Managing middleboxes

• “Middlebox manifesto” (ref. previous lecture) pointed out the need for automated middlebox management

• Many different solutions proposed afterwards
Managing middleboxes

• “Middlebox manifesto” (ref. previous lecture) pointed out the need for automated middlebox management

  • Many different solutions proposed afterwards

• Typical model: middleboxes are virtualized software appliances

  • Can be deployed on general-purpose servers

  • Area typically referred to as Network Function Virtualization
Managing middleboxes - II

- Implementing middleboxes in software offers great freedom...
- Allowing things like scaling and dynamic load balancing:
Managing middleboxes - II

- Implementing middleboxes in software offers great freedom…

- Allowing things like scaling and dynamic load balancing:

```
  Scaling
  
  Input traffic
  
  IDS 1
  
  IDS 2
```
Managing middleboxes - II

- Implementing middleboxes in software offers great freedom…

- Allowing things like scaling and dynamic load balancing:
Managing middleboxes - II

- Implementing middleboxes in software offers great freedom...
- Allowing things like scaling and dynamic load balancing:
Managing middleboxes - II

- Implementing middleboxes in software offers great freedom...
- Allowing things like scaling and dynamic load balancing:

Scaling

Input traffic

IDS 1

IDS 2

IDS 3

Load-balancing

IDS 1

IDS 2
Managing middleboxes - III

- The ability of shuffling middleboxes around creates various challenges.

- Most are related to the stateful nature of middlebox processing.

- E.g., an IDS keeps some detection state for each flow it is analyzing.

- What happens to the state if the IDS is terminated or the flow is remapped to another IDS node?
Example: IDS MB scaled out during portscan detection
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

IDS 1
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

Conn attempt #1

Conn attempt #2

Conn attempt #3

IDS 1
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

Conn attempt #1
Conn attempt #2
Conn attempt #3

IDS 1

Portscan
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

Conn attempt #1
Conn attempt #2
Conn attempt #3

IDS 1

Portscan

Scale-out mid-attack:

IDS 1
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

Conn attempt #1
Conn attempt #2
Conn attempt #3

Portscan

IDS 1

Scale-out mid-attack:

Conn attempt #1
Conn attempt #2

IDS 1
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

Conn attempt #1
Conn attempt #2
Conn attempt #3

IDS 1

Portscan

Scale-out mid-attack:

Conn attempt #1
Conn attempt #2

IDS 1

IDS 2
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

Conn attempt #1 → IDS 1
Conn attempt #2
Conn attempt #3
→ Portscan

Scale-out mid-attack:

Conn attempt #1 → IDS 1
Conn attempt #2
Conn attempt #3 → IDS 2
Example: IDS MB scaled out during portscan detection

All traffic processed by the same middlebox:

- Conn attempt #1
- Conn attempt #2
- Conn attempt #3

IDS 1

Portscan

Scale-out mid-attack:

- Conn attempt #1
- Conn attempt #2
- Conn attempt #3

IDS 1

IDS 2
OpenNF: Enabling Innovation in Network Function Control

Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella
University of Wisconsin-Madison
{agember,raajay,cprakash,rgrandl,junaid,souravd,akella}@cs.wisc.edu
http://opennf.cs.wisc.edu

ABSTRACT
Network functions virtualization (NFV) together with software-defined networking (SDN) has the potential to help operators satisfy tight service level agreements, accurately monitor and manipulate network traffic, and minimize operating expenses. However, in scenarios that require packet processing to be redistributed across a collection of network function (NF) instances, simultaneously achieving all three goals requires a framework that provides efficient, coordinated control of both internal NF state and network forwarding state. To this end, we design a control plane called OpenNF. We use carefully designed APIs and a clever combination of events and forwarding updates to address race conditions, bound overhead, and accommodate a variety of NFs. Our evaluation shows that OpenNF offers efficient state control without compromising flexibility, and requires modest additions to NFs.

Categories and Subject Descriptors
C.2.1 [Computer Communication Networks]: Network Architecture and Design; C.2.3 [Computer Communication Networks]: Network Operations

Keywords
Network functions, middleboxes, software-defined networking

1. INTRODUCTION
Network functions (NFs), or middleboxes, are systems that examine and modify packets and flows in sophisticated ways: e.g., intrusion detection systems (IDSs), load balancers, caching proxies, etc. NFs play a critical role in ensuring security, improving performance, and providing other novel network functionality [37].

Recently, we have seen a growing interest in replacing dedicated NF hardware with software-based NFs running on generic compute resources—a trend known as network functions virtualization (NFV) [12]. In parallel, software-defined networking (SDN) is being used to steer flows through appropriate NFs to enforce policies and jointly manage network and NF load [17, 20, 22, 26, 32].

Together, NFV and SDN can enable an important class of management applications that need to dynamically redistribute packet processing across multiple instances of an NF—e.g., NF load balancing [32] and elastic NF scaling [21]. In the context of such applications, “NFV + SDN” can help achieve three important goals: (1) satisfy tight service level agreements (SLAs) on NF performance or availability; (2) accurately monitor and manipulate network traffic, e.g., an IDS should raise alerts for all flows containing known malware; and (3) minimize NF operating costs. However, simultaneously achieving all three goals is not possible today, and fundamentally requires more control than NFV + SDN can offer.

To see why, consider a scenario where an IDS is overloaded and must be scaled out in order to satisfy SLAs on throughput (Figure 1). With NFV we can easily launch a new IDS instance, and with SDN we can reroute some in-progress flows to the new instance [17, 32]. However, attacks may go undetected because the necessary internal NF state is unavailable at the new instance. To overcome this problem, an SDN control application can wait for existing flows to terminate and only reroute new flows [22, 38], but this delays the mitigation of overload and increases the likelihood of SLA violations. NF accuracy may also be impacted due to some NF-internal state not being copied or shared.

In this example, the only way to avoid a trade-off between NF accuracy and performance is to allow a control application to quickly and safely move the internal IDS state for some flows from the original instance to the new instance, and update network forwarding state alongside. Similar needs arise in the context of other applications that rely on dynamic reallocation of packet processing: e.g., rapid NF upgrades and dynamic invocation of remote processing.

In this paper, we present OpenNF, a control plane architecture that provides efficient, coordinated control of both internal NF state and network forwarding state to allow quick, safe, and fine-grained reallocation of flows across NF instances. Using OpenNF, operators can create rich control applications that redistribute processing to optimally meet their performance, availability, security and cost objectives, thus avoiding the need to make undesirable trade-offs.

We address three major challenges in designing OpenNF:

CT1: Addressing race conditions. This is the most basic issue that arises when reallocating in-progress flows: When some internal NF state is being moved, packets may arrive at the source instance after the move starts, or at the destination instance before the state transfer finishes. Unless care is taken, updates to NF state due to such packets may either be lost or happen out of order, violating move safety. Similarly, when state is copied across NF instances, updates occurring contemporaneously may cause state to become inconsistent. Depending on the NF, these issues may hurt its accuracy.

To account for race conditions, we introduce two novel constructs: (1) an event abstraction to externally observe and prevent
Core problems

- Middleboxes tend to maintain state pertaining to the traffic they are processing

- Contrast to routers, switches etc. which do not need to “remember” anything besides the packet they are processing
Core problems

• Middleboxes tend to maintain state pertaining to the traffic they are processing

• Contrast to routers, switches etc. which do not need to “remember” anything besides the packet they are processing

• Problem: what happens to this state if processing switches to another middlebox?
OpenNF contributions

• **Mechanism** to coordinate scaling and load-balancing of middleboxes without service interruption

  • Based on state-transfer mechanism which can provide various useful properties (no packet lost/reordered)

  • Helps ensuring correctness (no state is lost/corrupted due to scaling/load-balancing)
OpenNF contributions

- **Mechanism** to coordinate scaling and load-balancing of middleboxes without service interruption
  - Based on state-transfer mechanism which can provide various useful properties (no packet lost/reordered)
  - Helps ensuring correctness (no state is lost/corrupted due to scaling/load-balancing)
- **Architecture** to coordinate state transfer operations
Relocating network functions

• Transferring network functions w/o service interruption requires coordination of:

  • **Application-level state transfer**: NF controller transfers state related to flows being redirected by the network level

  • **Network-level flow-steering**: SDN controller redirects flows from source to destination middlebox (already provided by standard OpenFlow network substrate)
OpenNF high-level architecture

1. **NF State Manager**
   - Retrieves state for NFs from SDN switches
   - Sends state updates to Flow Manager

2. **Flow Manager**
   - Sends flows to NFs
   - Sends flows to SDN switches
   - Manages state updates

3. **OpenNF Controller**
   - Receives commands from Control Application
   - Sends updates to NF State Manager

4. **Control Application**
   - Sends messages to OpenNF Controller

5. **Northbound API**
   - Allows communication between Control Application and OpenNF Controller

6. **Southbound API**
   - Allows communication between OpenNF Controller and NF State Manager

7. **SDN Switches**
   - Sends traffic to NFs

8. **NFs**
   - Processes traffic
   - Sends state updates to NF State Manager

**Key Components**
- **NF State Manager**: Manages state for NFs
- **Flow Manager**: Sends flows to NFs
- **OpenNF Controller**: Receives commands and sends updates
- **Control Application**: Sends messages to OpenNF Controller

**Communication Flows**
- **Northbound API**: Commands from Control Application to OpenNF Controller
- **Southbound API**: Updates from NF State Manager to OpenNF Controller
- **OpenNF Controller** to **Flow Manager**: Updates and commands
- **Flow Manager** to **NFs**: Flows
- **NFs** to **SDN Switches**: Traffic
- **SDN Switches** to **NFs**: Traffic

**Integration Points**
- **NF State Manager** with **OpenNF Controller**
- **Flow Manager** with **OpenNF Controller**
- **Control Application** with **OpenNF Controller**
OpenNF allows control applications to closely manage the behavior and performance of NFs to satisfy high level objectives without degrading NF accuracy. In particular, OpenNF supports the following objectives:

1. Fast failure recovery with low resource footprint. In the event of a failure, we can minimize downtime by rerouting flows to a non-failed instance.

2. Selectively invoking advanced remote processing. Depending on the specific needs of the network, control applications can decide to employ deeper and more advanced processing of a subset of incoming flows.

3. Cost of redirecting all traffic to the cloud. To achieve this, we again need the ability to create additional state for non-flow-based state, making it difficult to know the exact state to be moved when flows are rerouted.

As an example, consider a network scenario where an IDS module needs to process incoming flows to identify potential threats. If the processing of the flow is later determined to require more advanced processing, the NF instance is migrated to another location and the corresponding NF state is moved. This process allows for efficient and effective handling of network traffic, ensuring that critical data is processed accurately and timely.

In addition to these objectives, OpenNF also addresses issues related to state management, including:

1. One state allocation per flow, requiring some internal NF state manipulation.

2. Errors loss guarantee, ensuring that the data is delivered accurately and without loss.

3. Multiple NF instances from being moved and merged, preventing the creation of a combined instance.

4. Moving state as flows are rerouted, requiring some internal NF state manipulation.

5. Migrating instances (f), cloning and merging states at policy-defined frequencies.

Overall, OpenNF provides a flexible and scalable solution for managing NFs in complex network environments, ensuring that objectives are met while maintaining high performance and accuracy.
OpenNF allows control applications to closely manage the behavior of network functions (NFs). To support this, we need the ability to copy NF state, as we may want to back up pieces of NF state as they are updated. This eliminates the need to periodically create a backup of all NF state; this consumes network bandwidth and may not be feasible in all cases.

To support this, we need the ability to move, copy, or share NF state. Therefore, an NF control plane must offer the ability to combine NF state that applies to multiple flows in their entirety. The additional, unneeded state is only wasteful memory, but crucially, it can cause errors in the NF's behavior. Thus, the NF control plane should not restrict an NF's ability to move, copy, or share, and it should automatically maintain NF state to satisfy multiple objectives without degrading NF accuracy.

The network forwarding state update and resuming the flow of traffic (goal #1) are straightforward. Whenever an NF instance goes down or has to be terminated, the NF state and state operations are transferred back to the original NF instance. For the in-progress (and new) flows to a non-failed instance, for non-flow-based state, making it difficult to know the exact term of an instance. Furthermore, if one instance is late to move, copy or share, and terminating; but this can take a long time. Thus, the NF control plane must offer the ability to determine the precise sets of flows that specific NF instances should move, copy, or share, and to clone states at policy-defined frequencies. Vendors and open source projects, such as Pico Replication, provide an interface to NFs. However, these interfaces are not designed to address the challenges of NF state management. They provide a shared library that NFs use to create, access, and to clone states at policy-defined frequencies. They also provide a shared library that NFs use to create, access, and clone states at policy-defined frequencies.

Finally, the OpenNF high-level architecture (Figure 2) that combines existing control planes with techniques for VM migration and process replication does not address the above requirements and challenges. In this architecture, the network state in a way that fully satisfies all goals—e.g., it provides fast failure recovery with low resource footprint, allows selective invocation of advanced remote processing, and provides control over, and coordination of, traffic forwarding.

OpenNF is a novel control plane architecture that satisfies the aforementioned requirements and challenges. In this architecture, the network state is updated and resumed for traffic forwarding, allowing control applications to closely manage the behavior of NFs. It provides fast failure recovery with low resource footprint, allows selective invocation of advanced remote processing, and provides control over, and coordination of, traffic forwarding. It also provides the ability to capture this additional state while it is updated. This eliminates the need to periodically create a backup of all NF state, which can be wasteful in terms of network bandwidth and may not be feasible in all cases. Additionally, it allows control applications to access, move, copy, or share NF state at policy-defined frequencies, which is crucial for ensuring the accuracy of NFs.

In the OpenNF high-level architecture, a control application initiates and manages high-level operations (scaling, load-balancing, replication). It triggers two novel schemes to overcome underlying issues: (1) an NF instance may fail while in the middle of updating its state and state operations. We design two schemes to overcome this issue. (2) certain NF instances may migrate to another instance. We design two schemes to overcome this issue. (3) ask the controller to provide certain guarantees. We design two schemes to overcome this issue.

Based on NF output or external input, control applications: (1) initiate state for a specific NF instance, (2) direct the controller to provide the needed state, and (3) the controller provides the needed state. They provide a shared library that NFs use to create, access, and clone states at policy-defined frequencies. Vendor-supplied controllers that move, copy, and share NF state are ill-suited to support other complex control applications.

The OpenNF high-level architecture (Figure 2) that combines existing control planes with techniques for VM migration and process replication does not address the above requirements and challenges. In this architecture, the network state is transferred back to the original NF instance. This eliminates the need to periodically create a backup of all NF state, which can be wasteful in terms of network bandwidth and may not be feasible in all cases. Additionally, it allows control applications to access, move, copy, or share NF state at policy-defined frequencies, which is crucial for ensuring the accuracy of NFs.
OpenNF high-level architecture

Initiates/manages high-level operations (scaling, load-balancing, replication)

Control Application

OpenNF Controller

NF State Manager

Flow Manager

Northbound API

Southbound API

Performs NF state transfer

Performs flow steering

NFs

SDN Switches
High-level functions to state manipulation operations

- Goal: perform load-balancing by relocating a set of flows $S$ from middlebox $A$ to $B$:
  - Move state for flows in $S$ from $A$ to $B$
  - Steer flows in $S$ so that they traverse $B$ instead of $A$
High-level functions to state manipulation operations

- Goal: **perform load-balancing by relocating a set of flows S from middlebox A to B:**
  - Move state for flows in S from A to B
  - Steer flows in S so that they traverse B instead of A

- Goal: **perform scaling by terminating middlebox A and relocating its processing task to B**
  - Same as above…
  - …but at the end A is terminated
High-level functions to state manipulation operations

• Goal: perform load-balancing by relocating a set of flows S from middlebox A to B:
  • Move state for flows in S from A to B
  • Steer flows in S so that they traverse B instead of A

• Goal: perform scaling by terminating middlebox A and relocating its processing task to B
  • Same as above…
  • …but at the end A is terminated

• Goal: replicate a middlebox $A_1$ to another middlebox $A_2$ to provide a “hot spare”
  • Instantiate $A_2$
  • Copy state for the set of flows S managed by $A_1$ to $A_2$

• ...
State manipulation operations to low-level operations

**NFV controller:** implements copy/move/share operations by issuing appropriate sequences of low-level Southbound API calls
State manipulation operations to low-level operations

**Northbound API:**
- move(srcInst, dstInst, filter, scope, properties)
- copy(srcInst, dstInst, filter, scope)
- share(list<inst>, filter, scope, consistency)

**NFV controller:** implements copy/move/share operations by issuing appropriate sequences of low-level Southbound API calls
State manipulation operations to low-level operations

**Northbound API:**
- `move(srcInst, dstInst, filter, scope, properties)`
- `copy(srcInst, dstInst, filter, scope)`
- `share(list<inst>, filter, scope, consistency)`

**NFV controller:** implements copy/move/share operations by issuing appropriate sequences of low-level Southbound API calls

**Southbound API:**
- `getPerflow()`, `putPerflow()`, `delPerflow()`, `getMultiflow()`, `putMultiflow()`, `delMultiflow()`
- `getAllflows()`, `delAllflows()`, `enableEvents()`, `disableEvents()`
Types of state

- **Per-flow state**: program state keeping track of properties of a single network flow

- **Multi-flow state**: program state keeping track of aggregate properties of multiple flows

- **All-flow state**: program state keeping track of global properties of middlebox processing (information referring to and/or affecting all flows being processed)
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:

```
Flow of interest
```

```
1. getPerflow(F)
```

```
OpenFlow switch
```

```
SDN controller
```

```
State
```

```
MB_SRC
```

```
MB_DST
```

```
State
```
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:

```
1. getPerflow(F)
2. delPerflow(F)
```

```
OpenFlow switch  Flow of interest  MBSRC  State  MB_DST
```
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:

```
1. getPerflow(F)
2. delPerflow(F)
```

Diagram:
- OpenFlow switch
- SDN controller
- State
- MB_SRC
- MB_DST
- Flow of interest
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:

```
Flow of interest
```

```
1. getPerflow(F)
2. delPerflow(F)
3. putPerflow(F)
```

```
OpenFlow switch
```

```
SDN controller
```

```
MB_{SRC}
```

```
MB_{DST}
```

```
State
```
OpenNF: move operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:

```
1. getPerflow(F)
2. delPerflow(F)
3. putPerflow(F)
```

Flow of interest: OpenFlow switch -> SDN controller -> MB_SRC -> MB_DST

State
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:

```
1. getPerflow(F)
2. delPerflow(F)
3. putPerflow(F)
4. Reconfigure forwarding table
```

Flow of interest

```
OpenFlow switch → SDN controller → MB_{SRC} → MB_{DST} → State
```
OpenNF: *move* operation

- Can provide various properties (loss-freeness, order preservation)
  - Trade off: correctness vs delay
- Simplest case: move w/o guarantees:

```
1. getPerflow(F)
2. delPerflow(F)
3. putPerflow(F)
4. Reconfigure forwarding table
```

- OpenFlow switch
- SDN controller
- MB\textsubscript{SRC}
- MB\textsubscript{DST}
- State

*Flow of interest*
Move w/o guarantees: issues

- State created in $MB_{SRC}$ for packets arrived after getPerflow is not moved to $MB_{DST}$

- **Simplest solution:** instruct $MB_{SRC}$ (or SDN switch) to ignore (or drop) all packets received after getPerflow
Move w/o guarantees: issues

- State created in MB$_{SRC}$ for packets arrived after getPerflow is not moved to MB$_{DST}$

  - **Simplest solution:** instruct MB$_{SRC}$ (or SDN switch) to ignore (or drop) all packets received after getPerflow

- Only OK if processing is resilient to packet losses…
Move w/o guarantees: issues

- State created in MB$_{SRC}$ for packets arrived after getPerflow is not moved to MB$_{DST}$

  - **Simplest solution:** instruct MB$_{SRC}$ (or SDN switch) to ignore (or drop) all packets received after getPerflow

- Only OK if processing is resilient to packet losses…

- …and **still suboptimal!** E.g. IDS will continue to function but may miss attacks (e.g. portscans)
Move w/o guarantees: issues

- State created in MB$_{SRC}$ for packets arrived after getPerflow is not moved to MB$_{DST}$

  - **Simplest solution:** instruct MB$_{SRC}$ (or SDN switch) to ignore (or drop) all packets received after getPerflow

- Only OK if processing is resilient to packet losses…

- …and **still suboptimal**! E.g. IDS will continue to function but may miss attacks (e.g. portscans)

- OpenNF can provide additional guarantees: **loss-freeness**, **in-order delivery**
Loss-free move

• Leverage a novel OpenNF contribution: the event management API
Loss-free move

- Leverage a novel OpenNF contribution: the event management API

- Allows NFV controller to instruct middleboxes to process, buffer, or drop certain categories of packets while at the same time forwarding the same packets to the NFV controller
Loss-free move

• Leverage a novel OpenNF contribution: the event management API

• Allows NFV controller to instruct middleboxes to process, buffer, or drop certain categories of packets while at the same time forwarding the same packets to the NFV controller

• In loss-free move, these primitives are used to buffer and then forward all packets received by MB_{SRC} after state has been moved, to MB_{DST}
Loss-free move - part I

- OpenFlow switch
- Flow of interest
- $\text{MB}_{\text{SRC}}$
- State
- $\text{MB}_{\text{DST}}$
- SDN controller
Loss-free move - part I

Flow of interest

OpenFlow switch -> MB_{SRC} -> State -> MB_{DST}

1. enableEvents(F, drop)
Loss-free move - part I

Flow of interest

OpenFlow switch

SDN controller

MB_{SRC}

State

1: enableEvents(F, drop)

MB_{DST}
Loss-free move - part I

OpenFlow switch

Flow of interest

State

MB_{SRC}

Flow of interest

1.enableEvents(F, drop)

Packets

SDN controller

MB_{DST}
Loss-free move - part I

1. enableEvents(F, drop)
2. getPerflow(F)

Flow of interest

OpenFlow switch -> Flow of interest

Packets

SDN controller

MB_{SRC}

State

MB_{DST}
Loss-free move - part I

1. enableEvents(F, drop)
2. getPerflow(F)

Flow of interest

OpenFlow switch

Flow of interest

Packets

SDN controller

MB_SRC

State

MB_DST
Loss-free move - part I

1. enableEvents(F, drop)
2. getPerflow(F)
3. delPerflow(F)

Flow of interest

OpenFlow switch

Flow of interest

SDN controller

Packets

State

MB_{SRC}

State

MB_{DST}
Loss-free move - part I

1. enableEvents(F, drop)
2. getPerflow(F)
3. delPerflow(F)
Loss-free move - part II
Loss-free move - part II

Flow of interest

OpenFlow switch → Flow of interest → MB_SRC

State

SDN controller

Packets

4.putPerflow(F)
Loss-free move - part II

OpenFlow switch

Flow of interest

Packets

SDN controller

Flow of interest

4.putPerflow(F)

MB_{SRC}

MB_{DST}

State
Loss-free move - part II

OpenFlow switch → Flow of interest → Packets

SDN controller

Flow of interest → MB_SRC

4.putPerflow(F)

Flow of interest → MB_DST

State
Loss-free move - part II

5. Reconfigure forwarding table

4. putPerflow(F)

OpenFlow switch → Flow of interest → MB_{SRC} → Flow of interest → SDN controller → Packets

Packets → MB_{SRC} → Flow of interest → MB_{DST} → State
Loss-free move - part II

5. Reconfigure forwarding table

4. putPerflow(F)

Flow of interest

OpenFlow switch

SDN controller

Packets

Flow of interest

Flow of interest

MB_{SRC}

MB_{DST}

State
Loss-free move - part II

1. OpenFlow switch
2. SDN controller
3. MB_{SRC}
4. putPerflow(F)
5. Reconfigure forwarding table

Flow of interest
Loss-free move - part II

Issue: packets may still arrive out-of-order at MB\textsubscript{DST} (why?)
OpenNF: copy operation

- **Copy**: used e.g. for failure recovery (copy state from an active middle box to a backup instance)
  - Uses `getPerflow/putPerflow` (no deletion)
  - Provides **eventual consistency**
OpenNF: share operation

- **Share**: used to ensure that all state updates by a middlebox are directly reflected by the state of one or more other middleboxes (distributed processing)

- **Strict consistency**: state updates to a middlebox state are immediately reflected in the state of all other middleboxes (updates to the shared state reflects the order in which packets were received by `switch`)

- **Strong consistency**: order of state updates may differ from the order in which packets were received by `switch` but all updates performed by an individual middlebox must be “seen” by other middleboxes in the order in which they were performed
Results

Figure 10: Efficiency of \texttt{move} with no guarantees (NG), loss-free (LF), and loss-free and order-preserving (LF+OP) with and without parallelizing (PL) and early-release (ER) optimizations; traffic rate is 2500 packets/sec; times are averaged over 5 runs and the error bars show 95% confidence intervals.
Results - II

(a) Packet drops during a parallelized move with no guarantees  
(b) Total time for a parallelized loss-free move

Figure 11: Impact of packet rate and number of per-flows states on parallelized move with and without a loss-free guarantee
Other challenges in middlebox management

- Designing policy languages to describe middlebox deployments
- Offloading middleboxes to the cloud
Offloading middleboxes to the cloud

Making Middleboxes Someone Else’s Problem: Network Processing as a Cloud Service

Justine Sherry
UC Berkeley

Arvind Krishnamurthy
University of Washington

Shaddi Hasan
UC Berkeley

Sylvia Ratnasamy
UC Berkeley

Colin Scott
UC Berkeley

Vyas Sekar
Intel Labs

ABSTRACT
Modern enterprises almost ubiquitously deploy middlebox processing services to improve security and performance in their networks. Despite this, we find that today’s middlebox infrastructure is expensive, complex to manage, and creates new failure modes for the networks that use them. Given the promise of cloud computing to decrease costs, ease management, and provide elasticity and fault-tolerance, we argue that middlebox processing can benefit from outsourcing the cloud. Arriving at a feasible implementation, however, is challenging due to the need to achieve functional equivalence with traditional middleboxes deployments without sacrificing performance or increasing network complexity.

In this paper, we motivate, design, and implement APLOMB, a practical service for outsourcing enterprise middleboxes processing to the cloud. Our discussion of APLOMB is data-driven, guided by a survey of 57 enterprise networks, the first large-scale academic study of middleboxes deployment. We show that APLOMB solves real problems faced by network administrators, can outsource over 90% of middlebox hardware in a typical large enterprise network, and, in a case study of a real enterprise, imposes an average latency penalty of 1 ms and median bandwidth inflation of 3.8%.

Categories and Subject Descriptors
C.2.0 [Computer-Communication Networks]: General—Security and firewalls; C.2.1 [Network Architecture and Design]: Distributed applications; C.2.3 [Network Operations]: Network management

General Terms
Design, Management, Measurement

Keywords
Middlebox, Cloud, Outsourcing

1. INTRODUCTION
Today’s enterprise networks rely on a wide spectrum of specialized appliances or middleboxes. Trends such as the proliferation of smartphones and wireless video are set to further expand the range of middlebox applications [19]. Middleboxes offer valuable benefits, such as improved security (e.g., firewalls and intrusion detection systems), improved performance (e.g., proxies) and reduced bandwidth costs (e.g., WAN optimizers). However, as we show in §2, middleboxes come with high infrastructure and management costs, which result from their complex and specialized processing, variations in management tools across devices and vendors, and the need to consider policy interactions between these appliance and other network infrastructure.

The above shortcomings mirror the concerns that motivated enterprises to transition their in-house IT infrastructures to managed cloud services. Inspired by this trend, we ask whether the promised benefits of cloud computing—reduced expenditure for infrastructure, personnel and management, pay-by-use, the flexibility to try new services without sunk costs, etc.—can be brought to middlebox infrastructure. Beyond improving the status quo, cloud-based middlebox services would also make the security and performance benefits of middleboxes available to users such as small businesses and home and mobile users who cannot otherwise afford the associated costs and complexity.

We envision enterprises outsourcing the processing of their traffic to third-party middlebox service providers running in the cloud. Our proposal represents a significant change to enterprise networks, and hence we first validate that this exercise is worthwhile by examining what kind of a burden middleboxes impose on enterprises. The research literature, however, offers surprisingly few real-world studies; the closest study presents anecdotal evidence from a single large enterprise [42]. We thus start with a study of 57 enterprise networks, aimed at understanding (1) the nature of real-world middlebox deployments (e.g., types and numbers of middleboxes), (2) “pain points” for network administrators, and (3) failure modes. Our study reveals that middleboxes do impose significant infrastructure and management overhead across a spectrum of enterprise networks and that the typical number of middleboxes in an enterprise is comparable to its traditional L2/L3 infrastructure.

Our study establishes the costs associated with middlebox deployments and the potential benefits of outsourcing them. We then examine different options for architecting cloud-based middlebox services. To be viable, such an architecture must meet three challenges:

(1) Functional equivalence. A cloud-based middlebox must offer functionality and semantics equivalent to that of an on-site middlebox—i.e., a firewall must drop packets correctly, an intrusion detection system (IDS) must trigger identical alarms, etc. In contrast to traditional endpoint applications, this is challenging because middlebox functionality may be topology dependent. For example, traffic compression must be implemented before traffic leaves the enterprise access link, and an IDS that requires stateful processing must see all packets in both directions of a flow. Today, these requirements are met by deliberately placing middleboxes “on path” at network choke points within the enterprise—options that are not readily available in a cloud-based architecture. As we shall see, these topological constraints complicate our ability to outsource middlebox processing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCOMM’12, August 13-17, 2012, Helsinki, Finland.
Copyright 2012 ACM 978-1-4503-1419-0/12/08 . . . $15.00.
Middlebox deployments are expensive and complex

- Already outlined in “The Middlebox Manifesto”

- Sherry et al. provide more evidence by performing a study of 57 enterprise networks (!)

- Study further confirms that enterprise networks tend to use large number of middleboxes

- The study also investigates complexities and problems that arise when deploying middlebox in-house
Prevalence of middleboxes

Figure 1: Box plot of middlebox deployments for small (fewer than 1k hosts), medium (1k-10k hosts), large (10k-100k hosts), and very large (more than 100k hosts) enterprise networks. Y-axis is in log scale.
Complexity factor #1: management and updating

- Median interval between network-wide middlebox updates: 4 years
- Average number of vendors contacted per update: 4.9
Complexity factor #1: management and updating

- Median interval between network-wide middlebox updates: 4 years
- Average number of vendors contacted per update: 4.9
- Every 4 years, administrator have to evaluate/purchase/install/become familiar with a new set of devices
Complexity factor #1: management and updating

- Median interval between network-wide middlebox updates: 4 years

- Average number of vendors contacted per update: 4.9

- Every 4 years, administrator have to evaluate/purchase/install/become familiar with a new set of devices

- Also, technology evolves enough over 4 years that upgrading requires purchasing new hardware
Complexity factor #2: overload and failures

<table>
<thead>
<tr>
<th></th>
<th>Misconfig.</th>
<th>Overload</th>
<th>Physical/Electric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firewalls</td>
<td>67.3%</td>
<td>16.3%</td>
<td>16.3%</td>
</tr>
<tr>
<td>Proxies</td>
<td>63.2%</td>
<td>15.7%</td>
<td>21.1%</td>
</tr>
<tr>
<td>IDS</td>
<td>54.5%</td>
<td>11.4%</td>
<td>34%</td>
</tr>
</tbody>
</table>

Table 1: Fraction of network administrators who estimated misconfiguration, overload, or physical/electrical failure as the most common cause of middlebox failure.
The solution: offload middleboxes to the cloud

- “APLOMB” stands for “ApPLiance for Outsourcing MiddleBoxes”
The solution: offload middleboxes to the cloud

• “APLOMB” stands for “ApPLiance for Outsourcing MiddleBoxes”

• Benefits:
  
  • Cloud can leverage economies of scale to provide middlebox services for lower costs than in-house installation
  
  • Upgrade fatigue, failures etc. are entirely dealt with by cloud provider
The solution: offload middleboxes to the cloud

• “APLOMB” stands for “ApPLiance for Outsourcing MiddleBoxes”

• Benefits:
  
  • Cloud can leverage **economies of scale** to provide middlebox services for lower costs than in-house installation
  
  • Upgrade fatigue, failures etc. are entirely dealt with by cloud provider

• Drawbacks:
  
  • Middlebox processing can be **topology-dependent** (e.g. WAN optimizers, IDSs)
  
  • Offloading middlebox tasks implies traffic redirection with potentially high costs in terms of performance
Proposed approach

- General idea: DNS-based redirection of network flows to the cloud

- **Incoming traffic:** DNS requests for target website resolve to cloud address - cloud processes incoming traffic and tunnel output to target website

- **Outgoing traffic:** traffic generated by target website is intercepted by APLOMB gateway, routed through the cloud, and from the cloud to the final destination

(c) DNS-based redirection minimizes latency and allows providers to control PoP selection for each request.
Considerations

• This basic approach works for the majority of middleboxes...
  
  • ... but some location-dependent middleboxes are harder to offload
Considerations

• This basic approach works for the majority of middleboxes...

  • ... but some location-dependent middleboxes are harder to offload

• E.g. proxies, load-balancers, whose purpose is to decrease the amount of BW in and out of the network

  • Purpose defeated if traffic must be redirected outside the network for the middleboxes to work
Considerations

• This basic approach works for the majority of middleboxes...
  • ... but some **location-dependent middleboxes** are harder to offload

• E.g. proxies, load-balancers, whose purpose is to decrease the amount of BW in and out of the network
  • Purpose defeated if traffic must be redirected outside the network for the middleboxes to work

• **Solutions:** do not offload those middleboxes; or perform traffic compression between APLOMB gateway and cloud (does not solve all problems)
  • E.g., IDS analyzing internal traffic is harder to offload profitably
Results

- Does offload increase latency?

Figure 6: Round Trip Time (RTT) inflation when redirecting traffic between US PlanetLab nodes through Amazon PoPs.
Results

- Does offload increase latency?

Figure 6: Round Trip Time (RTT) inflation when redirecting traffic between US PlanetLab nodes through Amazon PoPs.

How is this possible?
Results

- Does offload increase latency?

How is this possible?

- Which percentage of middleboxes can be offloaded?

Figure 6: Round Trip Time (RTT) inflation when redirecting traffic between US PlanetLab nodes through Amazon PoPs.

Figure 14: Number of middleboxes in the enterprise with and without APLOMB+. The enterprise has an atypical number of ‘internal’ firewalls and NIDS.
Results

• Does offload increase latency?

![Figure 6: Round Trip Time (RTT) inflation when redirecting traffic between US PlanetLab nodes through Amazon PoPs.](image)

How is this possible?

• Which percentage of middleboxes can be offloaded?

![Figure 14: Number of middleboxes in the enterprise with and without APLOMB+. The enterprise has an atypical number of ‘internal’ firewalls and NIDS.](image)

Internal firewalls/IDS’s
Class dismissed!

- Hope you enjoyed it…
- … do not forget to fill in course evaluations!