Split/Merge: System Support for Elastic Execution in Virtual Middleboxes

Shriram Rajagopalan[†‡], Dan Williams[†], Hani Jamjoom[†], and Andrew Warfield[‡]

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

Abstract

Developing elastic applications should be easy. This paper takes a step toward the goal of generalizing elasticity by observing that a broadly deployed class of software—the network middlebox—is particularly well suited to dynamic scale. Middleboxes tend to achieve a clean separation between a small amount of per-flow network state and a large amount of complex application logic. We present a state-centric, systems-level abstraction for elastic middleboxes called *Split/Merge*. A virtual middlebox that has appropriately classified its state (e.g., per-flow state) can be dynamically scaled out (or in) by a Split/Merge system, but remains ignorant of the number of replicas in the system. Per-flow state may be transparently split between many replicas or merged back into one, while the network ensures flows are routed to the correct replica. As a result, Split/Merge enables load-balanced elasticity. We have implemented a Split/Merge system, called *FreeFlow*, and ported Bro, an open-source intrusion detection system, to run on it. In controlled experiments, FreeFlow enables a 25% reduction in maximum latency while eliminating hotspots during scale-out and a 50% quicker scale-in than standard approaches.

1 Introduction

The prevalence of Infrastructure as a Service (IaaS) clouds has given rise to a new breed of applications that better support *elasticity*: the ability to scale in or out to handle variations in workloads [17]. Fundamental to achieving elasticity is the ability to create or destroy virtual machine (VM) instances, or *replicas*, and partitioning work between them [14, 34]. For example, a 3-tier Web application may scale out the middle tier and balance requests between them. Consequently, the—virtual—middleboxes that these applications rely on (such as firewalls, intrusion detection systems, and protocol accelerators) must scale in a similar fashion. A recent survey of 57 enterprise networks of various sizes found that scalability was indeed critical for middleboxes [24].

Due to the diversity of cloud applications, supporting elasticity has been mostly the burden of the application or application-level framework [7]. For example, it is their responsibility to manage replicas and ensure that each replica will be assigned the same amount of work [1]. In the worst case, imbalances between replicas can result in inefficiencies, hotspots (e.g., overloaded replicas with degraded performance) or underutilized resources [33].

Unlike generic cloud applications, middleboxes share a unique property that can be exploited to achieve efficient, balanced elasticity. Despite the complex logic involved in routing or detecting intrusions, middleboxes are often implemented around the idea that each individual flow is an isolated context of execution [22, 26, 31]. Middleboxes typically classify packets to a specific flow, and then interact with data specific to that flow [9, 30]. By replicating a middlebox and adjusting the flows that each replica receives from the network—and the associated state held by each replica—any middlebox can maintain balanced load between replicas as the middlebox scales in or out.

To this end, we present a new hypervisor-level abstraction for virtual middleboxes called *Split/Merge*. A Split/Merge-aware middlebox may be replicated at will, yet remains oblivious to the existence of replicas. Split/Merge divides a middlebox application's state into two broad classes: *internal* and *external*. Internal state is treated similarly to application logic: it is required for a given replica to run, but is of no consequence outside that replica's execution. External state describes the application state that is actually scaled, and can be thought of as a large distributed data structure that is managed across all replicas. It can be further subdivided into to classes: *partitioned* and *coherent* state. Partitioned state is exclusively accessed, flow-specific data, and is the fun-

damental unit of reconfiguration in a Split/Merge system. Coherent state describes additional, often "global" state such as counters, that must remain consistent—either strongly or eventually—among all replicas.

We have designed and implemented *FreeFlow*, a system that implements Split/Merge to provide efficient, balanced elasticity for virtual middleboxes. FreeFlow splits flow-specific state among replicas and dynamically rebalances both existing and new flows across them. To enable middlebox applications to identify external state, associate it with network flows, and manage the migration of partitioned state between replicas, we have implemented an application-level *FreeFlow library*. In addition, we have implemented a *Split/Merge-aware software defined network* (SDN) that enables FreeFlow to partition the network such that each replica receives the appropriate network traffic even as partitioned state migrates between replicas.

FreeFlow enables elasticity by creating and destroying VM replicas, while balancing load between them. We have ported Bro [19], a real-world intrusion detection system, and built two synthetic middleboxes on FreeFlow. Using these middleboxes, we show that FreeFlow eliminates hotspots created during scale-out and inefficiencies during scale-in. In particular, it reduces the maximum latency by 25% after rebalancing flows during scale-out and achieves 50% quicker scale-in than standard approaches.

To summarize, the contributions of this paper are:

- a new hypervisor-level state abstraction that enables flow-related middlebox state to be identified, split, and merged between replica instances,
- a network abstraction that ensures that network input related to a particular flow-related piece of middlebox state arrives at the appropriate replica, and
- a system, FreeFlow, that implements Split/Merge alongside VM scale-in and scale-out to enable balanced elasticity for middleboxes.

The rest of this paper is organized as follows. Section 2 describes middleboxes and the Split/Merge abstraction. Section 3 describes the design and implementation of FreeFlow. Section 4 describes our experience in porting and building middleboxes for FreeFlow. Section 5 evaluates FreeFlow, Section 6 surveys related work, and Section 7 concludes.

2 Split/Merge

In this section, we describe the common structure in which middlebox state is organized. Motivated by this common structure, we define the three types of states exposed by the Split/Merge abstraction. We then describe how robust elasticity is achieved by tagging state and transparently partitioning network input across virtual middlebox replicas. We conclude the section with design challenges.

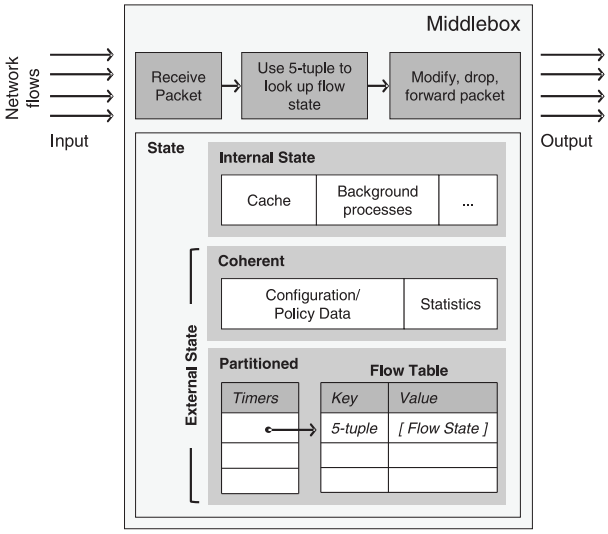

Figure 1: Typical structure of a middlebox

2.1 Anatomy of a Virtual Middlebox

A middlebox is defined as "any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host" [4]. Middleboxes can vary drastically in their function, performing such diverse tasks as network address translation, intrusion detection, packet filtering, protocol acceleration, and acting as a network proxy. However, middleboxes typically process packets and share the same basic structure [9,12,30].

Figure 1 shows the basic structure of a middlebox. State held by a middlebox can be characterized as policy and configuration data or as run-time responses to network flows [9, 26, 30, 31]. The former is provisioned, and can include, for example, firewall rules or intrusion detection rules. The latter, called *flow state* is created on-the-fly when packets of a new flow are received for the first time or through an explicit request. Flow state can vary in size. For example, on seeing a packet from a new flow, a middlebox may generate some small state like a firewall pinhole or a NAT translation entry, or it may begin to maintain a buffer to reconstruct a TCP stream.

Flow state is stored in a *flow table* data structure and accessed using flow identifiers (packet headers) as keys. (Figure 1) Models of middleboxes have been developed that represent state as a key-value database indexed by

addresses (e.g., a standard IP 5-tuple) [12]. A middlebox may have multiple flow tables (e.g., per network interface). It may also contain timers that refer to flow state, for example, to clean up stale flows.

We have performed a detailed analysis of the source code or specifications of several middleboxes to confirm that they fit into this model. We discuss three of them below:

Bro. Bro [19] is a highly stateful intrusion detection system. It maintains a flow table in the form of a dictionary of `Connection` objects, indexed by the standard IP 5-tuple without the protocol field. Inside the `Connection` objects, flow-related state varies depending on the protocol analyzers that are being used. Analyzer objects contain state machine data for a given protocol (e.g., HTTP) and reassembly buffers to reconstruct a request/response payload, leading to tens of kilobytes per flow in the common case. A dictionary of timers is maintained for each `Connection` object. Bro also contains statistics and configuration settings.

Application Delivery Controller (ADC). ADC [35, 38, 40] is a packet-modifying load balancer that ensures the addresses of servers behind it are not visible to clients. It contains a flow table that is indexed by the source IP address and port. Flow-specific data includes the internal address of the target server and a timestamp, resulting in only tens of bytes per flow. ADC also maintains timers for each flow, which it uses to clean up flow entries.

Stateful NAT64. Stateful NAT64 [15] translates IPv6 packets to IPv4 and vice-versa. NAT64 maintains three flow tables, which it calls session tables: for UDP, TCP, and ICMP Query sessions, respectively. Session tables are indexed using a 5-tuple. Flow state, called session table entries (STEs), consists of a source and destination IPv6 address and a source and destination IPv4 address, so is therefore tens of bytes in size. Timers, called STE lifetimes, are also maintained.

2.2 The Split/Merge Abstraction

The Split/Merge abstraction enables transparent and balanced elasticity for virtual middlebox applications. Using Split/Merge, middlebox applications can continue to be written and configured oblivious to the number of replicas that may be instantiated. Each replica perceives an identical VM abstraction, down to the details of the MAC address on the virtual network interface card.

As depicted Figure 2, using Split/Merge, the output of a middlebox application remains *consistent*, regardless of the number of replicas that have been instantiated or destroyed throughout its operation. Slightly more formally:

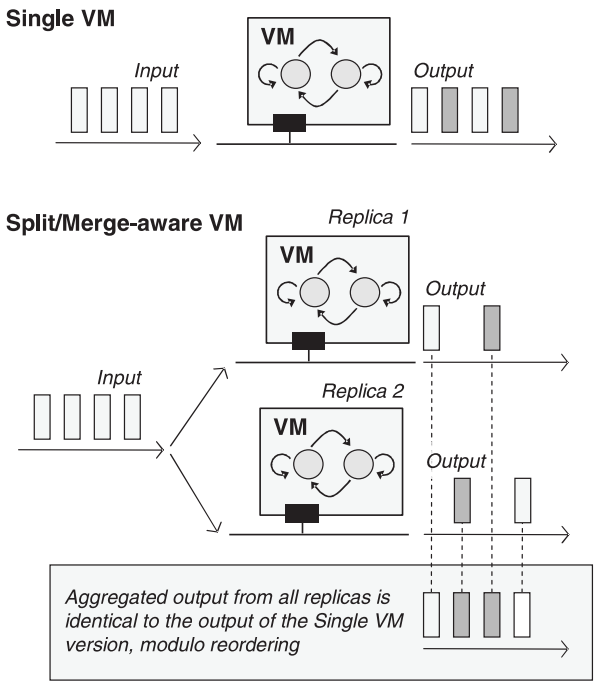

Figure 2: Split/Merge retains output consistency irrespective of the number of replicas.

Definition. *Let a VM be represented by a state machine that accepts input from the network, reads or writes some internal state, and produces output back to the network. A Split/Merge-aware VM is abstractly defined as a set of identical state machine replicas; the aggregate output of which—modulo some reordering—is identical to that of a single machine, despite the partitioning of the input between the replicas. Consistency is achieved by ensuring that each replicated state machine can access the state required to produce the appropriate output in response to its share of the input.*

There are two types of state in a Split/Merge-aware VM (Figure 1): *internal* and *external* state. Internal state is relevant only to a single replica. It can also be thought of as "ephemeral" [5]; its contents can deviate between replicas of the state machine without affecting the consistency of the output. Examples of internal state include background operating system processes, cache contents, and temporary side effects. External state, on the other hand, transcends a single replica. If accessed by *any* replica, external state cannot deviate from what it would have been in a single, non-replicated state machine without affecting output consistency. For example, a NAT may look up the port translation for a particular flow. Any deviation in the value of this state would cause the middlebox to malfunction, violating consistency.

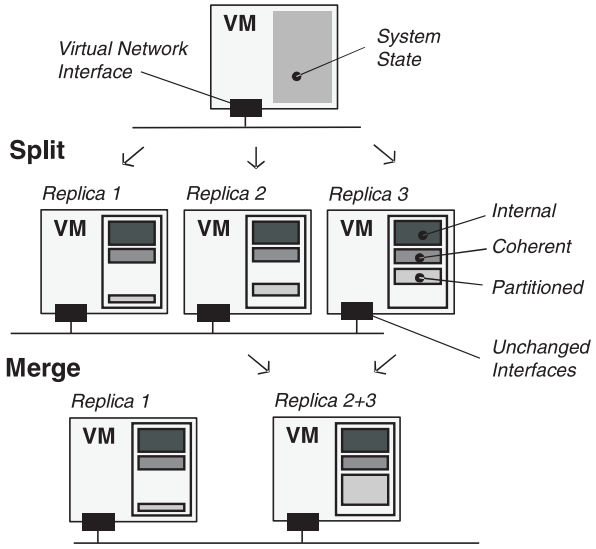

Figure 3: A Split/Merge-aware VM uses the different types of state to achieve transparent elasticity.

As depicted in Figure 1, external state can take two forms: *partitioned* or *coherent*. Partitioned state is made up of a collection of sub-states, each of which are intrinsically tied to a subset of the input and therefore only need to be accessed by the state machine replica that is handling that input. The NAT port translation state is an example of partitioned state, because only the replica handling the network flow in question must access the state. Coherent state, on the other hand, is accessed by multiple state machine replicas, regardless of how the input is partitioned. In Figure 1, the flow table and timers reside in partitioned state, while configuration information and statistics reside in coherent state.

2.3 Using Split/Merge for Elasticity

Figure 3 depicts how the state of a middlebox VM is split and merged when elastically scaling out and in. On scale-out, internal state is replicated with the VM, but begins to diverge as each replica runs independently. Coherent state is also replicated with the VM, but remains consistent (or eventually consistent) because access to coherent state from each replica is transparently coordinated and controlled. Partitioned state is split among the VM replicas, allowing each replica to work in parallel with its own sub-state. At the same time, the input to the VM is partitioned, such that each replica receives only the input pertinent to its partitioned sub-state.

On scale-in, one of the replicas is selected to be destroyed. Internal state residing at the replica can be safely discarded, since it is not needed for consistent output. Coherent state may be discarded when any outstanding updates are pushed to other replicas. The sub-states of the partitioned state residing at the dying replica are merged into a surviving replica. At the same time, the input that was destined for the dying replica is also redirected to the surviving replica now containing the partitioned sub-state.

2.4 Challenges

To implement a system that supports Split/Merge for virtual middleboxes, several challenges need to be met.

C1. **VM state must be classified.** For virtual middlebox applications to take advantage of Split/Merge, each application must identify which parts of its VM state are internal vs. external. Fortunately, the structure of middleboxes (Figure 1) is naturally well-suited to this task. The flow table of middleboxes already associates partitioned state with a subset of the input, namely network flows.

C2. **Transactional boundaries must be respected.** In some cases, a middlebox application may need to convey that it finished processing relevant input before partitioned state can be moved from one VM to another. For example, an IDS may continuously record information about a connection's state; such write operations must complete before the state can be moved. Other cases, such as a NAT looking up a port translation, do not have such transactional constraints.

C3. **Partitioned state must be able to move between replicas.** Merging partitioned state from multiple replicas requires at the most primitive level the ability to move the responsibility for a flow from one replica to another. In addition to moving the flow state, the replica receiving the flow must update its flow table data structures and timer structures so that it can readily access the state.

C4. **Traffic must be routed to the correct replica.** As partitioned state—associated with network flows—is split between VM replicas, the network must ensure that the appropriate flows arrive at the replica holding the state associated with those flows. Routing is complicated by the fact that partitioned state may move between replicas and each replica shares the same IP and MAC address.

The Split/Merge abstraction can be thought of in two parts: splitting and merging *VM state* between replicas (Figure 3), and splitting and merging *network input* between replicas (Figure 2). As such, the challenges can also be classified into those that deal with state management (C1, C2, C3) and those that deal with network management (C4).

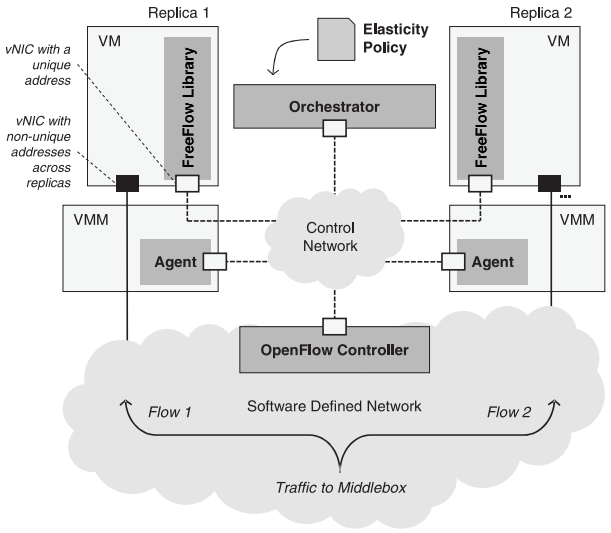

Figure 4: FreeFlow Architecture

```
// MIDDLEBOX−SPECIFIC PARTITIONED STATE HANDLING

create_flow(flow_key, size);     // alloc flow state
delete_flow(flow_key);           // free flow state

flow_state get_flow(flow_key); // increment refcnt
put_flow(flow_key);              // decrement refcnt

flow_timer(flow_key, timeout, callback);


// COHERENT STATE HANDLING

create_shared(key, size, cb); // if cb is null, then use
delete_shared(key);           // strong consistency

state get_shared(key, flags); // synch | pull | local
put_shared(key, flags);       // synch | push | local
```

Figure 5: Interface to the FreeFlow library

3 FreeFlow

FreeFlow implements the Split/Merge abstraction to enable virtual middleboxes to achieve transparent, balanced elasticity. The design of FreeFlow is shown in Figure 4. It consists of four components. First, the state aspects of the Split/Merge abstraction are implemented via the application-level *FreeFlow library*, which addresses the state-related challenges (C1, C2,C3). In particular, through the interface to the library, a middlebox application classifies its state as internal or external and communicates its transactional requirements. Additionally, the library manages all aspects of external state, including the migration of partitioned sub-states. Second, the network aspects of the Split/Merge abstraction are implemented in FreeFlow's *Split/Merge-aware software defined network* (SDN). The SDN addresses the final challenge (C4) and ensures that the correct network flows are routed to the replica maintaining the corresponding partitioned sub-state. Third, the *orchestrator* implements an elasticity policy: it decides when to create or destroy VM replicas and when to migrate flows between them. Finally, *VMM agents* perform the actual creation and destruction of replicas. The four components communicate with each other over a control network, distinct from the Split/Merge-aware SDN.

We have implemented a prototype of FreeFlow, including all of its components shown in Figure 4. Each physical machine runs Xen [2] as a VMM and Open vSwitch [20] as an OpenFlow-compatible software switch. In all components, flows are identified using the IP 5-tuple.

3.1 Guest Library: State Management

Middlebox applications interact with the FreeFlow library in order to classify state as external and identify transaction boundaries on such state. The interface to the library is shown in Figure 5. Behind the scenes, the library interfaces with the rest of the FreeFlow system to split and merge partitioned state between replicas and control access to coherent state.

To fulfill the task of identifying external state, the library acts a memory allocator, and is therefore the only mechanism the middlebox application can use to obtain partitioned or coherent sub-state. Partitioned state in middlebox applications generally consists of a flow table and a list of timers related to flow state; therefore, the library manages both. The library provides an interface, `create_flow` to allocate a new entry in the flow table against a *flow key*, which is usually an IP 5-tuple. A new timer (and its callback) can be allocated against a flow key using `flow_timer`. Coherent sub-state is allocated against a key by invoking `create_shared`, but the key is not necessarily associated with a network flow.

Transaction boundaries are inferred by maintaining reference counts for external sub-states. Using `get_flow` or `get_shared`, the middlebox application accesses external sub-state from the library, at which point a reference counter (refcnt) is incremented. When the application finishes with a transaction on the sub-state, it informs the library with `put_flow` or `put_shared`, which decrements the reference counter. The application must avoid dangling references to partitioned state. If it fails to inform the library that a transaction is complete, the state will be pinned to the current replica.

The library may copy partitioned sub-state across the

control network to another replica in response to a notification from the orchestrator (§ 3.3). When instructed to migrate a flow—identified with a flow key and a unique network address for a target replica on the control network—the library waits for the reference counter on the state to become zero, then copies the flow table entry and any timers across the control network. The flow table at the source is updated to record the fact that the particular flow state has migrated. Upon future `get_flow` calls, the library returns an error code indicating that the flow has migrated and the packet should be dropped. Similarly, when the target library receives flow data—and the flow key for it to be associated with—during a flow migration, the flow table and timer list are updated and the orchestrator is notified. At any one time, only one library instance maintains an active copy of the flow data for a particular flow.

The library also manages the consistency of coherent state across replicas. In most cases, strong consistency is not required. For example, the application can read and write counters or statistics locally most of the time (using the `LOCAL` flag on `get_shared`). Periodically, the application may require a consistent view of a counter. For example, an IDS may need to check an attack threshold value has not been exceeded. For periodic merging of coherent state between replicas, FreeFlow supports *combiners* [7, 16]. On `create_shared`, an application can specify a callback (`cb`) function, which takes a list of coherent state elements as an argument and combines them in an application specific way. In most cases, this function simply adds the values of the counters in the coherent state. The combiner will be invoked automatically by the library when a replica is about to be destroyed. It can also be invoked explicitly by the application either before a reference to the coherent state is obtained (using the `PULL` flag on `get_shared`) or after a transaction is complete (using the `PUSH` flag on `put_shared`). The combiner never runs in the middle of a transaction; `get_shared` using `PULL` may block until other replicas finish their transaction and the state can be safely read. In the rare case that strong consistency is required, the application does not specify a combiner, and library instead interacts with a distributed locking service [3,11]. On `get_shared` (with the `SYNCH` flag), the library obtains the lock associated with the specified key and ensures that it has the most recent copy of the coherent data. The library releases the lock on `put_shared` and the system registers the local copy of the coherent data as the most recent version.

We have implemented the FreeFlow library as a C library. In doing so, we addressed the implementation challenge of allowing flow state to include self-referential pointers to other parts of the flow state. To support unmodified pointers, the library must ensure that the flow state resides at same virtual address range regardless of which replica it is in. To accomplish this, the library allocates a large virtual address space *before* notifying the VMM agent to compute the initial snapshot. Within the virtual address range, the orchestrator provides each replica with a non-overlapping region to service new flow related memory allocations obtained with `create_flow`.

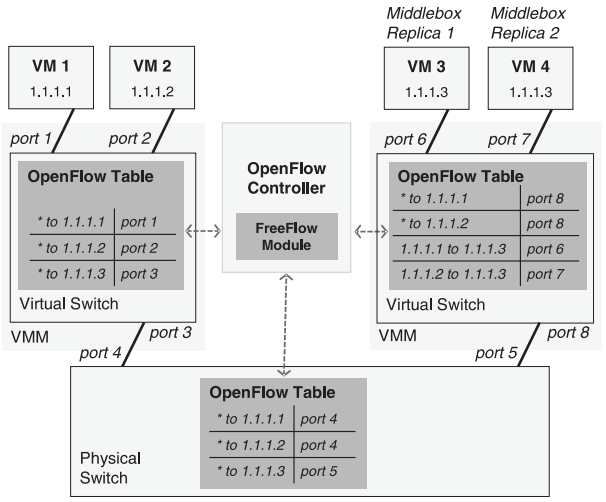

Figure 6: The SDN splits network input to replica VMs based on flow rules. The SDN ensures that traffic from VM 1 arrives at VM 3 and traffic from VM 2 arrives at VM 4. For clarity, we have omitted the flow rules for routing middlebox output.

3.2 Split/Merge-Aware SDN: Network Management

The Split/Merge-aware SDN implements the networking part of the Split/Merge abstraction. Each replica VM contains an identical virtual network interface. In particular, every replica has the same MAC and IP address. Maintaining consistent addresses in the replicas avoids breaking OS or application level address dependencies in the internal state within a VM.

As depicted in Figure 6, FreeFlow leverages OpenFlow-enabled [41] network elements (e.g., switches [20], routers) to enforce routing to various replicas. As packets flow through the OpenFlow network, each network element searches a local forwarding table for rules that match the headers of the packet, indicating they belong to a particular flow. If an entry is found, the network element forwards the packets along the appropriate interface on the fast path. If no entry exists, the packet (or just its header) is forwarded to an *OpenFlow controller*. The OpenFlow controller has a global view of the network and can make a routing decision for the new flow. The controller then pushes a new rule to one or more network

elements so that future packets belonging to the flow can be forwarded without consulting the controller.

The Split/Merge-aware SDN must ensure that packets arrive at the appropriate replica even as partitioned flow state migrates between replicas. To do this, FreeFlow contains a customized OpenFlow controller that communicates with the orchestrator (§ 3.3). When a flow is migrated between replicas, the orchestrator interfaces with the OpenFlow controller to communicate the new forwarding rules for the flow. Packets belonging to new flows are forwarded to the OpenFlow controller by default. The OpenFlow controller picks a replica toward which the new flow should be routed and notifies the orchestrator.

When a flow migration notification is received from the orchestrator, rules to route the flow are deleted from all network elements in the current path traversed by the flow. The flow is then considered *suspended*. Packets arriving from the switches are temporarily buffered at the OpenFlow controller until the flow is *resumed* by the controller, at the new replica. The flow is not resumed until partitioned sub-state has arrived at its new destination. The controller resumes a flow by calculating a new path for the flow that traverses the new replica, installing forwarding rules in the switches on the path, and injecting any buffered packets directly into the virtual switch connected to the new replica.[1]

We implemented the SDN in a module on top of POX [42], a python version of the popular NOX [10] OpenFlow controller. The controller provides a simple web API that allows it to receive notifications from the orchestrator about events like middlebox creation and deletion, or instructions to migrate one or more flows from one replica to another. We addressed three implementation challenges. First, the controller cannot use MAC learning techniques for middleboxes because every replica shares a MAC address. Instead, when replicas are created, the VMM agent registers a replica interface on a virtual switch port with the controller. Second, ARP broadcast requests may cause multiple replicas to respond or unexpected behavior, since they share a MAC address. To avoid this, the controller intercepts and replies to ARP requests that refer to the middlebox IP. Finally, the controller decides which replica a new flow is routed to, so must ensure that bi-directional flows are assigned to the same replica. This is achieved by maintaining a table that maps each flow to its replica that is checked before assigning new flows to replicas.

[1] Alternately, buffering could occur at the destination hypervisor and the controller could update the path immediately upon suspend, thereby reducing its load.

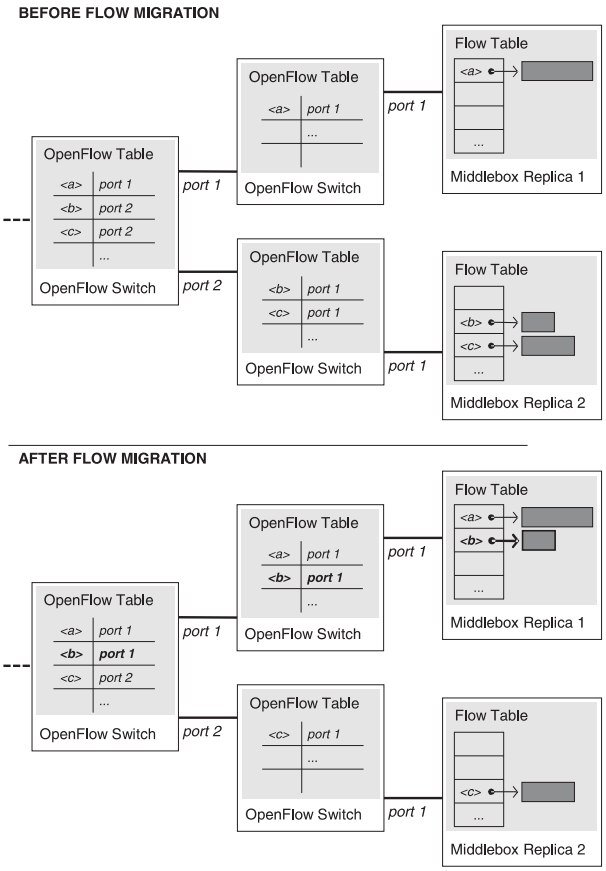

Figure 7: Migrating flow $\langle b \rangle$ from Replica 2 to Replica 1

3.3 Orchestrator: Splitting and Merging

The orchestrator implements the most fundamental primitive for enabling the splitting and merging of partitioned state between replicas: flow migration. Figure 7 shows the migration of a flow $\langle b \rangle$ between two replicas. The orchestrator interacts with other parts of the system as follows. It:

- instructs the SDN to suspend the flow $\langle b \rangle$ such that no traffic of the flow will reach either replica.
- instructs the guest library in Replica 2 to transfer the partitioned state associated with $\langle b \rangle$ to Replica 1.
- instructs the SDN to resume the flow by modifying the routing of flow $\langle b \rangle$ such that any new traffic belonging to the flow will arrive at Replica 1.

It is possible, although rare in practice, that some packets will arrive at Replica 2 after the flow state has been migrated to Replica 1. For example, packets may be buffered in the networking stack in the kernel of Replica 2 and not yet have reached the application. In case the application receives a packet after the flow state

is migrated, it should drop the packet.[2]

The orchestrator triggers the creation or destruction of replicas by the VMM Agent (§3.4) in order to scale in or out as part of an elasticity policy. It can trigger these operations automatically based on utilization (resembling the Autoscale functionality in Amazon EC2 [34]) or explicitly in response to user input.

3.4 VMM Agent: Scaling In and Out

The VMM agent creates or destroys replica VMs in response to instructions from the orchestrator. Replica VMs are instantiated from a point-in-time snapshot of the first instantiation of the middlebox VM, before it began processing packets. During library initialization, after variables in the internal state are initialized but before the VM has allocated any external state, the FreeFlow library instructs the VMM agent to compute the snapshot. By definition, the internal state in the replica VM can diverge from the snapshot.

We have implemented the VMM agent in Xen's control domain (Domain 0). Communication with the library is implemented using Xenstore, a standard, non-network based virtual communication channel used between guest VMs and Domain 0 in Xen. Checkpoints are computed with the `xm save` command, and replicas are instantiated with the `xm restore` command.[3] The time to create a new fully operational replica is on the order of a few seconds; it may be possible to reduce this delay with rapid VM cloning techniques [14].

3.5 Limitations

Virtual middleboxes cannot use FreeFlow (or the Split/Merge paradigm in general) unless their structure roughly matches that described in Figure 1. While most middleboxes we have examined do fit this architecture, it should be noted that some middleboxes are more difficult to adapt to FreeFlow than others. The main cause of difficulty is how the middleboxes deal with partitioning granularity and coherent state.

Middleboxes can be composed of numerous layers and modules, each of which may refer to flows using a different granularity. For example, in an IDS, like Bro, one module may store coarse-grained state (e.g., concerning all traffic in an IP subnet), while another may store fine-grained state (e.g., individual connection state). There are two approaches to adapting such a middlebox to FreeFlow. First, the notion of a flow could be expanded to the largest granularity of all modules. In the preceding example, this would mean using the same flow key for all data related to all flows in an IP subnet, fundamentally limiting FreeFlow's ability to balance load. Second, a fine-grained flow key could be used to identify partitioned state, causing the coarse-grained state to be classified as coherent. If strong consistency is required for the coarse-grained state or a combiner cannot be specified, this approach may cause high overhead due to state synchronization.

[2] In this case, the library returns an error code when flow-specific state is accessed (§ 3.1).

[3] In our prototype, the distribution of VM disk images to physical hosts is performed manually.

4 Experience Building Split/Merge Capable Middleboxes

To validate that the Split/Merge abstraction is well suited to virtual middleboxes, we have ported Bro, an open-source intrusion detection system, to run on FreeFlow. To evaluate a wider range of middleboxes, we have also implemented two synthetic FreeFlow middleboxes.

4.1 Bro

Bro is composed of two key components: an Event Engine and a Policy Script Interpreter. Packets captured from the network are processed by the Event Engine. The Event Engine runs a protocol analysis, then generates one or more predefined events (e.g., connection establishment, HTTP request) as input to the Policy Script Interpreter. The Policy Script Interpreter executes code written in the Bro scripting language to handle events. As explained in Section 2.1, the Event Engine maintains a flow table with each table entry corresponding to an individual connection. Each event handler executed by the Policy Script Interpreter also maintains state that is related to one or more flows.

Our porting effort focused on Bro's Event Engine and one event handler.[4] The event handler scans for potential SQL injection strings in HTTP requests to a webserver. The handler tracks—on a per-flow basis—the number of HTTP requests (`num_sqli`) that contain a SQL injection exploit. When `num_sqli` exceeds a predefined threshold (`sqli_thresh`), Bro issues an alert.

Porting Bro to FreeFlow. Porting Bro to FreeFlow involved the straightforward classification of external state and interfacing with the FreeFlow library to manage it. First, we identified all points of memory allocation in the code. If the memory allocation was for flow-specific data, we modified the allocation to use FreeFlow-provided memory instead of the heap. In certain cases, we had to provide custom implementations of standard C++ constructs like `std::List`, to avoid leaking references to FreeFlow-managed memory.

[4] For ease of implementation, we ported the event handler to C++ instead of using the Bro scripting language.

After ensuring partitioned state was allocated in FreeFlow-managed memory, we checked for external references to it. The only two references were from the global dictionary of `Connection` objects and the global dictionary of timers. Since FreeFlow manages access to flow-related objects and timers, we could replace these two global collections. We found that Bro always accesses flow-related state in the context of processing a single packet, and therefore has well-defined transactional boundaries. References from FreeFlow-managed classes to external memory occur only to read static configuration data (internal state).

As expected, there was very little data that we classified as coherent state. We used FreeFlow's support for combiners for non-critical global statistics counters. The combiners were configured to only be invoked by the system (i.e., on replica VM destruction). We did not find any variables that required strong consistency or real-time synchronization across replicas.

Verification. To validate the correctness of the modified system, we used a setup consisting of a client and a webserver, separated by two middlebox replicas running the modified version of Bro. At a high level, we used the client to issue a single flow of HTTP requests containing SQL injection exploits while FreeFlow migrated the flow between the two replicas multiple times. We check for the integrity of state and execution by ensuring (a) Bro generates an alert, (b) the number of exploits detected exactly matches those sent by the client (c) both replicas remain operational after each flow migration. Assuming Bro sees all packets on the flow, the first two conditions cannot be satisfied if the state becomes corrupted during migration. Additionally, the system would crash on flow migration when objects inside FreeFlow memory refer to external memory that does not exist on the local replica.

4.2 Synthetic Middlebox Applications

We built two synthetic FreeFlow based middlebox applications that capture the essence of commonly used real world middlebox applications. The first application is compute-bound. It performs a set of computations on each packet of a flow, resembling the compute intensive behavior of middlebox applications like an Intrusion Prevention System (IPS) or WAN optimizer. The second application modifies packets in a flow in both directions, using a particular application-level (layer 7) protocol, resembling a NAT or Application Layer Gateway. Both middleboxes were built in userspace using the Linux netfilter [39] framework to interpose on packets arriving at the VM. The userspace applications inspect and/or modify packets before forwarding them to the target.

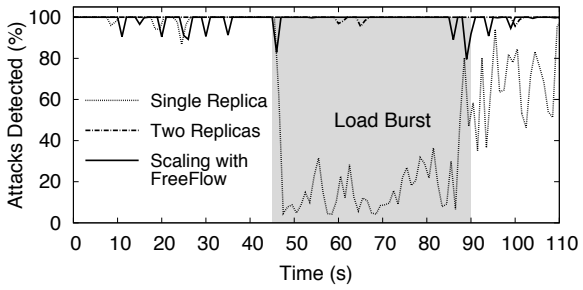

Figure 8: Splitting/Merging Bro for Stateful Elasticity

5 Evaluation

FreeFlow enables balanced elasticity by leveraging the Split/Merge abstraction to distribute—and migrate—flows between replicas. In this section, we evaluate FreeFlow with the following goals:

- demonstrate FreeFlow's ability to provide dynamic and stateful elasticity to complex real world middleboxes (§ 5.1),
- demonstrate FreeFlow's ability to alleviate hotspots created by a highly skewed load distribution across replicas (§ 5.2),
- measure the gain in resource utilization when scaling in a deployment using FreeFlow (§ 5.3), and
- quantify the performance overhead of migrating a single flow under different application loads (§ 5.4).

In our experimental setup, a set of client and server VMs are placed on different subnets. Traffic—TCP or UDP—is routed between the VMs via a middlebox. We evaluate FreeFlow using Bro or one of the synthetic middleboxes described in Section 4.2.

5.1 Stateful Elasticity with Split/Merge

Figure 8 shows FreeFlow's ability to dynamically scale Bro out and in during a load burst, splitting and merging partitioned state. In this experiment, the generated load contains SQL injection exploits; we measure the percentage of attacks detected by Bro to determine Bro's ability to scale to handle the load burst.

Load is generated by a configurable number of cURL-based [36] HTTP clients in the form of a continuous sequence of POST requests to a webserver. The requests contain SQL injection exploits; an attack comprises 31 consecutive requests. Each client is configured to generate 50 requests/second. Throughout the experiment (for 120 seconds), 30 clients generate a base load. We inject a load burst 45 seconds into the experiment by introducing an additional 30 clients and 10 UDP flows

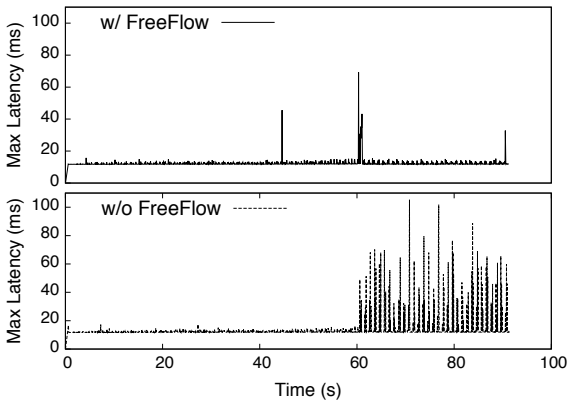

Figure 9: Eliminating hotspots with FreeFlow

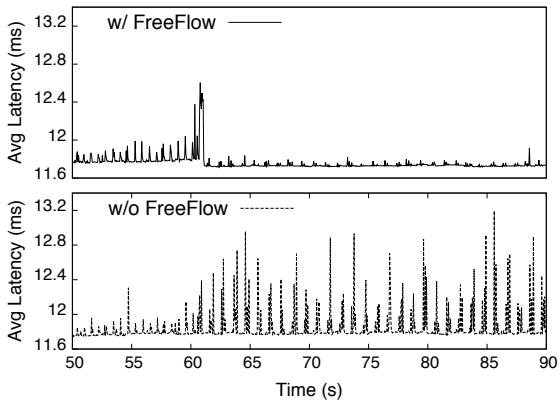

Figure 10: Performance impact of FreeFlow's load rebalancing on hotspots

(1 Mbps each) that do not contain attacks. The load burst lasts 45 seconds, after which the additional client and UDP traffic ceases.

We compare three scenarios: a single Bro instance that handles the entire load burst, a pair of Bro replicas that share load (flows are assigned to replicas in a round-robin fashion), and Bro running with FreeFlow. The FreeFlow scenario begins with a single replica and FreeFlow is configured to create a new replica and split flows and state between them when the number of flows handled by the replica exceeds 60. Similarly, it is configured to merge flows and state and destroy a replica when the number of flows handled by a replica drops below 40.

As shown in Figure 8, until the load burst at $t = 45$s, all three configurations have a 100% detection rate. During the load burst, the performance of the single replica reduces drastically because packets are dropped and attacks are missed. The two replica cluster does not experience any degradation as it has enough capacity and the load is well balanced between the two replicas.

The FreeFlow version of Bro behaves in the same manner as a single replica, until the load burst is detected around $t = 45$s. While partitioned state is being split to a new replica, packets are dropped and attacks are missed. However, the detection rate quickly rises because the two replicas have enough capacity for the load burst. After the load burst ($t = 85$s), FreeFlow detects a drop in load, so merges partitioned state and destroys one of the replicas. The FreeFlow version of Bro continues to detect attacks at the base load with a single replica. FreeFlow therefore enables Bro to handle the load burst without wasting resources by running two replicas throughout the entire experiment.

5.2 Hotspot Elimination

In this experiment, we demonstrate FreeFlow's ability to eliminate hotspots that arise when the load distribution across middleboxes becomes skewed. For the purpose of this discussion, we define a hotspot as the degradation in network performance due to high CPU or network bandwidth utilization at the middlebox.

We use the compute-bound middlebox application described in Section 4.2 under load from 1 Mbps UDP flows. We define our scale-out policy to create a new replica once the number of flows in a replica reaches 100 (totaling 100 Mbps per replica). Flows are gradually added to the system every 500 ms up to a total of 101 flows. After scaling out, the system has two replicas: one with 100 flows and another with just one flow.

As expected, the replica handling 100 flows experiences much higher load than the other replica. The resulting hotspot is reflected by highly erratic packet latencies experienced by the clients, shown in Figure 9 and Figure 10. Figure 9 shows the maximum latency, while Figure 10 shows the fluctuations in the average latency during the last 40s of the experiment. FreeFlow splits the flows evenly among the two replicas thereby redistributing the load and alleviating the hotspot. Ultimately, FreeFlow achieves a 26% reduction in the average maximum latency during the hotspot, with a 73% lower standard deviation.

Irrespective of flow duration and traffic patterns, without FreeFlow's ability to balance flows, an over-conservative scale-out policy may be used to ensure hotspots do not occur, leading to low utilization and wasted resources. By balancing flows, FreeFlow enables less conservative scale-out policies leading to higher overall utilization.

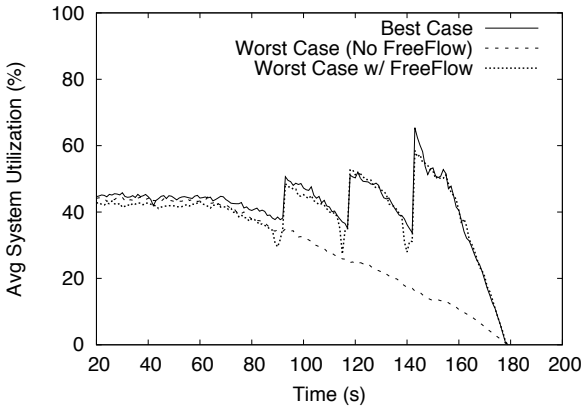

Figure 11: Scaling in with FreeFlow

5.3 Efficient Consolidation

In this experiment, we show how FreeFlow's ability to statefully merge flows between two or more replicas can be used to consolidate resources during low load and improve overall system utilization. We measure how quickly FreeFlow can scale in compared to a standard *kill-based* technique, in which a replica is killed only when all its flows have expired. We also measure the average system utilization per live replica during scale in, shown in Figure 11.

We start with 4 replicas running the compute-bound middlebox application (§4.2), handling 50 UDP flows of 1 Mbps each. One flow expires every 500 ms according to a best case or worst case scenario.

In the best case scenario, the first 50 flows expire from the first replica in the first 25 seconds, enabling the kill-based technique to destroy the replica. The second 50 flows expire from the second replica in the next 25 seconds, enabling the second replica to be destroyed, and so on. In this case, the average system utilization remains high throughout the scale-in process, with a sawtooth pattern as shown in Figure 11.

In the worst case scenario, flows expire from replicas in a round-robin fashion. In a kill-based system, each of the 4 replicas contains one or more flows until the very end of the experiment, preventing the system from destroying replicas. This results in steadily degrading average system utilization over the duration of the experiment.

On the other hand, even in the worst case, FreeFlow can destroy a replica every 25 seconds. To accomplish this, FreeFlow is configured with a scale-in policy that triggers once the average number of flows per replica falls below 50. When scaling in, FreeFlow kills a replica after merging its state and flows with the remaining replicas. Subsequently, in the worst case, FreeFlow maintains average system utilization close to that of the kill-based strategy in the best case scenario and improves the average system utilization by up to 43% in the worst case scenario. Based on the the time at which the first replica was killed in the worst case scenario, FreeFlow can scale in 50% faster than the standard kill-based system.

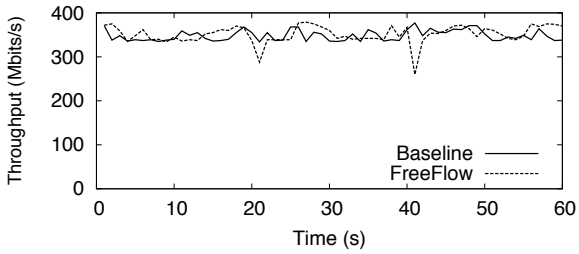

Figure 12: Impact of flow migration on TCP throughput (migration at 20s & 40s)

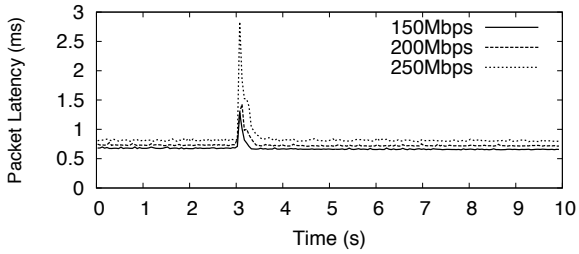

Figure 13: Latency overhead of flow migration

FreeFlow does impact the performance of flows during the experiment; in particular, packet drops are caused by flow migrations that happen when a replica is merged. However, performance impact is low: the average packet drop rate per-flow was 0.9%.

5.4 Migrating Application Flow State

Flow state migration is a fundamental unit of operation in FreeFlow, when splitting or merging partitioned state between replicas. Figure 12 shows the impact on TCP throughput during flow migration compared to a baseline where no migration is performed. We use the Iperf [37] benchmark to drive traffic on a single TCP stream between the client and the server, through the compute-bound middlebox. We perform two flow migrations: one at 20th and another at 40th second, respectively. When sampled at 1 second intervals, we observe a 14 – 31% drop in throughput during the migration period, lasting for a maximum of 1 second.[5]

We further study the overhead of flow migration on a

[5] Due to Iperf's limitation on the minimum reporting interval, (1 second), we are unable to calculate the exact duration of the performance impact.

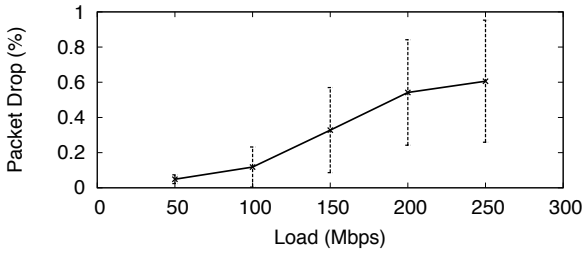

Figure 14: Packet drop rate during flow migration

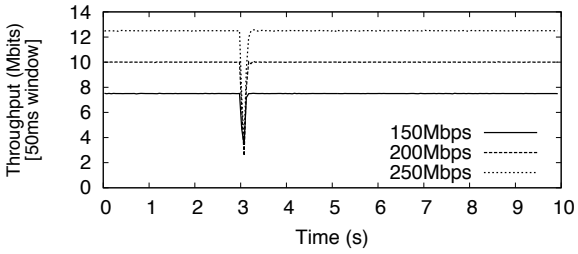

Figure 15: Throughput overhead of flow migration (50 ms window)

single UDP flow using the packet modifier middlebox application (§4.2). For these experiments, the flows are 10 seconds in duration and the migration is initiated after three seconds from the start of the flow. The impact of a single flow migration on end-to-end latency for different flow rates is shown in Figure 13. We observe a maximum of 1 ms increase in latency during flow migration. The latency fluctuations last for a very short period of time (500 ms). Figure 14 shows the overall packet drop rate for the entire duration of the flow. The overall packet drop rate is less than 1% including any disruption caused by the migration. Figure 15 shows the impact on throughput as observed by the client, when the flow migration occurs. The plotted throughput is based on a 50 ms moving window. As the load on the network increases, there is an increase in throughput loss due to flow migration. However, the drop in throughput occurs only for a brief period of time and quickly ramps up to pre-migration levels.

6 Related Work

Split/Merge relies on the ability to identify per-flow state in middleboxes. The behavior and structure of middleboxes has been characterized through the use of models [12]. In other work, state in middleboxes has been identified as global, flow-specific, or ephemeral (per-packet) [30]. On a single machine granularity, MLP [31], HILTI [26], and multi-threaded Snort [21, 22] all exploit the fact that flow-related processing rarely needs access to data for other flows or synchronization with them. CoMb [23] exploits middlebox structure to consolidate heterogeneous middlebox applications onto commodity hardware, but does not address the issue of scaling, parallelism, or elasticity.

Clustering techniques have traditionally been used to scale-out middleboxes. The NIDS Cluster [28] is a clustered version of Bro [19] that is capable of performing coordinated analysis of traffic, at large scale. By exposing policy layer state and events as serializable state [27], individual nodes are able to obtain a global view of the system state. The NIDS Cluster cannot scale dynamically and statefully, as it lacks the ability to migrate lower layer (event engine) flow state and their associated network flows across replicas.

FreeFlow leverages OpenFlow in its Split/Merge-aware SDN. Load balancing has been implemented in an SDN using OpenFlow with FlowScale [25], and wildcard rules can accomplish load balancing in the network while reducing load on the controller [32]. The Flowstream architecture [8] includes modules—for example, VMs—that handle flows and can be migrated, relying on OpenFlow to redirect network traffic appropriately. However, Flowstream does not characterize external state within an application. Olteanu and Raiciu [18] similarly attempt to migrate per-flow state between VM replicas without application modifications.

There are many ways in which different types of applications are dynamically scaled in the cloud [29]. Knauth and Fetzer [13] describe scaling up general applications using live VM migration [6] and oversubscription. Amazon's Autoscaling [34] automatically creates or destroys VMs when user-defined thresholds are exceeded. SnowFlock [14] provides sub-second scale-out using a *VM fork* abstraction. These approaches do not enable balancing of existing load between instances, potentially resulting in load imbalance [33].

7 Conclusion

We have described a new abstraction, Split/Merge, and a system, FreeFlow, that enables transparent, balanced elasticity for stateful virtual middleboxes. Using FreeFlow, middleboxes identify partitioned state, which can be split among replicas or merged together into a single replica. At the same time, FreeFlow partitions the network to ensure packets are routed to the appropriate replica. As networks become increasingly virtualized, FreeFlow addresses a need for elasticity in middleboxes, without introducing the configuration complexity of running a cluster of independent middleboxes. Further, as virtual servers become increasingly mobile, utilizing live VM migration across or even between data centers, the

ability to migrate flows—or split and merge them between replicas—will become even more important.

8 Acknowledgments

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

References

[1] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2010).

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[3] BURROWS, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2006).

[4] CARPENTER, B., AND BRIM, S. Middleboxes: Taxonomy and Issues. RFC 3234, `https://tools.ietf.org/rfc/rfc3234.txt`, 2002.

[5] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The Collective: A Cache-Based System Management Architecture. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2005).

[6] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2005).

[7] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM 51*, 1 (2008).

[8] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow Processing and the Rise of Commodity Network Hardware. *ACM SIGCOMM Computer Communications Review 39*, 2.

[9] GU, Y., SHORE, M., AND SIVAKUMAR, S. A Framework and Problem Statement for Flow-associated Middlebox State Migration. `http://tools.ietf.org/html/draft-gu-statemigration-framework-02`, 2012.

[10] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communications Review 38*, 3.

[11] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIX Annual Technical Conference (ATC)* (2010).

[12] JOSEPH, D. A., AND STOICA, I. Modeling Middleboxes. *IEEE Network 22*, 5 (2008).

[13] KNAUTH, T., AND FETZER, C. Scaling non-elastic Applications Using Virtual Machines. In *IEEE International Conference on Cloud Computing* (2011).

[14] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proc. of ACM European Conference on Computer Systems (EuroSys)* (2009).

[15] M. BAGNULO, P. MATTHEWS, I. V. B. Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. `http://tools.ietf.org/id/draft-ietf-behave-v6v4-xlate-stateful-12.txt`, 2010.

[16] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-Scale Graph Processing. In *Proc. of ACM SIGMOD* (2010).

[17] MELL, P., AND GRANCE, T. The NIST Definition of Cloud Computing. In *National Institute of Standards and Technology Special Publication 800-145* (2011).

[18] OLTEANU, V. A., AND RAICIU, C. Efficiently Migrating Stateful Middleboxes. In *ACM SIGCOMM - Demo* (2012).

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

[20] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. of ACM Workshop on Hot Topics in Networks* (2009).

[21] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proc. of USENIX Conference on System Administration* (1999).

[22] SCHUFF, D. L., CHOE, Y. R., AND PAI, V. S. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Proc. of ACM Symposium on Principles and Practice of Parallel Programming* (2007).

[23] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2012).

[24] SHERRY, J., AND RATNASAMY, S. A Survey of Enterprise Middlebox Deployments. Tech. Rep. UCB/EECS-2012-24, EECS Department, University of California, Berkeley, 2012.

[25] SMALL, C. FlowScale. GENI Engineering Conference (Poster), `http://groups.geni.net/geni/attachment/wiki/OFIU-GEC12-status/FlowScale_poster.pdf`, 2012.

[26] SOMMER, R., CARLI, L. D., KOTHARI, N., VALLENTIN, M., AND PAXSON, V. HILTI: An Abstract Execution Environment for Concurrent, Stateful Network Traffic Analysis. Tech. Rep. TR-12-003, ICSI, 2012.

[27] SOMMER, R., AND PAXSON, V. Exploiting Independent State For Network Intrusion Detection. In *Proc. of Computer Security Applications Conference* (2005).

[28] VALLENTIN, M., SOMMER, R., LEE, J., LERES, C., PAXSON, V., AND TIERNEY, B. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. of International Conference on Recent Advances in Intrusion Detection* (2007).

[29] VAQUERO, L. M., RODERO-MERINO, L., AND BUYYA, R. Dynamically Scaling Applications in the Cloud. *ACM SIGCOMM Computer Communications Review 41*, 1.

[30] VERDÚ, J., NEMIROVSKY, M., GARCÍA, J., AND VALERO, M. Workload Characterization of Stateful Networking Applications. In *Proc. of International Symposium on High-Performance Computing* (2008).

[31] VERDÚ, J., NEMIROVSKY, M., AND VALERO, M. MultiLayer Processing - An Execution Model for Parallel Stateful Packet Processing. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2008).

[32] WANG, R., BUTNARIU, D., AND REXFORD, J. OpenFlow-based Server Load Balancing Gone Wild. In *Proc. of USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (2011).

[33] WELSH, M., AND CULLER, D. Adaptive Overload Control for Busy Internet Servers. In *Proc. of USENIX Symposium on Internet Technologies and Systems* (2003).

[34] Amazon EC2: Auto Scaling. `http://aws.amazon.com/autoscaling/`.

[35] BIG-IP Local Traffic Manager (LTM). `http://www.f5.com/products/big-ip/big-ip-local-traffic-manager/`.

[36] libcurl - The Multiprotocol File Transfer Library. `http://www.tcpdump.org/`.

[37] Iperf: TCP and UDP Bandwidth Performance Measurement Tool. `http://iperf.sourceforge.net/`.

[38] Linux Virtual Server. `http://www.linuxvirtualserver.org/`.

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

[40] Citrix NetScaler ADC. `http://www.citrix.com/netscaler`.

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

[42] POX OpenFlow Controller. `http://www.noxrepo.org/pox/about-pox/`.