Design and Implementation of a Framework for Software-Defined Middlebox Networking

Aaron Gember, Robert Grandl, Junaid Khalid, Shan-Hsiang Shen, Aditya Akella
University of Wisconsin-Madison, Madison, WI, USA
{agember,rgrandl,junaid,shan-hsi,akella}@cs.wisc.edu

ABSTRACT

Increasingly, middleboxes are being deployed as software components and, with the advent of software defined networking, can be deployed at arbitrary locations. However, existing approaches for controlling the operations of middleboxes continue to be rudimentary and ad hoc. As such, a variety of *dynamic* network control scenarios that are crucial to enhancing the security, availability and performance of enterprise applications cannot be realized today.

In this paper, we ask: what is the right way to exercise unified control over the actions of middlebox that enables sophisticated dynamic network control scenarios? Inspired by SDN, we argue that a *software-defined middlebox networking* (SDMBN) framework—which provides fine-grained, programmatic control over *all MB state* in concert with control over the network—is the answer to this question. Thus, we present the design and implementation of OpenMB. OpenMB consists of slightly modified middleboxes that expose a southbound API for importing/exporting middlebox state, a middlebox controller that implements a northbound API to define how state can be accessed or placed, and scenario-specific control applications that orchestrate middlebox and network changes in tandem.

1. INTRODUCTION

Middleboxes (MBs) are network components operating at layers 4-7 through which network traffic passes for inspection and/or modification. As recent quantitative studies have shown, MBs are used widely, for security, facilitating network access, or providing other novel functionality [27, 30].

Modern MB deployments are driven by two trends. First, MBs are increasingly deployed as software components: as VMs, in hypervisors, on end-hosts [19], or as collections of processes [27]. Second, the advent of software defined networking (SDN) has enabled MBs to be deployed at arbitrary locations. These trends align well with the flexibility enabled by recent advances in compute and network virtualization, making it possible to offer novel services, e.g., enabling the creating of rich virtual network topologies.

However, existing approaches for controlling the operations of MBs continue to be rudimentary and ad hoc. As such, a variety of *dynamic* network control scenarios that are crucial to enhancing the security, availability and performance of enterprise applications cannot be realized today.

MB operations are determined by configuration policies and parameters, traffic streams flowing to them, and their internal algorithms and state. A variety of heterogeneous mechanisms are used today to affect the first two factors: e.g., tweaking routing configurations and MB-specific configuration engines [11]. As for the third factor, there is no way today to directly access and modify internal MB algorithms and state, because most MBs are closed systems.

This apparent lack of unified fine-grained control over MBs and their state precludes correct and performant implementation of control scenarios that require re-allocating live flows across MBs. Examples include live migration of enterprise applications for security reasons or resource constraints, elastic scaling up/down of MBs to meet cost-performance tradeoffs, and transparent MB failure recovery (§2).

In this paper, we ask: what is the right way to exercise unified control over the actions of MBs that enables sophisticated dynamic network control scenarios? Inspired by SDN, we argue that a *software-defined MB networking* (SDMBN) framework is the answer to this question.

An ideal SDMBN framework offers *useful abstractions for unified software-driven control* of MB functions across a range of MBs (perhaps from different vendors). At the same time, the framework should not wrest too much control away from the MBs themselves so that vendors can continue to innovate and improve their MB offerings. Such a carefully balanced SDMBN framework can simplify management of complex MB deployments and engender a wide-variety of rich dynamic MB control applications. We believe that it can also help create new hitherto unseen MBs.

Based on an analysis of key scenarios, we argue that SDMBN requires fine-grained, programmatic control over *all MB state* happening in concert with control over the network (§2). There are two key roadblocks to realizing this.

First, compared to switch forwarding state, MB state is highly diverse. We survey of a wide range of MBs and find that there exist coarse commonalities in the structure of MB state. Based on this, we present a comprehensive taxonomy of MB state and argue for the use of a novel scheme to represent state. We then show how to use the taxonomy and rep-

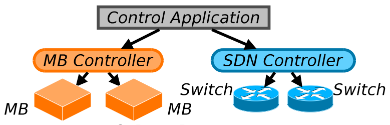


Figure 1: Overview of the elements OpenMB encompasses

resentation to design appropriate state control mechanisms for different types of state.

Second, internal MB logic is complex, and as such, ripping it out of MBs is challenging; it also restricts innovation in the design of MBs themselves. We argue for a novel division of functionality whereby MBs are largely autonomous and continue to be responsible for creating and modifying crucial internal state according to proprietary logic, whereas the location of, and consistency across, these pieces of state is externally controlled. MBs also provide limited external introspection of their actions.

We present the design and implementation of *OpenMB*, an SDMBN framework driven by these insights. Our architecture (§3) consists of an MB controller, control applications and slightly modified MBs as shown in Figure 1.

We design an MB-facing ("southbound") API that defines how MBs receive and export state (§4). The semantics of state operations (e.g., Should an operation be disallowed in some cases?) and representation of state (e.g., Should state be encrypted? Exported per flow?) are tied to the type of state in question. We argue that this API must be augmented with a properly detailed event abstraction that allows MBs to notify the MB controller of the occurrence of, but not the reason behind, internal state establishment or manipulation actions. This helps ensure consistent operation at no loss of performance, and enables rich cross-MB actions, while preserving MB autonomy.

We design a control ("northbound") API for MB state that defines how MB state can be accessed and placed or changed by applications (§5). We carefully trade-off richness of the API for enabling simple and correct application designs. The API helps control applications make network state changes in sync with MB state changes. We design corresponding functionality in the controller for translating between northbound and southbound API calls; these help prevent applications from issuing illicit actions to MBs while also limiting the amount of extra functionality MBs must implement (in addition to the southbound API) to support OpenMB.

We implement the northbound API as a module in Floodlight [5] and modify several MBs—Bro [24], SmartRE [16], and PRADS [10]—to implement the southbound API (§7). We also construct two control applications—live migration and elastic scaling—(§6) that leverage both our northbound API for MBs and OpenFlow [23].

We evaluate the OpenMB framework using these control applications, along with traffic traces captured from an operational enterprise network. In particular, we show that using OpenMB preserves the correctness and performance of MBs in the presence of dynamic changes to an MB deployment: the output of both unmodified MBs and OpenMB-enabled

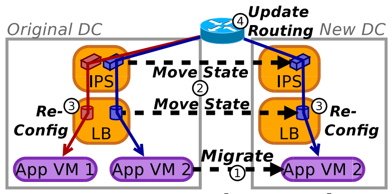


Figure 2: Live migration between data centers

MBs in a live migration scenario is the same, and there is at most a 2% increase in packet processing latency only while MBs are processing southbound API calls. This contrasts with existing techniques which can cause upwards of 9% of flows to be mishandled and up to a 100x increase in packet latency while MB state is being moved.

2. MOTIVATING SCENARIOS

We now describe a few dynamic enterprise scenarios including, live data center migration, elastic MB scaling and load balancing, and MB failure recovery. With each scenario, we derive key requirements that need to be satisfied to support it effectively. We argue that alternate state-of-the-art approaches address some but not all requirements. In particular, because MB state is complex and closed in nature (see §3.1 for a taxonomy of MB state), and these approaches only offer rudimentary and/or indirect control over the state, they can lead to correctness issues or performance degradations (§2.1). We conclude that supporting such dynamic scenarios requires fine-grained, programmatic control over all MB state in tandem with control over the network.

Live Migration Between Data Centers. Enterprises may desire to live migrate application virtual machines (VMs) between private, public, and cloud data centers for reasons of performance, cost, security, resource availability, etc. MBs must be considered as part of the migration process to ensure the security and performance of these applications is preserved, both during and after migration.[1]

MB migration is particularly complex when only a subset of the VMs of an application are migrated, because it requires changes to the configuration and/or internal state residing at MBs; we show an example in Figure 2. In particular, new instances of the appropriate MBs, e.g., an intrusion prevention system (IPS) and a load balancer (LB), should be launched in the new data center and loaded with the internal state for the specific flows or flow groups associated with the migrated app VMs; this state should be derived from the original MB instances in the old data center. In other words:

> **R1:** We need the ability to *move* internal MB state at *fine-granularity*.

In some cases it may be necessary to copy, rather than move, the internal state from existing MB instances. For example, consider replacing the IPS in Figure 2 with a redundancy elimination (RE) decoder [16]. When the RE decoder

[1]Live network migration was considered in LIME [22], but the focus was solely on network forwarding state.

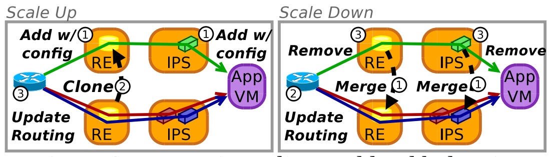


Figure 3: Dynamic scaling and load balancing

is migrated to a new location, it needs the appropriate cache entries to correctly decode received packets. This means:

R2: We need the ability to *clone* internal state.

The new and old MBs should operate with the same semantics (high level policy) during and after live migration. Additionally, live migration may require MB configuration changes to ensure correct MB operation: e.g., the load balancer instances in both the new and original data centers should be reconfigured to balance traffic only among the app VMs in the respective data centers. This means:

R3: We need the ability to *clone and dynamically modify MB configurations*.

When shifting flows to a new MB, the transfer of corresponding internal MB state must happen in tandem with modifying network routing (to route traffic to the new MB). Migrating internal state without regard to routing changes or failing to migrate internal state can cause serious correctness issues: e.g., a load balancer will assign an in-progress transaction to a different server (when the routing update happens before state is moved), and an intrusion prevention system (IPS) may miss (when state has moved and the routing update did not yet take effect) or generate false alerts. Thus:

R4: We need the ability to coordinate MB state migration and configuration update with changes to network forwarding state.

Dynamic Scaling and Load Balancing. Deploying MBs as software components makes it easy to add and remove MB instances as network load changes. Recent work has developed techniques to determine when MBs should be scaled and how many instances should be added or removed [20].

However, making such scaling effective requires meeting the above requirements, plus a few new requirements. As Figure 3 shows, scaling up requires copying (and potentially modifying) the configuration state (requirement R3) from an existing MB instance. For some MBs (e.g., RE), scaling up also requires cloning internal state (requirement R2) from an existing MB instance. On the other hand, scaling down requires consolidating several MB instances into fewer instances, or running multiple MB instances in parallel on the same compute resource with fewer resources assigned to each instance. In both cases, unless we have a way of combining state, we will not be able to operate correctly within the new resource constraints. Thus:

R5: We need the ability to *merge* internal state from multiple MBs.

Second, scale up/down must happen in concert with load balancing among MB replicas, as it helps maximize MB efficiency and reduces the need for additional instances [20]. When flows are short-lived, load can be balanced by carefully assigning new flows to specific MB instances. However, when flows are long-lived, in-progress flows needs to be reassigned to different MB instances to achieve an optimal load distribution. This requires moving the appropriate state (R1) and updating routing (R4).

Failure Recovery. Deploying MBs as software components enables replacement (failover) instances to be launched more quickly. However, since most MBs are stateful, replacement MBs must be loaded with the requisite internal state.

There a few possible approaches for this. One option is to run two instances of the same MB in parallel, with a copy of each packet sent to both instances. But, this requires twice as many compute and network resources. A second option is to create a copy of all MB state in real-time to bootstrap a replacement instance.This is computationally expensive. While the overhead can be reduced by snapshotting at set intervals, some state may still be lost when a failure occurs. A more viable option which is as effective as the first approach, without the overhead or cost, is to keep (and move upon failure) a *minimal live snapshot of only critical state* (e.g., IP address and port mappings from a NAT), with non-critical state (e.g., mapping timeouts) set to default values when a failed MB instance is replaced.

Thus, to support efficient failure recovery across a range of MBs, it helps greatly to know when an MB created such criticial state, and what the state created was. In other words:

R6: We need support for providing introspection into MB operations.

2.1 Existing Techniques are Insufficient

Existing techniques can address a subset of the requirements, or offer alternative options for some of the scenarios discussed above. However, these approaches have limited applicability, tend to waste resources, reduce performance, or lead to correctness issues. We provide qualitative arguments below and present quantitative evidence in §8.

Virtual Machine Snapshots. Running MBs as VMs (or Linux containers) enables the use of VM snapshots [17] as a mechanism for moving and cloning MB state in its entirety. In the live migration scenario, for example, each MB will have the necessary internal state for flows in its data center. However the MBs will also have unneeded state (for flows in the other data center). This wastes MB memory, but more crucially, it can cause incorrect MB behavior, e.g., an IDS might generate false alerts. VM snapshots cannot be used when state from multiple MBs must be moved and merged, e.g., in the case of scale down.

Configuration Protocols. SIMCO [11] is an attempt to provide a standardized MB configuration protocol that can help dynamically modify MB configuration. Unfortunately, it is limited to firewalls and NATs due to its very specific syntax. SIMCO also does not help manage internal state.

Controlling MB Configuration and Routing. Partial control can be achieved by performing MB configuration (using existing interfaces) and routing (using SDN) in tandem [28].

By periodically probing topology, traffic patterns, and MB constraints, a controller can automatically compute an optimal configuration of MBs and the network to satisfy a high level policy. However, this is incomplete: re-routing in-progress flows according to the new configuration without moving, cloning, and merging internal state from MBs that the traffic had touched in the old configuration can impact correctness: e.g., an IPS will have no record of earlier packets from the flows. Only re-routing new flows avoids incorrect MB operations, but prohibits the new configuration from fully taking effect until all existing flows have finished: e.g., a scaled down MB cannot be destroyed until all flows passing through it have completed.

Application-level Libraries. In Split/Merge [25] MBs are modified to use an application-level library which (*i*) provides methods for MBs to allocate, free, and reference internal state, (*ii*) exposes the internal state to a controller so it can be migrated between MBs, and (*iii*) leverages SDN to control the flow of traffic as state is moved. However, Split/Merge abstractions fall short in generality: the abstractions are focused on scaling specific MB types, and it is unclear if they can apply to other MBs/scenarios. A key reason is that there is no way to handle shared internal state, e.g., moving or cloning the cache on a RE decoder, or achieve introspection into MB operations.

3. OpenMB ARCHITECTURE

OpenMB achieves the requirements highlighted in §2 by introducing: (1) programmatic, fine-grained control over all forms of MB state and (2) unifying this control with existing SDN frameworks for controlling L2 and L3 network elements. Together these enable the design of rich control applications to support the scenarios in §2.

In SDMBN, programmatic control (#1) is achieved through the introduction of an MB controller (see Figure 1) and two novel APIs. Unification of MB and network control (#2) is achieved by having the control applications coordinate the control functions they invoke at the SDN and MB controllers. We illustrate these through the example in Figure 4. A control application running on SDMBN, e.g., *migrate*, operates on a view of network switches, links and MBs. To achieve control over MB state, the application invokes the **control API**—this defines how MB state can be accessed and changed by control applications—e.g., *migrate* issues `move(k)` to transfer a subset of state, identified by the key `k`, from MB A to MB B. The MB controller relays these actions to the appropriate physical MBs using the **MB-facing API**—this defines how MBs export and receive state—e.g., the MB controller issues `get(k)`, receives state `s`, and issues `put(s)` to move state.The MB-facing API also defines when MBs need to notify the MB controller that they have established/manipulated state internally to ensure atomicity and provide introspection into MB actions, e.g., an event to re-process packet `p` is raised by MB A; the controller passes it to MB B. When `move` returns successfully, the application

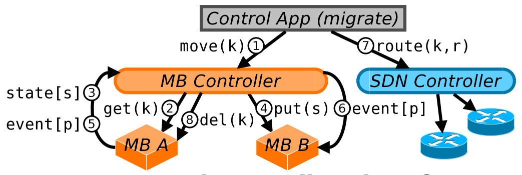


Figure 4: Example API calls and notifications

triggers an update of network forwarding state, e.g., *migrate* issues `route(k,r)` to the SDN controller to change the forwarding for flows identified by key `k` to the route `r`. Finally, the MB controller issues a `delete(k)` to MB A after a set time has passed since `put(s)` returned.

There are two basic roadblocks in designing OpenMB:

- **MB state is highly diverse.** The ability to programmatically control all MB state hinges on identifying commonalities in the structure and semantics of a diverse range of MB state. In §3.1, we argue that such commonalities do exist and present a state taxonomy that forms the basis of our APIs.
- **Internal MB logic is complex.** Indeed, each MB features intricate and unique packet processing logic that is closely tied to internal state. In §3.2, we argue that ripping this logic out—akin to SDN—is difficult, and also undesirable. We then describe an appropriate split of responsibility between MBs and the controller/applications. This defines the richness of our APIs and of the functionality at the controller/applications.

Building upon these insights, in §4 and §5, we describe our design choices for the MB-facing and control APIs, respectively. We describe the design of control applications that use these APIs in §6.

3.1 MB State Taxonomy

In contrast with SDN where switches have a forwarding information base, MBs in SDMBN rely on complex pieces of state that serve diverse purposes. A single MB may receive dozens of configuration inputs, and its internal logic may establish and manipulate hundreds of pieces of in-depth state based on received traffic whose structure and semantics varies significantly across MB types and vendors.

We reviewed several types of MBs from a variety of vendors—firewalls [8, 3], intrusion detection systems [24, 26, 21], load balancers [1], WAN optimizers[2][15, 4, 7], proxies [13], and monitoring systems [10, 9]—to identify commonalities in the structure, semantics, and purpose of their state. We make a few key observations:

- Each piece of MB state fulfills one of three purposes: specifying policies and parameters, supporting packet processing, or reporting MB observations/decisions.
- Each piece of MB state applies either to a specific "flow"[3] or all traffic at the MB.

[2]This includes MBs that perform caching, compression, and redundancy elimination.
[3]We use "flow" to loosely refer to a transport connection, an application session, a pair of communicating hosts, etc.

Role	Description	Partitioning	MB Ops
Configuring	*Policies and parameters that define and tune MB behavior*	Shared Only	MB reads
Supporting	*Details on past traffic to guide MB decisions and actions*	Per-Flow & Shared	MB reads & writes
Reporting	*Quantify observations and decisions*	Per-Flow & Shared	MB writes

Table 1: Taxonomy of MB state

- The same basic data structure tends to be used across different MBs for pieces of state with the same semantics, e.g., MBs maintain an index of current transport connections using a hash table, tuning parameters are specified as key/value pairs, etc.

Based on this, we classify MB state along two dimensions: its role in MB operation—configuring, supporting, or reporting—and its partitioning—per-flow or shared (Table 1). We also consider a key property for MB state: is it read, written, or read & written by the MB? This has implications for whether the controller should be allowed to modify the state, and whether state should be moved, cloned, and/or merged when moving flows between MBs.

3.2 Division of Responsibility

MBs examine and modify network traffic according to complex internal logic. At launch time, this logic may parse configuration inputs (e.g., rule definitions), initialize supporting and reporting state structures (e.g., connection index, timer manager, etc.), or perform other startup tasks. As the MB runs and examines traffic, the logic may: read configuration state; access, establish, and manipulate supporting state; create or update reporting state; and drop, forward, modify, and/or generate network traffic.

In contrast, the internal logic of network switches, and the writing and reading of forwarding state, is cleanly divided between a control plane and a data plane [23]. SDN leverages this to divide responsibility between network switches and a controller/applications: All forwarding state[4] is established by the SDN controller, and network switches simply forward packets based on this state.

The complexity of internal MB logic makes the "SDN model"—i.e., the controller makes all state changes—unsuitable for MBs. The majority of internal MB logic would essentially need to be re-implemented in the controller; since this removes too much control from MBs themselves, it could constrain innovation in the design of MBs. Additionally, MBs which make complex state updates as they process every packet (e.g., an IPS) would need to send almost all MB-received traffic to the controller.

OpenMB thus divides responsibility for state changes between MBs and the MB controller/control applications. MBs are responsible for *creating and modifying supporting and reporting state*, as they do today. The MB controller, under direction from the control application, is responsible for *manipulating where (i.e., on which MB) specific pieces of supporting and reporting state reside*. Additionally, the controller is responsible for *creating and updating all configuration state*. With this division of responsibility, the way MBs access, create, and update state is unchanged.

[4]Excluding timers and packet/byte counters.

4. MB-FACING API

The MB-facing ("southbound") API defines (*i*) how MBs import and export state, and (*ii*) when MBs need to signal that they have created or updated state internally.

4.1 State Interface

Providing programmatic, fine-grained control over all forms of MB state requires individual MBs to expose an external interface for reading and writing state. This should complement the state operations (read, update, add, and remove) performed by internal MB logic (§3.2). The key question is *at what granularity and in what form should MBs allow state to be externally read/written?* We rely on our state taxonomy (§3.1) to help address this question. Since the answer is subtly different for the three classes of MB state, we talk about the interface for each class separately.

4.1.1 Configuration State

Today, MBs support a variety of interfaces for setting, updating or querying configuration state. Unfortunately, the syntax used varies across MB types and vendors, e.g., iptables [8] vs. Cisco IOS Firewall [3] rules. Prior attempts at standardizing MB configuration interfaces have been narrowly focused. For example, SIMCO [11] only targets firewalls and NATs, and VRT rules [12] are only designed for IPSs [12, 14, 2]. Even with SDN, non-routing-related switch configuration (e.g., managing VLANs, configuring ports) occurs using vendor-specific syntax. These issues make cloning and dynamically modifying MB configuration state complex.

In SDMBN, we ask: how should configuration state be logically organized to enable fine-grained control? Based on our state taxonomy (§3.1), we advocate organizing configuration state as a hierarchy of keys and values. Each value is a single unit of configuration state, e.g.: a single parameter (e.g., cache size, replacement algorithm), a single policy item (e.g., firewall rule, IPS rule), etc. Each key is associated with either an unordered set of sub-keys or an ordered set of values. The exact hierarchy, key names, and value syntax/semantics is unique to each MB. The API for accessing/updating configuration state then is:

```
getConfig(⟨HierarchicalKey⟩)
setConfig(⟨HierarchicalKey⟩,
     [⟨ConfigurationValue⟩, ⟨ConfigurationValue⟩,...])
delConfig(⟨HierarchicalKey⟩)
```

4.1.2 Supporting State

Supporting state guides MB decisions and actions (§3.1). Correspondingly, the contents and structure of supporting state depend on the functions the MB provides. Typically, supporting state contains deep, detailed information, including: portions of the headers/payloads of received traffic, e.g., IP addresses, TCP flags, HTTP header fields; actions to apply to received traffic, e.g., forward traffic to a specific IP, re-write the internal IP address to a specific public-facing IP address, drop all traffic from a specific host; meta-data for locating pieces of state, e.g., hashes of flow identifiers, hashes of packet payloads; and other important data. Each MB defines its own organizational structure for this information. E.g., Bro [24] defines more than 100 different structures for storing supporting state.

One option is to share the syntax of these structures, and the semantics of the data they contain, with control applications so that they can inspect/modify/create supporting state in sophisticated ways. This has crucial disadvantages: (*i*) Control applications may modify or create supporting state in a way that is inconsistent with MB logic, leading to unexpected or incorrect MB behavior. (*ii*) MB mendors may be unlikely to share the syntax and semantics of internal MB structures for proprietary reasons (§3.2).

Thus, we must reconcile the desire of MBs to conceal and protect the integrity of supporting state with the requirement of being able to move, clone, and merge supporting state at fine granularity (§2). We address this challenge separately for per-flow supporting state and shared supporting state.

Per-Flow Supporting State. Per-flow state is intrinsically organized into self-contained chunks, one for each flow. Exposing state at the granularity of these per-flow chunks reveals nothing about an MB's internal structures that could not already be deduced from knowledge of the MB's functionality. More crucially, MBs can encrypt (decrypt) chunks of per-flow supporting state before exporting (after importing) to protect the state. The nature of per-flow state also provides an inherent mechanism for identifying a specific state chunk: We can simply use the same identifiers that are used by the MB to determine to which network traffic the per-flow state applies, e.g., IP addresses, ports, protocol numbers.

Thus, MBs should export/import pieces of per-flow supporting state as a key/value pair:
`[`⟨*HeaderFieldList*⟩`:`⟨*EncryptedChunk*⟩`]`.
Additionally, MBs should support three basic operations for controlling which per-flow supporting state resides at an MB:

`getSupportPerflow(`⟨*HeaderFieldList*⟩`)`
`putSupportPerflow([`⟨*HeaderFieldList*⟩`:`⟨*EncryptedChunk*⟩`])`
`delSupportPerflow(`⟨*HeaderFieldList*⟩`)`

Note that the identifiers which an MB uses to determine which per-flow supporting state applies to which packets also determines the finest granularity at which per-flow supporting state for that MB can be accessed. For example, Balance [1] only maintains a chunk of per-flow state based on source IP/ port, since the destination IP/port is the same for all connections, namely, the IP/port of the load balancer. We do not preclude requesting per-flow supporting state at a coarser granularity than the MB uses, e.g., identifying based only on source IP. Such a request will always return all matching pieces of per-flow supporting state at the finest granularity. However, requests for per-flow state at a granularity finer than the MB uses will return an error.

Shared Supporting State. Shared state requires a different interface because it applies to all traffic passing through a MB. The key constraints imposed by shared supporting state are: we cannot move it out of an MB if any flows will remain on the MB (e.g., during live migration) as the MB will not have the necessary state for the flows that remain, and we cannot move to an MB which already has flows (e.g., during scale down) as this would overwrite the shared state that already exists on the MB to where the flows are moved.

Alternate ways of managing shared state may not face the above issues to start with: e.g., one possibility is to maintain one global copy of each piece of shared supporting state across all instances of a specific type of MB. However, this requires MBs' internal logic to be significantly modified to read/write shared supporting state differently, e.g., using a distributed hash table.

The approach we adopt eliminates the need for complex modifications to MB logic. We synchronize MBs' shared state only when flows move, allowing the state to diverge independently in the interim. In the case where a subset of flows are moved to a new MB (e.g., live migration), the shared state on the new and original MBs can be synchronized by *cloning* the shared state from the original MB. In the case where all flows are moved to an existing MB (e.g., scale down), the shared state from the two MBs must be *merged*. Because shared supporting state applies to all traffic, *all* shared state must be cloned/merged, and hence MBs should export/import shared supporting state in a single chunk.

Thus, the MB-facing API for shared supporting state contains two operations:
`getSupportShared()`
`putSupportShared(`⟨*EncryptedChunk*⟩`)`
Cloning/merging can be implemented using these calls (§5).

Note that the merge operation at the MB can be complex and dependent on the semantics of the shared state, which we do not want to expose to the controller (§3.2). Thus, the MB must implement the needed logic for merging (invoked when `put` is called at an MB that is already maintaining shared supporting state): E.g., if two content caches (e.g., redundancy elimination (RE) decoders) are being merged, the MB may require extra meta-data (e.g., hit counts) for each cache entry to determine from which piece of state a particular entry should be retained.

4.1.3 *Reporting State*

Reporting state is intended to quantify observations or decisions that have already happened. Internal MB logic does not rely on reporting state for packet processing functions and decision making; MBs solely maintain this state for use by external entities, e.g, a network-wide alarm system.

Reporting state must be carefully managed to avoid "double reporting". E.g., packet counters on one MB should not be duplicated on another MB when traffic is moved, otherwise summing the counters from both MBs will doublecount packets which traversed the original MB prior to duplication. Avoiding double reporting of per-flow information is straightforward and we adopt techniques similar to per-flow supporting state.

However, dealing with shared reporting state is less straightforward. In particular, when a subset of flows are moved to a new MB, shared reporting state should *not* be cloned (unlike shared supporting state) as it leads to double reporting; instead, we start tracking fresh reporting state at the new MB. When consolidating two MBs, moving all traffic away from an MB without doing anything with reporting state will result in under reporting, since the reporting state will be lost when the MB is deprecated. In such cases, the consolidated MB can decide to merge the shared reporting state using appropriate logic if possible (e.g., for aggregate traffic counters), or it may decide to start afresh when the state does not permit merge (e.g., for traffic volume percentiles).

Thus, the MB-facing API for reporting supporting state contains the following operations:

```
getReportPerflow(⟨HeaderFieldList⟩)
putReportPerflow([⟨HeaderFieldList⟩:⟨EncryptedChunk⟩])
delReportPerflow(⟨HeaderFieldList⟩)
getReportShared()
putReportShared(⟨EncryptedChunk⟩)
```

4.2 State Events

The operations discussed above enable the controller to move state in and out of MBs. But MB themselves may locally initiate/change state (§3.2) unbeknownst to the controller, i.e., establish, update, or remove both supporting and reporting state when either a *packet is received* or a *timer fires*. For example, an IPS updates a connection record when a packet is received, a load balancer creates a new connection-to-server mapping when the first packet of a flow is received, and a NAT removes an IP address/port mapping when no matching packets are received for some time period. The southbound API "hides" both the *logic behind* and *the occurance of* such actions from the controller.

In SDN, forwarding state at switches may change as a result of the same triggers. But switches don't change state themselves; they raise an event in response to the trigger which the controller views as a *request for a change in forwarding state* and makes necessary state changes.

We argue that limited support for "SDN-like" events can prove quite useful OpenMB. The architectural difference from SDN, rooted in how we divide functionality, is that events in OpenMB are raised *when an MB establishes or updates state in response to a trigger*, not when the trigger itself (packet received or timer fires) occurs. Thus, the events augment the southbound API to provide visibility into occurance of a specific set of MB actions, but the underlying logic is still hidden. This helps ensure correct operations at no loss of performance as state is moved or cloned across MBs (§4.2.1), and supporting rich functionality (§4.2.2).

4.2.1 Atomicity Without Loss in Performance

Changes that involve both a controller-initiated state operation and a network update (or other non-MB change) typically need to happen atomically to ensure the correctness of MB operations. For example, shifting flows from a particular subnet from one MB to another requires moving per-flow supporting state and updating network routing as a single logical transaction. Such a transaction occurs atomically if:

(*i*) all affected packets are received and processed by at least one of the MBs (old or new),
(*ii*) external side-effects from packet processing—e.g, a packet is injected back into the network, or an alert is generated—only occur once for each packet,
(*iii*) no state creations or updates are lost, and
(*iv*) a complete, up-to-date copy of the state involved in the controller-initiated state action resides at the appropriate MB (s) when the transaction is finished.

Guaranteeing atomicity is complicated by our desire to minimize delay or suspension of MB operations.

One option for guaranteeing atomicity is to suspend the flow of traffic to MBs while a transaction is occurring. This requires identifying which traffic may trigger modifications to the state involved in the operation, and temporarily buffering this traffic at network switches. Additionally, packets that are already in transit to the affected MBs must be dropped upon arrival. The flow of traffic can be resumed only after the operation and network update have both completed. While this approach is straightforward, it requires suspending the processing of some flows for significant periods of time (up to several 100s of ms to a few seconds) which can lead to, e.g., user applications timing out. Conducting state operations and network updates at small granularity, e.g., one flow at a time, can minimize such downtime, but this only works for per-flow MB state and requires installing many fine-grained forwarding entries in network switches.

Instead, OpenMB allows MBs to continue processing traffic while a transaction is occurring, and use *events to "replay" state processing that occurs during (and slightly after) the state action and network update*. This avoids delaying packet processing for significant periods of time and does not require network switches or end hosts to buffer (potentially) large volumes of packets.

To understand how this approach works, consider the scenario in Figure 4: Since the flow of traffic is not suspended in OpenMB, packets corresponding to the state being moved may continue to arrive at MB A during the move. Furthermore, after `move` returns and the applications proceeds to update network state, packets corresponding to moved state may arrive at MB A until routing change kicks in, plus for a short time after this change (as packets may already have been in transit to MB A when the routing change occurred).

There are several possible approaches for dealing with these two sets of packets:

- **Discard the packets.** violates requirement *(i)* above.

- **Process the packets at MB A and re-send the affected state to MB B.** This avoids a violation of atomicity requirement *(iv)*. However, once the routing change takes effect, MB B may begin receiving packets corresponding to the moved state and the internal MB logic may update the supporting state that was originally transfered from MB A. A transfer of updated state from MB A will wipe out the updates made on MB B and violate atomicity requirement *(iii)*.
- **Redirect the packets to MB B.** Processing all packets for the state being moved on MB B ensures atomicity requirements *(i)*, *(ii)*, and *(iv)* are met. However, if the redirected packets arrive at MB B before the state from MB A, MB B will establish a new piece of per-flow supporting state. This will result in incorrect MB operation, and the state established by MB B will be lost when the state from MB A finally arrives, violating atomicity requirement *(iii)*. MB A must therefore buffer the packets until MB B has acknowledged the corresponding per-flow supporting state has been installed; this unnecessarily delays packet processing. More crucially, redirection is unsuitable in the case of a clone action, since both MBs need to process the packet to keep the clone up-to-date while the transaction is in progress (§6.1).

OpenMB both guarantees atomicity and avoids processing delay using three steps: (1) These packets are processed at MB A as normal, including the occurrence of external side-effects, e.g., forwarding of the packet and, (2) If a piece of state that was moved or cloned is updated while the packet is being processed, MB A sends a *packet re-process* event, which includes a copy of the packet, to MB B.

(3) When MB B receives the event, it processes the packet as normal to update state, *except it does not perform external side-effects*. Processing the packet in this manner at MB A preserves atomicity requirements *ii*, *iii*, and *iv*. MB A stops raising packet re-process events when it stops receiving packets which trigger updates to moved or cloned state.

One caveat of this approach is later packets in the flow may already have arrived at and been processed by MB B before the event arrived. This is not a problem for MBs that can handle such out-of-order arrivals by design. Some MBs, e.g., an RE decoder [16], cannot handle reordering; this is an intrinsic limitation of the MB and SDMBN does not introduce new complications.

4.2.2 Introspection

MBs also raise events to provide introspection into their operations. For example, a control application may be interested in knowing when a NAT has created a new IP address/port mapping or when a load balancer has a assigned a new flow to a server. These events are broadly triggered when an MB creates or updates supporting or reporting state; the exact triggers for these events are MB-specific. The events always include a key that identifies the relevant state

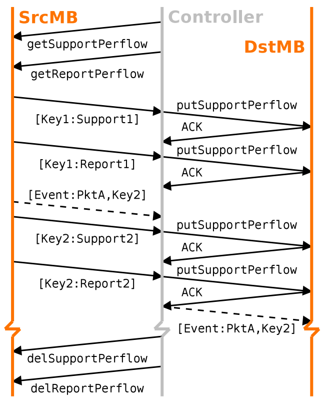


Figure 5: Sequence of actions for `moveInternal`.

and an event code; additional MB-specific values (e.g., the server to which a flow was assigned) may be included to provide more information. For example, a NAT could generate an event to announce the creation of a new mapping. The event would include both the header fields of the affected flow and the new mapping. Generally, points in internal MB logic where information is written to a log file are likely places for triggering events.

To ensure that the controller, network, and MB are not at risk for overload, OpenMB makes it possible to enable or disable the generation of introspection events based on event codes and keys. For example, a controller application can request to receive events only when a load balancer assigns new flows for a specific subnet to a server, or receive all events only for a limited period of time.

5. CONTROL API

The application-facing ("northbound") API encapsulates the intricacies of state operations on individual MBs. We discuss in § 6 how control applications leverage this API. The API consists of six operations:

```
readConfig(⟨SrcMB⟩,⟨HierarchicalKey⟩)
writeConfig(⟨DstMB⟩,⟨HierarchicalKey⟩,
   [⟨ConfigurationValue⟩,⟨ConfigurationValue⟩,...])
stats(⟨SrcMB⟩,⟨HeaderFieldList⟩)
moveInternal(⟨SrcMB⟩,⟨DstMB⟩,⟨HeaderFieldList⟩)
cloneSupport(⟨SrcMB⟩,⟨DstMB⟩)
mergeInternal(⟨SrcMB⟩,⟨DstMB⟩)
```

The controller serves as a broker for all of these operations.
Controller's Actions. We now discuss what sequence of steps the controller executes when applications call the functions above. The `readConfig` and `writeConfig` operations are simple: The controller issues a `get-config` or `set-config` call to the appropriate MB. We could also include a `cloneConfig` operation that would be a composition of the `readConfig` and `writeConfig` calls. The `stats` operation is used for informational purposes. It allows applications to query how much shared and per-flow supporting and reporting state exists for a given key.

The `moveInternal`, `cloneSupport` and `mergeInternal` operations are more complex because they involve events, in addition to `get`, `put`, and `del` calls. Figure 5 illustrates `moveInternal`: the controller begins the operation by calling both the `getSupportPerflow` and the `getReportPerflow` operations on the *SrcMB*, using the *HeaderFieldList* provided to the `moveInternal` operation. The *SrcMB* will begin returning pieces of per-flow state to the controller. The controller will subsequently call the `putSupportPerflow` or `putReportPerflow` operation the *DstMB*, providing the piece of per-flow state that came from the *SrcMB*. The *DstMB* will send an ACK to the controller after each put operation completes successfully, and the *SrcMB* will send an ACK to the controller after both get operations complete successfully. In parallel, the controller receives and forwards `reprocess` events. The *SrcMB* may begin generating events as soon as it sends the first piece of per-flow state to the controller. When the events arrive at the controller they are buffered until the *DstMB* has not ACK'd the put for the piece of per-flow state to which the event applies. The `moveInternal` operation returns once all puts have been ACK'd. However, the controller may continue to process events related to this operation. When no events have been received from the *SrcMB* for a fixed amount of time (e.g., 5 seconds), it is assumed the routing change has taken place. At this time, the controller calls both the `delSupportPerflow` and `delReportPerflow` operations on the *SrcMB* to complete the move.

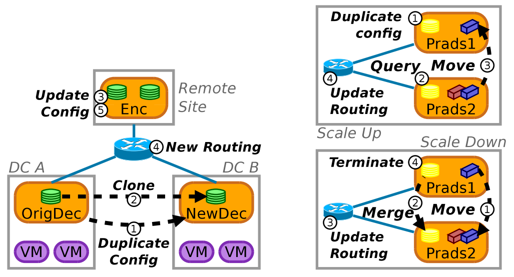
The sequence of actions performed by the controller for the `cloneSupport` and `mergeInternal` operations is similar, except: (*i*) the get and put operations for shared supporting (and reporting, in the case of merge) state are called on the *SrcMB* and *DstMB*, and (*ii*) no delete operation is called when events stop arriving.

The above discussion and §4 imply that the controller actively intervenes *all* exchange of state and events. Alternatively, MBs can exchange state and events directly, based on a request (move, clone, or merge) from the controller. We did not adopt the latter approach as it means that the MBs must include the appropriate communication logic, carefully order puts and events, and handle failure cases. In our design, this logic only needs to be implemented once (at the controller) and processing burden of MBs is not increased.

Why A Separate API. Exposing a separate API, operating at a higher level of abstraction, to control applications has several benefits over directly exposing the MB-facing API:

First, decoupling the two APIs helps evolution: e.g., the MB-facing API can evolve without control applications changing. This is vital given the rapid pace of MB innovation.

Second, it simplifies the design of applications. For example, a move operation requires issuing `gets` to one MB, `puts` to another MB, and forwarding events from one MB to the other; these are now handled by the controller. Further, the controller can implement appropriate disciplines for scheduling the finer-grained actions comprising multiple northbound API calls to ensure, e.g., that northbound calls complete in a reasonable time-frame; application logic is unburdened by these considerations.



(a) Live migration with RE (b) Scaling PRADS

Figure 6: Control application examples

Third, it limits the potential for control applications to make state changes that will lead to correctness or performance issues. For example, the restrictive API prohibits control applications from retrieving per-flow supporting state from one MB but failing to it to another MB.

It is still up to control applications to issue multiple MB state control operations, and SDN control operations, in the correct sequence. For example, a move operation must be issued and completed before initiating a network routing change. Note that exposing supporting state as an encrypted blob (§ 4.1) also helps eliminate illicit state changes.

6. CONTROL APPLICATIONS

We describe two scenarios below, which face different challenges due to the nature of state resident at the MBs in question and the interaction between network and middlebox state. While these challenges are tricky to overcome in general, we show how our northbound API helps, making control application design simple and easy to reason about.

6.1 Live Migration

We first consider a live migration scenario involving redundancy elimination (RE) MBs (Figure 6(a)). Initially, all application VMs reside in a single data center (DC A). Traffic destined for these VMs passes through an RE encoder at a remote site, traverses the WAN to reach DC A, and passes through an RE decoder. At some point, half of the application VMs are migrated to a new data center (DC B). Following the migration, traffic destined for the VMs in DC B passes through the same RE encoder at the remote site, traverses the WAN to reach DC B, and passes through a new RE decoder in DC B; traffic destined for the VMs remaining in DC A follow the same path as before.

Both the RE encoder and decoder rely only on shared supporting state. The encoder adds each received packet to a packet cache (implemented as a ring buffer) and inserts hashes of the packets' contents into a fingerprint table (implemented as a hash table) [16]. Redundant portions of a packet are replaced by a small shim that specifies the location of the original content within the packet cache. The decoder reconstructs the original packet from its own packet cache, which is implemented and updated exactly the same as the packet cache in the encoder.

The control application must carefully manage the state at the encoder and decoders to ensure packets can be appropriately decoded when they arrive at a decoder. RE's assumption that the encoder and decoder's packet caches are tightly synchronized [16] makes this especially challenging.[5]

The simplest solution would be to launch the new decoder with an empty cache (and create a corresponding empty cache at the encoder). However, packets which have been encoded based on the encoder's new cache may be routed to the old decoder, which will be unable to reconstruct the packets (§8.1.2). This situation can occur due to a delay between the encoder switching to use the new cache and the routing update. Using OpenMB avoids both of these issues.

When application VMs are migrated, our *migrate* control application performs the following actions:

1. Launch a new RE decoder in DC B; duplicate the configuration of the original RE decoder:
   `values = readConfig(OrigDec,"*")`
   `writeConfig(NewDec,"*",values)`
2. Clone the original decoder's cache:
   `cloneSupport(OrigDec,NewDec)`
3. Add a second cache to the encoder:
   `writeConfig(Enc,"NumCaches",[2])`
   Internally, the encoder will clone its original cache to create a new second cache.
4. Update the network routing by making the appropriate calls to the SDN controller.
5. Tell the encoder to start using the second cache for traffic going to the VMs in DC B and the first cache for traffic going to the VMs in DC A: `writeConfig(Enc, "CacheFlows",["1.1.1.0/24","1.1.2.0/24"])`

6.2 Scaling

We now consider a scaling scenario involving monitoring (PRADS [10]) MBs (Figure 6(b)). When network load is high, additional MB instances are added to process the traffic. In-progress flows are redistributed across the new instance(s) to balance load. The additional instances are scaled down when the traffic volume reduces, and flows are re-balanced among the remaining instance(s).

The collective monitoring behavior of the PRADS instances should remain the same regardless of any scaling, i.e., there should be no over-reporting or under-reporting of packet/flow counters. This requires carefully controlling both the per-flow and shared reporting state associated with PRADS.

When the control application determines scale up should occur, it performs the following actions:

1. Launch a new PRADS instance and duplicate the configuration from an existing instance:
   `values = readConfig(Prads1,"*")`
   `writeConfig(Prads2,"*",values)`
2. Query how much per-flow state exists for specific subnets to determine how in-progress flows should be rebalanced: `stats(Prads1,[nw_src=1.1.1.0/24])`
3. Move a subset of the per-flow state: `moveInternal( Prads1,Prads2,[nw_src=1.1.1.0/24])`
4. Route the moved flows to the new instance.

A slightly different set of actions occur during scale down:

1. Transfer the per-flow reporting state for all flows:
   `moveInternal(Prads2,Prads1,[])`
2. Merge the shared reporting state:
   `mergeInternal(Prads2,Prads1)`
3. Route flows to the remaining instance(s).
4. Terminate the unneeded instance.

7. IMPLEMENTATION

Our OpenMB prototype consists of an MB controller that implements our control API (§5), three MBs—IPS, monitor, and RE—modified to support our MB-facing API (§4), and the control applications discussed in §6.

Our MB controller is a module running atop Floodlight [5] (≈1600 lines of Java code). The controller listens for connections from MBs and, for each MB, launches one thread for handling state operations and one thread for handling events. Additionally, the controller maintains a hash table for each MB to buffer re-process events (raised due to a get) and track acknowledgements (of puts). JSON messages are exchanged by the controller and MBs to invoke operations, send/receive state, and raise/forward events.

We modified three different MBs to support our MB-facing API (§4): Bro [24], Prads [10], and SmartRE [16]. Each MB relies on a common code base for MB-controller communications (≈500 LOC); the code leverages standard UNIX sockets and the JSON-C library. Additional MB-specific modifications are made to retrieve, insert, and remove per-flow/shared state and to generate and process events.

Bro. Bro maintains a `Connection` object, and a tree of associated objects, for each flow. The `Connection` objects are stored in one of three hash tables (depending on whether the flow is TCP, UDP, or ICMP). When `getPerflowSupport` is invoked, we perform a linear search[6] of the hash table(s) of `Connection` objects to identify and send all matching per-flow state. We added serialization functions to the `Connection` class and all referenced classes (>100 classes), using libboost's serialization library, to allow the state for a given flow to be moved. Additionally, we added a `moved` flag to a subset of these classes—to prevent Bro from logging errors when the state for a flow is deleted, following a successful move—and a mutex to the `Connection` class—to prevent Bro from modifying a `Connection` object, or an object it references, while serialization is occurring. When `putPerflowSupport` is invoked, we reverse the serialization and insert the `Connection` object into the appropriate

[5]We assume the encoder maintains a separate packet cache and fingerprint table for each decoder.

[6]Techniques used by network switches for wildcard matches on packet headers could be adopted here for improved performance.

hash table. Lastly, we made two additions to Bro's main packet processing loop: lock/unlock the `Connection` object with which a packet is associated while the packet is being processed, and raise an event (that includes the packet) when the `moved` flag is set in the `Connection` object.

PRADS. PRADS maintains a `connection` object for each flow as well as a `prads_stat` object that is shared across all flows. The `connection` objects are stored in buckets, where each bucket is a doubly-linked list of `connection` objects. The handling of `getPerflowReport` and `putPerflowReport` calls is similar to the handling of the `getPerflowSupport` and `putPerflowSupport` calls in Bro. The only difference is that there is no need for complex serialization because there is only a single structure for each flow. To handle `putSharedReport`, we add the counter values stored in the `prads_stat` structure provided in the put call to the counter values stored in the `prads_stat` structure already residing at the PRADS instance. We modify PRADS main packet processing loop the same as we did for Bro. In total, we added ≈200 lines of C code (1.5% increase).

RE. RE maintains a `cache` object that includes cached content, `size_of_cache` state, a pointer `current_pos` indicating where to insert a new cache entry, and a `max_reached` indicating if cache is full. The `cache` object is shared by all flows. An encoder maintains multiple `cache` objects. Each of them corresponds to a decoder. An encoder also maintains a `num_of_decoder` to remember the number of caches that need to be maintained and a `fingerprint_table` for each decoder. `cloneSupport` call clones original cache content and `current_pos` to a new decoder. We use `writeConfig` call to tell an encoder the number of existing decoders; the, the encoder creates a new cache for a new decoder. Original content and `fingerprint_table` are cloned to the new cache. We also use `writeConfig` to tell the encoder when to switch to a new bucket. In total, we added 140 lines of C++ code, excluding the OpenMB common code base.

8. EVALUATION

In this section, we evaluate our OpenMB prototype in a variety of scenarios using both real and synthetic workloads. Our goals are to examine the following issues:

- Are OpenMB abstractions useful to construct rich network control applications that can achieve fine-grained control over MB deployments in a wide variety of dynamic scenarios?
- What are the advantages of OpenMB relative to existing point solutions for achieving control over MBs in these scenarios?
- Does OpenMB interfere with correct functioning of MBs? What is the impact of OpenMB on both the implementation and processing performance of MBs?
- What is the performance of our OpenMB controller? What aspects of OpenMB's design constrain its performance the most?

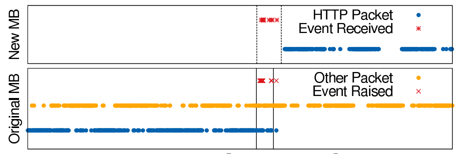


Figure 7: MB actions during *scale up* scenario

We use a testbed consisting of an OpenFlow-enabled HP ProCurve 5400 switch, a mid-range server (dual 2.7GHz Quad-Core Intel Xeon, 12GB, 1Gbps NIC) that runs the controller, and six low-end desktops (2.4GHz Quad-Core Intel Core 2, 4GB RAM, 1Gbps NIC) that run the modified MBs. The traffic used in our experiments comes from three different network traces: (*i*) all traffic exchanged between a large university campus and two major cloud providers (Amazon EC2 and Microsoft Azure), captured at the campus network border for ≈15 minutes; (*ii*) a subset of traffic exchanged in a university data center over ≈1 hour [18]; and (*iii*) a high-redundancy trace constructed from traffic exchanged in a campus network [29].

8.1 Control Application Design

We first present snapshots of OpenMB actions when running the *scaling* control application (§6.2), illustrating how OpenMB helps achieve dynamic fine-grained control in this scenario. We then present a qualitative evaluation of why some state-of-the-art alternatives are unsuitable for the elastic scaling and live migration scenarios.

8.1.1 *OpenMB Behavior at Run Time*

We capture the actions occurring at MBs for the *scale-up* scenario discussed in §6.2. Figure 7 shows the packet processing, event raising/processing, and operation handling that occurs over a 3-second window at the original (bottom) and new (top) Prads MBs. (We exclude the configuration operations for brevity.) The solid lines indicate the start and end of the `getPerflowReport` operation at the original MB, and the dashed lines indicate the start of the first and the end of the last `putPerflowReport` operations at the new MB. First, we observe that HTTP packets are processed by the original MB until slightly after the final put operation completes at the new MB, at which point all HTTP packets are processed by the new MB. This is due to the controller returning from the `moveInternal` operation, and the *migrate* control application issuing a routing update via the SDN controller. OpenMB enables this careful sequencing of MB state changes and routing updates. Second, we observe that the original MB begins raising re-process events soon after the get operation begins, and continue to be raised until slightly after the get operation completes. These events are received and processed by the new MB after the corresponding state has been put. This highlights OpenMB's use of events to ensure state updates are not lost the new MB while waiting for the routing change to take effect.

8.1.2 *Other Alternatives*

	Scale up	Scale Down	Migration
SDMBN	✓	✓	✓
Snapshot	≈	X	≈
Controlling Config & Routing	≈	≈	≈
Split/Merge	✓	≈	✓

Table 2: Applicability of different schemes for MB control to different dynamic scenarios: ✓ fully supported, ≈ partially supported, X not supported. We separate scale up/down to show some approaches support one but not both.

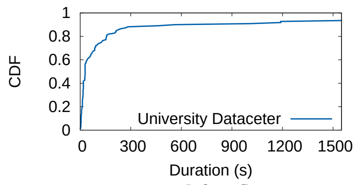


Figure 8: Time required for flows completion

We now consider the alternatives in §2.1 in terms of their basic ability to support elastic scaling and live migration, how efficiently resources are used and the overall correctness of MB operations. Our results are summarized in Table 2.

Virtual Machine Snapshots. We study if VM snapshotting is applicable to cloning or otherwise moving state. When state at an MB needs to move/clone to a new MB, we simply create the new MB from a snapshot of the old. We then update routing to send traffic to it for the migrated flows.

We experiment with this approach in the context of the live migration scenario using a Bro MB (variation of Figure 6(a), top). We compared OpenMB against an approach where the MB we use in DC B is a snapshot of the Bro VM in DC A. As argued in §2.1, both Bro MBs have unneeded state. We quantify this by comparing (byte-by-byte) a base image of Bro, i.e., a snapshot without any traffic ("BASE") against a memory snapshot taken at the instant of migration ("FULL"), and two snapshots with just the HTTP and other traffic substreams taken at the instant of migration ("HTTP" and "OTHER"). BASE and FULL differed by 22MB. HTTP and OTHER differed from base by 19MB and 4MB, respectively; these numbers indicate the overhead imposed by the unneeded state at the two Bro instances. In contrast, we found that the amount of state moved by SDMBN (i.e., per-flow supporting state for all HTTP flows) was 8.1MB. But more interesting are the correctness implications of the unneeded state: We found that this state results in 3173 and 716 incorrect entries in the conn.log at the two MBs; these arise because the migrated HTTP (other) flows terminate abruptly at the old (new) Bro MB, which Bro considers an anomaly.

VM snapshotting does not apply to scale down (Table 2).

Controlling MB configuration and routing. SDN provides a reasonable solution for achieving control over MB configuration and routing. However, this is insufficient because of lack of fine-grained MB state control. We illustrate this for both scale down and live migration.

One way to support scale down (Figure 6(b), bottom) is to carefully manage configuration and routing: Active HTTP flows are left for processing in the MB to be deprecated (the MB on top), whereas only new HTTP flows are forwarded to

	Encoded Bytes (MB)	Undecodable bytes (MB)
SDMBN	148.42	0
Config + routing	97.33	97.33

Table 3: Performance of RE in live migration.

the consolidated MB (the MB on the bottom). However this approach unnecessarily "holds up" the MB to be deprecated as long as flows stay active. In Figure 8 we show a CDF of the duration of flow lengths for HTTP traffic in traces: we see that around 9% of flows take more than 1500 secs to complete. Indeed, we saw in our trace-driven experiments that the deprecated MB was held up for over 1500s!

We also evaluate the migration scenario using an RE decoder MB (Figure 6(a)). To support migration of the decoder, we create a new decoder in DC B with an *empty* cache; correspondingly, we create an empty encoder at the remote site. All HTTP traffic eventually traverses the new pair of RE MBs, while all other traffic traverses the old pair.

An interesting quirk of the RE decoder is that it assumes that packet contents are stored locally at the exact same memory locations as they are stored at the encoder. Thus, without the ability to clone the cache, starting with the empty cache is necessary to minimize the potential for correctness issues (i.e., all encoded packets need to be decoded, which requires caches to be in perfect sync).[7] Even with empty caches, ensuring correctness is fundamentally hard in this approach. This is because the encoder, decoder *and* the router need to be in perfect sync: e.g., if the new encoder starts being used for HTTP traffic, but routing has not been updated yet, then the encoded traffic reaches the old decoder where it cannot be recovered. Since this causes the new decoder to miss out on some packets from its encoder, the two caches get out of sync and stay that way even after routing has been updated.

The difference in performance and the potential correctness issues in this approach w.r.t. OpenMB are illustrated in Table 3; the cache sizes we use are modest (500MB). We assume that the routing change takes effect after the encoder has sent 10 packets. We see that this approach encoded 51MB fewer redundant bytes (34% during the cache warmup time) relative to OpenMB. More importantly, *none* of the encoded bytes can be decoded. The caches need to be forcefully evicted in full and started afresh.

Split/Merge. Split/Merge is designed specifically for elastic scaling, but its mechanism for providing atomicity—halting all traffic while state is moved—introduces latency costs.

We experiment with this approach in the context of the scale up scenario using Bro MBs (variation of Figure 6(a), top). We assume 1000 pieces of per-flow state need to be moved and packets are arriving at a rate of 1000 packets/second. We observe that 244 packets must be buffered while the move operation is occurring. More crucially, the average processing latency of these packets increases by 863ms as a result of this buffering.

Split/Merge is insufficient for some cases of scale down

[7]Using VM snapshot is difficult because we need synchronized snapshots for both the encoder and the decoder.

(e.g., when using RE or Prads MBs), since it lacks support for merging shared state (Table 2).

8.2 MB Correctness and Performance

We now evaluate the impact of OpenMB on the correctness and performance of individual MBs to ensure neither is sacrificed. Additionally, we measure the time required to process get/put operations, and the number of events generated during these operations, to understand how the implementation of the southbound API on individual MBs affects the overall performance of OpenMB.

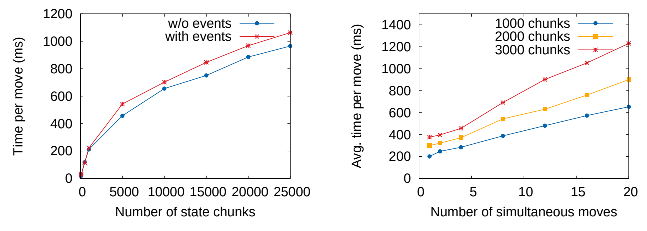
Correctness. We verify the correctness of MB operations by comparing the output produced by a single, unmodified MB versus the output produced by an OpenMB-capable MB while running the *migrate* control application. For Bro, we replayed the cloud traffic trace for both scenarios and compared the conn.log and http.log files, which reflect connection state/statistics and HTTP requests/replies; we observed no differences in either log file. Similarly, we compared the statistics output by Prads under both scenarios and found no discrepancies. We verified the correctness of RE's operation by comparing the high-redundancy trace with the packets output by the decoder(s); all packets were properly decoded.

Performance. We evaluated the impact of OpenMB on MB performance by comparing the average per-packet processing latency (including queueing time) during normal MB operation and when an MB is processing a get call. For Bro, there is no significant change in the average per-packet processing latency: 6.93ms during normal operation and 7.06ms when processing a get call. For RE, there is no significant change in the average time from when a packet leaves the encoder to when it leaves the decoder: 0.781ms during normal operation and 0.790ms when processing a get call.

API Call Processing. The time required to process get/put operations has a direct influence on how quickly a move, clone, or merge operation completes, and, subsequently, how long a control application must wait before updating network routing. A longer time window between the start of a control API call and the corresponding routing update taking effect, means more packets may arrive at the original MB and trigger re-processing events. These events in turn introduce additional processing work for the new MB.

Figures 9(a) and 9(b) show the time required to complete a single get and all corresponding puts, respectively, as a function of the number of chunks of per-flow state involved. For both Prads and Bro, we observe a linear increase in get and puts processing time as the number of per-flow state chunks increases. Additionally, the collective put processing time is ≈6x lower than get; we attribute this difference to the inefficient linear search that is performed by both Prads and Bro for get calls (§7). Overall, the processing time is higher for Bro because of the size and complexity of the per-flow state. We also measured the get processing time for RE: it takes 34.8 seconds to retrieve a 500 MB cache.

Figures 9(c) and 9(d) show the number of re-processing events generated by Prads and Bro, respectively, as a function of the packet rate and the number of chunks of per-flow state involved in the get operation that caused the events to be raised. For both Prads and Bro, we observe a linear increase in the number of events as the packet rate increase. This intuitively makes sense because more packets will arrive in the time window between the start of a control API call and the corresponding routing update taking effect.



(a) Single operation. (b) Concurrent operations.

Figure 10: Time per move operation.

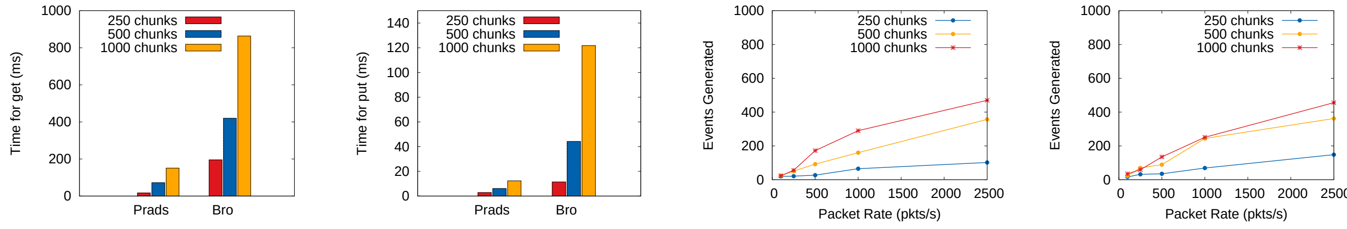
8.3 MB Controller Performance

In this section, we analyze the performance and scalability of our prototype MB controller. Recall that the MB controller brokers all MB state operations (§5), making the controller's performance a key contributor to the overall performance of the OpenMB framework. We focus on two important questions: (*i*) how quickly can the controller execute state operations and process events? and (*ii*) how many simultaneous operations can the controller support?

To isolate the performance and scalability of the MB controller from the performance of individual MBs, we use "dummy" MBs that simply replay traces of past state in response to gets, send acks in response to puts, and infinitely generate events during the lifetime of the experiment. The traces we use are derived from actual state and events sent by Prads while processing our cloud traffic trace. All state and events are small (202 bytes and 128 bytes, respectively) for consistency, and to maximize the processing demand at the controller and minimize the impact due to network transfer.

Single Operation. We first analyze how quickly the controller can process a single operation (`moveInternal`). Figure 10(a) shows the time required to complete this operation, relative to the number of chunks of per-flow state processed as part of the operation, both with and without the controller receiving and forwarding events. We observe that both the amount of migrated state and the presence of events impact performance. Crucially, even at high rates, events increase operation processing time by at most 9%.

Concurrent operations. Next, we evaluate how well our controller can handle many northbound API calls issued simultaneously. We use a similar setup as before except that multiple pairs of MBs are involved in `move` operations. Figure 10(b) shows the average time per move operation as a function of the number of simultaneous operations. We observe that the average time per move operation increases linearly with both the number of simultaneous operations and



(a) `getPerflowSupport` time per operation on Prads and Bro. (b) `putPerflowSupport` time per operation on Prads and Bro. (c) Events generated by Prads during `moveInternal` (d) Events generated by Bro during `moveInternal`.

Figure 9: MBs performance.

the number of state chunks per operation.

To better understand the cause of the observed performance, and identify potential techniques for reducing operation processing latency, we profiled our MB controller using HPROF [6]. We analyzed CPU usage using both time and samples profiling. First, time profiling revealed that, with 2 (4, 8) simultaneous operations, 49% (40%, 29%) of CPU time is spent context switching between threads and locking shared objects. However, we cannot distinguish whether this is a result of our MB controller implementation or an artifact of the core Floodlight [5] code base. Regardless, this time could be reduced by using finer grained synchronization primitives in our MB controller module. Second, samples profiling revealed that, with 2 (4, 8) simultaneous operations, threads are busy reading from sockets 47% (62%, 76%) of the time. We believe this is the main cause of increased operation latency when then are more state chunks. This can be improved by optimizing the size of state transfers using compression. We ran a simple experiment and observed that, for a move operation with 500 chunks states, state can be compressed by 38%, decreasing the operation execution latency from 110ms to 70ms.

9. CONCLUSION

Effectively supporting sophisticated dynamic enterprise scenarios requires the introduction of a software-defined MB networking framework that useful abstractions for unified software-driven control of MB functions across a range of MBs. As we have shown, designing such a framework requires coping with highly diverse MB state and complex internal MB logic. OpenMB addresses these challenges through two novel APIs: an MB-facing API that defines how MBs receive/export state and a control API that defines how MB state can be accessed and placed. Our implementation of an MB controller, several control applications, and OpenMB-enabled MBs allow a variety of dynamic scenarios to be realized without affecting the correctness or performance of MBs. Going forward, we believe that a framework like OpenMB is crucial to continued innovation in MBs and the network.

10. REFERENCES

[1] Balance. `http://inlab.de/balance.html`.
[2] Check Point IPS Software Blade. `http://checkpoint.com/products/ips-software-blade`.
[3] Cisco IOS Firewall. `http://cisco.com/en/US/products/sw/secursw/ps1018`.
[4] CloudOpt. `http://cloudopt.com`.
[5] Floodlight OpenFlow Controller. `http://floodlight.openflowhub.org`.
[6] Hprof. `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`.
[7] HyperIP. `http://netex.com/hyperip`.
[8] iptables. `https://help.ubuntu.com/community/IptablesHowTo`.
[9] nDPI. `http://ntop.org/products/ndpi`.
[10] Passive real-time asset detection system. `http://prads.projects.linpro.no`.
[11] RFC 4540: NEC's Simple Middlebox Configuration (SIMCO) Protocol Version 3.0. `http://tools.ietf.org/html/rfc4540`.
[12] Snort. `http://snort.org`.
[13] Squid. `http://squid-cache.org`.
[14] Suricata. `http://openinfosecfoundation.org/index.php/download-suricata`.
[15] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in Network Traffic: Findings and Implications. In *SIGMETRICS*, 2009.
[16] A. Anand, V. Sekar, and A. Akella. SmartRE: An Architecture for Coordinated Network-wide Redundancy Elimination. In *SIGCOMM*, 2009.
[17] 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 *SOSP*, 2003.
[18] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.
[19] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *NSDI*, 2011.
[20] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-Class Entities. Technical Report TR1771, University of Wisconsin-Madison, 2012.
[21] Y. Gu, A. McCallum, and D. Towsley. Detecting Anomalies in Network Traffic Using Maximum Entropy Estimation. In *IMC*, 2005.
[22] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford. Live Migration of an Entire Network (and its Hosts). In *HotNets*, 2012.
[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2), 2008.
[24] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security Symposium (SSYM)*, 1998.
[25] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *NSDI*, 2013.
[26] S. E. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In *RAID*, 2004.
[27] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.
[28] V. Sekar, R. Krishnaswamy, A. Gupta, and M. K. Reiter. Network-Wide Deployment of Intrusion Detection and Prevention Systems. In *CoNEXT*, 2010.
[29] S.-H. Shen, A. Gember, A. Anand, and A. Akella. REfactor-ing Content Overhearing to Improve Wireless Performance. In *MobiCom*, 2011.
[30] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.