Pico Replication: A High Availability Framework for Middleboxes

Shriram Rajagopalan[†‡] Dan Williams[†] Hani Jamjoom[†]

[†]IBM T. J. Watson Research Center, Yorktown Heights, NY
[‡]University of British Columbia, Vancouver, Canada

Abstract

Middleboxes are being rearchitected to be service oriented, composable, extensible, and elastic. Yet system-level support for high availability (HA) continues to introduce significant performance overhead. In this paper, we propose *Pico Replication (PR)*, a system-level framework for middleboxes that exploits their flow-centric structure to achieve low overhead, fully customizable HA. Unlike generic (virtual machine level) techniques, PR operates at the flow level. Individual flows can be checkpointed at very high frequencies while the middlebox continues to process other flows. Furthermore, each flow can have its own checkpoint frequency, output buffer and target for backup, enabling rich and diverse policies that balance—per-flow—performance and utilization. PR leverages OpenFlow to provide near instant flow-level failure recovery, by dynamically rerouting a flow's packets to its replication target. We have implemented PR and a flow-based HA policy. In controlled experiments, PR sustains checkpoint frequencies of 1000Hz, an order of magnitude improvement over current VM replication solutions. As a result, PR drastically reduces the overhead on end-to-end latency from 280% to 15.5% and throughput overhead from 99.5% to 3.2%.

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1.
http://dx.doi.org/**10.1145/2523616.2523635**

1 Introduction

Middleboxes [36–42] pervade data center networks of various scales. Recently, middlebox architectures are being re-engineered to be more service oriented [13, 43, 47, 52], composable [24], extensible [5] and dynamically scalable [21]. In this new vision, services like protocol acceleration, load balancing, and intrusion detection/prevention can be easily customized, managed, and scaled to match the needs of the flows in the data center. Despite the renewed interest in middleboxes, high availability support is limited and requires that each middlebox service implements its own HA mechanism and policy.[1] This paper re-examines system support for HA in middleboxes.

Current middlebox HA approaches often deploy a cluster of replicas or configure a pair replicas in active/standby or active/active replication setups [29, 31]. In such configurations, middlebox failure is typically not transparent to the endpoints. Failure causes existing flows in the failed replica to drop [30]; endpoints must explicitly reestablish the lost connections. Router redundancy protocols like HSRP [56] and VRRP [62] only address part of the problem: how to re-route flows to a standby appliance in case of a failure. These protocols do not address the problem of persisting session (flow) state, which is essential to maintaining end-to-end connectivity. This paper shows that preserving flow state—a key ingredient for graceful recovery—can be achieved without introducing substantial design complexity in the middlebox or sacrificing performance.

One straightforward approach to achieving generic middlebox HA is to directly apply HA solutions for virtual machines (VMs). VM-level checkpointing [9, 17] and event logging [11, 12, 23] techniques can be used to protect arbitrary middleboxes, transparently. However, such approaches are heavyweight because (1) the entire VM must be suspended to ensure a consistent checkpoint, (2) all flows—including delay-sensitive flows—

[1]High availability is crucial for middleboxes, as evidenced by recent outages related to middlebox services [44, 45, 50].

Class	State Access Pattern	Examples	HA
Flow Independent	N/A	Stateless Firewalls	No Sync
Flow Dependent	write once, then read	NAT, Vyatta [41,42]	Sync Once
	read/write for every packet	IPS [61], IDS [18], ADC [36–40]	Sync Continuously

Table 1: Taxonomy of Middleboxes for the purposes of HA

are delayed for every checkpoint, and (3) all flows are replicated to the same target, limiting the scope of possible recovery policies.

In this paper, we introduce and fully implement *Pico Replication (PR)*—a system level HA framework specifically tailored towards middlebox applications. Pico Replication operates on individual flows. It takes advantage of how flows in a middlebox represent separate execution contexts that can be migrated between replicas [21]. This allows PR to independently and transparently replicate flow-specific state using techniques from VM replication systems [9, 15, 23], specifically through continuous live migration [8] and output buffering [27]. Unlike existing approaches to VM replication, the middlebox state machine continues to process packets belonging to other flows in the system during checkpointing. More specifically, PR fragments the set of flows on a replica into disjoint subsets called *replication groups*. Each replication group has its own output buffer, checkpoint frequency and a replication (middlebox) target, independent of other groups in the same replica. PR leverages OpenFlow to track the location of individual flows (and flow-specific state) across the system. On failure, individual flows are re-routed to their respective backup targets by updating the flow forwarding rules in the OpenFlow network.

The flow-level granularity of control has two key advantages. First, it allows PR to operate at much higher frequencies than existing VM-based replication. In our evaluation, PR is able to achieve replication frequencies of 1000Hz, an order of magnitude higher than Remus. When compared to Remus, PR only increases end-to-end latency by 8.5 ms (over 54.5 ms base latency), a 264% drop; PR also minimizes the throughput overhead from 99.5% to 3.2%. The second advantage of flow-level granularity is that it allows PR to control the replication frequencies and targets of different flows independently. This creates new opportunities for balancing performance and utilization and to embed a broad set of HA policies, including those popularized by Chord [26]. Most importantly, these policies can be mixed and matched depending on the needs of the each flow.

To summarize, Pico Replication makes the following contributions:

- system-level support for middlebox HA that enables custom per-flow replication with transparent failure recovery,
- an order of magnitude improvement in replication performance over existing system-level HA solutions, and
- a middlebox aware HA policy framework that enables various dynamically adaptive policies.

The rest of the paper proceeds as follows: Section 2 presents a classification of middleboxes and limitations of existing solutions. Section 3 and 4 describe the design and implementation of the Pico Replication framework, respectively. Section 5 provides example Pico Replication policies. Section 6 evaluates Pico Replication and Section 7 concludes the paper.

2 Middlebox HA

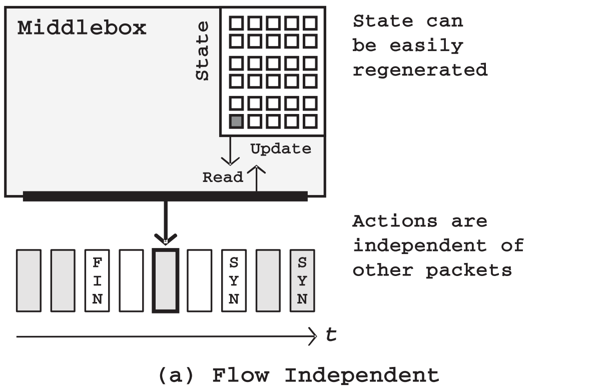
In cloud environments, design for failure is critical, including middlebox failure.[2] In this section, we look into which categories of middleboxes can benefit from HA solutions, the requirements for providing effective HA for middleboxes, and the current state of the art.

2.1 Middlebox Classification

Middleboxes cover a broad spectrum of applications. Some act as gate keepers (e.g., firewalls), inspecting packet headers and making decisions independent of previous or future packets. Others analyze packets at line speed and maintain state for each flow (e.g., intrusion prevention systems). We classify middlebox applications into two types based on their per-flow *statefulness* (Table 1).

Flow Independent. *Flow independent* refers to a middlebox that either maintains no state, contains state that

[2]With even moderately long-lived flows (on the order of few minutes), commonly found on ecommerce websites, session-oriented web applications [16], etc., losing middlebox state results in widespread outage. Previous work [3] has shown that middleboxes can indeed reduce end-to-end availability to 99.9%, compared to the five 9s that today's users typically expect. Apart from disrupting connectivity, loss of a middlebox increases resource usage at the server, as hung TCP connections linger for 50s or more, until a TCP timeout [2].


Middlebox
State
Update
Read
F I N
S Y N
S Y N
t
State can be easily regenerated
Actions are independent of other packets
(a) Flow Independent

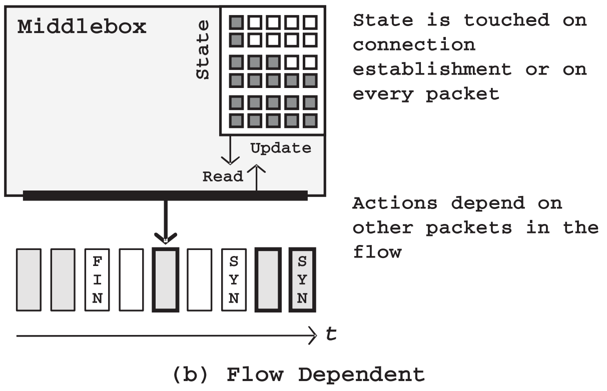

Middlebox
State
Update
Read
F I N
S Y N
S Y N
t
State is touched on connection establishment or on every packet
Actions depend on other packets in the flow
(b) Flow Dependent

Figure 1: Middlebox Types

depends only on the packet being inspected (independent of past or future packets), or contains state can be regenerated automatically (Figure 1(a)). Examples of applications in this class include stateless firewalls that block all traffic matching a specified set of rules.

From an HA perspective, flow independent middleboxes can be made highly available by having redundant instances. When one or more instances fail, traffic to the failed instance can be re-routed via the remaining active instances, without any impact to end-to-end connectivity.

Flow Dependent. *Flow dependent* middleboxes generate per-flow state (Figure 1(b)). Some do so only during flow establishment, while others update flow state on every packet. In flow dependent middleboxes, the per-flow state is vital to maintaining the end-to-end connectivity.

Middleboxes like NAT, for example, only create state—a port mapping—during flow establishment. Processing of other packets in the same flow involves only reads from the port mapping table. Vyatta's vRouter [41, 42] is one example of a commercial product that generates flow state on flow establishment. On failure, since a new flow state is created only during flow establishment, merely re-routing ongoing flows to standby hosts will not suffice. The standby hosts would drop the packets. For HA, middleboxes like NAT require flow state synchronization *only once* during flow establishment. This approach is commonly found in Netfilter Connection Tracking based HA systems, such as those employed by Vyatta [31].

Flow dependent middleboxes can also manipulate the state for every packet in the flow. For example, an inline Intrusion Prevention System (IPS) such as Suricata [61], intercepts each packet entering or exiting the network, performing deep packet inspection (DPI), tracking the protocol state machine, etc. Other common examples include full proxies, layer-7 firewalls, protocol accelerators, traffic managers, web content optimizers, etc. These are collectively known as application delivery controllers (ADC) [36–40]. To provide HA to this class of middleboxes, the per-flow state has to be continuously replicated and kept up to date, should one wish to ensure a seamless failover. In case of an inline IPS with no HA support, simply failing over to a redundant instance potentially terminates all existing TCP streams since the backup instance possess no knowledge of prior traffic. Existing HA solutions for flow dependent middlebox are either ad hoc, such as the one employed by F5's BIG-IP [29] or decide to drop flows altogether causing downtime, such as one used by Riverbed's Stingray Traffic Manager [30].

2.2 Requirements for Middlebox HA

Our goal in this paper is to develop a generic, system level HA solution for flow dependent middleboxes. We assume a fail-stop failure model, in which network partitions are not tolerated. We present the following requirements that an ideal middlebox HA solution should satisfy.

R1. **Recovery-transparency.** The middlebox is often an invisible entity that lies along the network path between two end points.Thus, it is insufficient to transparently failover to a redundant middlebox for new flows. State from the failed middlebox must be recovered with consistency, so that existing flows can continue with minimal interruption.

R2. **Low Performance Overhead.** Middleboxes process millions of packets per second. The performance overhead of the HA framework (latency and throughput) on individual flows must be minimal.

R3. **Tunable Policies.** When attempting to provide a transparent HA mechanism for a wide array of middlebox applications, the policy developer should be able to easily create HA policies that control where to place the backup for a given set of flows and when (with what frequency) to checkpoint the flows. These two parameters allow the designer to make tradeoffs between end-to-end latency and the overall system utilization in the middlebox cluster.

2.3 State of the Art

When considering the applicability of previous work, we focus on systems that can preserve the runtime state of the middlebox application in the failed replica, ensure transparent failover and maintain consistency of state after recovery. We find that none of the existing solutions can be readily applied to virtual middlebox applications in a cloud infrastructure.

Ad Hoc Solutions. Commercial middlebox products use ad hoc HA implementations. For example, one common approach involves a clustered configuration with DNS load balancing and failover [30]. When a node fails, ongoing client connections are lost. Another typical approach is to use active-standby or active-active configurations with custom state replication techniques [29, 31] to provide transparent failure recovery. However, with such point solutions, the management complexity increases dramatically as the number of middleboxes in the network increases. This is apparent in enterprise networks today where there are as many middleboxes in the network as L3 devices [25], frequently resulting in device sprawl, network downtime due to misconfiguration, etc. Thus, at large scale, it becomes infeasible to configure and manage different HA solutions specific to each middlebox vendor. Our goal is to develop a generic framework at the system level that can be leveraged by a large class of applications, whose inner workings are well understood.

VM Replication. VM level HA techniques [7, 9, 11, 12, 15, 17, 20, 23] can be applied to arbitrary applications, to provide a completely stateful, consistent and transparent recovery (R1). However, these solutions do not satisfy requirement R2, since the coarse-grained protection granularity (the entire VM) degrades the performance of the middlebox application during normal operation. Similarly, at the VM granularity, very few HA policies can be instituted (R3).

The root cause of the performance degradation in replication based solutions arises from output buffering, a critical requirement for achieving consistent and transparent failover. Output buffering and commit involves buffering the VM's output (e.g., network packets) during an epoch and releasing the output only after the checkpoint for that epoch is committed at the backup. Suspending and resuming the entire VM, as is done for every checkpoint in systems like Remus [9, 17], delay the release of the output buffer and limit the frequency of checkpointing.

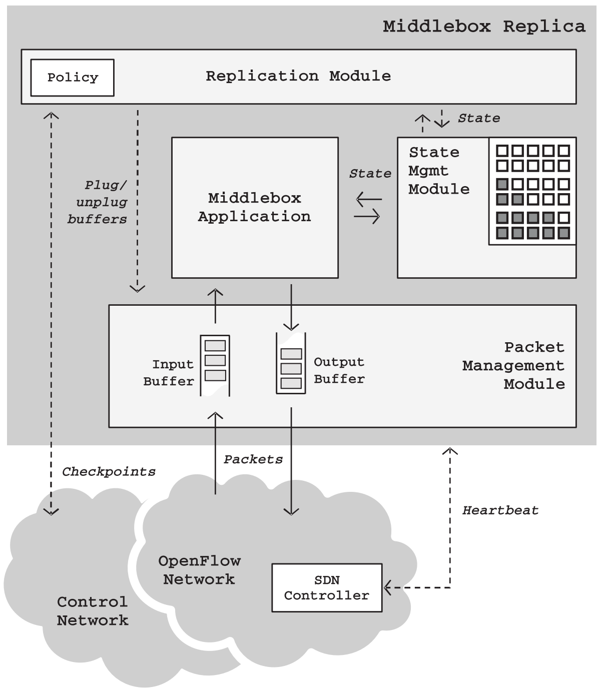


Figure 2: High level design of Pico Replication

3 Pico Replication

The Pico Replication framework proposes a new approach to preserving state. Instead of suspending and checkpointing an entire middlebox at the VM level, Pico Replication capitalizes on the unique structure of middlebox applications to enable fine-grained flow-level replication. By operating on this fine-grained level, the middlebox can continue to process packets from other flows even as one flow is suspended for checkpointing. With Pico Replication, the most valuable pieces of data in a middlebox are replicated at very high frequencies (1000 checkpoints per second), with little impact on the application's execution; satisfying all three requirements mentioned in Section 2.2.

Figure 2 shows the high level components that make up the Pico Replication framework. We assume a deployment model in which a dynamically scaling middlebox cluster comprises a number of replica VMs. Each replica runs three modules. First, the *State Management Module* (SMM) is responsible for managing flow-related state. The SMM manages and controls access to both a set of primary flow states for flows that are currently being processed by the local replica and a set of backup flow states for flows that are currently being processed elsewhere. Second, the *Packet Management Module* (PMM) is responsible for ensuring that packets enter and exit the middlebox at appropriate times. In particular, the PMM buffers incoming and outgoing packets

HA Operation	Pico Replication Component
Identify State	State module
Replicate State	State, packet, and replication modules
Failover	State module and SDN

Table 2: Pico Replication components involved in each HA operation

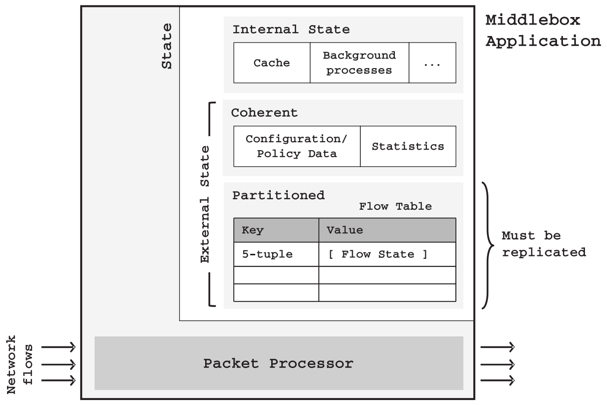


Figure 3: Anatomy of Middlebox State

in one of a set of per-flow or per-group-of-flow buffers to maintain output consistency in case of failure. Third, the *Replication Module* (RM) implements a replication policy by instructing the PMM to plug or unplug input buffers, while copying state from the SMM over the network to other replicas.

Finally, each replica is placed on a Software Defined Network (SDN) (e.g., an OpenFlow [60] network). The SDN controller assigns network flows to replicas and sets up flow forwarding paths in the network. The SDN controller also detects replica failure and reroutes network flows to failover.

The three fundamental operations involved in any state replication based HA system are: *identify state*, *replicate state* and *failover*. The Pico Replication framework is no exception; Table 2 describes the components responsible for providing each of these operations. The remainder of this section describes each component in more detail.

3.1 State Management Module

Pico Replication relies on the middlebox application to interface with the State Management Module to identify flow-related state. State identification reduces the overhead of replication and eliminates the need to suspend an entire VM for checkpointing. To identify critical pieces of state in the middlebox, we leverage concepts from our earlier work, Split/Merge [21].

Identifying Flow State. Figure 3 shows the Split/Merge classification of middlebox state. Split/Merge classifies the state inside a middlebox VM into two types: *internal* and *external*. Internal state is specific to every replica in a middlebox cluster, such as the operating system's buffer cache, other processes in the replica, etc. and does not need to be replicated. External state is further classified into *partitioned* or *coherent*. Partitioned state represents the set of per-flow states maintained by the middlebox application. Each replica exclusively owns a subset of the per-flow states. The per-flow states (partitioned state) represent the critical pieces of data that need to be persisted across failures, to ensure uninterrupted end-to-end connectivity. Coherent state represents global shared data such as static configuration information, non-critical statistics counters, etc. Since it is already shared (and likely replicated), we do not consider further replication of coherent state.

Flow-State Transactions. Like the Split/Merge system, the State Management Module exposes an interface to the application to not just identify, but control access to flow states. Importantly, it maintains a notion of a *transaction*. In other words, the State Management Module keeps track of when the middlebox is in the middle of processing a packet belonging to a particular flow and accessing the corresponding state. This transaction boundary provides information as to whether or not it is safe to replicate flow state over the network. The Replication Module (§3.3) will not copy flow state for flows that are in the middle of a transaction.

One-time Checkpointing. As described in Section 2.1, some flow dependent middleboxs create the per-flow state once during flow establishment (e.g., stateful NAT) and the state remains read-only throughout the lifetime of the flow. The SMM allows an application to indicate whether the flow's state was modified or not during the course of packet processing. By querying the SMM, the Packet Management Module (§3.2) or Replication Module (§3.3) will perform less work for unmodified state.

Active vs. Standby States. Unlike Split/Merge, the State Management Module also contains a set of flow state that consists of backups of other replica's state, called *standby state*. Figure 4 depicts three replicas, each storing a set of active and standby state. In contrast to active state, standby state is not released to the application unless a failure has been signaled to the SMM (from the SDN controller §3.4). The transition from standby state to active state is simple: the SMM simply begins responding with the state when a middlebox application requests access to it. This transition is depicted in Figure 5.

Overall, the Split/Merge [21] system provides the ability to classify middlebox application state and control access to it; howerver, it does not provide HA ca-

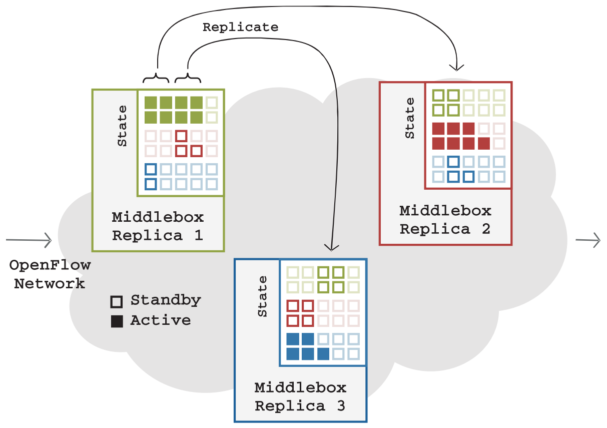


Figure 4: Pico replication of state

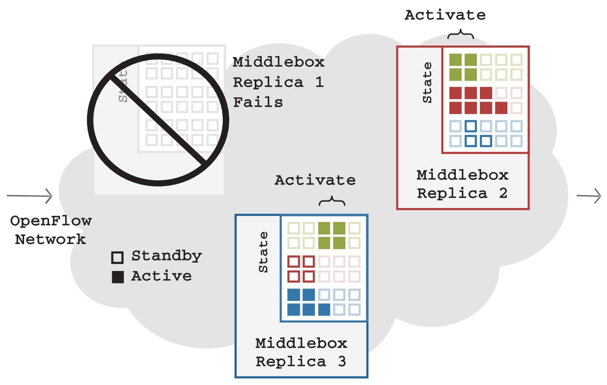


Figure 5: Transparent failure recovery

pabilities. PR and Split/Merge together provide robust system support for both elasticity and HA.

3.2 Packet Management Module

The Packet Management Module ensures per-flow state replication maintains consistency. It does this through input and output buffers and acting as an intermediary between a middlebox application and the network.

Output Buffering. Using output buffering, the system ensures that the network output from a middlebox replica is not seen by the external world until a checkpoint of the flow states is committed at the backup node. The rate at which the output buffer is released corresponds to the checkpoint frequency. Output buffering can introduce delay and burstiness into the egress network traffic if the checkpoint frequency is lower than the arrival rate of packets at the middlebox. In order to minimize the impact of output buffering, the checkpoint frequency has to be high.

PR leverages the independence of flows in a middlebox to perform efficient output buffering, using two techniques. First, given that a middlebox is composed of hundreds of flows operating independently of one another, PR can maintain a different output buffer for each flow, eliminating the need to wait for other flows to be replicated. While one set of flows are being checkpointed, the middlebox can continue processing packets belonging to other flows. Second, individual flows can be checkpointed at a very high frequency, since their state is small relative to the rest of the state in the system. High frequency flow-level checkpointing allows the flow's output buffers to be released quickly, resulting in lower latency overhead and minimal impact on the shape of egress flow traffic.

Certain classes of applications may have reasons to require flow state checkpointing without output buffering. One example is Bro [18], an IDS, that analyzes every packet in the flow and maintains extensive state information. However, its state is not critical to forwarding of the flow. As a result, the PMM exposes an interface for an application to explicitly disable output buffering.

Input Buffering. A typical checkpoint begins by suspending any execution that may affect the state that comprises the checkpoint. In Pico Replication, the fact that no packets belonging to a particular flow are being processed implies that the flow state associate with that flow will not be accessed. Therefore, halting input of a particular flow is effectively a suspend operation and is sufficient for a checkpoint of the flow state to commence. Once a flow is suspended and its state is copied elsewhere, the appropriate output buffer is released.

Replication Groups. One or more flows can be grouped together into a single logical unit that can be replicated to the same target with the same checkpoint frequency. Batching flows into replication groups reduces the computational overhead of maintaining multiple replication streams. The replication target and the checkpoint frequency of each group can be configured independent of other replication groups in the system. Flows can also be moved across replication groups, allowing the system to dynamically adapt to changes in network traffic.

3.3 Replication Module

The Replication Module interacts with the State Management Module and the Packet Management Module in order to implement a policy for replication. A replication policy controls flow checkpointing in the following manner. First, the RM instructs the PMM to halt a flow. Then, the RM obtains the flow state from the SMM. Unless the SMM reports that no changes have been made to the flow state, the RM copies the flow state to an SMM elsewhere in the network. Finally, after receiving confirmation that the flow state is backed up, the RM instructs the PMM to release the output buffer. The RM also informs the SDN (§3.4) of the flow backup targets so that it can quickly recover from failure.

Replication policies can modulate a number of variables, including the number of replication groups in the system, assignment of flows to replication groups, the checkpoint frequency and the replication target for each group.

3.4 SDN Controller

The SDN controller is responsible for detecting and recovering from failure. Failure detection has been extensively researched in the past and there are several well known algorithms [1, 14, 32] and tools [33–35] to detect failures in a timely manner. When a replica failure is detected, the SDN controller signals the SMMs in other replicas to activate the hot-standby copies of flow states that belonged to the failed replica. The SDN then re-routes the flows to the (new) instances responsible for them. Apart from temporary packet loss, end-to-end connectivity remains intact.

4 Implementation

Our Pico Replication implementation extends FreeFlow, an implementation of the Split/Merge design paradigm [21]. FreeFlow provides some of the necessary building blocks: namely, it is able to identify and migrate flow states across replicas and reprogram packet forwarding rules in the network accordingly. The FreeFlow system consists of two main components: (a) an application library to interact with the middleboxes and the underlying hypervisor, (b) an OpenFlow [60] SDN controller, based on Floodlight [55]. Figure 6 shows how the FreeFlow library contributes to the implementation of Pico Replication. Figure 7 sketches the execution of a middlebox application that uses a combination of FreeFlow API's (`get_flow`, `put_flow`) and new APIs from our Packet Management Module implementation (`pkt_read`, `pkt_write`). The remainder of this section describes the implementation in more detail.

4.1 State Management Module

The State Management Module is implemented as an extension to the FreeFlow userspace library. By default, the FreeFlow library exposes a set of APIs that allow middlebox applications to offload state management to FreeFlow, while focusing on the application logic. Specifically, the APIs enable the application to create per-flow and coherent states (Figure 3).[3] The FreeFlow `get_flow`/`put_flow` interface enables the application to access the relevant state bound to a flow, as shown in Figure 6. This interface also enables the application to define the boundaries of a state transaction. We augmented the `put_flow` interface with an optional flag that allows the application to indicate whether the flow state was modified during the transaction (i.e., packet processing). Unmodified flows will not be replicated.

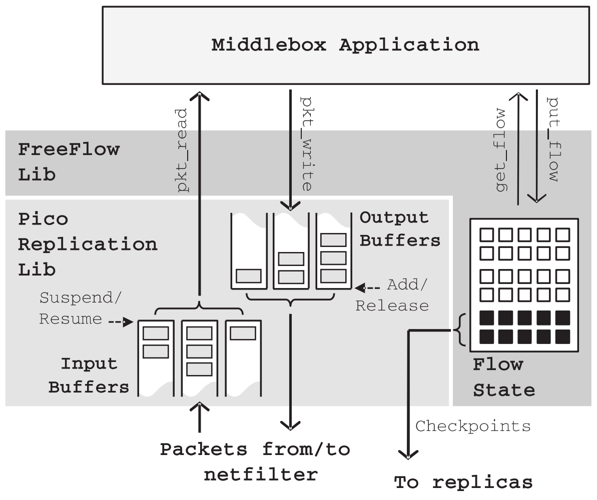


Figure 6: Input and output buffers in Pico Replication

4.2 Packet Management Module

In order to be able to suspend and resume flows and control the release of application's outputs, the PMM interposes on the ingress path of a packet from the network to the application and vice versa. PR organizes flows in the system into replication groups, as described in Section 3.2. Every replication group has an associated input and output buffer. Packets typically enter an input buffer in the PMM before reaching the application and enter an output buffer in the PMM before exiting to the network.

The PMM is implemented using the Netfilter [58] kernel module to copy packets arriving at the network interface to a user space memory region, corresponding to an input buffer. Similarly, when packets are eventually sent from the output buffer out onto the network, the kernel module copies packets from the user specified buffer into kernel space and forwards it onto the appropriate network interface. Copying to/from user

[3]Depending on application needs, the coherent state can either be strongly or eventually consistent. In our experience analyzing and building middleboxes, most shared middlebox state, statistics counters for example, require only eventual consistency. Similar to the Bayou [28] storage system approach, FreeFlow uses application-specific combiner functions to merge updates on eventually consistent state from multiple replicas.

```
while (1)
{
  //read packet from network
  pkt *p = pkt_read();

  //get flow key from packet
  flow_key k = extract_key(p);

  //increment refcnt of flow k
  flow_state *f = get_flow(k);

  ... process packet ...

  //write packet to network
  pkt_write(k, p);

  //decrement refcnt of flow k
  put_flow(k);
}
```

Figure 7: A skeletal code demonstrating the control flow inside the middlebox application using the augmented FreeFlow API.

space incurs a high throughput overhead on the order of 50%. We are working on supporting additional frameworks for fast user space packet processing using direct I/O, like netmap [22], Intel's Data Plane Development Kit [54] and vPF_RING [59]. Alternatively, input and output buffers could be implemented at the Open vSwitch layer in the hypervisor, at the cost of increased hypercall overhead to manage the buffers.

Shown in Figure 6, from the middlebox's perspective, packets are received from an input buffer or sent to an output buffer using a library interface consisting of two primary functions: `pkt_read` and `pkt_write`, respectively. The application issues `pkt_read` to obtain a reference to packets arriving from the network. The PMM scans the input buffers for available packets and returns a reference to a packet from one of the buffers. If all input buffers are currently suspended, the API call blocks or returns an error code if called in non-blocking mode. The application can forward modified or unmodified packets or inject new ones onto an output buffer using the `pkt_write` functions. If the output buffer is full, the packet is dropped and the application is informed accordingly. We also supply a `pkt_write_direct` function to explicitly inform the system that a particular packet can be sent out immediately, rather than being sent to an output buffer. A worker thread scans the output buffers of various replication groups for available packets that can be injected back into the network.

4.3 Replication Module

The Replication Module implementation associates a separate thread with each replication group for checkpointing and replicating flow state to the group's target. At the start of a checkpoint epoch, the replication group's worker thread creates an output buffer. At the end of the epoch, the thread suspends the flows in the replication group by instructing the PMM to suspend the respective input buffer. Once the application has released any lingering references to flows in the replication group (this condition is exposed by the FreeFlow library based on reference tracking with `get_flow`/`put_flow`), the flow states are copied from the SMM to a temporary buffer. The PMM is instructed to resume the input buffer, thereby the flows in the group are also resumed. The RM uses the control network for carrying replication stream traffic between various nodes in the cluster. Once the checkpoint (in the temporary state buffer) is replicated to the target, the PMM is instructed to release the output buffer corresponding to the checkpoint.[4]

If the backup replica for a given replication group fails, the RM at the primary retries a few times before terminating the checkpointing thread and disabling input/output buffering for the associated replication group. It informs the SDN controller of this failure and waits until instructed by the controller to restart the replication process to a new backup target. During this period the flows associated with the replication group will not be protected against failures.

4.4 SDN Controller

We augment the FreeFlow SDN controller to detect failure and activate standby flows in other replicas. When a middlebox replica VM fails, the SDN controller receives an OpenFlow [60] `PORT_DOWN` message from the host hypervisor's Open vSwitch [19]. Host failures (i.e., switch failures) are detected using the OpenFlow protcol's `ECHO` messages that serve as heartbeats between the controller and OpenFlow switches. The failure recovery process involves leveraging the FreeFlow SDN controller to re-route the flows—using OpenFlow—to the (new) instances responsible for them. Since the recovery procedure and the FreeFlow scale-in procedure are the same from the standby replica's standpoint, no special recovery logic is needed beyond that implemented in the FreeFlow library. The application is unaware of both scale-in and failure recovery.

[4]Flows that are not protected are put into a default group whose input and output buffers are not subject to suspend/resume and add/release respectively.

5 Example Replication Policies

Using the tuning knobs provided by Pico Replication, one can implement a variety of policies ranging from a simple daisy chain replication to much more sophisticated and adaptive approaches, that can take into account parameters such as network load distribution, QoS, HA resource overhead, network topology, fault domains, etc. In our implementation, policies are specified to the SDN controller, which co-ordinates replication related tasks across the cluster, such as configuring replication groups and failure recovery. Here we describe two example policies.

Differentiated Pico Streams. Platform as a Service (PaaS) providers like Heroku [48] and Cloud Foundry [46] offer services such as key-value data stores, media streaming, SMS and push notification, big data analytics, and so on. When protecting middleboxes operating in these environments, the HA policy has to take into account the QoS requirements of the network flows and the system resources available for HA purposes.

A naïve HA policy could set a very high replication frequency and apply it to all flows in the system irrespective of their QoS requirements. While the performance impact on end-to-end throughput and latency would be minimal, the HA resource overhead (CPU and replication bandwidth) can quickly overwhelm the system.

Using Pico Replication's ability to replicate subsets of flows at different frequencies, an adaptive policy can be implemented to take advantage of the variegated QoS demands of the flows. By monitoring the flows in the system, replication can be configured to checkpoint the flows as groups with different frequencies based on their QoS requirements. As a simple example, consider a scenario wherein the middlebox application cluster is processing a large number of HTTP flows and a relatively small number of flows involving memcached [51] clients and servers. Since a memcached transaction could complete in just one RTT, it is essential to replicate the flow state at a very high frequency, to keep latency impact negligible. HTTP transactions on the other hand could transfer hundreds of kilobytes of data [57]. If the latency impact on page load time is not perceivable, the HTTP flows can be replicated at a lower frequency, thereby conserving system resources.

Elastic Pico Chords. This policy is similar to techniques used in systems like Chord [26], Dynamo [10] and FAWN [4]. In an N node cluster, the nodes can be arranged in a ring using consistent hashing. To ensure even distribution of load, multiple virtual nodes can be assigned to each physical node, such that each virtual node owns a single non-contiguous slice of the ring. Incoming flows are assigned to the virtual node that follows the flow in the ring, in the clockwise direction. The backup for a given flow is placed at the virtual node immediately following its current virtual (primary) node.

Replication groups on a node can be automatically formed by grouping flows according to their backup destinations. Such groups can be split into smaller groups if subsets of its flows are to be replicated at different replication frequencies. When a node fails, its load is evenly dispersed across the rest of the cluster, such that the overall load balance in the cluster is maintained. However, unlike the DHT style query forwarding approach adopted by Chord, in an SDN environment, the network controller can be leveraged to automatically (re)route a flow to the appropriate replica (primary or backup).

Since the Chord style algorithm adapts to changes in cluster memberships, elastic middlebox applications can maintain the load distribution of both the primary and backup as the cluster scales out or scales in according to load.

6 Evaluation

We evaluated Pico Replication across different middlebox setups. We focused on the following goals:

- Demonstrate Pico Replication's ability to transparently failover a flow dependent middlebox, while incurring very low performance overhead during normal operation. (Requirements R1& R2)
- Demonstrate a simple Differentiated Pico Stream HA policy that provides low overhead HA to a class of flows, while maintaining performance isolation at scale. (Requirement R3)
- Using a homogeneous workload, study the impact on system utilization and performance as the replication frequency and the number of replication groups vary.

We evaluate PR's performance with applications that require continuous [18, 36–40, 61], rather than one-time synchronization (Table 1), as the former fully exercises the replication subsystem. When comparing PR with other HA techniques, we do not evaluate HA approaches used in commercial middlebox systems. Some do not preserve flow state on failover [30] and many resort to in-house proprietary implementations [29] whose details are not disclosed publicly. Open source implementations such as Netfilter's Connection Tracking [58] perform one-time or highly coarse-grained synchronizations (e.g., during TCP state machine transitions). At the time of this writing, we were unable to find non-proprietary flow state preserving HA solutions for middleboxes requiring continuous synchronization. As a baseline comparison, we evaluate the performance of Remus [9], a VM replication based HA solution, as it provides transparent failure recovery (Requirement R1).[5]

Middlebox Application and Workload. We implemented a generic middlebox service, MBServ, that performs packet inspection and request routing. The functions performed by MBServ are representative of the operations carried out by popular middlebox applications like the Suricata [61] intrusion prevention system and ADCs like Riverbed's Stingray Traffic Manager [39] that are used for layer-7 firewalling, request rate shaping and load balancing. Note that, by default, these systems do not support stateful failure recovery [30]. While traffic load is redistributed, ongoing connections handled by the failed node are dropped by other nodes in the cluster, due to lack of related flow state.

The MBServ application runs inside a VM that sits in front of a pool of backend servers on a protected network. MBServ interposes on Client TCP requests from an external network. MBServ interacts with the Pico Replication library. It reads packets from the network interface, using the `pkt_read` call. The application maintains a TCP state machine for every flow and inspects the payload for occurrences of a predefined set of malicious strings. It then forwards packets to the destination network using the `pkt_write` call.

Incoming packets that do not conform to the protocol state machine of their respective flows, or have no corresponding flow in the system, are dropped. Hence, lack of middlebox state replication will result in loss of end-to-end connectivity on failover because, on failure, the flow will be reassigned to a new middlebox that has no prior knowledge of the flow.

Unless otherwise noted, all experiments use a fixed request and response size of 1400 bytes. Latency and CPU utilization values reported in this section correspond to the 95th percentile values, indicating peak latency and peak utilization, respectively.

Experimental Setup. Our experimental setup consists of a cluster of 8 heterogeneous physical hosts running the Xen [6] 4.2 hypervisor, Linux 3.4 kernel and Open vSwitch [19] based software OpenFlow switches. A hardware OpenFlow switch, BNT G8264, connects the physical hosts together. Pico Replication HA policies and FreeFlow's SDN modules are implemented on top of the Floodlight [55] OpenFlow controller. Unless stated explicitly, the middlebox VMs used throught evaluation were provisioned with two CPUs, on physical hosts with a quad-core Intel Xeon processor.

[5]The version of Remus used in our evaluation uses checkpoint compression, an optimization introduced in RemusDB [17] to performance overhead.

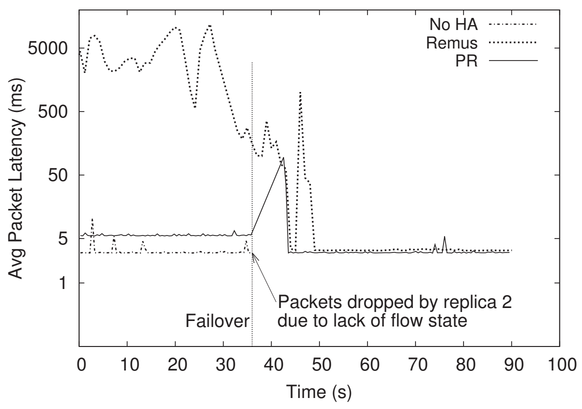


Figure 8: Transparent failure recovery in Pico Replication.

6.1 Stateful Failover

In this experiment, we demonstrate PR's ability to transparently failover a set of flows to another middlebox, without disrupting end-to-end connectivity. We provision a middlebox cluster with three VMs (MBServ1, MBServ2, and MBServ3) on three different hosts (H1, H2, and H3). Figure 8 shows the average end-to-end latency experienced by the client over a 500ms sampling window. MBServ1 is subjected to 50Mbps network traffic spread over 10 flows. After 35 seconds into the experiment, MBServ1 is destroyed. Failure is detected when the SDN controller receives an OpenFlow `PORT_DOWN` message from the host hypervisor's Open vSwitch.

Three different HA configurations are used in this experiment. In each case, the SDN controller reroutes flows from the failed replica (MBServ1) to one or both of the other replicas in the cluster. The HA configurations are as follows:

No HA. In the first case, when MBServ1 fails, its load is redistributed by the SDN controller equally between MBServ2 and MBServ3. No state is replicated between the replicas in the cluster.

Remus. In the second case, we use Remus [9] as an example of a VM-level HA technique that provides transparent failure recovery. The state of MBServ1 is constantly replicated by Remus at a frequency of 40Hz to a different host in the cluster.[6] We denote MBServ1's passive backup copy created by Remus as MBServ1-B. We applied checkpoint compression [17] to reduce the latency overhead of checkpointing. On failure detec-

[6]It is possible to reach replication frequencies of 100Hz with Remus. However, no useful work gets accomplished by the VM at such frequencies since the VM suspend/resume calls themselves take up approximately 7-8ms to complete.

tion, the SDN controller reroutes the flows accordingly to VM1-B.

PR. In the third case, Pico Replication divides the 10 flows handled by MBServ1 into two replication groups, targeted to MBServ2 and MBServ3, respectively. Since only the flow state is replicated, PR is able to achieve a very high replication frequency of 1000Hz. On failure detection, the SDN controller immediately activates the relevant flow states in MBServ2 and MBServ3 and reroutes the flows accordingly.

Figure 8 shows the performance of each HA strategy. In the "No HA" scenario, even though the controller automatically reassigns the flows from MBServ1 to MBServ2 and MBServ3, they do not have the state pertaining to the flows. As a result, the packets belonging to flows from MBServ1 are dropped. With Remus, when MBServ1 fails, its backup copy MBServ1-B resumes execution in host H2, from the most recent and consistent checkpoint of MBServ1. Since a consistent copy of all the flow states is available at MBServ1-B, traffic processing continues uninterrupted.[7] With PR, each replica absorbs a share of the load from MBServ1 and is able to continue service the flows, since it possesses the up-to-date copy of the flow states required to process the packets.

Without any replication, the end-to-end latency is approximately 3ms. With Remus, the overhead is prohibitively high due to the much lower replication frequency (40Hz), and the overhead associated with suspending and resuming the entire VM. With Pico Replication, the latency is very close to native case, approximately 5ms. This is due to the fact that PR replicates relevant flow states at a very high frequency (1000Hz) to MBServ2 and MBServ3, thereby ensuring quick release of the output buffers associated with each flow. After failure of MBServ1, all scenarios eventually reach performance matching "No HA" because they are no longer replicating the flow state or the VM.

6.2 Pico vs. VM Replication

To demonstrate Pico Replication's performance benefits compared to Remus, we evaluate the performance overhead on a HTTP style flow. The traffic was generated using the Flowgrind [63] tool, for a period of 120 seconds. The request sizes follow a lognormal distribution, with of 10KB, variance of 1KB and a maximum request size of 100KB. The same random seed was used for all trials of the experiment. We measured the application perceived response times per request and the total number of transactions per second.

	Replication Freq. (Hz)	Throughput (txn/s)	Latency (ms)
No HA	N/A	6683 [95]	54.48
Pico Replication	1000	6468 [687]	62.95
Remus	100	30 [2]	207.72

Table 3: Performance overhead of Pico Replication vs. Remus. Throughput metric is HTTP transactions per second [w/standard deviation]. Latency metric is the application perceived RTT per HTTP transaction. 95th percentile latency values are shown.

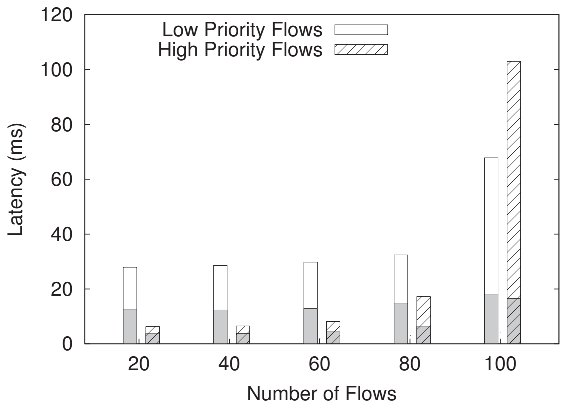


Figure 9: A simple policy that creates two replication groups with different frequencies and targets based on flow priority. The shaded portion of the bar at the bottom indicates the baseline (unprotected) system performance. At 100 flows, MBServ handles a sustained load of 175Mbps.

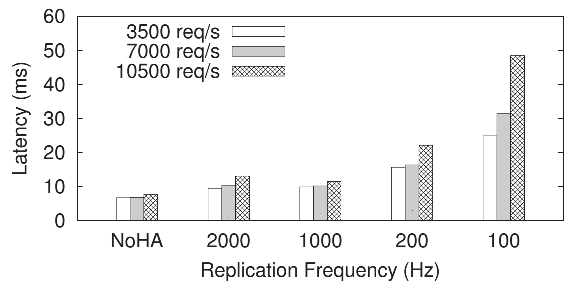
As shown in Table 3, Pico Replication imposes a 15.5% latency overhead and a 3.2% throughput drop compared to a 2.8X latency overhead and a 99.5% drop in throughput imposed by Remus.[8] Again, the low overhead of PR can be attributed to its relatively high checkpoint frequency over Remus (1000Hz vs. 100Hz).

6.3 Differentiated Pico Streams - An Example

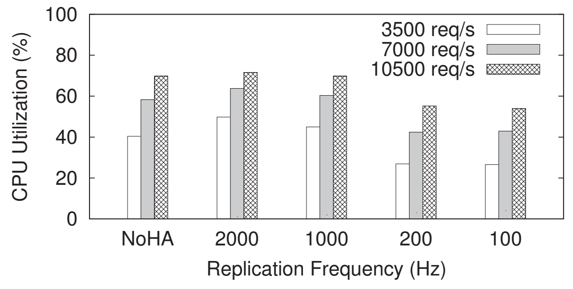
In this experiment, we showcase PR's ability to support various flexible and adaptive HA policies with a simple example. Consider a hypothetical scenario where MBServ processes a set of low and high priority flows. We

[7]With Remus, on failure recovery, host H2 now has both MBServ2 and MBServ1-B. In other words, the entire load from MBServ1 is shed onto host H2. This is an unfortunate consequence of VM replication solutions—the inability to provide fine grained load balanced failover.

[8]The higher variability in throughput with PR is due to frequent context switches between user and kernel mode, during packet processing. The context switches are an artifact of the PMM's heavy use of the Netfilter [58] framework.



(a) Performance



(b) System utilization

Figure 10: Impact of replication frequency at various loads

assume that the ratio of low to high priority flows is 3:1. We setup the low priority flows with a rate of 1Mbps and the high priority flows with a rate of 4Mbps each. Then, we created a simple HA policy that identifies and classifies flows according to their priority (e.g., by their port numbers) and creates two replication groups for each priority respectively, each with a different replication target. By replicating low priority flows at a lower frequency (100Hz) than high priority flows (1000Hz), the middlebox application can conserve CPU resources (the effect of checkpoint frequency on CPU utilization is examined in more detail in Section 6.4).

The performance impact of the differentiated replication policy described above is shown in Figure 9 As the number of flows in the system (load) increases from 20 to 80, PR is able to provide performance isolation to the high priority group despite the increasing number of lower priority flows in the system. At 100 flows, the performance begins to degrade quickly, indicating that more resources need to be allocated to the system to maintain the same quality of service. However, given a fixed amount of resources, PR enables a HA policy to be instantiated on a middlebox that will trade performance for resources on a per-flow basis by adjusting checkpoint frequencies.

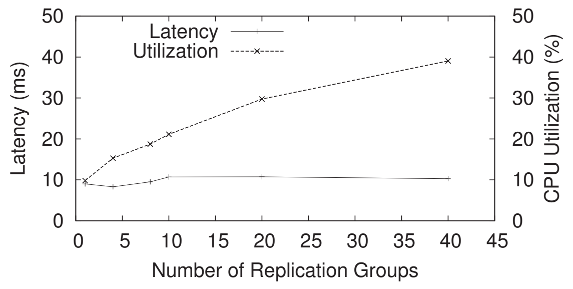


Figure 11: Impact of number of replication groups on latency and system utilization on a 8 CPU middlebox VM. (Latency values are computed using the geometric mean.)

6.4 Performance vs. Utilization Trade-Off

In the following set of experiments, we analyze the parameters that are available at a policy designer's disposal, when attempting to strike a balance between performance and system utilization. The workload for this experiment models communication between two tiers of a multi-tiered application deployment, where each tier is in a separate network. The clients in one tier generate traffic at a rate of 3500 requests per second using a fixed sized connection pool. The two tiers are scaled gradually by adding more resources (VMs) such that the request rate increases in units of 3500.

Impact of Replication Frequency. In general, higher checkpoint frequencies (or lower checkpoint intervals) result in quicker release of network output buffers, thereby minimizing the impact on end-to-end latency, at the cost of higher CPU usage. Figure 10a shows the impact on end-to-end latency at various replication frequencies when the load on MBServ increases progressively.

Figure 10b shows the corresponding CPU utilization. As the replication frequency decreases, the CPU usage decreases. At very low replication frequencies of 200Hz and 100Hz, the TCP endpoints themselves backoff, thus reducing the overall load on the system.

While PR can be configured to replicate state at 2kHz (or 500us), the performance improvement is negligible at the cost of increased CPU usage, denoting a point of diminishing return. Also, at high loads (e.g., 10K requests/s), the 2kHz replication frequency begins to have a negative impact on the performance of MBServ, as observed in Figure 10a. This is due to the fact that checkpointing at very low intervals leaves no room for the system to do any useful work (packet processing).

Impact of Replication Groups. Figure 11 shows the performance impact and CPU utilization as a function of number of concurrent replication groups in the system. We allocate 8 CPUs to the middlebox VM in this experiment. MBServ is processing 80Mbits/s worth of traffic, spread across 40 flows. The flow state is replicated at a rate of 1000Hz. Given a high replication frequency and a fixed load, the number of concurrent replication groups has minimal impact on the performance. However, as the number of replication groups (a thread per replication group) increase, the context switch overhead is no longer negligible, resulting in increased CPU utilization.

7 Conclusion

Middleboxes have become an integral part of networks of various scales, to an extent that today they constitute some of the most critical pieces in enterprise infrastructures [25]. As a result, when failures occur, entire infrastructures crash in a spectacular fashion [44, 49], taking down hosted applications for hours at a stretch [53]. Yet, despite recent trends that exploit the inherent characteristics of middleboxes to provide system support for this important class of applications, high availability support remains limited. We have presented Pico Replication, a system that provides HA for flow-oriented middlebox applications with low overhead. Pico Replication is an important addition to the growing set of tools that help middleboxes become more "cloud ready": software-defined, extensible, scalable, and highly available.

8 Acknowledgments

We would like to thank our shepherd, Marvin Theimer, and the anonymous referees for their helpful comments.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science*, 220, 1999.

[2] Z. Al-Qudah, M. Rabinovich, and M. Allman. Web Timeouts and Their Implications. In *Proc. of International Conference on Passive and Active Measurement*, 2010.

[3] M. Allman. On the Performance of Middleboxes. In *Proc. of Internet Measurement Conference*, 2003.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[5] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proc. of Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2012.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[7] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault-tolerance. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2005.

[9] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2008.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2002.

[12] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proc. of ACM Conference on Virtual Execution Environments VEE*, 2008.

[13] A. Gember, R. Grandl, J. Khalid, S.-H. Shen, and A. Akella. Design and Implementation of a Framework for Software-Defined Middlebox Networking. Technical Report TR1794, University of Wisconsin-Madison, 2013.

[14] D. K. Gifford. Weighted Voting for Replicated Data. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 1979.

[15] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. ReSpec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Proc. of ASPLOS*, 2010.

[16] B. C. Ling, E. Kiciman, and A. Fox. Session state: Beyond soft state. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2004.

[17] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent High Availability for Database Systems. *PVLDB*, 4(11), 2011.

[18] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24), 1999.

[19] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of ACM Workshop on Hot Topics in Networks*, 2009.

[20] S. Rajagopalan, B. Cully, R. O'Connor, and A. Warfield. SecondSite: Disaster Tolerance as a Service. In *Proc. of ACM Conference on Virtual Execution Environments VEE*, 2012.

[21] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2013.

[22] L. Rizzo. Netmap: A Novel Framework For Fast Packet I/O. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2012.

[23] D. J. Scales, M. Nelson, and G. Venkitachalam. The Design and Evaluation of a Practical System for Fault-Tolerant Virtual Machines. Technical Report VMWare-RT-2010-001, VMWare, Inc., 2010.

[24] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2012.

[25] J. Sherry and S. Ratnasamy. A Survey of Enterprise Middlebox Deployments. Technical Report UCB/EECS-2012-24, EECS Department, University of California, Berkeley, 2012.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communications Review*, 31, 2001.

[27] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3), 1985.

[28] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[29] F5 Networks Inc., BIG-IP Configuring High Availability. `http://support.f5.com/kb/en-us/products/big-ip_ltm/manuals/product/tmos_management_guide_10_0_0/tmos_high_avail.html`.

[30] Riverbed Technology, Stingray Traffic Manager - User Manual. `https://support.riverbed.com/software/stingray/trafficmanager.htm`.

[31] Vyatta Inc., High Availability Reference Guide. `http://www.vyatta.com/downloads/documentation/VC6.5/Vyatta-HA_6.5R1_v01.pdf`.

[32] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-style Failure Detection Service. In *Proc. of International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998.

[33] The Corosync Cluster Engine. `http://corosync.org/`.

[34] Linux-HA Project. `http://www.linux-ha.org/doc/`.

[35] Pacemaker: A Scalable High Availability Cluster Resource Manager. `http://clusterlabs.org/`.

[36] F5 Networks Inc., BIG-IP Product Suite. `http://www.f5.com/products/big-ip/`.

[37] Embrane Inc., heleos. `http://www.embrane.com/products/heleos`.

[38] Citrix Systems Inc., NetScaler ADC. `http://www.citrix.com/netscaler`.

[39] Riverbed Technology, Stingray Product Family. http://www.riverbed.com/products-solutions/products/application-delivery-stingray/.

[40] A10 Networks Inc., SoftAX Virtual ADC: Software-based Application Delivery Controller. http://www.a10networks.com/products/axseries-softax.php.

[41] Vyatta Inc., Vyatta Network OS for Amazon. http://www.vyatta.com/product/vyatta-network-os/amazon.

[42] Vyatta Inc., Brocade Vyatta vRouter available as a Service. http://www.vyatta.com/content/vyatta-rackspace-cloud.

[43] Riverbed Technology: Infographic - ADC as a Service. http://media-cms.riverbed.com/documents/Riverbed-ADCaaS-Infographic-May7-2013.pdf, 2013.

[44] Amazon AWS Outage Takes Down Netflix On Christmas Eve. http://www.forbes.com/sites/kellyclay/2012/12/24/amazon-aws-takes-down-netflix-on-christmas-eve/, December 2012.

[45] Amazon AWS Outage Summary. http://aws.amazon.com/message/67457/, June 2013.

[46] Cloud Foundry. http://www.cloudfoundry.com/.

[47] Amazon Web Services, Elastic Load Balancing. http://aws.amazon.com/elasticloadbalancing/.

[48] Heroku. https://www.heroku.com/.

[49] Heroku learns the hard way from Amazon EC2 outage. http://SearchCloudComputing.com, January 2010.

[50] Heroku - Incidence Report. https://status.heroku.com/incidents/386, June 2013.

[51] Memcached - A Distributed Memory Object Caching System. http://memcached.org.

[52] Rackspace: Load Balancing as a Service. http://www.rackspace.com/cloud/load-balancing/, 2013.

[53] RightScale Infographic Shows Average of 7.5 Hours to Recover from Data Center and Cloud Outages. http://www.rightscale.com/news_events/press_releases/2013/rightscale-infographic-shows-average-of-7.5-hours-to-recover-from-data-center-and-cloud-outages.php, 2013.

[54] Intel DPDK: Data Plane Development Kit. http://dpdk.org.

[55] Big Switch Networks Inc., Floodlight OpenFlow Controller. http://www.projectfloodlight.org/floodlight/.

[56] Hot Standby Router Protocol (HSRP). http://tools.ietf.org/html/rfc2281.

[57] HTTP Archive - Interesting Stats. http://httparchive.org/interesting.php.

[58] Netfilter Packet Filtering Framework. http://www.netfilter.org.

[59] Virtual PF_RING. http://www.ntop.org/products/pf_ring/vpf_ring/.

[60] The OpenFlow Switch Specification. http://www.openflow.org.

[61] Open Information Security Foundation: Suricata IDS/IPS. http://www.openinfosecfoundation.org/index.php.

[62] Virtual Router Redundancy Protocol (VRRP). http://tools.ietf.org/html/rfc3768.

[63] A. Zimmermann, A. Hannemann, and T. Kosse. Flowgrind: A New Performance Measurement Tool. In *Global Telecommunications Conference (GLOBECOM)*, 2010.