC); a new approach that replicates state along the chain to provide fault tolerance.

3.1 Requirements

We design FTC to provide fault tolerance for a wide variety of middleboxes. FTC adheres to four requirements:

Correct recovery: FTC ensures that the middlebox behavior after a failure recovery is consistent with the behavior prior to the failure [54]. To tolerate f failures, a packet can only be released outside of a chain once all necessary information needed to reconstruct the internal state of all middleboxes is replicated to $f + 1$ servers.

Low overhead and fast failure recovery: Fault tolerance for a chain must come with low overhead. A chain processes a high traffic volume and middlebox state can be modified very frequently. At each middlebox of a chain, latency should be within 10 to 100 μs [51], and the fault tolerance mechanism must support accessing variables 100 k to 1 M times per second [51]. Recovery time must be short enough to prevent application outages. For instance, highly available services timeout in just a few seconds [3].

Resource efficiency: Finally, the fault tolerance solution should be resource efficient. To isolate the effect of possible failures, replicas of a middlebox must be deployed on separate physical servers. We are interested in a system that dedicates the fewest servers to achieve a fixed replication factor.

3.2 Design Choices

We model packet processing as a transaction. FTC carefully collects updated values of state variables modified during a packet transaction and appends them to the packet. As the packet passes through the chain, FTC replicates piggybacked state updates in servers hosting the middleboxes.

Transactional packet processing: To accommodate non-determinism due to concurrency, we model the processing of a packet as a transaction, where concurrent accesses to shared state are serialized to ensure that consistent state is captured and replicated. In other systems, the interleaved order of lock acquisitions and state variable updates between threads is non-deterministic, yet externally observable. Capturing and replaying this order is complex and incurs high performance overheads [51]. FTC uses transactional packet processing to avoid the complexity and overhead.

This model is easily adaptable to hybrid transactional memory, where we can take advantage of the hardware support for transactions [13]. This allows FTC to use modern hardware transactional memory for better performance, when the hardware is present.

We also observe that this model does not reduce concurrency in popular middleboxes. First, these middleboxes already serialize access to state variables for correctness. For instance, a load balancer and a NAT ensure *connection persistence* (i.e., a connection is always directed to a unique destination) while accessing a shared flow table [9, 53]. Concurrent threads in these middleboxes must coordinate to provide this property.

Moreover, most middleboxes share only a few state variables [29, 32]. Kablan et al. surveyed five middleboxes for their access patterns to state [29]. These middleboxes mostly perform only one or two read/write operations per packet. The behavior of these middleboxes allow packet transactions to run concurrently most of the time.

In-chain replication: Consensus-based state replication [36, 40] requires $2f + 1$ replicas for each middlebox to reliably detect and recover from f failures. A high-availability cluster approach requires $f + 1$ replicas as it relies on a fault tolerant coordinator for failure detection. For a chain of n middleboxes, these schemes need $n \times (2f + 1)$ and $n \times (f + 1)$ replicas. Replicas are placed on separate servers, and a *naïve* placement requires the same number of servers.

FTC observes that packets already flow through a chain; each server hosting a middlebox of the chain can serve as a replica for the other middleboxes. Instead of allocating dedicated replicas, FTC replicates the state of middleboxes across the chain. In this way, FTC tolerates f failures without the cost of dedicated replica servers.

State piggybacking: To replicate state modified by a packet, existing schemes send separate messages to replicas. In FTC, a packet carries its own state updates. State piggybacking is possible, as a small number of state variables [33] are modified with each packet. Since state updated during processing a packet is replicated in servers hosting the chain, relevant state is already transferred and replicated when the packet leaves the chain.

No checkpointing and no replay: FTC replicates state values at the granularity of packet transactions, rather than taking snapshots of state or replaying packet processing operations. During normal operation, FTC removes state updates that have been applied in all replicas to bound memory usage of replication. Furthermore, replicating the values of state variables allows for fast state recovery during failover.

Centralized orchestration: In our system, a central orchestrator manages the network and chains. The orchestrator deploys fault tolerant chains, reliably monitors them, detects their failures, and

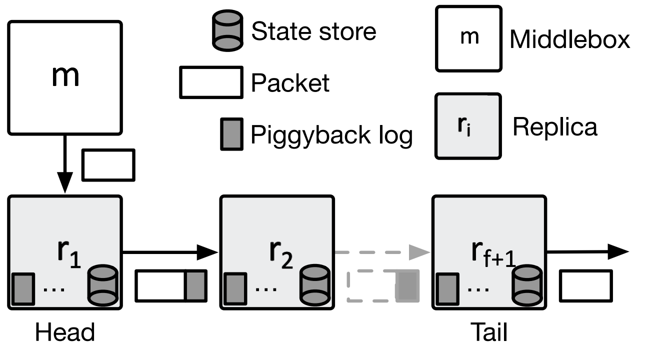


Figure 2: Normal operation for a single middlebox. The head and middlebox reside in the same server. The head tracks state updates due to middlebox packet processing and appends a piggyback log containing these updates to the packet. As the packet passes through the chain, other replicas replicate the piggyback log and apply the carried state updates to their state stores. Finally, the tail strips the piggyback log and releases the packet.

initiates failure recovery. The orchestrator functionality is provided by a fault tolerant SDN controller [7, 31, 41]. After deploying a chain, the orchestrator is not involved in normal chain operations to avoid becoming a performance bottleneck.

In the following sections, we first describe our protocol for a single middlebox in § 4, then we extend this protocol for a chain of middleboxes in § 5.

4 FTC FOR A SINGLE MIDDLEBOX

In this section, we present our protocol for a single middlebox. We first describe our protocol with a single threaded middlebox where state is replicated by single threaded replicas. We extend our protocol to support multithreaded middleboxes and multithreaded replication in § 4.2 and § 4.3. Our system allows middlebox instances to scale the number of threads in response to load, but scaling the number of instances is outside scope of this work.

4.1 Middlebox State Replication

We adapt the *chain replication* protocol [58] for middlebox state replication. For reliable state transmission between servers, FTC uses *sequence numbers*, similar to TCP, to handle out-of-order deliveries and packet drops within the network.

Figure 2 shows our protocol for providing fault tolerance for a middlebox. FTC replicates the middlebox state in $f + 1$ replicas during normal middlebox operations. Replicas $r_1, \ldots, r_{f+1}$ form the *replication group* for middlebox m where r_1 and r_{f+1} are called the *head* and *tail* replicas. Each replica is placed on a separate server whose failure is isolated. With state replicated in $f + 1$ replicas, the state remains available even if f replicas fail.

The head is co-located with the middlebox in the same server. The middlebox state is separated from the middlebox logic and is stored in the head's *state store.* The head provides a state management API for the middlebox to read and write state during packet processing. For an existing middlebox to use FTC, its source code must be modified to call our API for state reads and writes.

Normal operation of protocol: As shown in Figure 2, the middlebox processes a packet, and the head constructs and appends a *piggyback log* to the packet. The piggyback log contains a sequence number and a list of state updates during packet processing. As the packet traverses the chain, each subsequent replica replicates the piggyback log and applies the state updates to its state store. After replication, the tail strips the piggyback log and releases the packet.

The head tracks middlebox updates to state using a monotonically increasing sequence number. After a middlebox finishes processing a packet, the head increments its sequence number only if state was modified during packet processing. The head appends the state updates (i.e., state variables modified in processing the packet and their updated values) and sequence number to the packet as a piggyback log. If no state was updated, the head adds a no-op piggyback log. The head then forwards the packet to the next replica.

Each replica continuously receives packets with piggyback logs. If a packet is lost, a replica requests its predecessor to retransmit the piggyback log with the lost sequence number. A replica keeps the largest sequence number that it has received in order (i.e., the replica has already received all piggyback logs with preceding sequence numbers). Once all prior piggyback logs are received, the replica applies the piggyback log to its local state store and forwards the packet to the next replica.

The tail replicates state updates, strips the piggyback log from the packet, and releases the packet to its destination. Subsequently, the tail periodically disseminates its largest sequence number to the head. The sequence number is propagated to all replicas so they can prune their piggyback logs up to this sequence number.

Correctness: Each replica replicates the per-packet state updates in order. As a result, when a replica forwards a packet, it has replicated all preceding piggyback logs. Packets also pass through the replication group in order. When a packet reaches a replica, prior replicas have replicated the state updates carried by this packet. Thus, when the tail releases a packet, the packet has already traversed the entire replication group. The replication group has $f + 1$ replicas allowing FTC to tolerate f failures.

Failure recovery: FTC relies on a fault tolerant orchestrator to reliably detect failures. Upon failure detection, the replication group is repaired in three steps: adding a new replica, recovering the lost state from an alive replica, and steering traffic through the new replica.

In the event of a head failure, the orchestrator instantiates a new middlebox instance and replica, as they reside on the same server. The orchestrator also informs the new replica about other alive replicas. If the new replica fails, the orchestrator restarts the recovery procedure.

Selecting a replica as the source for state recovery depends on how state updates propagate through the chain. We can reason about this using the *log propagation invariant*: for each replica except the tail, its successor replica has the same or prior state, since piggyback logs propagate in order through the chain.

If the head fails, the new replica retrieves the state store, piggyback logs, and sequence number from the immediate successor to the head. If other replicas fail, the new replica fetches the state from the immediate predecessor.

To ensure that the log propagation invariant holds during recovery, the replica that is the source for state recovery discards any out-of-order packets that have not been applied to its state store and will no longer admit packets in flight. If the contacted replica fails during recovery, the orchestrator detects this failure and re-initializes the new replica with the new set of alive replicas.

Finally, the orchestrator updates routing rules in the network to steer traffic through the new replica. If multiple replicas have failed, the orchestrator waits until all new replicas acknowledge that they have successfully recovered the state. Then, the orchestrator updates the necessary routing rules from the tail to the head.

4.2 Concurrent Packet Processing

To achieve higher performance, we augment our protocol to support multithreaded packet processing and state replication in the middlebox and the head. Other replicas are still single threaded. Later in § 4.3, we will support multithreaded replications in other replicas.

In concurrent packet processing, multiple packets are processed in interleaving threads. The threads can access the same state variables in parallel. To accommodate this parallelism, FTC must consistently track parallel state updates. We introduce *transactional packet processing* that effectively serializes packet processing. This model supports concurrency if packet transactions access disjoint subsets of state.

Transactional Packet Processing: In concurrent packet processing, the effects on state variables must be serializable. Further, state updates must be applied to replicas in the same order so that the system can be restored to a consistent state during failover. To support this requirement, replay based replication systems, such as FTMB [51], track all state accesses, including state reads, which can be challenging to perform efficiently.

In transactional packet processing, state reads and writes by a packet transaction have no impact on another concurrently processed packet. This isolation allows us to only keep track of the relative order between transactions, without needing to track all state variable dependencies.

We realize this model by implementing a *software transactional memory* (STM) API for middleboxes. When a packet arrives, the runtime starts a new packet transaction in which multiple reads and writes can be performed. Our STM API uses *fine grained strict two phase locking* (similar to [14]) to provide serializability. Our API uses a *wound-wait scheme* that aborts transaction to prevent possible deadlocks if a lock ordering is not known in advance. An aborted transaction is immediately re-executed. The transaction completes when the middlebox releases the packet.

Using two phase locking, the head runtime acquires necessary locks during a packet transaction. We simplify lock management using *state space partitioning*, by using the hash of state variable keys to map keys to partitions, each with its own lock. The state partitioning is consistent across all replicas, and to reduce contention, the number of partitions is selected to exceed the maximum number of CPU cores.

At the end of a transaction, the head atomically increments its sequence number only if state was updated during this packet transaction. Then, the head constructs a piggyback log containing the state updates and the sequence number. After the transaction completes, the head appends the piggyback log to the packet and forwards the packet to the next replica.

Correctness: Due to *mutual exclusion*, when a packet transaction includes an updated state variable in a piggyback log, *no* other concurrent transaction has modified this variable, thus the included value is consistent with the final value of the packet transaction. The head's sequence number maps this transaction to a *valid* serial order. Replicated values are consistent with the head, because replicas apply state updates of the transaction in the sequence number order.

4.3 Concurrent State Replication

Up to now FTC provides concurrent packet processing but does not support concurrent replication. The head uses a single sequence number to determine a total order of transactions that modify state partitions. This total ordering eliminates multithreaded replication at successor replicas.

To address the possible replication bottleneck, we introduce *data dependency vectors* to support concurrent state replication. Data dependency tracking is inspired by the *vector clocks* algorithm [19], but rather than tracking points in time when events happen for processes or threads, FTC tracks the points in time when packet transactions modify state partitions.

This approach provides more flexibility compared to tracking dependencies between threads and replaying their operations to replicate the state [51]. First, it easily supports vertical scaling as a running middlebox can be replaced with a new instance with different number of CPU cores. Second, a middlebox and its replicas can also run with different number of threads. The state-of-the-art [51] requires the same number of threads with a one-to-one mapping between a middlebox and its replicas.

Data dependency vectors: We use data dependency vectors to determine a partial order of transactions in the head. Each element of this vector is a sequence number associated to a state partition. A packet piggybacks this partial order to replicas enabling them to replicate transactions with more concurrency; a replica can apply and replicate a transaction in a different serial order that is still equivalent to the head.

The head keeps a data dependency vector and serializes parallel accesses to this vector using the same state partition locks from our transactional packet processing. The head maintains its dependency vector using the following rules. A read-only transaction does *not* change the vector. For other transactions, the head increments the sequence number of a state partition that received any read or write.

In a piggyback log, we replace the sequence number with a dependency vector that represents the effects of a transaction on state partitions. If the transaction does not access a state partition, the head uses a "*don't-care*" value for this partition in the piggyback log. The head obtains the sequence number of other partitions from the head's dependency vector before incrementing their sequence numbers.

Each successor replica keeps a dependency vector *MAX* that tracks the latest piggyback log that it has replicated in order, i.e., it has already received all piggyback logs prior to *MAX*. In case

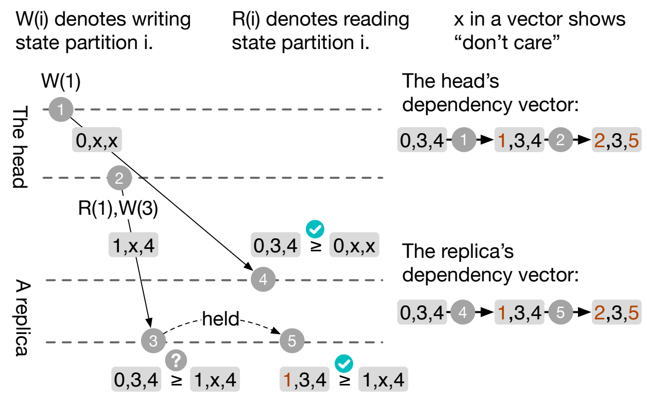

W(i) denotes writing state partition i.
R(i) denotes reading state partition i.
x in a vector shows "don't care"
W(1)
The head
0,x,x
R(1),W(3)
1,x,4
0,3,4 ≥ 0,x,x
A replica
held
0,3,4 ≥ 1,x,4
1,3,4 ≥ 1,x,4
The head's dependency vector:
0,3,4 → 1,3,4 → 2,3,5
The replica's dependency vector:
0,3,4 → 1,3,4 → 2,3,5

Figure 3: Data dependency vectors. The head and the replica run two threads and maintain a dependency vector for three state partitions.

a packet is lost, a replica requests its predecessor to retransmit missing piggyback logs.

Upon receiving a packet, a replica compares the piggybacked dependency vector with its *MAX*. The replica ignores state partitions with "don't care" from this comparison. Once all prior piggyback logs have been received and applied, the replica applies and replicates the piggyback log. For other state partitions, the replica increments their associated sequence numbers in *MAX*.

Example: Figure 3 shows an example of using data dependency vectors in the head and a successor replica with two threads. The head and the replica begin with the same dependency vector for three state partitions. First, the head performs a packet transaction that writes to state partition 1 and increments the associated sequence number. The piggyback log belonging to this transaction contains "don't care" value for state partitions 2 and 3 (denoted by x), since the transaction did not read or write these partitions. Second, the head performs another transaction and forwards the packet with a piggyback log.

Third, as shown the second packet arrives to the replica before the first packet. Since the piggybacked dependency vector is out of order, the replica holds the packet. Fourth, the first packet arrives. Since the piggybacked vector is in order, the replica applies the piggyback log and updates its local dependency vector accordingly. Fifth, by applying the piggyback log of the first packet, the replica now can apply the piggyback log of the held packet.

5 FTC FOR A CHAIN

Our protocol for a chain enables each middlebox to replicate the chain's state while processing packets. To accomplish this, we extend the original chain replication protocol [58] during both normal operation and failure recovery. FTC supports middleboxes with different functionalities to run across the chain, while the same process must be running across the nodes in the original chain replication protocol. FTC's failure recovery instantiates a new middlebox at the failure position to maintain the chain order, while the traditional protocol appends a new node at the end of a chain.

Figure 4 shows our protocol for a chain of n middleboxes. Our protocol can be thought of as running n instances (per middlebox) of the protocol developed earlier in § 4. FTC places a replica per each middlebox. Replicas form n replication groups, each of which provides fault tolerance for a single middlebox.

Viewing a chain as a *logical ring*, the replication group of a middlebox consists of a replica and its f succeeding replicas. Instead of being dedicated to a single middlebox, a replica is shared among $f + 1$ middleboxes and maintains a state store for each of them. Among these middleboxes, a replica is the head of one replication group and the tail of another replication group. A middlebox and its head are co-located in the same server. For instance in Figure 4, if $f = 1$ then the replica r_1 is in the replication groups of middleboxes m_1 and m_n, and r_2 is in the replication groups of m_1 and m_2. Subsequently, the replicas r_n and r_1 are the head and the tail of middlebox m_n.

FTC adds two additional elements, the *forwarder* and *buffer* at the ingress and egress of a chain. The forwarder and buffer are also multithreaded, and are collocated with the first and last middleboxes. The buffer holds a packet until the state updates associated with all middleboxes of the chain have been replicated. The buffer also forwards state updates to the forwarder for middleboxes with replicas at the beginning of the chain. The forwarder adds state updates from the buffer to incoming packets before forwarding the packets to the first middlebox.

5.1 Normal Operation of Protocol

Figure 4 shows the normal operation of our protocol. The forwarder receives incoming packets from the outside world and *piggyback messages* from the buffer. A piggyback message contains middlebox state updates. As the packet passes through the chain, a replica detaches and replicates the relevant parts of the piggyback message and applies associated state updates to its state stores. A replica r_i tracks the state updates of a middlebox m_i and updates the piggyback message to include these state updates. Replicas at the beginning of the chain replicate for middleboxes at the end of the chain. The buffer withholds the packet from release until the state updates of middleboxes at the end of the chain are replicated. The buffer transfers the piggyback message to the forwarder that adds it to incoming packets for state replication.

The forwarder receives incoming packets from outside world and piggyback messages from the buffer. A piggyback message consists of a list of piggyback logs and a list of *commit vectors*. The tail of each replication group appends a commit vector to announce the latest state updates that have been replicated $f + 1$ times for the corresponding middlebox.

Each replica constantly receives packets with piggyback messages. A replica detaches and processes a piggyback message before the packet transaction. As mentioned before, each replica is in the replication group of f preceding middleboxes. For each of them, the replica maintains a dependency vector *MAX* to track the latest piggyback log that it has replicated in order. The replica processes a relevant piggyback log from the piggyback message as described in § 4.3. Once all prior piggyback logs are applied, the replica replicates the piggyback log, applies state updates to the associated state store, and updates the associated dependency vector *MAX*.

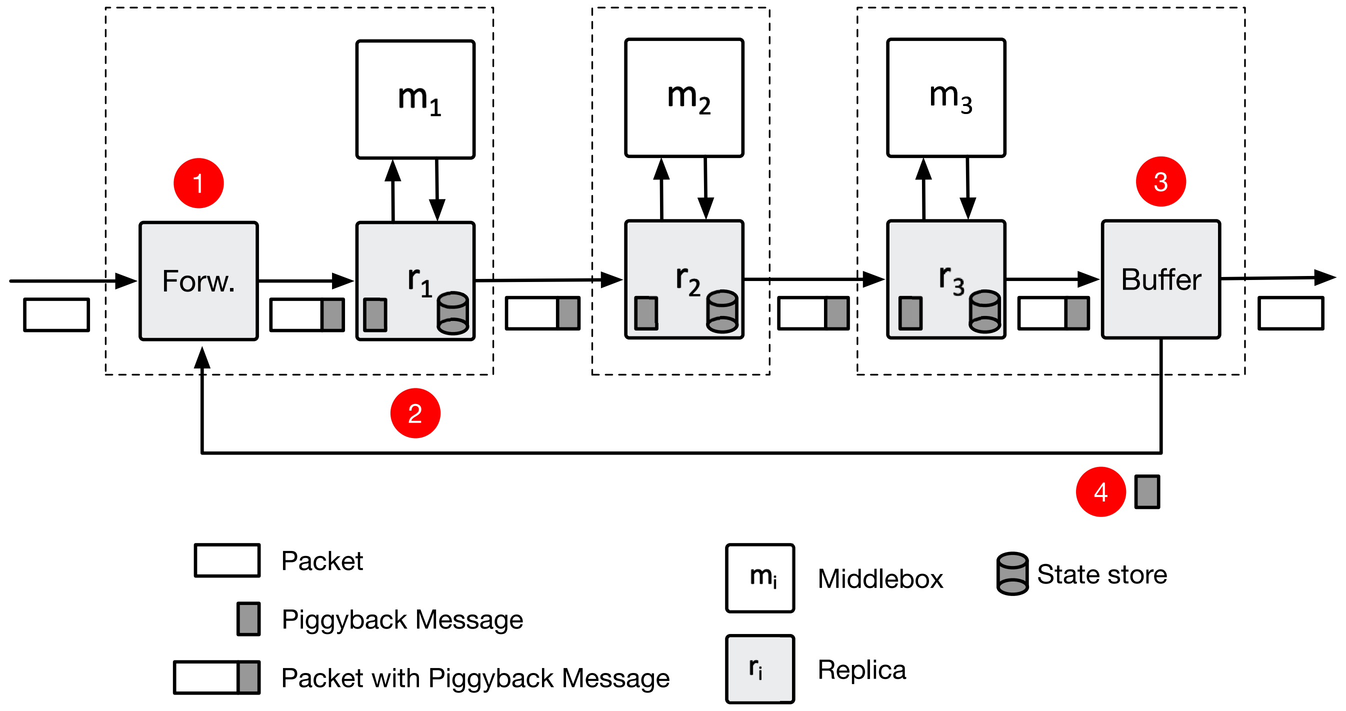


Figure 4: Normal operation for a chain. A middlebox and its head replica reside in the same server. The forwarder and buffer are located in the first and last servers. ① The forwarder appends a state message containing the state updates of the last f middleboxes in the chain from the buffer to an incoming packet, and forwards this packet to the first replica. ② Each replica applies the piggybacked state updates, allows the middlebox to process the packet, and appends new state updates to the packet. ③ Replicas at the beginning of the chain replicate for middleboxes at the end of the chain. The buffer holds packets and releases them once state updates from the end of the chain are replicated. ④ The buffer transfers the piggyback message to the forwarder that adds it to incoming packets for state replication.

Once the middlebox finishes the packet transaction, the replica updates and reattaches the piggyback message to the packet, then forwards the packet. For the replication group where the replica is the head, it adds a piggyback log containing the state updates of processing the packet. If the replica is a tail in the replication group of a middlebox m, it removes the piggyback log belonging to middlebox m to reduce the size of the piggyback message. The reason is that a tail replicates the state updates of m for $f+1$-th time. Moreover, it attaches its dependency vector *MAX* of middlebox m as a commit vector. Later by reading this commit vector, the buffer can safely release held packets. Successor replicas also use this commit vector to prune obsolete piggyback logs.

To correctly release a packet, the buffer requires that the state updates of this packet are replicated, specifically for each middlebox with a preceding tail in the chain. The buffer withholds a packet from release until an upcoming packet piggybacks commit vectors that confirm meeting this requirement. Upon receiving an upcoming packet, the buffer processes the piggybacked commit vectors to release packets held in the memory.

Specifically, let m be a middlebox with a preceding tail, and V_2 be the end of updated range from a piggyback log of a held packet belonging to m. Once the commit vector of each m from an upcoming packet shows that all state updates prior to and including V_2 have been replicated, the buffer releases the held packet and frees its memory.

Other considerations: There may be time periods that a chain receives no incoming packets. In such cases, the state is not propagated through the chain, and the buffer does not release packets. To resolve this problem, the forwarder keeps a timer to receive incoming packets. Upon the timeout, the forwarder sends a *propagating packet* carrying a piggyback message it has received from the buffer. Replicas do not forward a propagating packet to middleboxes. They process and update the piggyback message as described before and forward the packet along the chain. The buffer processes the piggyback message to release held packets.

Some middlebox in a chain can filter packets (e.g., a firewall may block certain traffic), and consequently the piggybacked state is not passed on. For such a middlebox, its head generates a propagating packet to carry the piggyback message of a filtered packet.

Finally, if the chain length is less than $f+1$, we extend the chain by adding more replicas prior to the buffer. These replicas only process and update piggyback messages.

5.2 Failure Recovery

Handling the failure of the forwarder or the buffer is straightforward. They contain only soft state, and spawning a new forwarder or a new buffer restores the chain.

The failure of a middlebox and its head replica is not isolated, since they reside on the same server. If a replica fails, FTC repairs $f+1$ replication groups as each replica replicates for $f+1$ middleboxes. The recovery involves three steps: spawning a new replica and a new middlebox, recovering the lost state from other alive replicas, and steering traffic through the new replica.

After spawning a new replica, the orchestrator informs it about the list of replication groups in which the failed replica was a member. For each of these replication group, the new replica runs an independent state recovery procedure as follows. If the failed replica was the head of a replication group, the new replica retrieves the state store and the dependency vector *MAX* from the immediate successor in this replication group. The new replica restores the dependency matrix of the failed head by setting each of its row to the retrieved *MAX*. For other replication groups, the new replica fetches the state from the immediate predecessors in these replication groups.

Once the state is recovered, the new replica notifies the orchestrator to update routing rules to steer traffic through the new replica. For simultaneous failures, the orchestrator waits until all new replicas confirm that they have finished their state recovery procedures before updating routing rules.

6 IMPLEMENTATION

FTC builds on the ONOS SDN controller [7] and Click [34]. We use the ONOS controller as our NFV orchestrator to deploy middleboxes built on Click. Click is a popular toolkit to develop virtual middleboxes using modular elements. The forwarder and buffer are implemented as Click elements. Our implementation consists of 1K lines of Java for the orchestrator and 6K lines of C++ for FTC and middleboxes.

A replica consists of *control* and *data plane* modules. The control module is a daemon that communicates with the orchestrator and the control modules in other replicas. In failover, the control module spawn a thread to fetch state per each replication group. Using a reliable TCP connection, the thread sends a fetch request to the appropriate member in the replication group and waits to receive state.

The data plane module processes piggyback messages, sends and receives packets to and from a middlebox, constructs piggyback messages, and forwards packets to a next element in the chain (the data-plane module of the next replica or the buffer). In our experiments, the data plane also routes packets through a chain by rewriting their headers.

FTC appends the piggyback logs to the end of a packet, and inserts an IP option to notify our runtime that a packet has a piggyback message. As a piggyback message is appended at the end of a packet, its process and construction can be performed in-place, and there is no need to actually strip and reattach it. Before sending a packet to the middlebox, the relevant header fields (e.g., the total length in IP header) is updated to not account for the piggyback message. Before forwarding the packet to next replica, the header is updated back to reconsider the piggyback message. For middleboxes that may extend the packet, the data plane module operates on the copy of a piggyback message.

Middlebox	State reads	State writes
`MazuNAT`	Per packet	Per flow
`SimpleNAT`	Per packet	Per flow
`Monitor`	Per packet	Per packet
`Gen`	No	Per packet
`Firewall`	N/A	N/A

Chain	Middleboxes in chain
Ch-n	$\texttt{Monitor}_1 \longrightarrow \cdots \longrightarrow \texttt{Monitor}_n$
Ch-Gen	$\texttt{Gen}_1 \longrightarrow \texttt{Gen}_2$
Ch-Rec	`Firewall` $\longrightarrow$ `Monitor` $\longrightarrow$ `SimpleNAT`

Table 1: Experimental middleboxes and chains

7 EVALUATION

We describe our setup and methodology in § 7.1. We micro benchmark the overhead of FTC in § 7.2. We measure the performance of FTC for middleboxes in § 7.3 and for chains in § 7.4. Finally, we evaluate the failure recovery of FTC in § 7.5.

7.1 Experimental Setup and Methodology

We compare FTC with `NF`, a non fault-tolerant baseline system, and `FTMB`, our implementation of [51]. Our `FTMB` implementation is a performance upper bound of the original work that performs the logging operations described in [51] but does not take snapshots. Following the original prototype, `FTMB` dedicates a server in which a middlebox *master* (M) runs, and another server where the fault tolerant components *input logger* (IL) and *output logger* (OL) execute. Packets go through IL, M, then OL. M tracks accesses to shared state using packet access logs (PALs) and transmits them to OL. In the original prototype, no data packet is released until all corresponding dropped PALs are retransmitted. Our prototype assumes that PALs are delivered on the first attempt, and packets are released immediately afterwards. Further, OL maintains only the last PAL.

We used two environments. The first is a local cluster of 12 servers. Each server has an 8-core Intel Xeon CPU D-1540 clocked at 2.0 Ghz, 64 GiB of memory, and two NICs, a 40 Gbps Mellanox ConnectX-3 and a 10 Gbps Intel Ethernet Connection X557. The servers run Ubuntu 14.04 with kernel 4.4 and are connected to 10 and 40 Gbps top-of-rack switches. We use MoonGen [17] and pktgen [57] to generate traffic and measure latency and throughput, respectively. Traffic from the generator server, passed in the 40 Gbps links, is sent through middleboxes and back to the generator. FTC uses a 10 Gbps link to disseminate state changes from buffer to forwarder.

The second environment is the SAVI distributed Cloud [30] comprised of several datacenters deployed across Canada. We use virtual machines with 4 virtual processor cores and 8 GiB memory running Ubuntu 14.04 with Kernel 4.4. We use the published ONOS docker container [39] to control a virtual network of OVS switches [38] connecting these virtual machines. We follow the *multiple interleaved trials methodology* [4] to reduce the variability that come from performing experiments on a shared infrastructure.

We use the middleboxes and chains shown in Table 1. The middleboxes are implemented in Click [34]. `MazuNAT` is an implementation of the core parts of a commercial NAT [2], and `SimpleNAT`

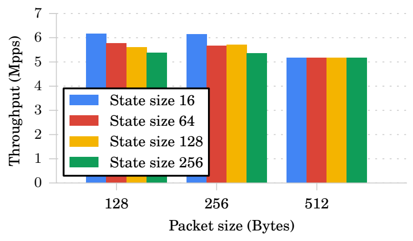


Figure 5: Throughput vs. state size

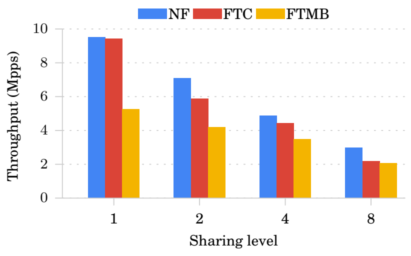


Figure 6: Throughput of `Monitor`

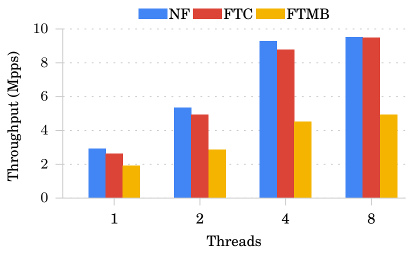


Figure 7: Throughput of `MazuNAT`

provides basic NAT functionalities. They represent read-heavy middleboxes with a moderate write load on the shared state. `Monitor` is a read/write heavy middlebox that counts the number of packets in a flow or across flows. It takes a *sharing level* parameter that specifies the number of threads sharing the same state variable. For example, no state is shared for the sharing level 1, and all 8 threads share the same state variable for sharing level 8. `Gen` represents a write-heavy middlebox that takes a state size parameter, which allows us to test the impact of a middlebox's state size on performance. `Firewall` is stateless. Our experiments also test three chains comprised of these middleboxes, namely `Ch-`n, `Ch-Gen`, and `Ch-Rec`.

For experiments in the first environment, we report latency and throughput. For a latency data point, we report the average of hundreds of samples taken in a 10 second interval. For a throughput data point, we report the average of maximum throughput values measured every second in a 10 second interval. Unless shown, we do not report confidence intervals as they are negligible. Unless specified otherwise, the packet size in our experiments is 256 B, and $f = 1$.

7.2 Microbenchmark

Performance breakdown: To benchmark FTC, we breakdown the performance of the MazuNAT middlebox configured with eight threads in a chain of length two. We show the results for a single thread, but we observed similar results across all threads (assuming low contention). The results only show the computational overhead and exclude device and network IO. In § 7.3, we discuss FTC's impact on end-to-end latency.

Table 2 shows the per packet processing cost in CPU cycles. Packet transaction execution, which includes both packet processing and locking, is the primary contributor to packet latency. The latency due to copying piggybacked state is negligible, because the state is updated per network flow, and the size of updated state is small. The latencies of the forwarder and buffer are also small, and they are independent of the chain length. Only the first and last middlebox contain the forwarder and buffer respectively.

State size performance impact: We also use a micro-benchmark to determine the impact of a state size on the performance of FTC. We measured the latency overhead for the middlebox `Gen` and the chain `Ch-Gen`. We observed that under 2 Mpps for 512 B packets, varying the size of the generated state from 32–256 B has a negligible impact on latency for both `Gen` and `Ch-Gen` (the difference is less than 2 μs). Thus, we focus on the throughput overhead.

	CPU cycles
Packet processing	355 ± 12
Locking	152 ± 11
Copying piggybacked state	58 ± 6
Forwarder	8 ± 2
Buffer	100 ± 4

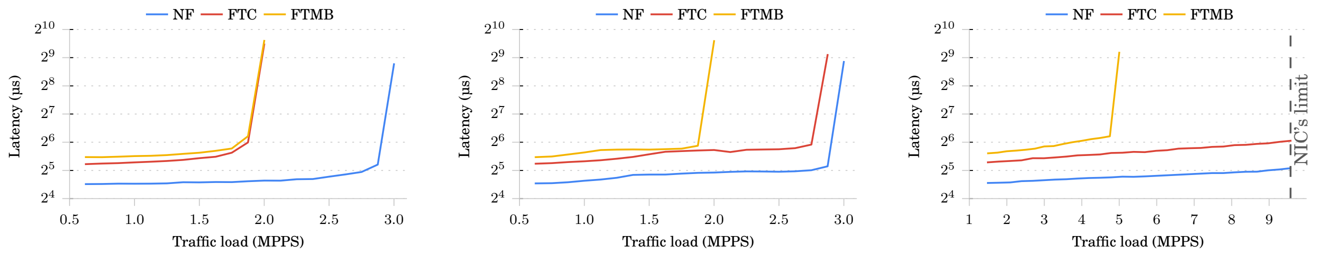
Table 2: Performance breakdown for `MazuNAT` running in a chain of length two. This table shows a breakdown of the CPU overhead for an FTC enabled middlebox.

Figure 5 shows the impact of state size generated by `Gen` on throughput. `Gen` runs a single thread. We vary the state size and measure `Gen`'s throughput for different packet sizes. As expected, the size of piggyback messages impacts the throughput only if it is proportionally large compared to packet sizes. For 128 B packets, throughput drops by only 9% when `Gen` generates states that are 128 B in size or less. The throughput drops by less than 1% with 512 B packets and state up to 256 B in size.

We expect popular middleboxes to generate state much smaller than some of our tested values. For instance, a load balancer and a NAT generate a record per traffic flow [9, 28, 53] that is roughly 32 B in size (2×12 B for the IPv4 headers in both directions and 8 B for the flow identifier). FTC can use jumbo frames to encompass larger state sizes exceeding standard maximum transmission units.

7.3 Fault-Tolerant Middleboxes

Throughput: Figures 6 and 7 show the maximum throughput of two middleboxes. In Figure 6, we configure `Monitor` to run with eight threads and measure its throughput with different sharing levels. As the sharing level for `Monitor` increases, the throughput of all systems, including `NF`, drops due to the higher contention in reading and writing the shared state. For sharing levels of 8 and 2, FTC achieves a throughput that is 1.2× and 1.4× that of `FTMB`'s and incurs an overhead of 9% and 26% compared to `NF`. These overheads are expected since `Monitor` is a write-heavy middlebox, and the shared state is modified non-deterministically per packet. For sharing level 1, `NF` and FTC reach the NIC's packet processing



(a) `Monitor` - Sharing level 8 **(b) `MazuNAT` - 1 thread** **(c) `MazuNAT` - 8 threads**

Figure 8: Latency of middleboxes

capacity[1]. `FTMB` does not scale for sharing level 1, since for every data packet, a PAL is transmitted in a separate message, which limits `FTMB`'s throughput to 5.26 Mpps.

Figure 7 shows our evaluation for `MazuNAT`'s throughput while varying the number of threads. FTC's throughput is 1.37–1.94× that of `FTMB`'s for 1 to 4 threads. Once a traffic flow is recorded in the NAT flow table, processing next packets of this flow only requires reading the shared record (until the connection terminates or times out). The higher throughput compared for `MazuNAT` is because FTC does not replicate the reads, while `FTMB` logs them to provide fault tolerance [51]. We observe that FTC incurs 1–10% throughput overhead compared to `NF`. Part of this overhead is because FTC has to pay the cost of adding space to packets for possible state writes, even when state writes are not performed.

The pattern of state reads and writes impacts FTC's throughput. Under moderate write workloads, FTC incurs 1–10% throughput overhead, while under write-heavy workloads, FTC's overhead remains less than 26%.

Latency: Figure 8 shows the latency of `Monitor` (8 threads with sharing level 8) and `MazuNAT` (two configurations, 1 thread and 8 threads) under different traffic loads. For the both middleboxes, the latency remains under 0.7 ms for all systems as the traffic load increases, until the systems reach their respective saturation points. Past these points, packets start to be queued, and per-packet latency rapidly spikes.

As shown in Figure 8a, under sustainable loads, FTC and `FTMB` respectively introduce overhead within 14–25 μs and 22–31 μs to the per packet latency, out of which 6–7 μs is due to the extra one way network latency to forward the packet and state to the replica. For this write heavy middlebox, FTC adds a smaller latency overhead compared to `FTMB`.

Figure 8b shows that, when running `MazuNAT` with one thread, FTC can sustain nearly the same traffic load as `NF`, and FTC and `FTMB` have similar latencies. For eight threads shown in Figure 8c, both FTC and `NF` reach the packet processing capacity of the NIC. The latency of FTC is largely independent of the number of threads, while `FTMB` experiences a latency increase of 24–43 μs when going from one to eight threads.

7.4 Fault Tolerant Chains

In this section, we report the performance of FTC for a chain of middleboxes during normal operation. For a `NF` chain, each middlebox is deployed in a separate physical server. We do not need more servers, while we dedicate twice the number of servers to `FTMB`: A server for each middlebox (Master in `FTMB`) and a server for its replica (IL and OL in `FTMB`).

Chain length impact on throughput: Figure 9 shows the maximum traffic throughput passing in four chains (`Ch-2` to `Ch-5` listed in Table 1). `Monitors` in these chains run eight threads with sharing level 1. We also report for `FTMB+Snapshot` that is `FTMB` with snapshot simulation. To simulate the overhead of periodic snapshots, we add an artificial delay (6 ms) periodically (every 50 ms). We get these values from [51].

As shown in Figure 9, FTC's throughput is within 8.28–8.92 Mpps and 4.83–4.80 Mpps for `FTMB`. FTC imposes a 6–13% throughput overhead compared to `NF`. The throughput drop from increasing the chain length for FTC is within 2–7%, while that of `FTMB+Snapshot` is 13–39% (its throughput drops from 3.94 to 2.42 Mpps).

This shows that throughput of FTC is largely independent of the chain length, while, for `FTMB+Snapshot`, periodic snapshots taken at all middleboxes significantly reduce the throughput. No packet is processed during a snapshot. Packet queues get full at early snapshots and remain full afterwards because the incoming traffic load is at the same rate. More snapshots are taken in a longer chain. Non-overlapping (in time) snapshots cause shorter service time at each period and consequently higher throughput drops. An optimum scheduling to synchronize snapshots across the chain can reduce this overhead; however, this is not trivial [10].

Chain length impact on latency: We use the same settings as the previous experiment, except we run single threaded `Monitors` due to a limitation of the traffic generator. The latter is not able to measure the latency of the chain beyond size 2 composed of multi-threaded middleboxes. We resort to use single threaded `Monitors` under the load of 2 Mpps, a sustainable load by all systems.

As shown in Figure 10, FTC's overhead compared to `NF` is within 39–104 μs for `Ch-2` to `Ch-5`, translating to roughly 20 μs latency

[1]Although the 40 GbE link is not saturated, our investigation showed that the bottleneck is the NIC's packet processing power. We measured that the Mellanox ConnectX-3 MT 27500, at the receiving side and working under the DPDK driver, at most can process 9.6–10.6 Mpps for varied packet sizes. Though we have not found any official document by Mellanox describing this limitation, similar behavior (at higher rates) has been reported for Intel NICs (see Sections 5.4 and 7.5 in [17] and Section 4.6 in [26]).

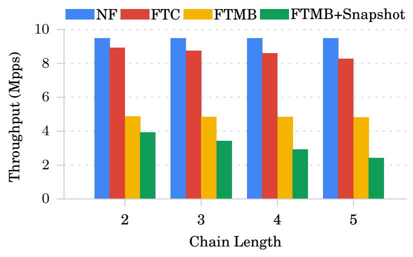


Figure 9: Tput vs. chain length

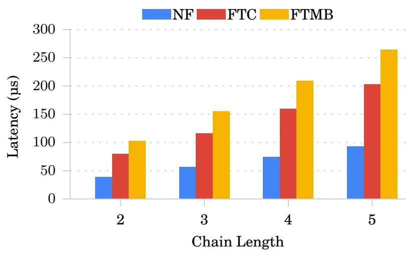


Figure 10: Latency vs. chain length

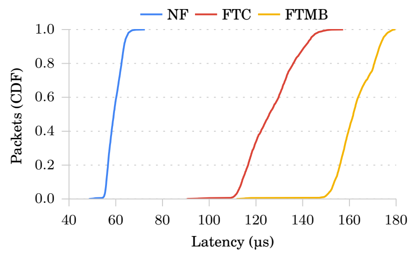


Figure 11: `Ch-3` per packet latency

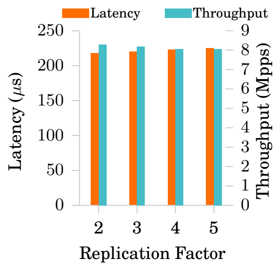


Figure 12: Repl. factor

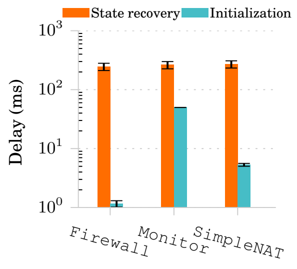


Figure 13: Recovery time

per middlebox. The overhead of FTMB is within 64–171 μs, approximately 35 μs latency overhead per middlebox in the chain. As shown in Figure 11, the tail latency of individual packets passing through `Ch-3` is only moderately higher than the minimum latency. FTC incurs 16.5–20.6 μs per middlebox latency which is respectively three and two orders of magnitudes less than Pico's and REINFORCE's, and is around 2/3 of FTMB's.

In-chain replication eliminates the communication overhead with remote replicas. Doing so also does not cause latency spikes unlike snapshot-based systems. In FTC, packets experience constant latency, while the original FTMB reports up to 6 ms latency spikes at periodic checkpoints (e.g., at every 50 ms intervals) [51].

Replication factor impact on performance: For replication factors of 2–5 (i.e., tolerating 1 to 5 failures), Figure 12 shows FTC's performance for `Ch-5` in two settings where `Monitors` run with 1 or 8 threads. We report the throughput of 8 threaded `Monitor`, while only report the latency of 1 threaded `Monitor` due to a limitation of our test harness.

To tolerate 2.5× failures, FTC incurs only 3% throughput overhead as its throughput decreases to 8.06 Mpps. The latency overhead is also insignificant as latency only increases by 8 μs. By exploiting the chain structure, FTC can tolerate a higher number of failures without sacrificing performance. However, the replication factor cannot be arbitrarily large as encompassing the resulting large piggyback messages inside packets becomes impractical.

7.5 FTC in Failure Recovery

Recall from § 6, failure recovery is performed in three steps: initialization, state recovery, and rerouting delays. To evaluate FTC during recovery, we measure the recovery time of `Ch-Rec` (see Table 1), when each of its middleboxes fails separately. Each middlebox is placed in a different region of our Cloud testbed. As the orchestrator detects a failure, a new replica is placed in the same region as the failed middlebox. The head of `Firewall` is deployed in the same region as the orchestrator, while the heads of `SimpleNAT` and `Monitor` are respectively deployed in a neighboring region and a remote region compared to the orchestrator's region. Since the orchestrator is also a SDN controller, we observe negligible values for the rerouting delay, thus we focus on the state recovery delay and initialization delay.

Recovery time: As shown in Figure 13, the initialization delays are 1.2, 49.8, and 5.3 ms for `Firewall`, `Monitor`, and `SimpleNAT`, respectively. The longer the distance between the orchestrator and the new replica, the higher the initialization delay. The state recovery delays are in the range of 114.38 ± 9.38 ms to 270.79 ± 50.47 ms[2]. In a local area network, FTMB paper [51] reports comparable recovery time of ∼100 ms to 250 ms for `SimpleNAT`. Upon any failure, a new replicas fetches the state from a remote region in the cloud, which causes the WAN latency to dominate delay.

Using ping, we measured the network delay between all pairs of remote regions, and the observed round-trip times confirmed our results.

FTC replicates the values of state variables, and its state recovery delay is bounded by the state size of a middlebox. The replication factor also has a negligible impact on the recovery time of FTC, since a new instantiated replica fetches state in parallel from other replicas.

8 RELATED WORK

In addition to NFV related work discussed in § 2.2, this Section discusses other relevant systems.

Fault tolerant storage: Prior to FTC, the distributed system literature used chain and ring structures to provide fault tolerance. However, their focus is on ordering read/write messages at the process level (compared to, middlebox threads racing to access shared state in our case), at lower non-determinism rates (compared to, per-packet frequency), and at lower output rates (compared to, several Mpps releases).

A class of systems adapt the chain replication protocol [58] for key-value storage systems. In HyperDex [18] and Hibari [21],

[2]The large confidence intervals reported are due to latency variability in the wide area network connecting different regions.

servers shape multiple logical chains replicating different key ranges. NetChain [27] replicates in the network on a chain of programmable switches. FAWN [5], Flex-KV [43], and parameter server [37] leverage consistent hashing to form a replication ring of servers. Unlike these systems, FTC takes advantage of the natural structure of service function chains, uses transactional packet processing, and piggybacks state updates on packets.

Primary backup replication: In *active* replication [49], all replicas process requests. This scheme requires determinism in middlebox operations, while middleboxes are non-deterministic [15, 26]. In passive replication [8], only a *primary* server processes requests and sends state updates to other replicas. This scheme makes no assumption about determinism. Generic virtual machine high availability solutions [12, 16, 48] pause a virtual machine per each checkpoint. These solutions are not effective for chains, since the chain operations pauses during long checkpoints.

Consensus protocols: Classical consensus protocols, such as Paxos [36] and Raft [40] are known to be slow and cause unacceptable low performance if used for middleboxes.

9 CONCLUSION

Existing fault tolerant middlebox frameworks can introduce high performance penalties when they are used for a service function chain. This paper presented FTC, a system that takes advantage of the structure of a chain to provide efficient fault tolerance. Our evaluation demonstrates that FTC can provide high degrees of fault tolerance with low overhead in terms of latency and throughput of a chain. Our implementation is available https://github.com/eljalalpour/FTSFC.git. *This paper does not raise any ethical issues.*

ACKNOWLEDGMENTS

We would like to thank our shepherd Vyas Sekar and the anonymous reviewers for their valuable feedback. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2017. *NFV Whitepaper.* Technical Report. European Telecommunications Standards Institute. https://portal.etsi.org/NFV/NFV_White_Paper.pdf

[2] 2019. mazu-nat.click. https://github.com/kohler/click/blob/master/conf/mazu-nat.click.

[3] 2020. Tuning Failover Cluster Network Thresholds. https://bit.ly/2NC7dGk. [Online].

[4] Ali Abedi and Tim Brecht. 2017. Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17).* ACM, New York, NY, USA, 287–292. https://doi.org/10.1145/3030207.3030229

[5] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09).* ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/1629575.1629577

[6] P Ayuso. 2006. Netfilter's connection tracking system. *;login* 31, 3 (2006).

[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. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14).* ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2620728.2620744

[8] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. 1993. Distributed Systems (2Nd Ed.). In *Distributed Systems (2Nd Ed.),* Sape Mullender (Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, Chapter The Primary-backup Approach, 199–216. http://dl.acm.org/citation.cfm?id=302430.302438

[9] B. Carpenter and S. Brim. 2002. *Middleboxes: Taxonomy and Issues.* RFC 3234. RFC Editor. 1–27 pages. http://www.rfc-editor.org/rfc/rfc3234.txt

[10] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[11] Adrian Cockcroft. 2012. A Closer Look At The Christmas Eve Outage. http://techblog.netflix.com/2012/12/a-closer-look-at-christmas-eve-outage.html.

[12] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08).* USENIX Association, San Francisco, CA.

[13] Dave Dice, Yossi Lev, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2010. Simplifying Concurrent Algorithms by Exploiting Hardware Transactional Memory. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10).* Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1810479.1810537

[14] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06).* Springer-Verlag, Berlin, Heidelberg, 194–208. https://doi.org/10.1007/11864219_14

[15] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09).* ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/1629575.1629578

[16] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. 2013. COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13).* ACM, New York, NY, USA, Article 3, 16 pages. https://doi.org/10.1145/2523616.2523630

[17] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 2015 Internet Measurement Conference (IMC '15).* ACM, New York, NY, USA, 275–287. https://doi.org/10.1145/2815675.2815692

[18] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12).* ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2342356.2342360

[19] Colin J Fidge. 1987. *Timestamps in message-passing systems that preserve the partial ordering.* Australian National University. Department of Computer Science.

[20] N. Freed. 2000. *Behavior of and Requirements for Internet Firewalls.* RFC 2979. RFC Editor. 1–7 pages. http://www.rfc-editor.org/rfc/rfc2979.txt

[21] Scott Lystig Fritchie. 2010. Chain Replication in Theory and in Practice. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang (Erlang '10).* ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/1863509.1863515

[22] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. 2016. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16).* ACM, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/2901318.2901352

[23] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14).* ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/2619239.2626313

[24] Y. Gu, M. Shore, and S. Sivakumar. 2013. A Framework and Problem Statement for Flow-associated Middlebox State Migration. https://tools.ietf.org/html/draft-gu-statemigration-framework-03.

[25] T. Hain. 2000. *Architectural Implications of NAT.* RFC 2993. RFC Editor. 1–29 pages. http://www.rfc-editor.org/rfc/rfc2993.txt

[26] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated Software Router. *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 195–206. https://doi.org/10.1145/1851275.1851207

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* USENIX Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[28] D. Joseph and I. Stoica. 2008. Modeling middleboxes. *IEEE Network* 22, 5 (September 2008), 20–25. https://doi.org/10.1109/MNET.2008.4626228

[29] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17).* USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[30] J. M. Kang, H. Bannazadeh, and A. Leon-Garcia. 2013. SAVI testbed: Control and management of converged virtual ICT resources. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. 664–667.

[31] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. 2015. Ravana: Controller Fault-tolerance in Software-defined Networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/2774993.2774996

[32] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 501–516. https://www.usenix.org/conference/nsdi19/presentation/khalid

[33] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 239–253. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khalid

[34] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297. https://doi.org/10.1145/354871.354874

[35] Sameer G Kulkarni, Guyue Liu, KK Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. 2018. REINFORCE: Achieving Efficient Failure Resiliency for Network Function Virtualization based Services. In *15th USENIX International Conference on emerging Networking EXperiments and Technologies (CoNEXT) 18)*. USENIX Association, 35–49.

[36] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (Dec. 2001), 18–25.

[37] Mu Li, David G. Anderson, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Operating Systems Design and Implementation (OSDI)*. 583–598.

[38] NiciraNetworks. 2019. OpenvSwitch: An open virtual switch. http://openvswitch.org.

[39] NiciraNetworks. 2019. The published ONOS Docker images. https://hub.docker.com/r/onosproject/onos/.

[40] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 305–319.

[41] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. 2017. SCL: Simplifying Distributed SDN Control Planes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 329–345. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-aurojit-scl

[42] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/2168836.2168870

[43] Amar Phanishayee, David G. Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. 2012. Flex-KV: Enabling High-performance and Flexible KV Systems. In *Proceedings of the 2012 Workshop on Management of Big Data Systems (MBDS '12)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/2378356.2378361

[44] Rahul Potharaju and Navendu Jain. 2013. 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)*. ACM, New York, NY, USA, 9–22. https://doi.org/10.1145/2504730.2504737

[45] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2486001.2486022

[46] Paul Quinn and Thomas Nadeau. 2015. *Problem Statement for Service Function Chaining*. Internet-Draft. IETF. https://tools.ietf.org/html/rfc7498

[47] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. 2013. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2523616.2523635

[48] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. 2010. *The design and evaluation of a practical system for fault-tolerant virtual machines*. Technical Report. Technical Report VMWare-RT-2010-001, VMWare.

[49] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[50] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. 2012. Design and implementation of a consolidated middlebox architecture. In *NSDI 12*. 323–336.

[51] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 227–240. https://doi.org/10.1145/2785956.2787501

[52] Robin Sommer, Matthias Vallentin, Lorenzo De Carli, and Vern Paxson. 2014. HILTI: An Abstract Execution Environment for Deep, Stateful Network Traffic Analysis. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)*. ACM, New York, NY, USA, 461–474. https://doi.org/10.1145/2663716.2663735

[53] P. Srisuresh and K. Egevang. 2001. *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022. RFC Editor. 1–16 pages. http://www.rfc-editor.org/rfc/rfc3022.txt

[54] Rob Strom and Shaula Yemini. 1985. Optimistic Recovery in Distributed Systems. *ACM Trans. Comput. Syst.* 3, 3 (Aug. 1985), 204–226. https://doi.org/10.1145/3959.3962

[55] The AWS Team. 2012. Summary of the October 22, 2012 AWS Service Event in the US-East Region. https://aws.amazon.com/message/680342/.

[56] The Google Apps Team. 2012. Data Center Outages Generate Big Losses. http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/appsstatus/ir/plibxfjh8whr44h.pdf.

[57] Daniel Turull, Peter Sjödin, and Robert Olsson. 2016. Pktgen: Measuring performance on high speed networks. *Computer Communications* 82 (2016), 39 – 48. https://doi.org/10.1016/j.comcom.2016.03.003

[58] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=1251254.1251261

[59] O. Huang M. Boucadair N. Leymann Z. Cao J. Hu W. Liu, H. Li. 2014. Service function chaining use-cases. https://tools.ietf.org/html/draft-liu-sfc-use-cases-01.