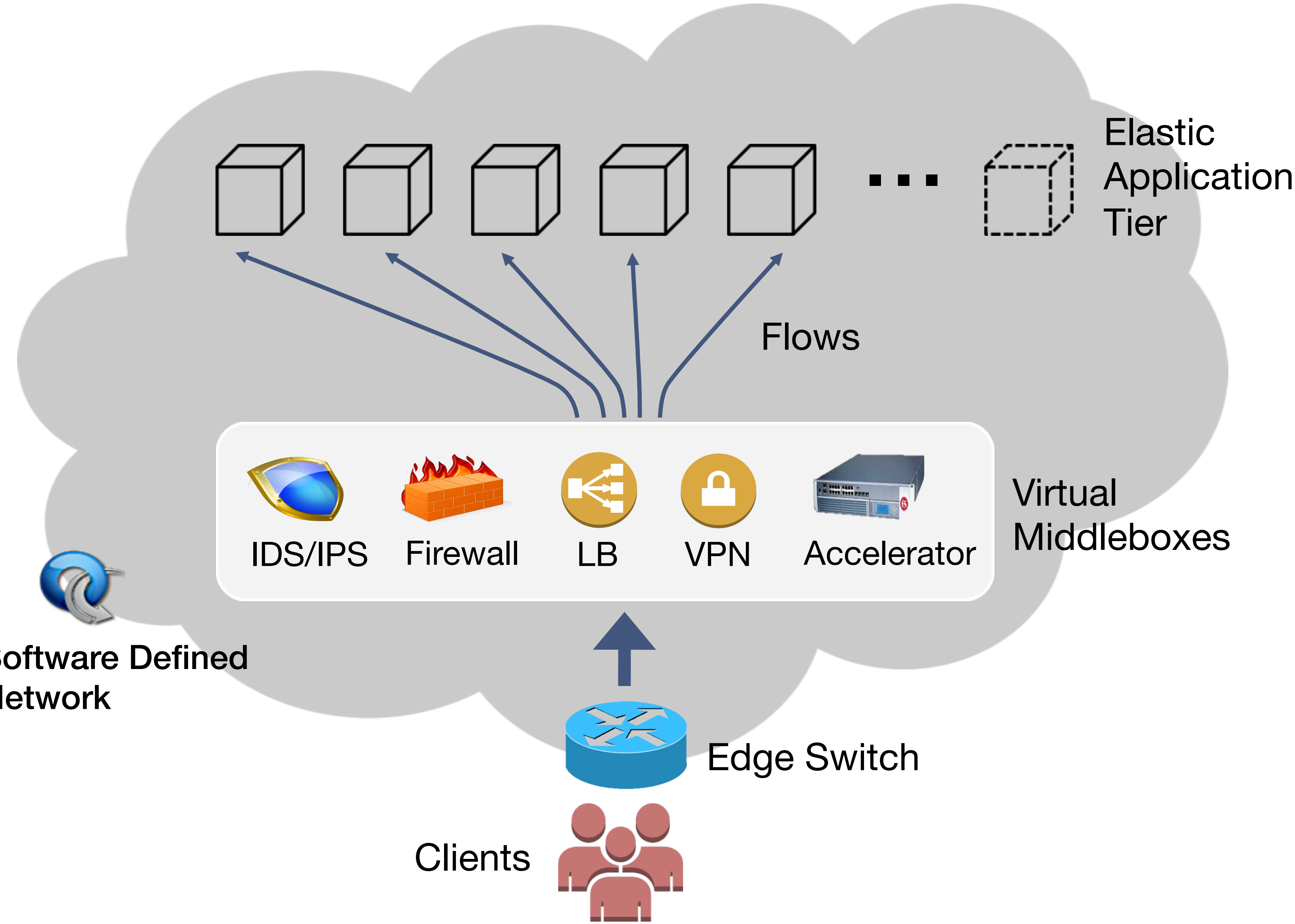
Split / Merge

System Support for Elastic Execution in Virtual Middleboxes

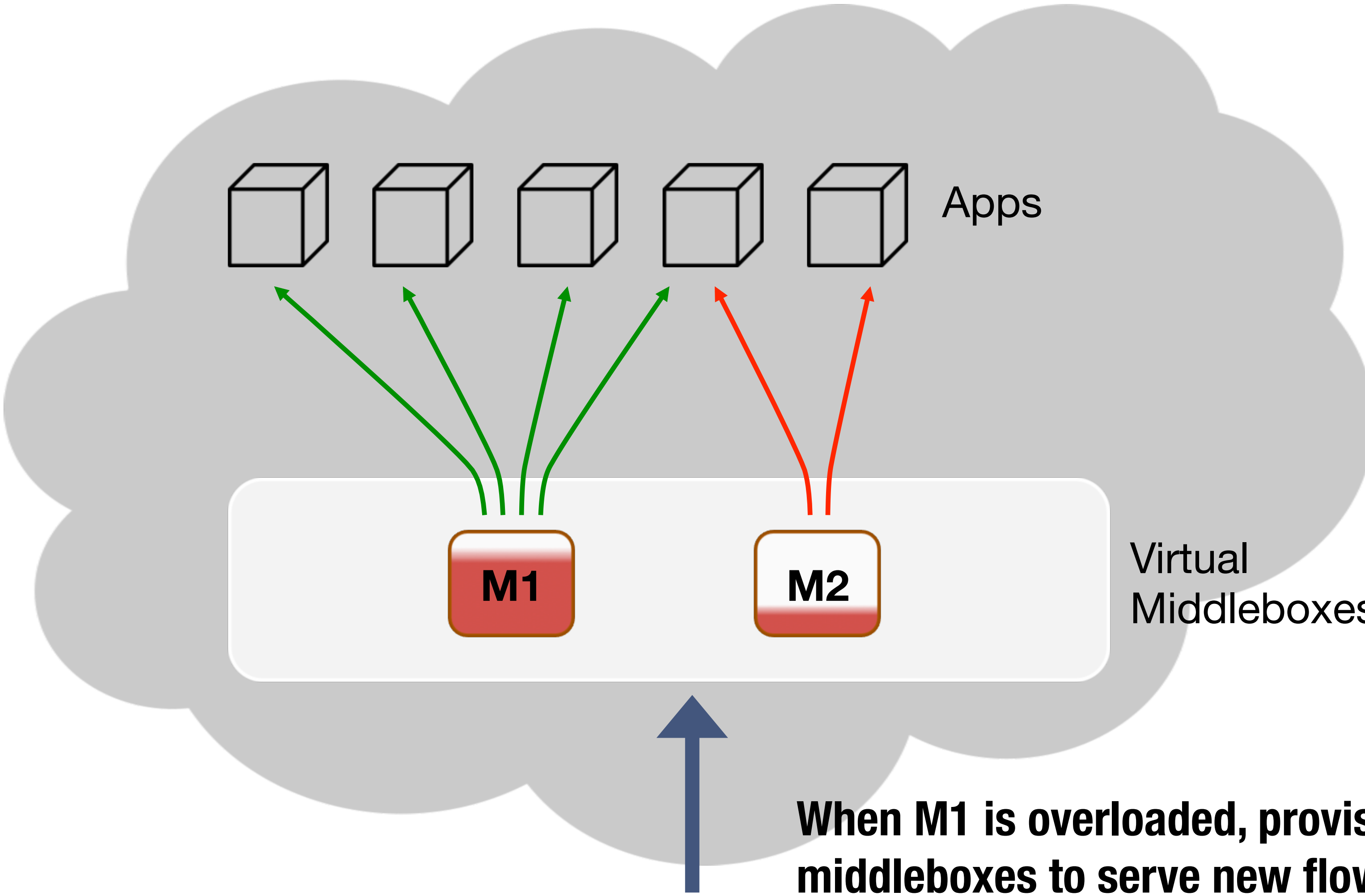
Shriram **RAJAGOPALAN** IBM Research & UBC
Dan **WILLIAMS** IBM Research
Hani **JAMJOOM** IBM Research
Andrew **WARFIELD** UBC

The Problem

Elastic Applications Need Elastic Middleboxes

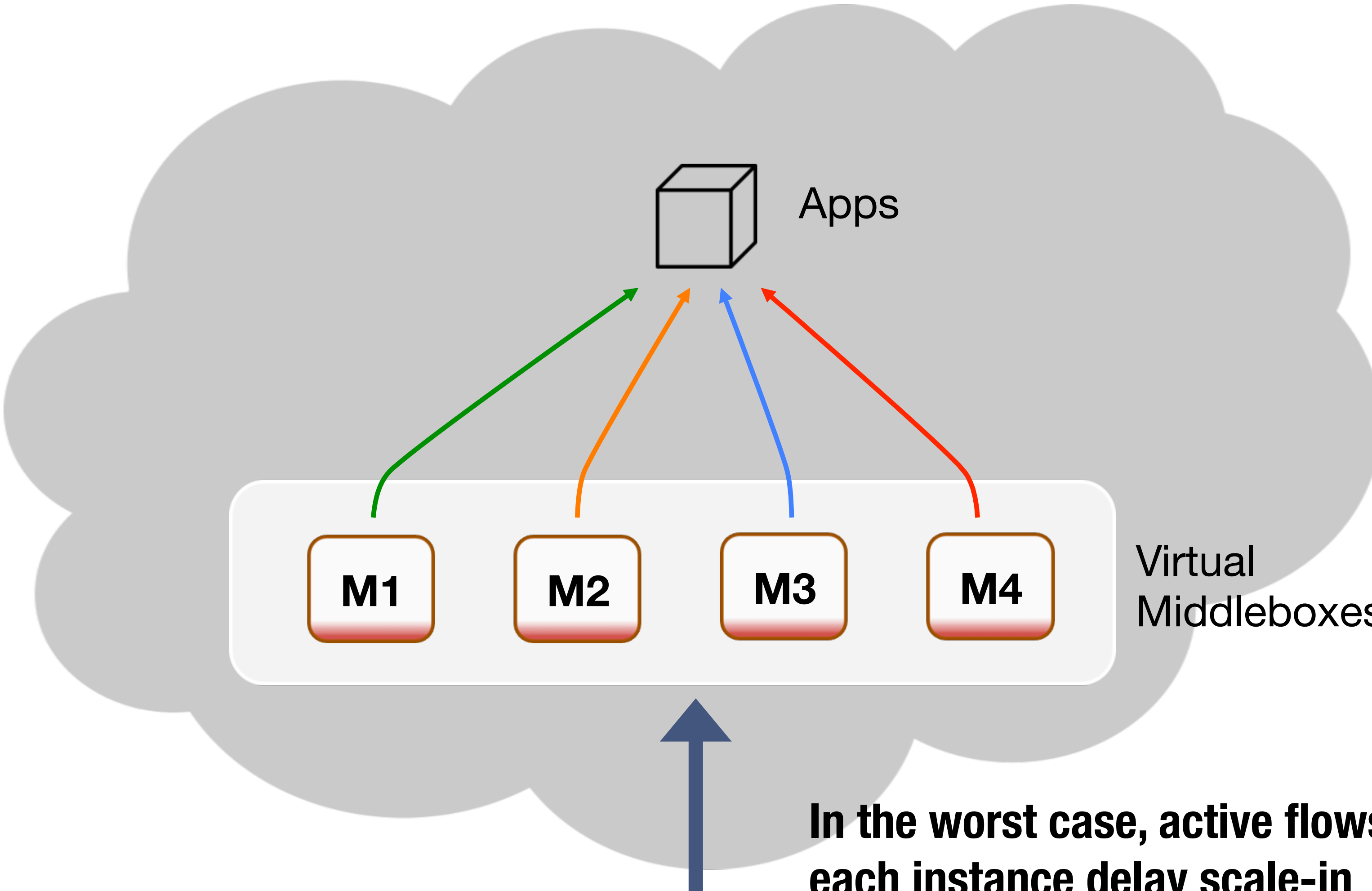


Hotspots Cannot be Alleviated Quickly



When M1 is overloaded, provision more middleboxes to serve new flows

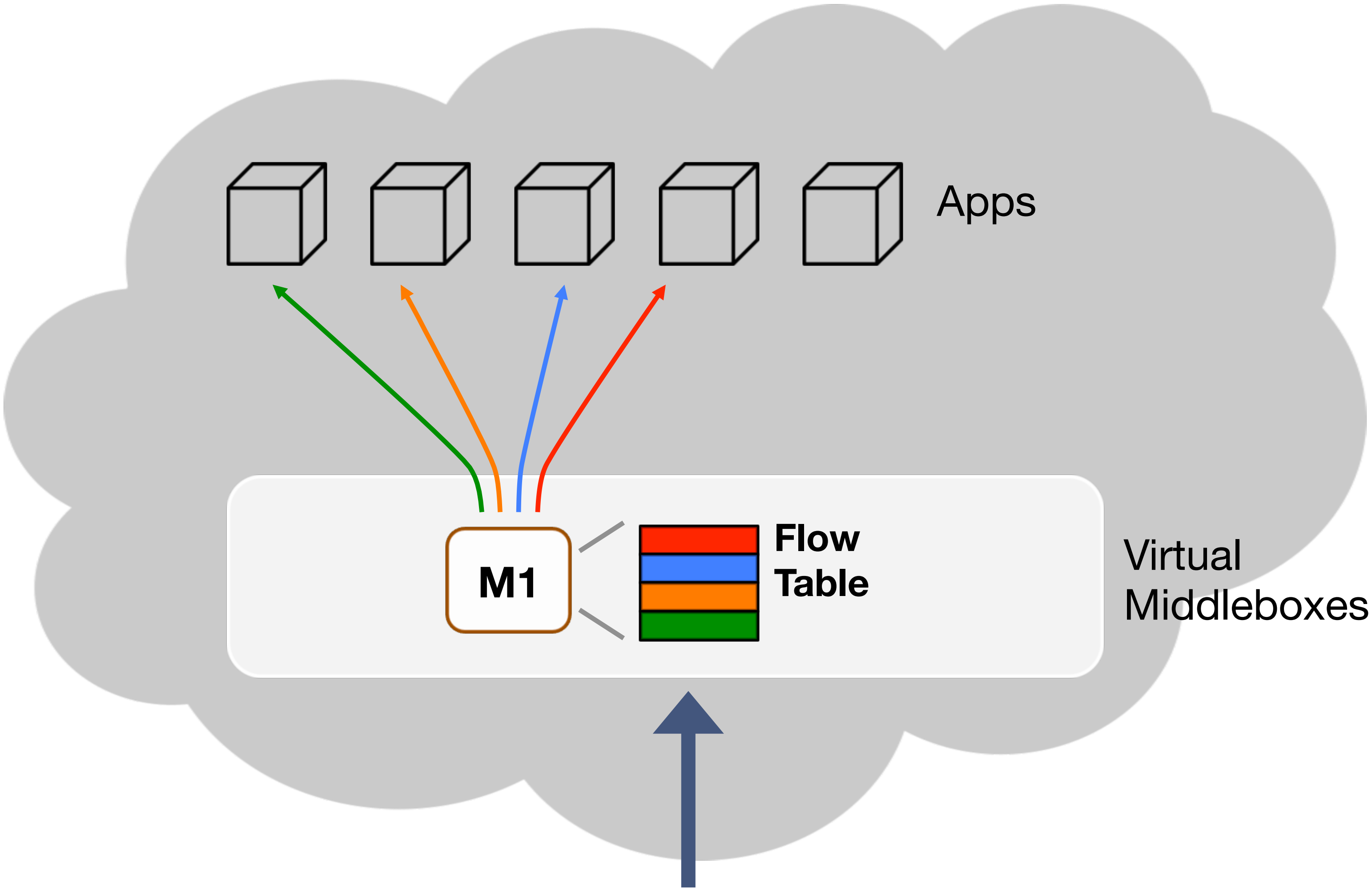
Scaling Inefficiencies Lead to Poor Utilization



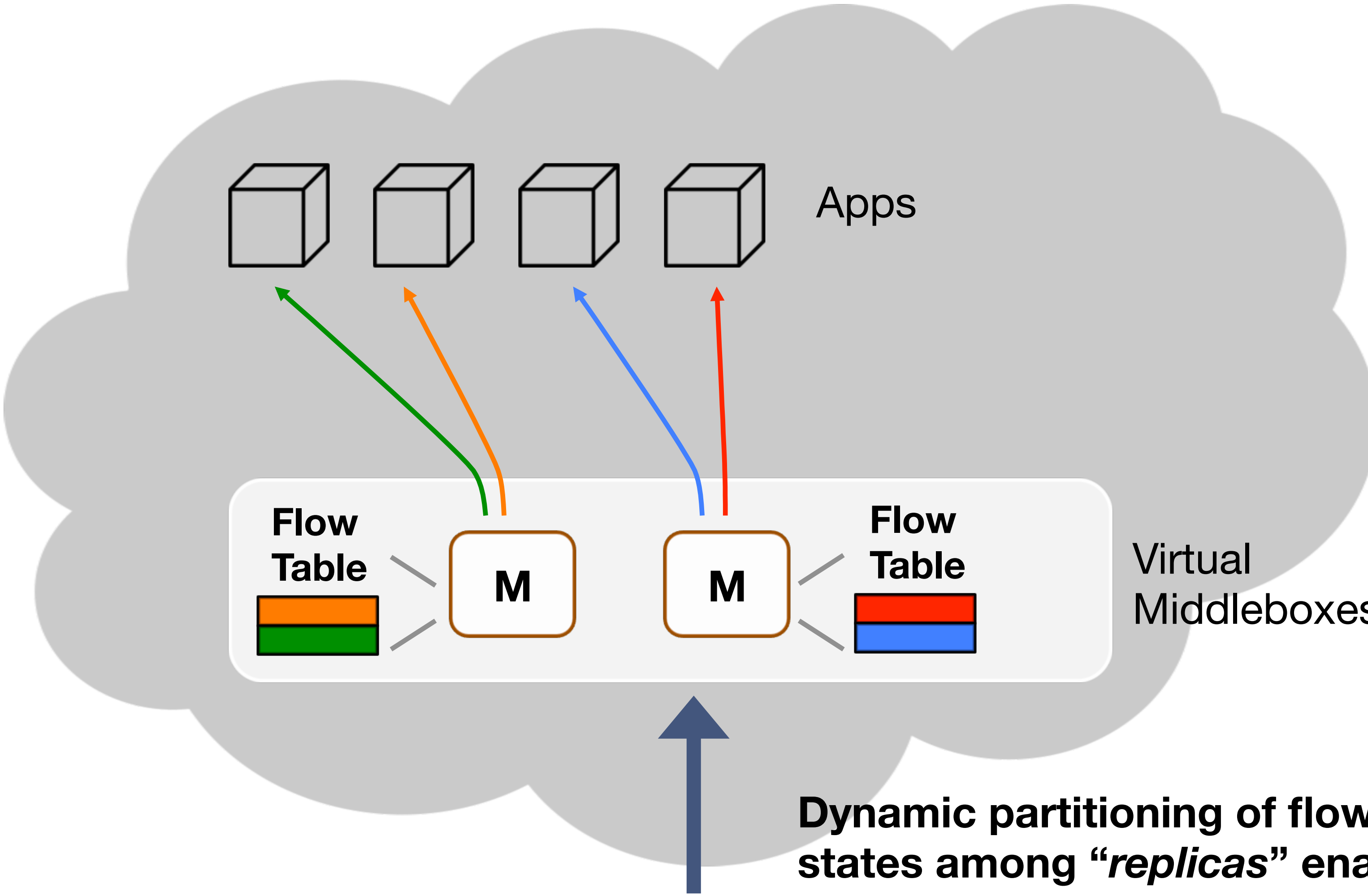
In the worst case, active flows in each instance delay scale-in

The Insight

Flow State is Naturally Partitioned

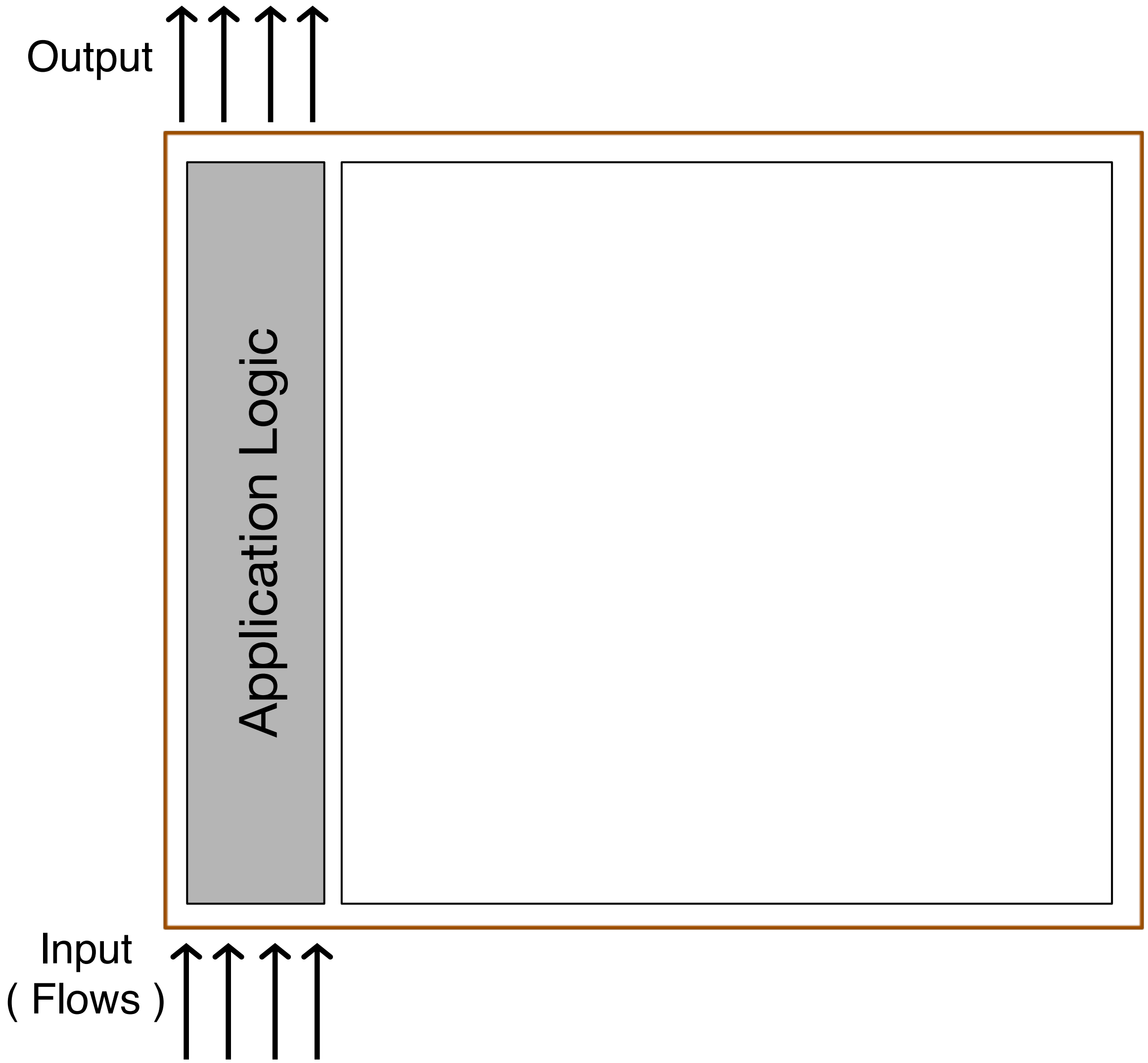


Enabling Elasticity in Virtual Middleboxes

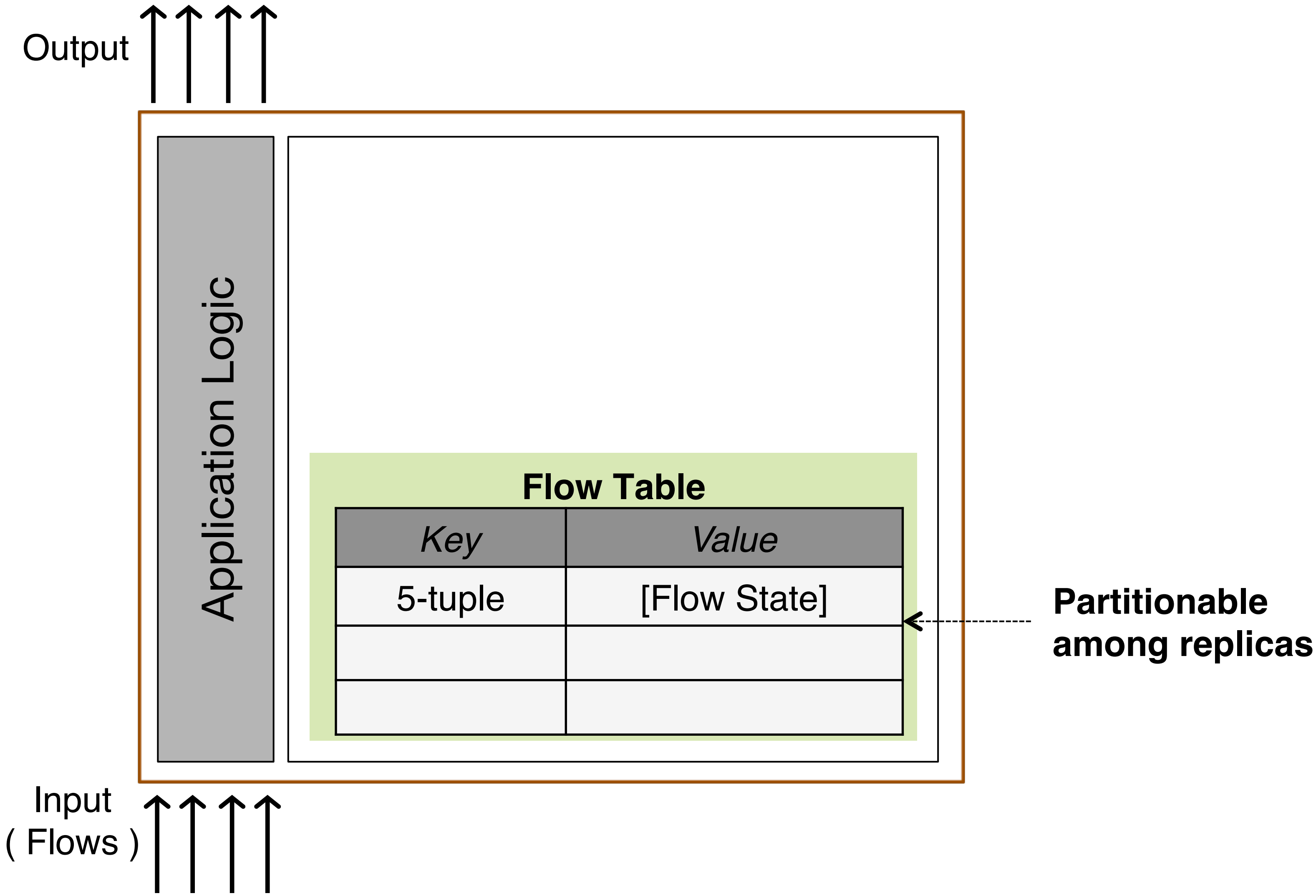


Dynamic partitioning of flow states among “*replicas*” enables elastic execution

Understanding the State Inside a Middlebox

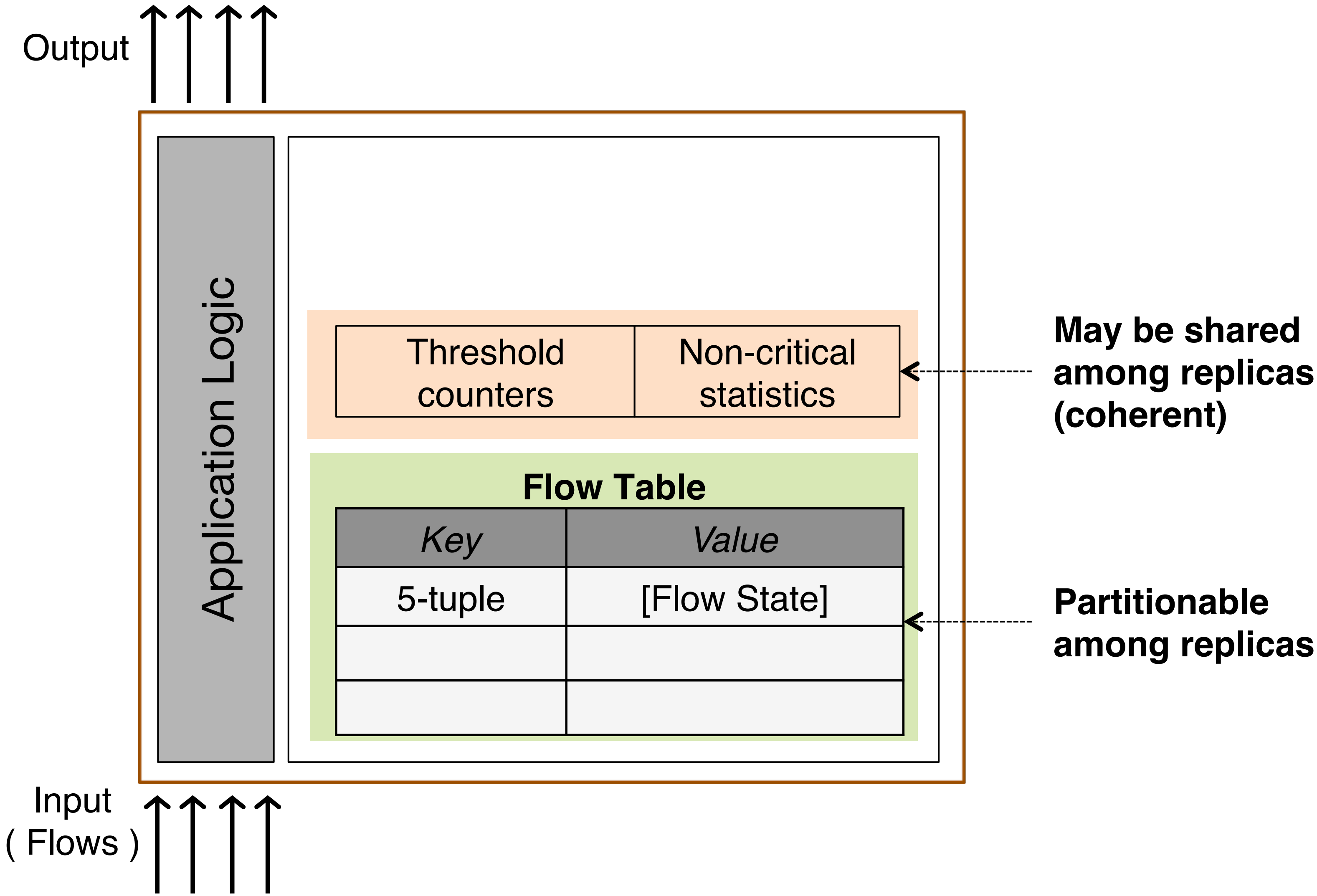


Understanding the State Inside a Middlebox

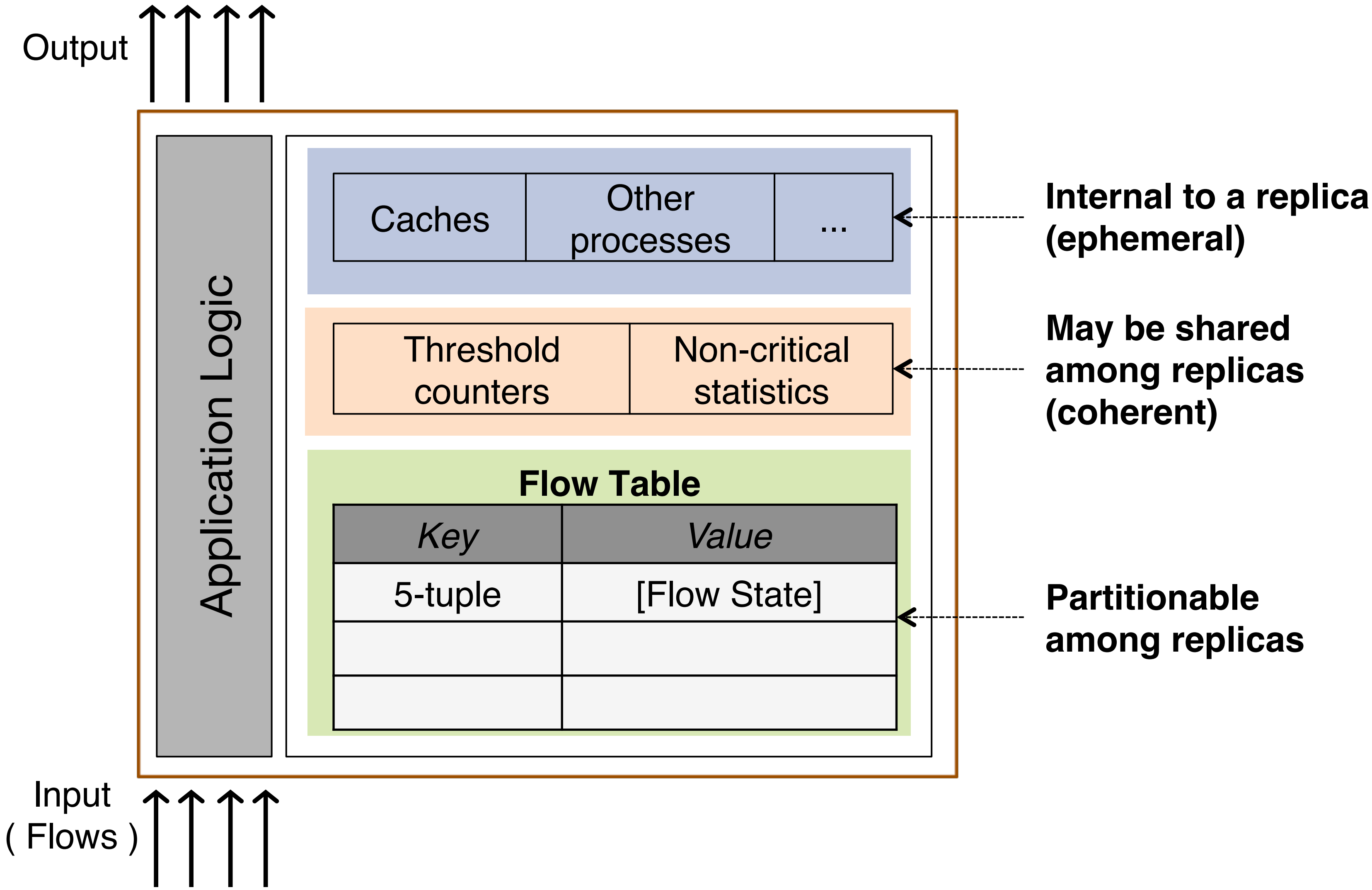


Key	Value
5-tuple	[Flow State]

Understanding the State Inside a Middlebox

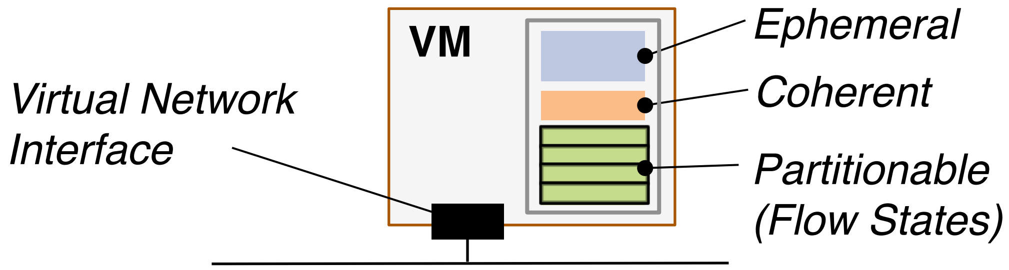


Understanding the State Inside a Middlebox

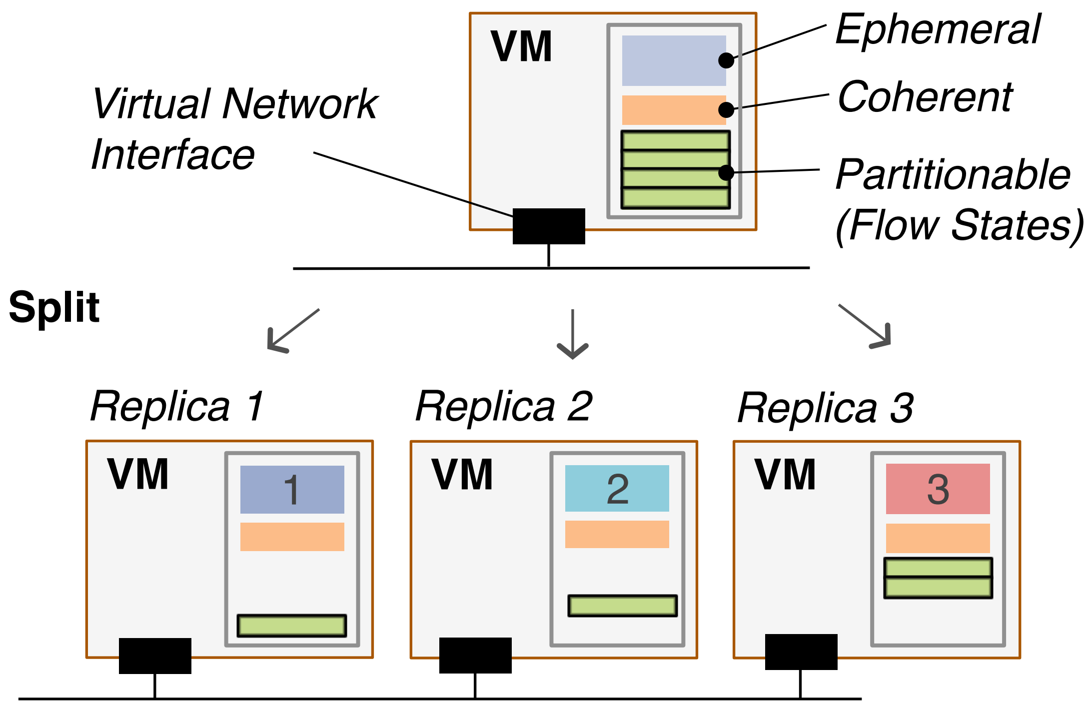


Our Contribution

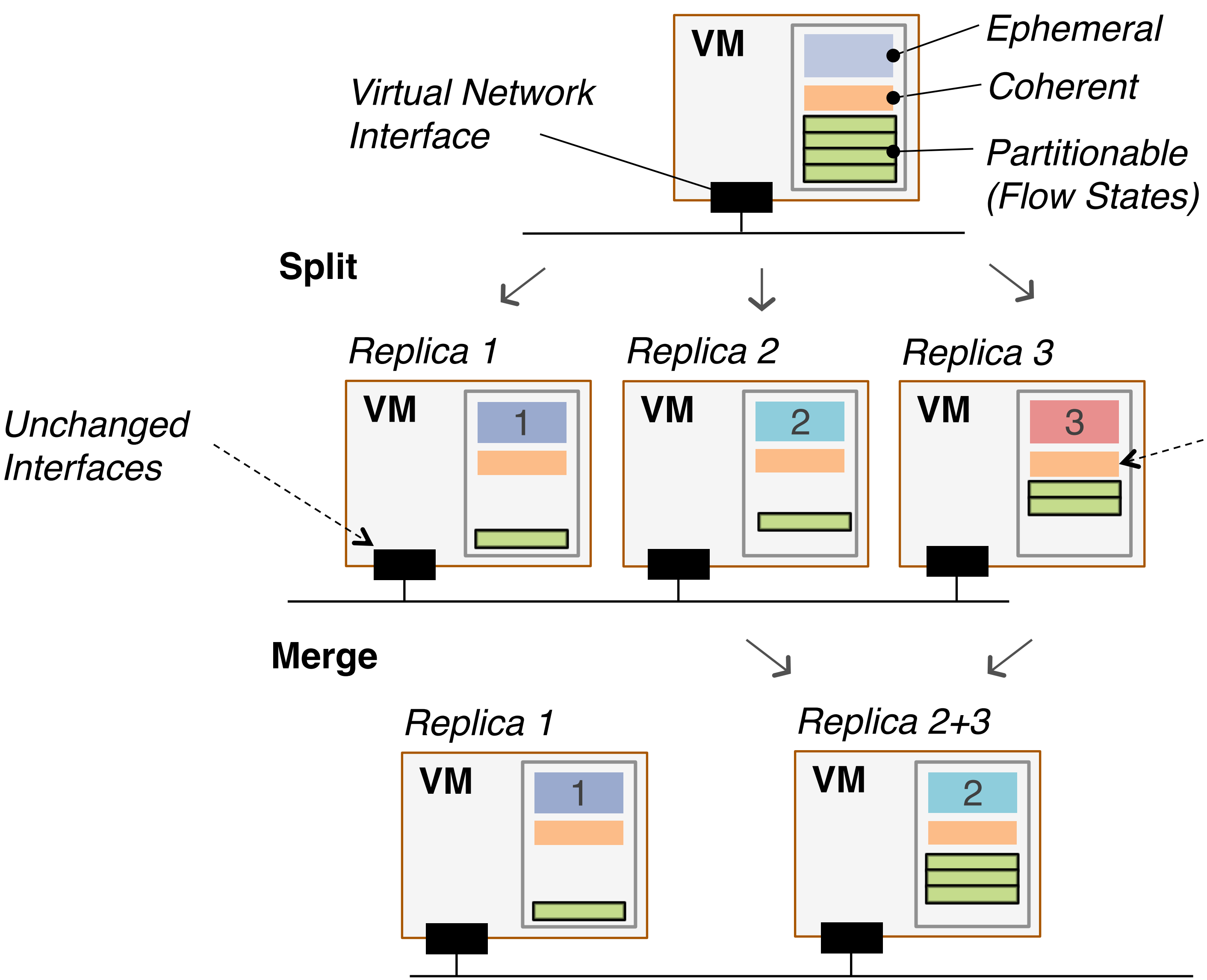
Split/Merge: A State-Centric Approach to Elasticity



Split/Merge: A State-Centric Approach to Elasticity



Split/Merge: A State-Centric Approach to Elasticity

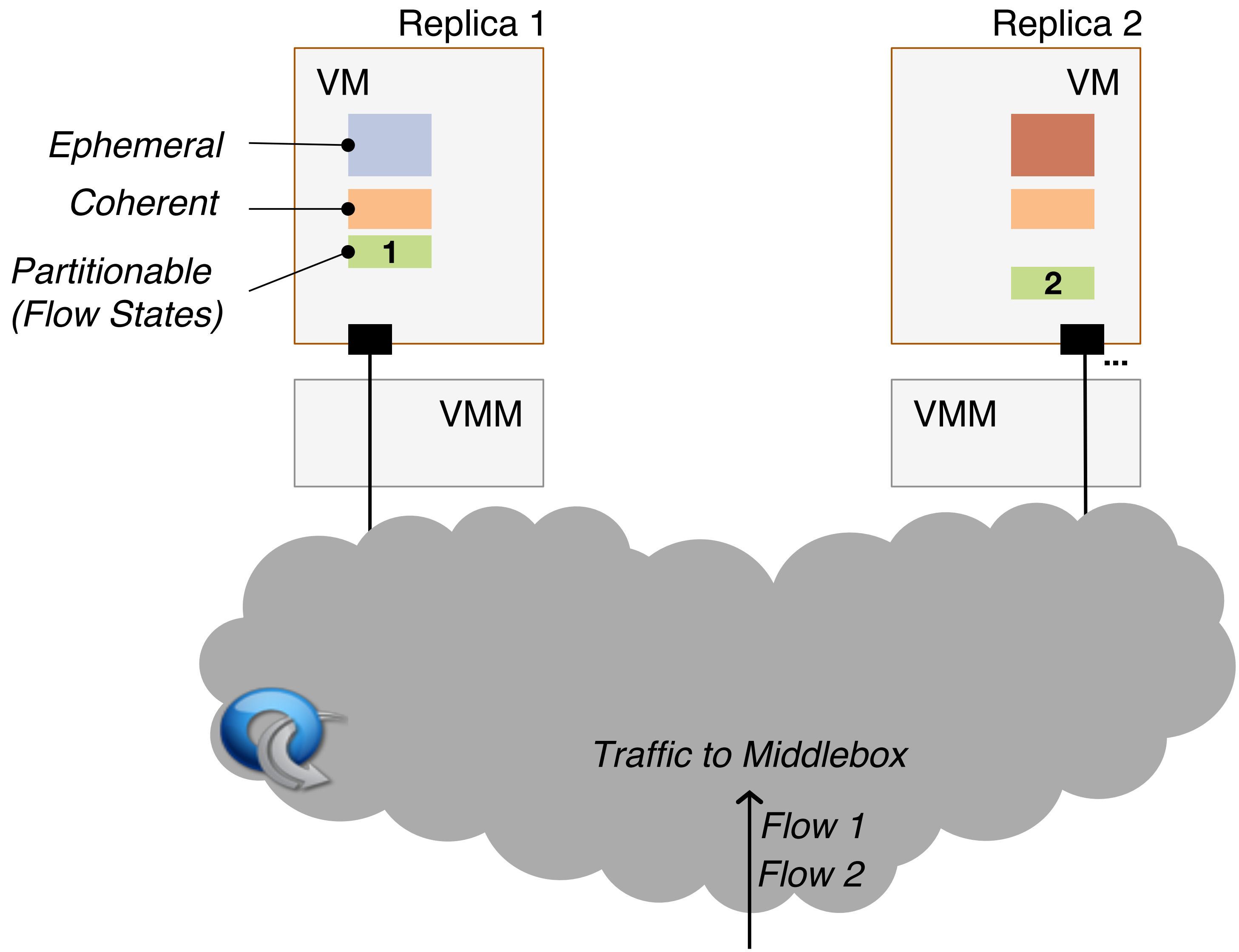


Implementation

FreeFlow

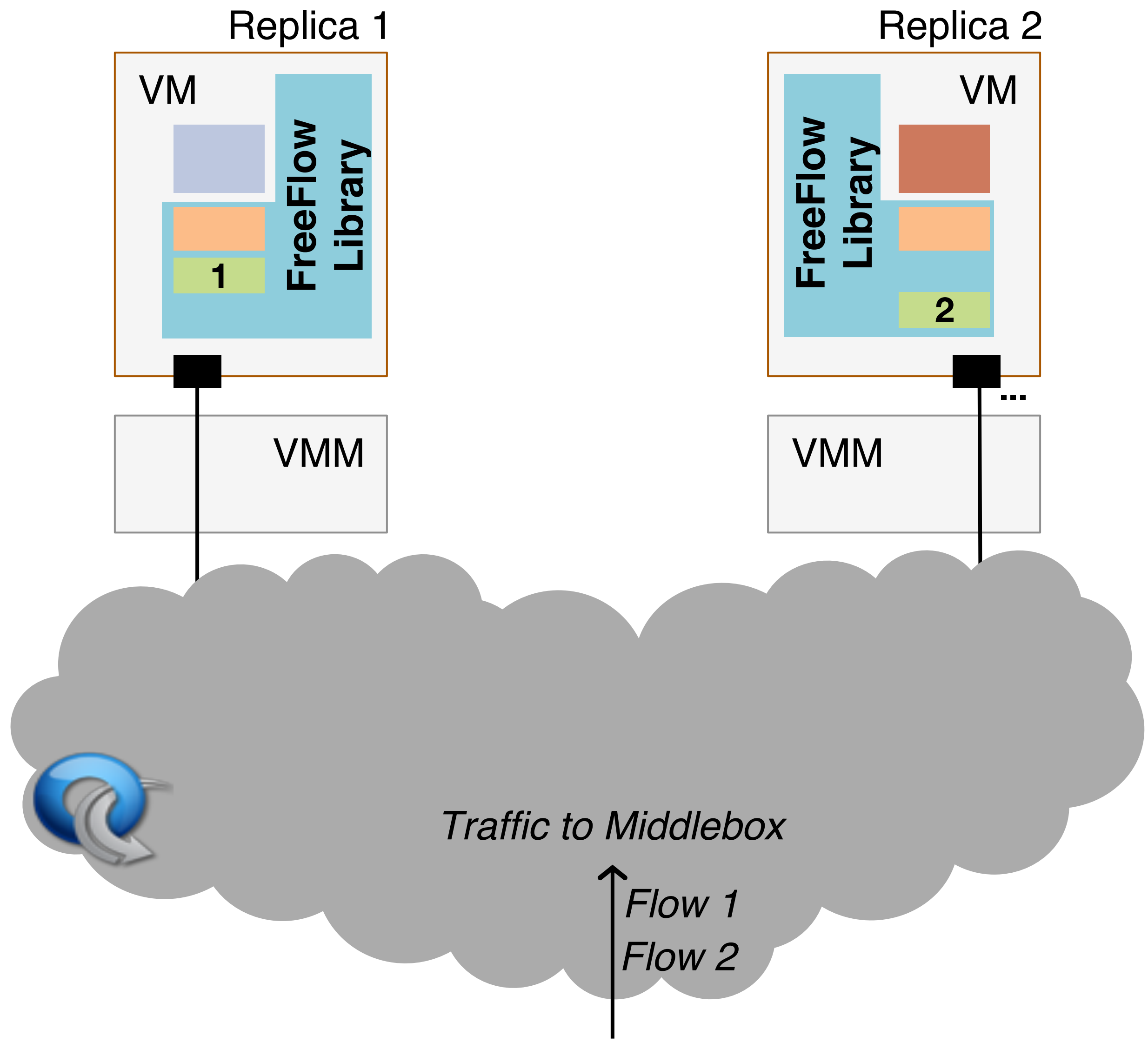
- A VMM based runtime that provides Split/Merge abstraction to applications
- Developers modify application code to annotate flow state
- FreeFlow takes care of the rest!

FreeFlow: A Split/Merge Implementation



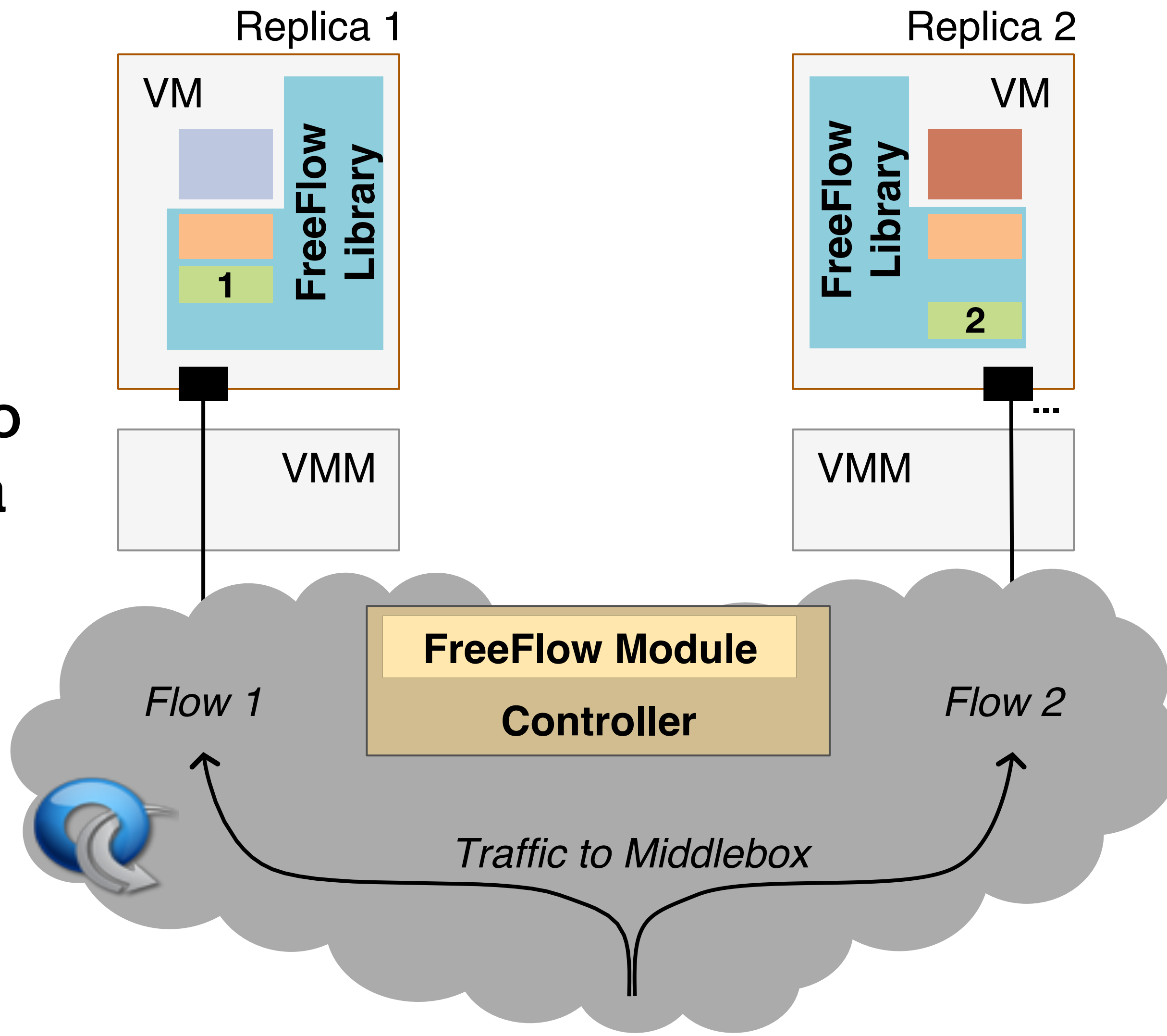
FreeFlow: A Split/Merge Implementation

- Need to manage application state



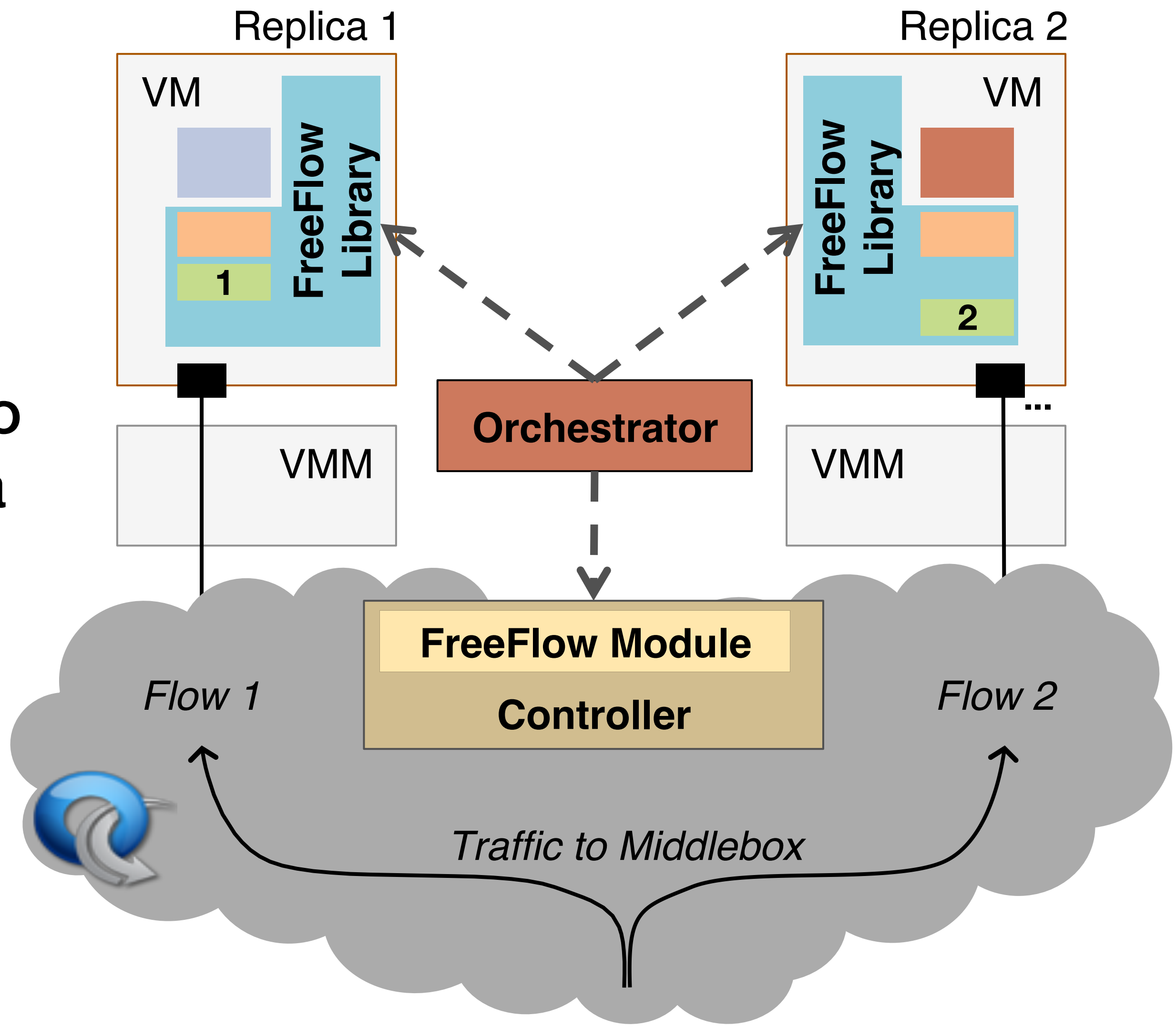
FreeFlow: A Split/Merge Implementation

- Need to manage application state
- Need to ensure flows are routed to the correct replica



FreeFlow: A Split/Merge Implementation

- Need to manage application state
- Need to ensure flows are routed to the correct replica
- Need to decide when to split or merge a replica



Annotating State using FreeFlow API

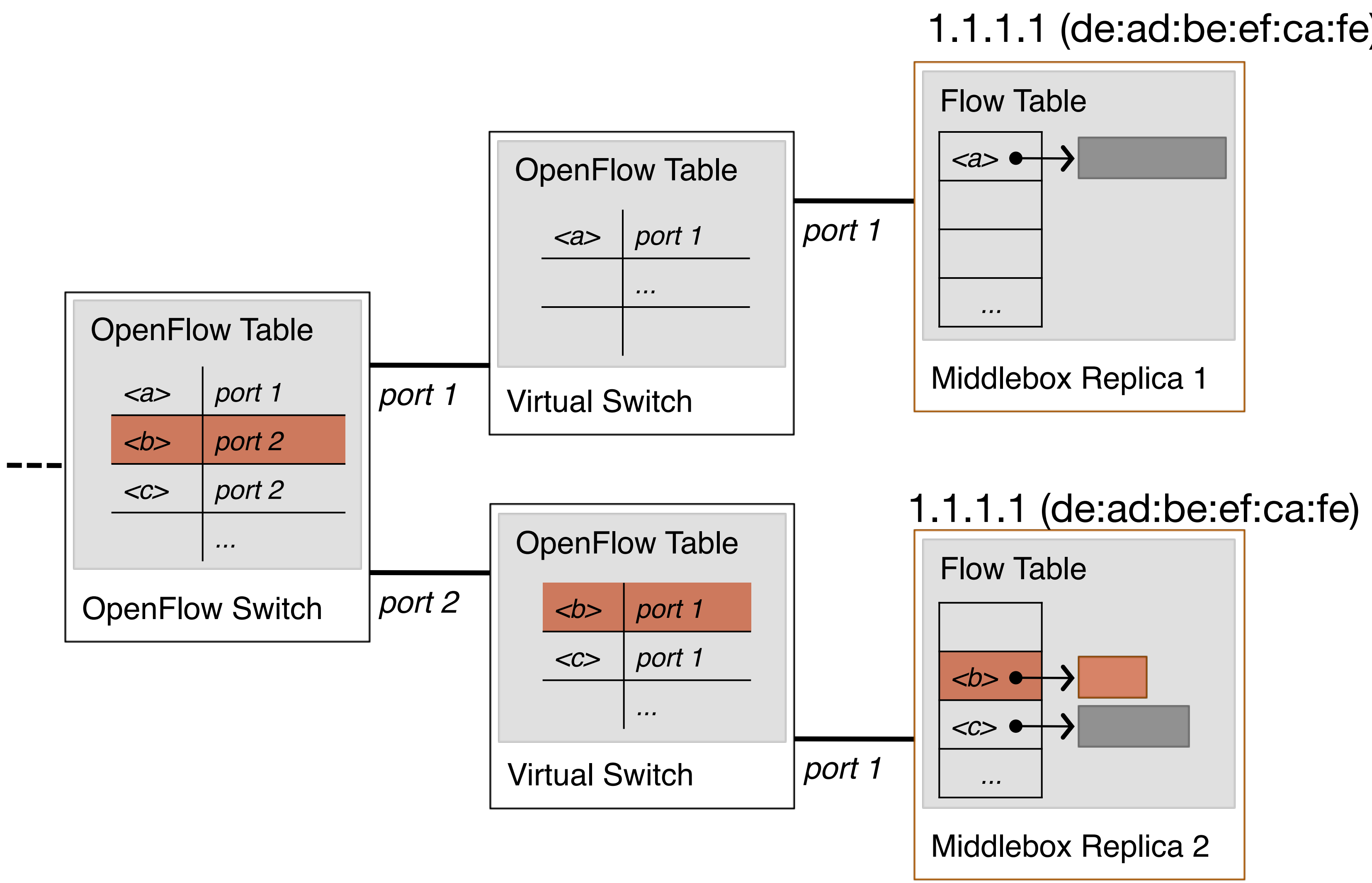
Partitioned State API

```
create_flow(flow_key, size)
delete_flow(flow_key)
flow_state get_flow(flow_key)
put_flow(flow_key)

flow_timer(flow_key, timeout, callback)
```

Coherent State API

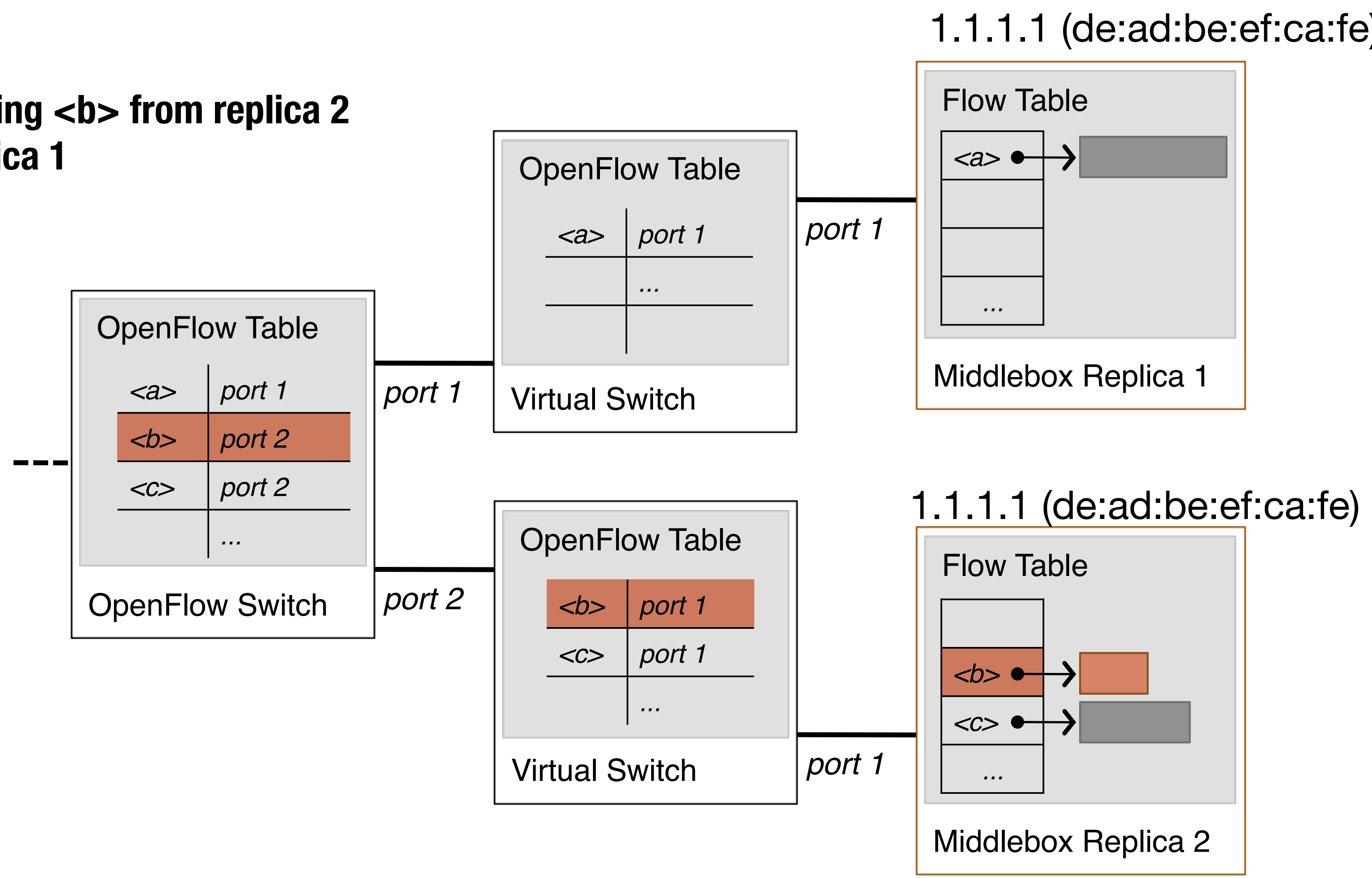
```
create_shared(key, size, cb)
delete_shared(key)

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

Forwarding Flows Correctly using OpenFlow



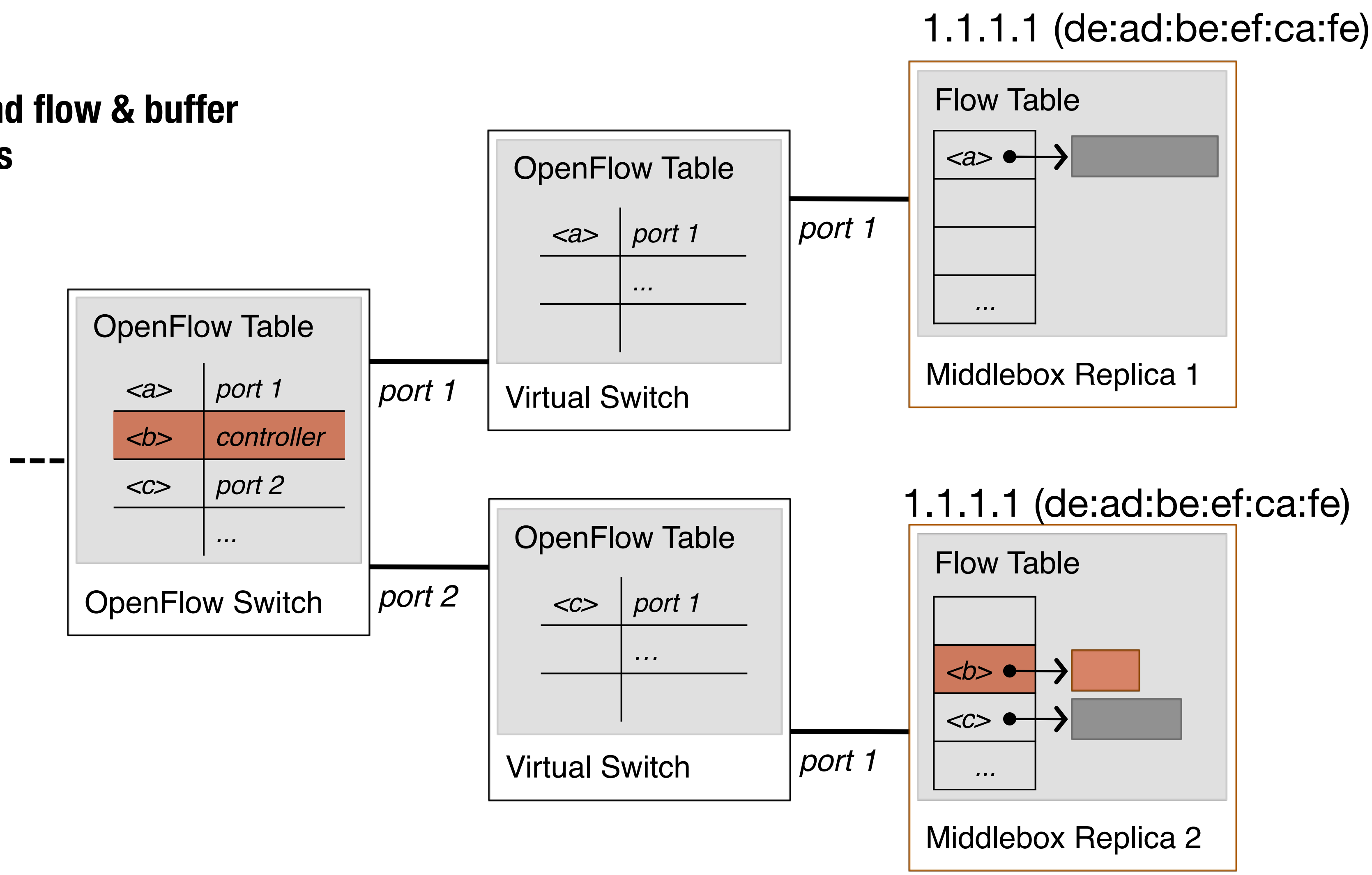
Flow Migration

Migrating <b> from replica 2 to replica 1



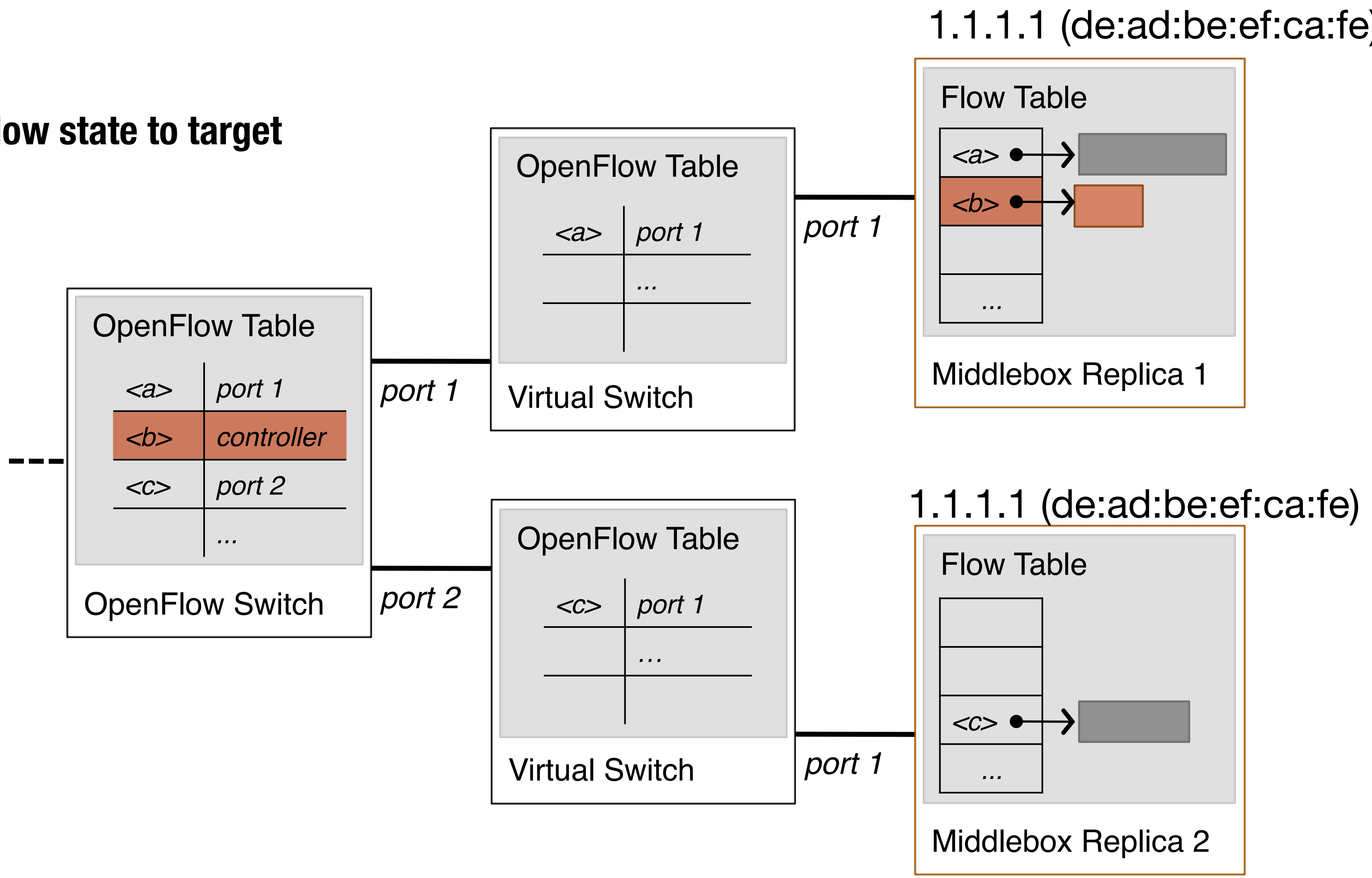
Flow Migration

Suspend flow & buffer packets



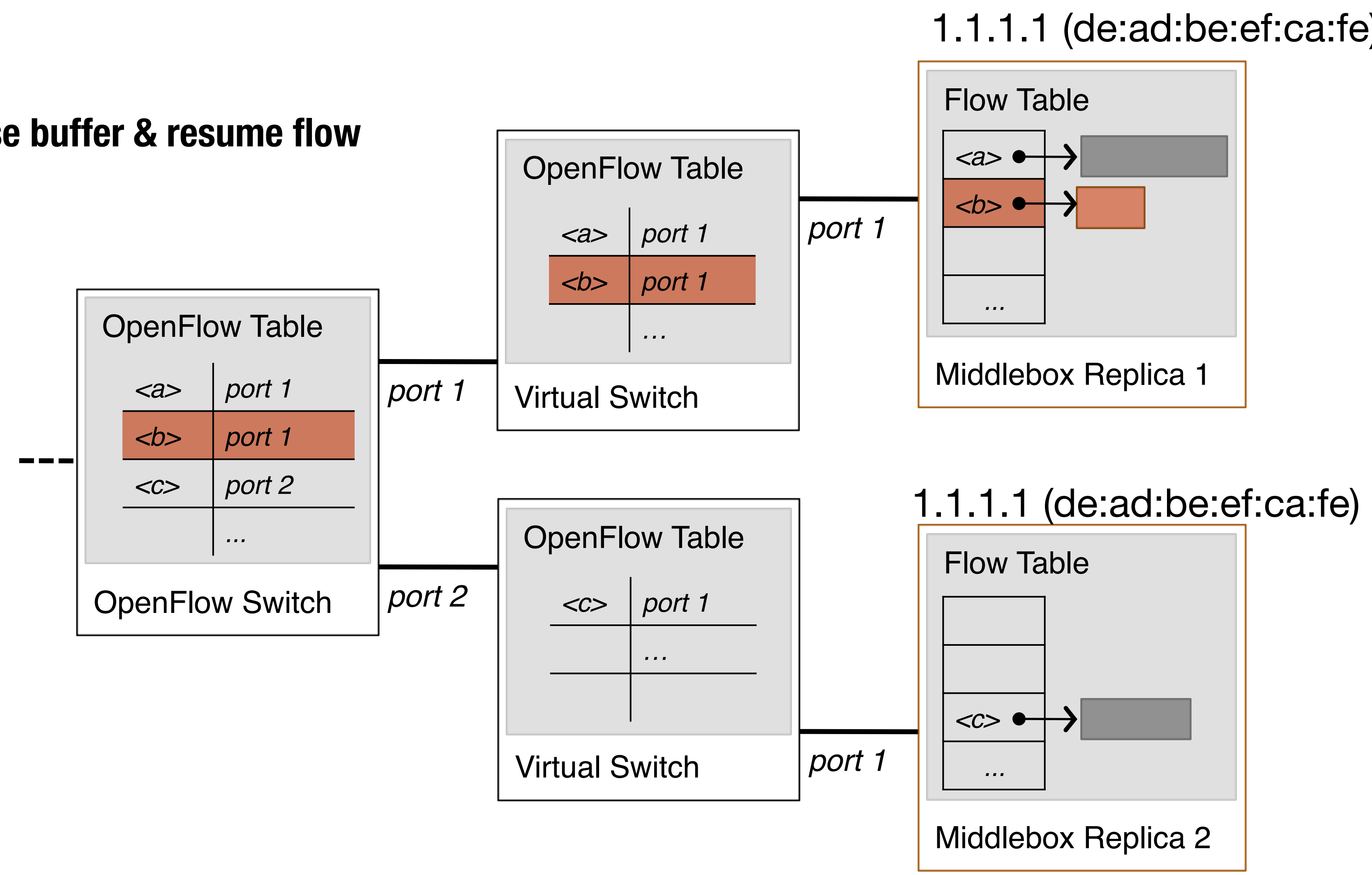
Flow Migration

Move flow state to target



Flow Migration

Release buffer & resume flow



Managing Coherent State

```
create_shared(key, size, cb)
delete_shared(key)

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

Managing Coherent State

Strong Consistency

Distributed lock for every update

```
create_shared("foo", 4, NULL)
while (1)
        process_packet()
        p_foo  = get_shared("foo", synch)
        val = (*p_foo)++
        put_shared("foo", synch)

        if (val > threshold)
              bar()
```

Middlebox applications rarely need strong consistency!

Managing Coherent State

Eventual Consistency

Hi frequency local updates

Periodic global updates

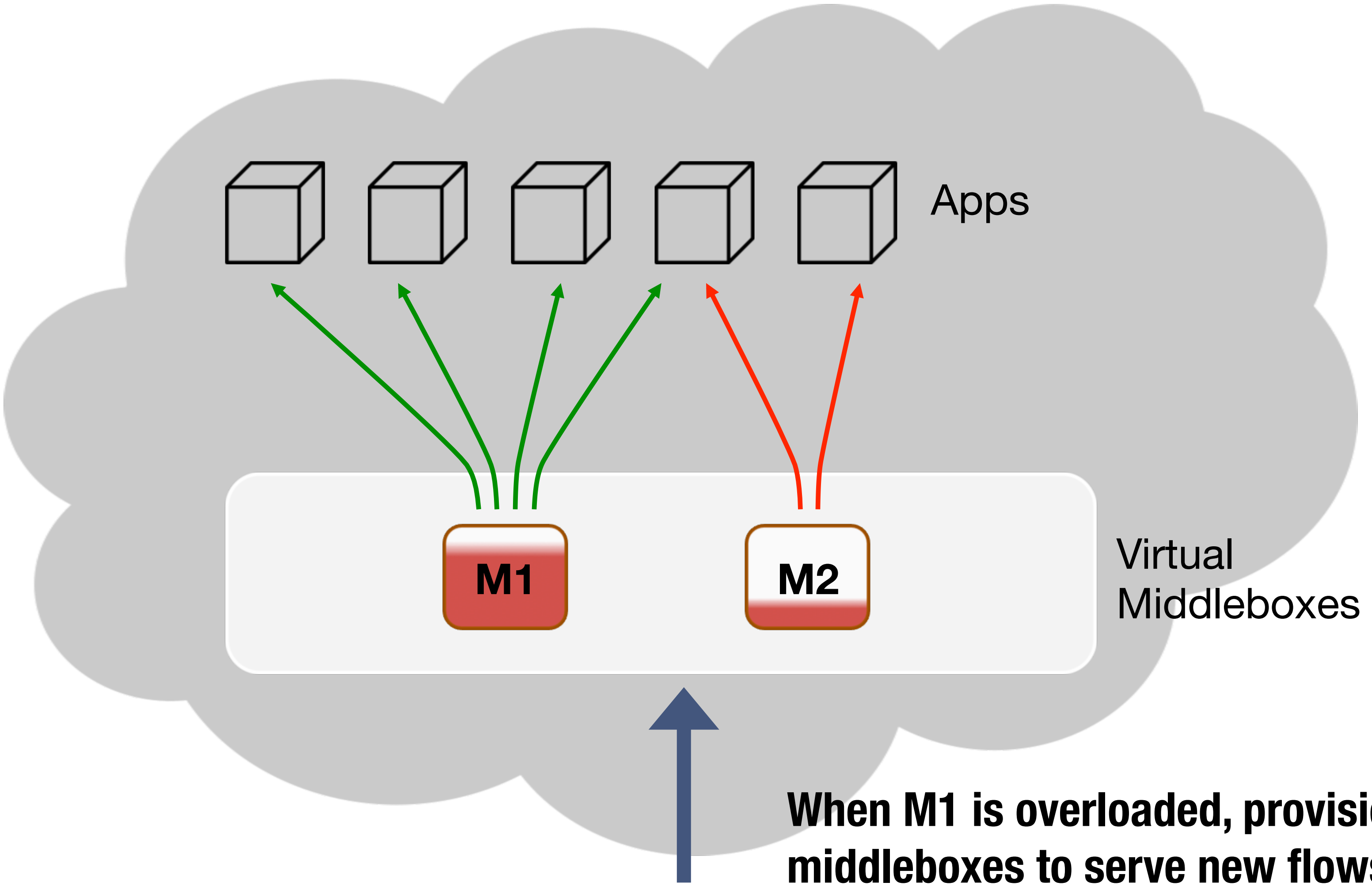
```
create_shared("foo", 4, merge_fn)
while (1)
     process_packet()
     p_foo = get_shared("foo", local)
     val = (*p_foo)++
     put_shared("foo", local)

     if (val > threshold)
        bar()
        put_shared("foo", push)
```

Evaluation

Evaluation Overview

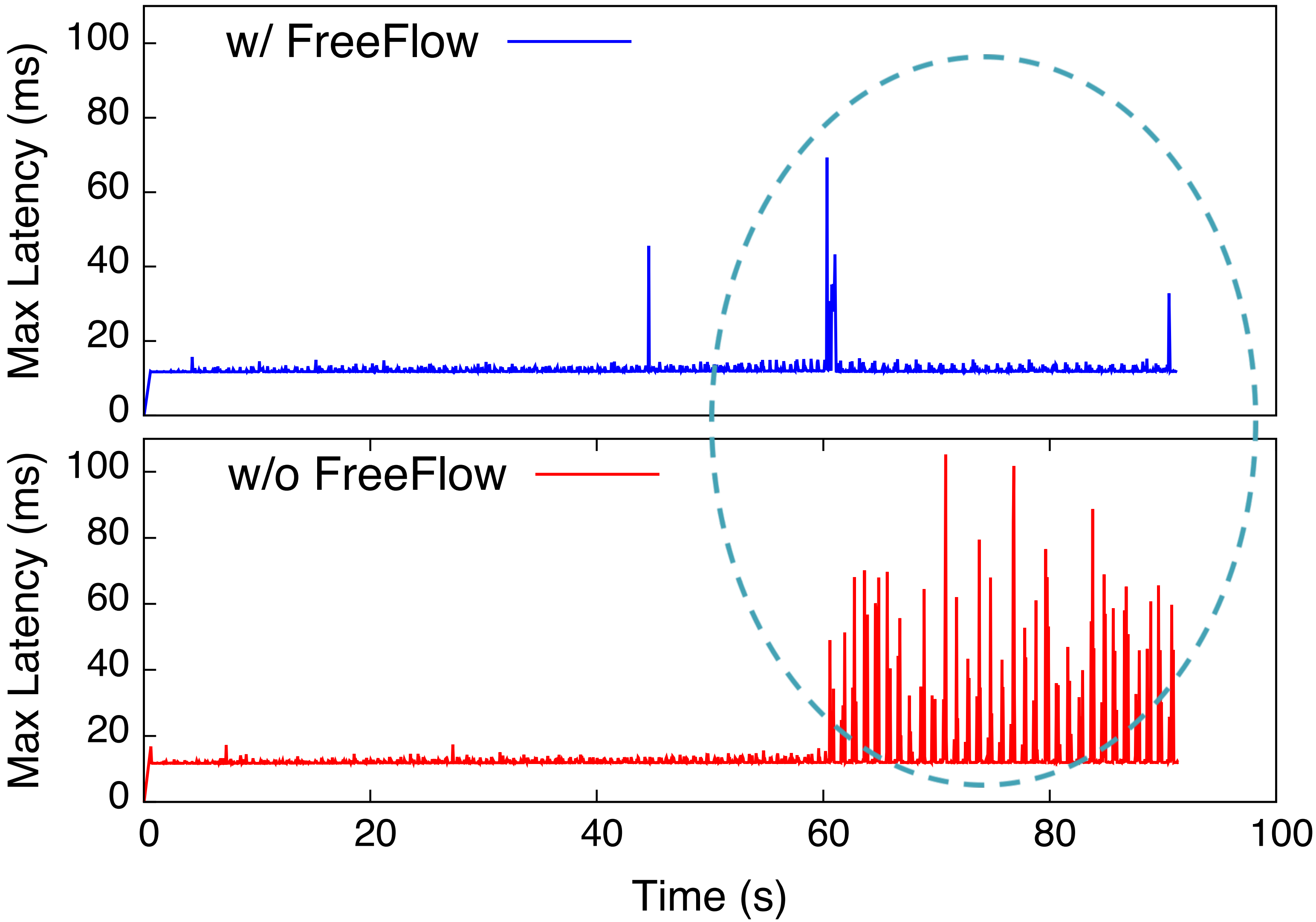
- Eliminating hotspots during scale-out
- Fast and efficient scale-in
- Split/Merge Bro during a load burst

Hotspots Cannot be Alleviated Quickly

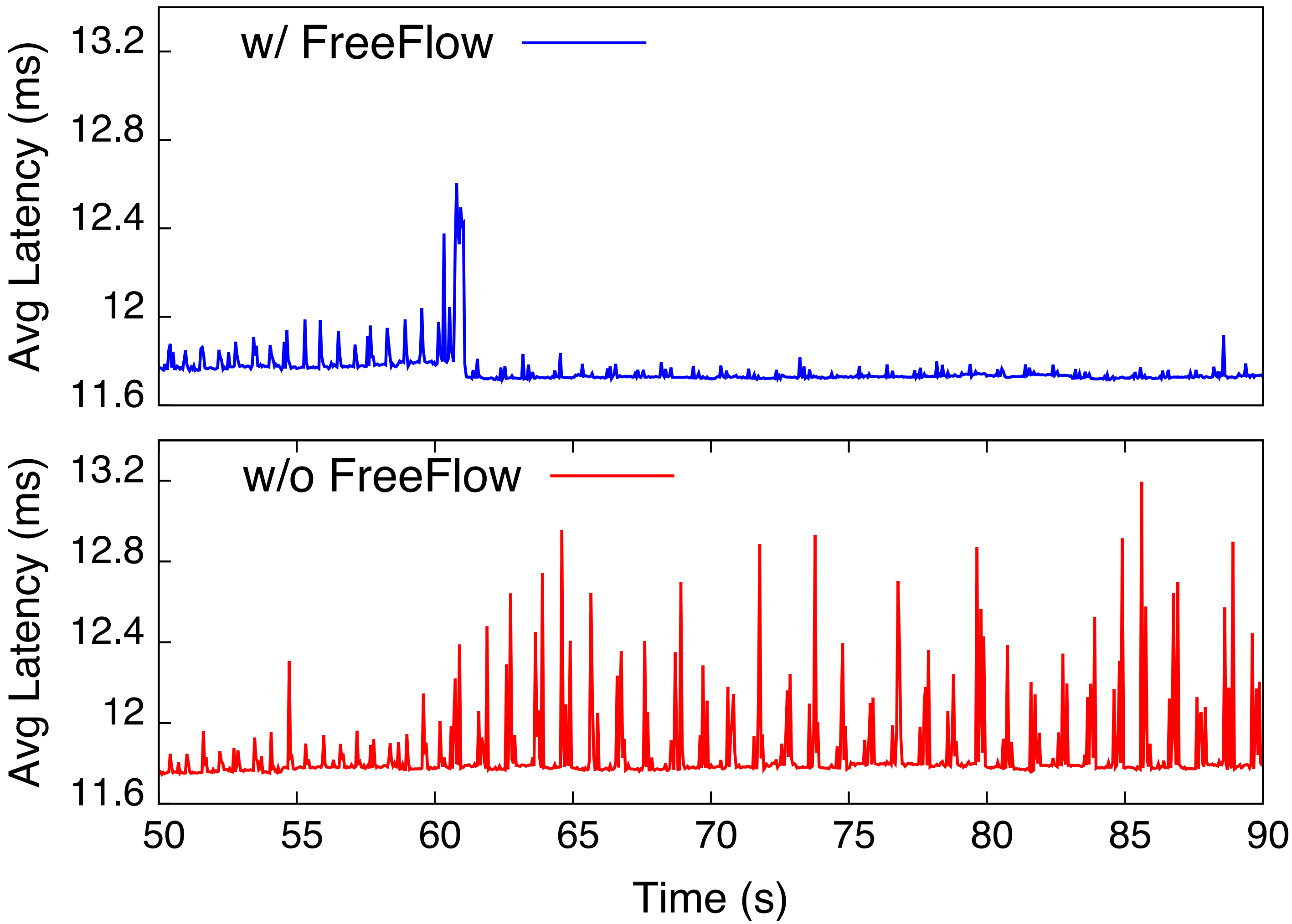


When M1 is overloaded, provision more middleboxes to serve new flows

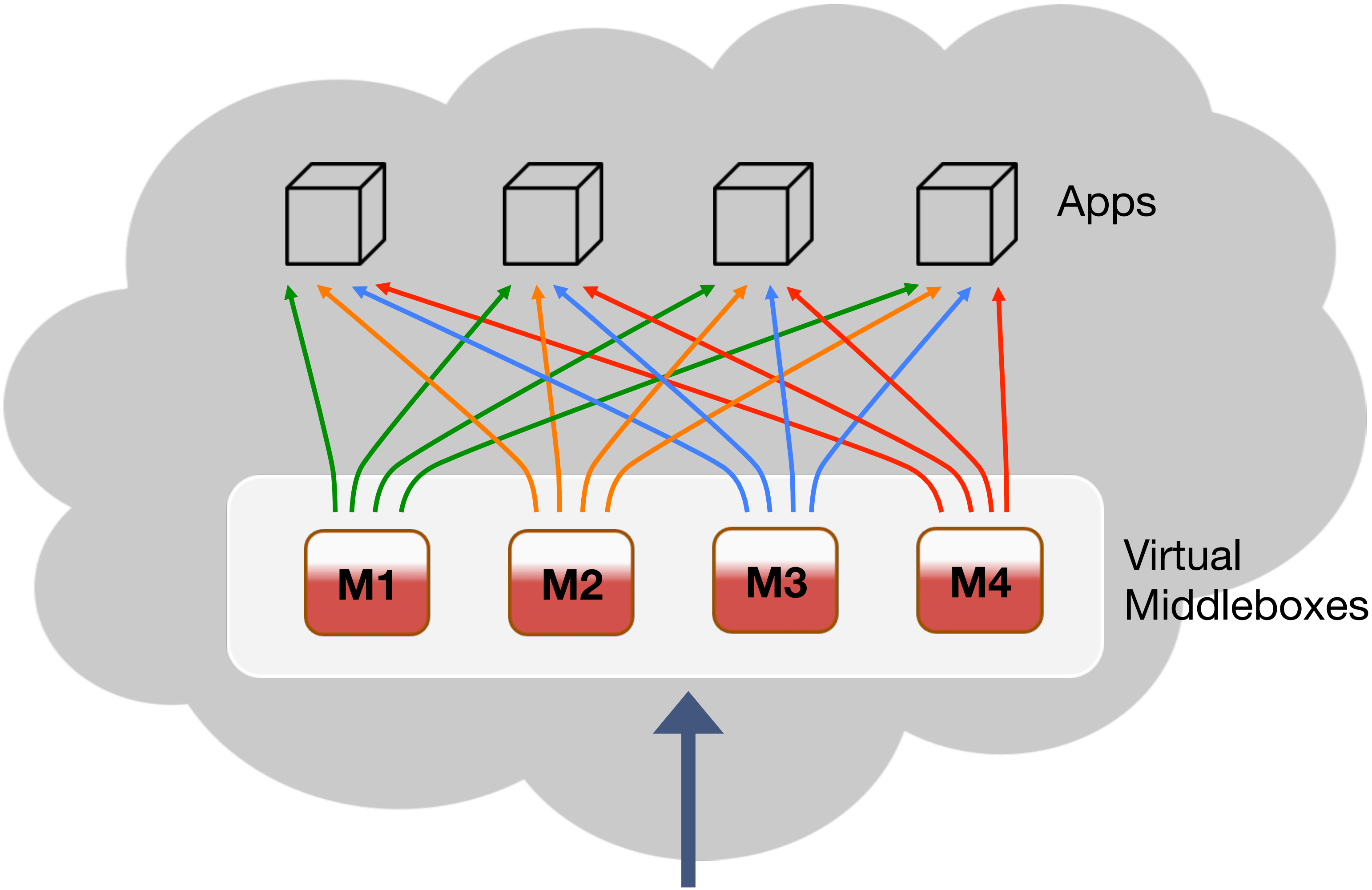
Eliminating Hotspots by Shedding Load



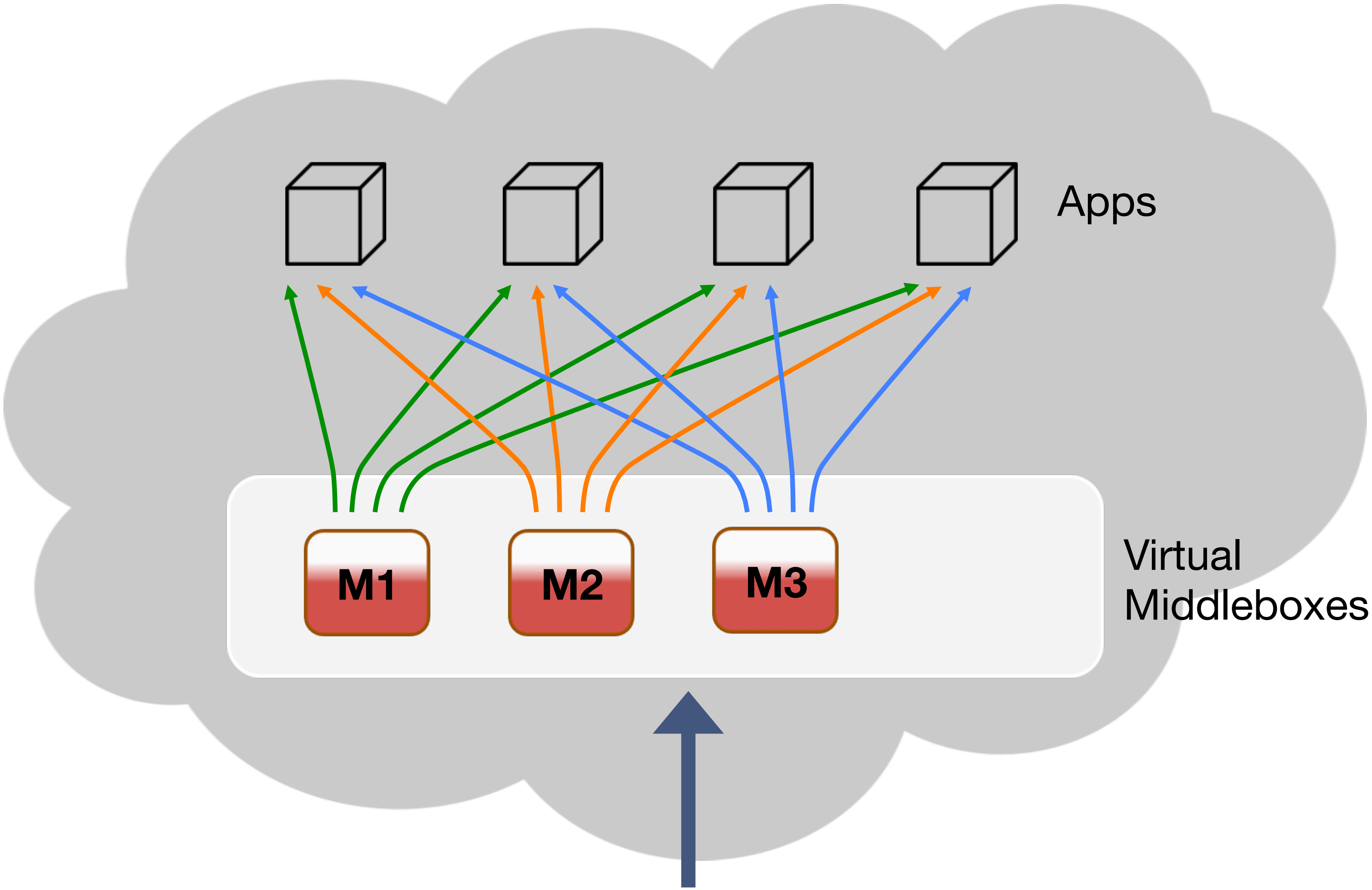
Eliminating Hotspots by Shedding Load



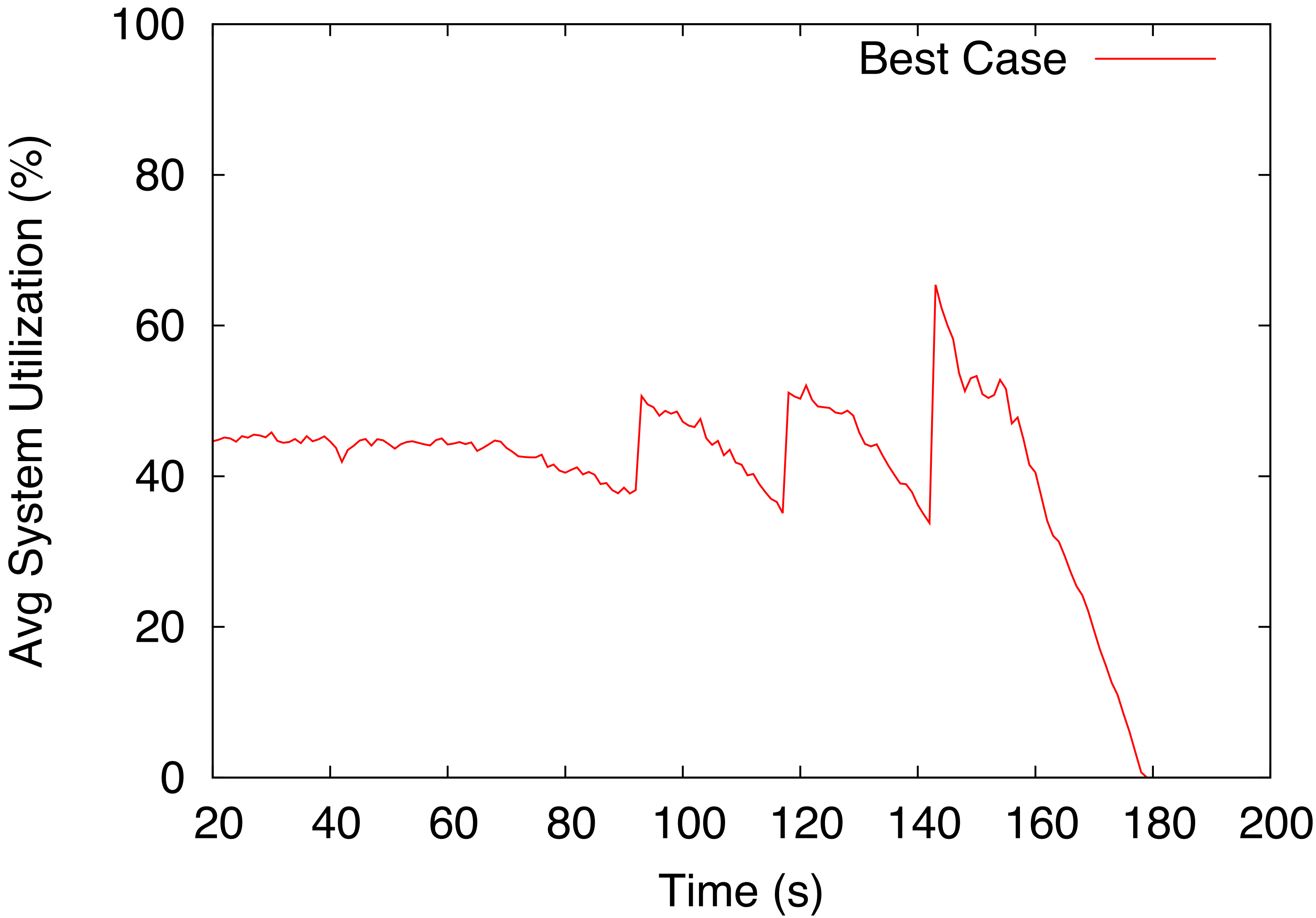
Scaling-In a Deployment : Best Case Scenario



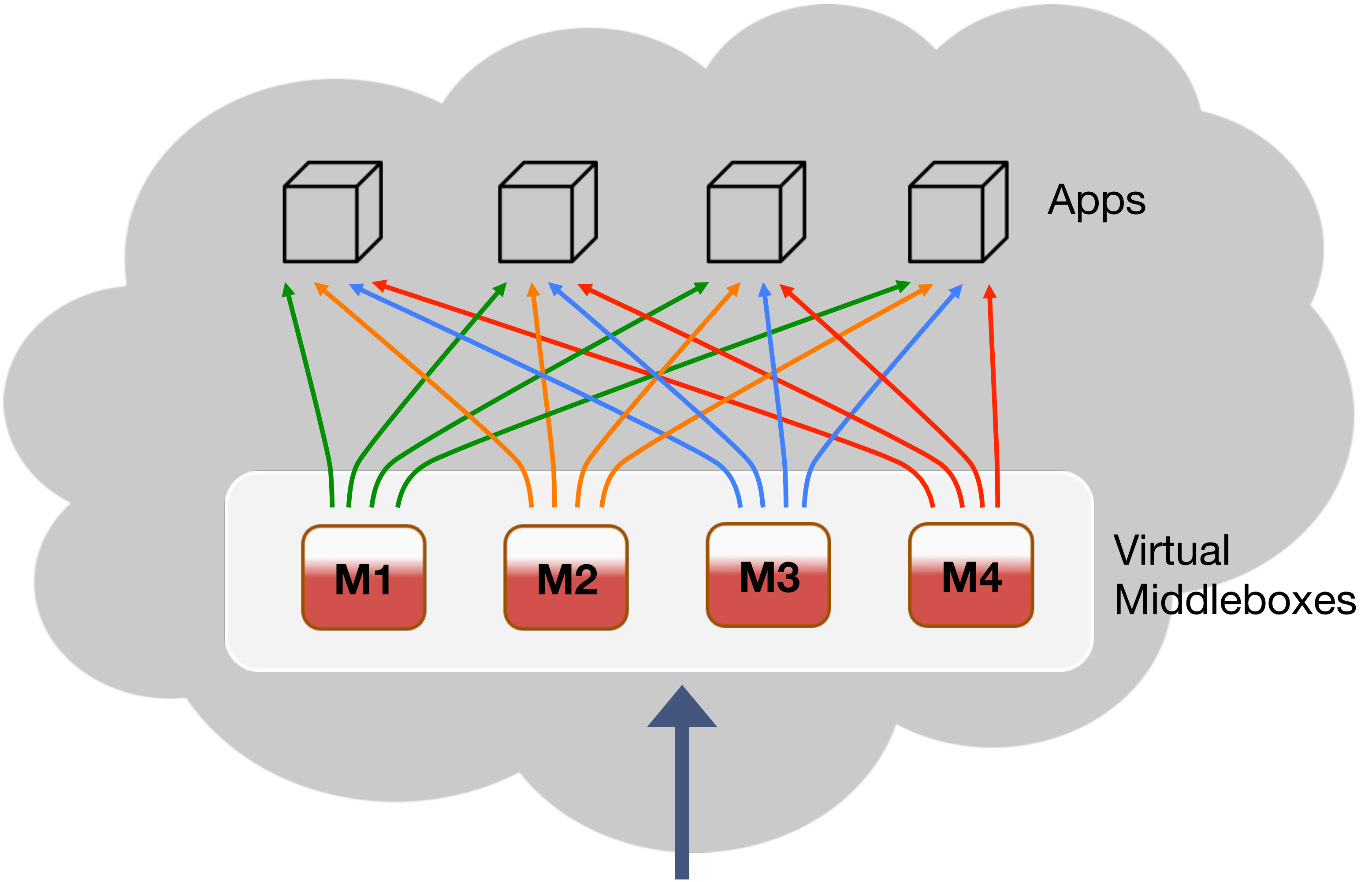
Scaling-In a Deployment : Best Case Scenario



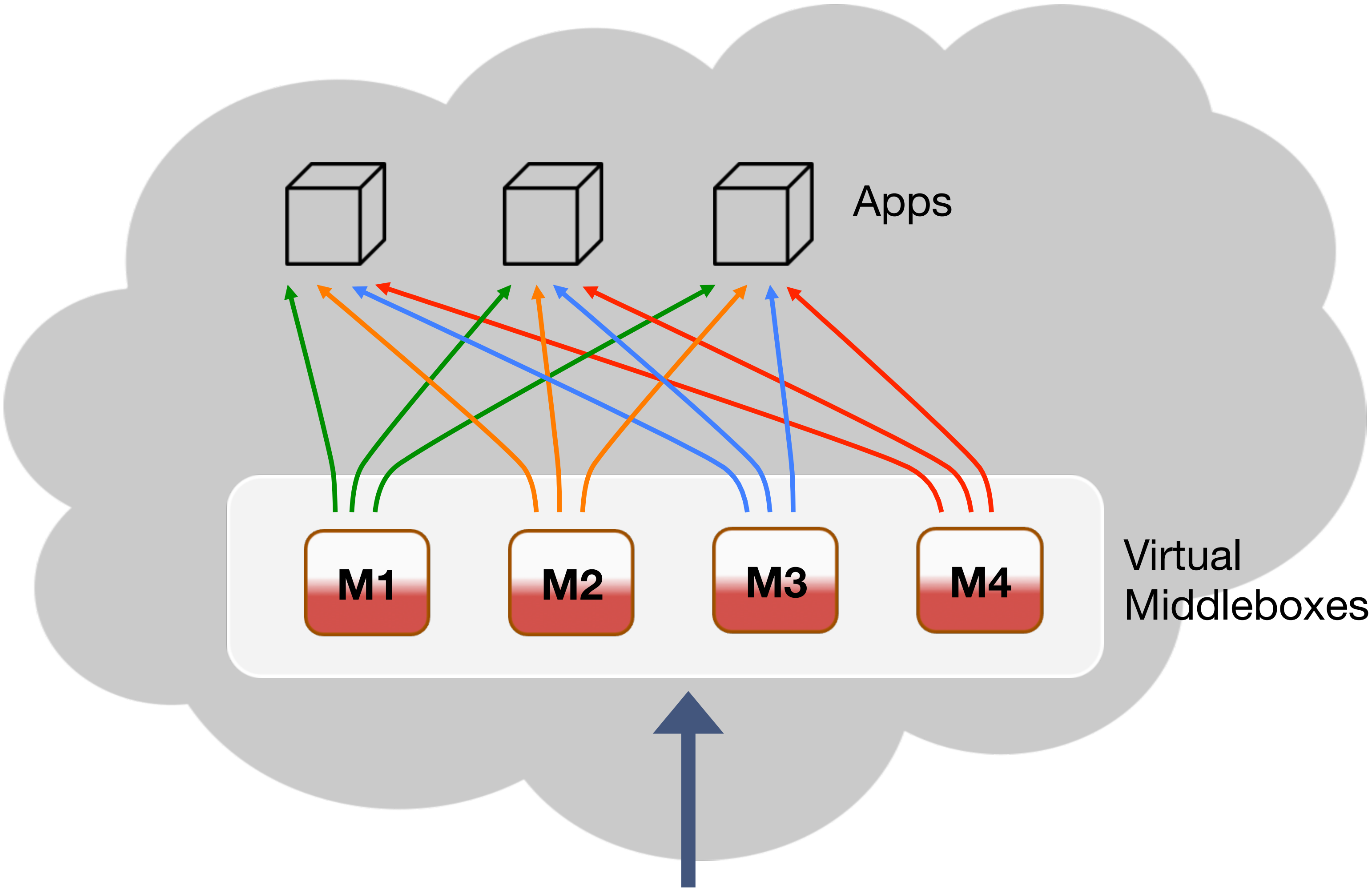
Scaling-In: Best Case Scenario



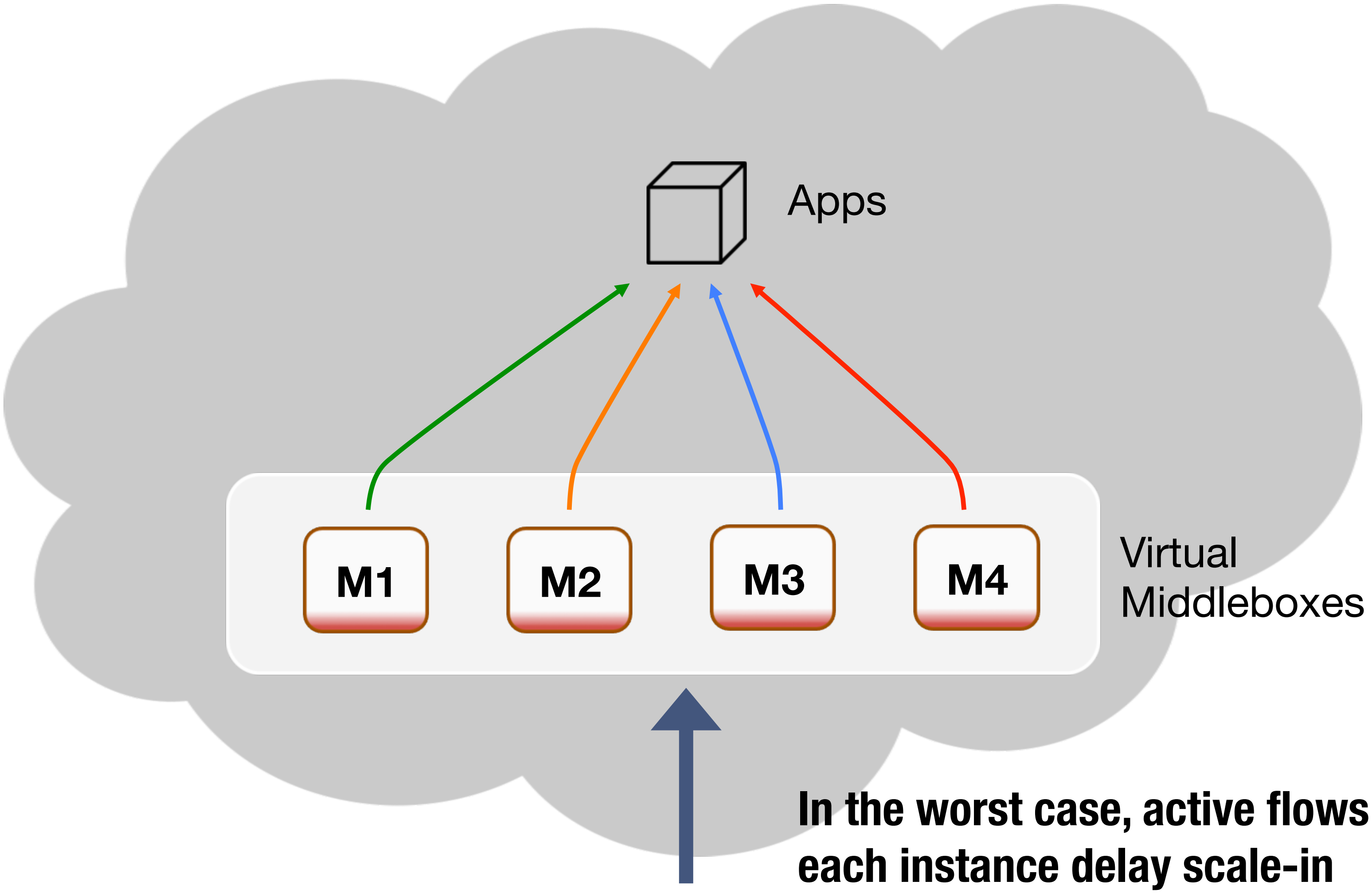
Scaling-In using *kill* : Worst Case Scenario



Scaling-In using *kill* : Worst Case Scenario

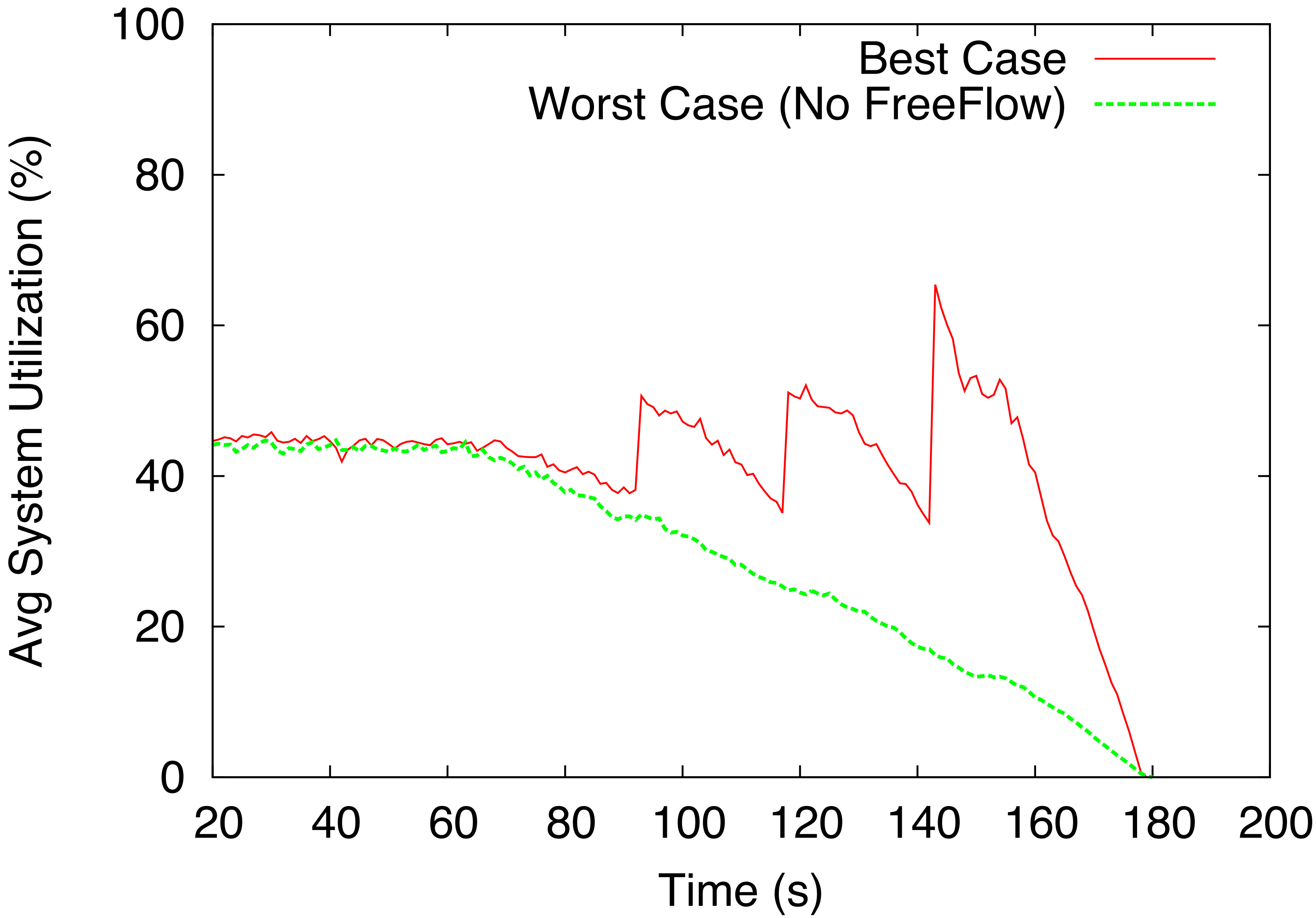


Scaling-In using *kill* : Worst Case Scenario

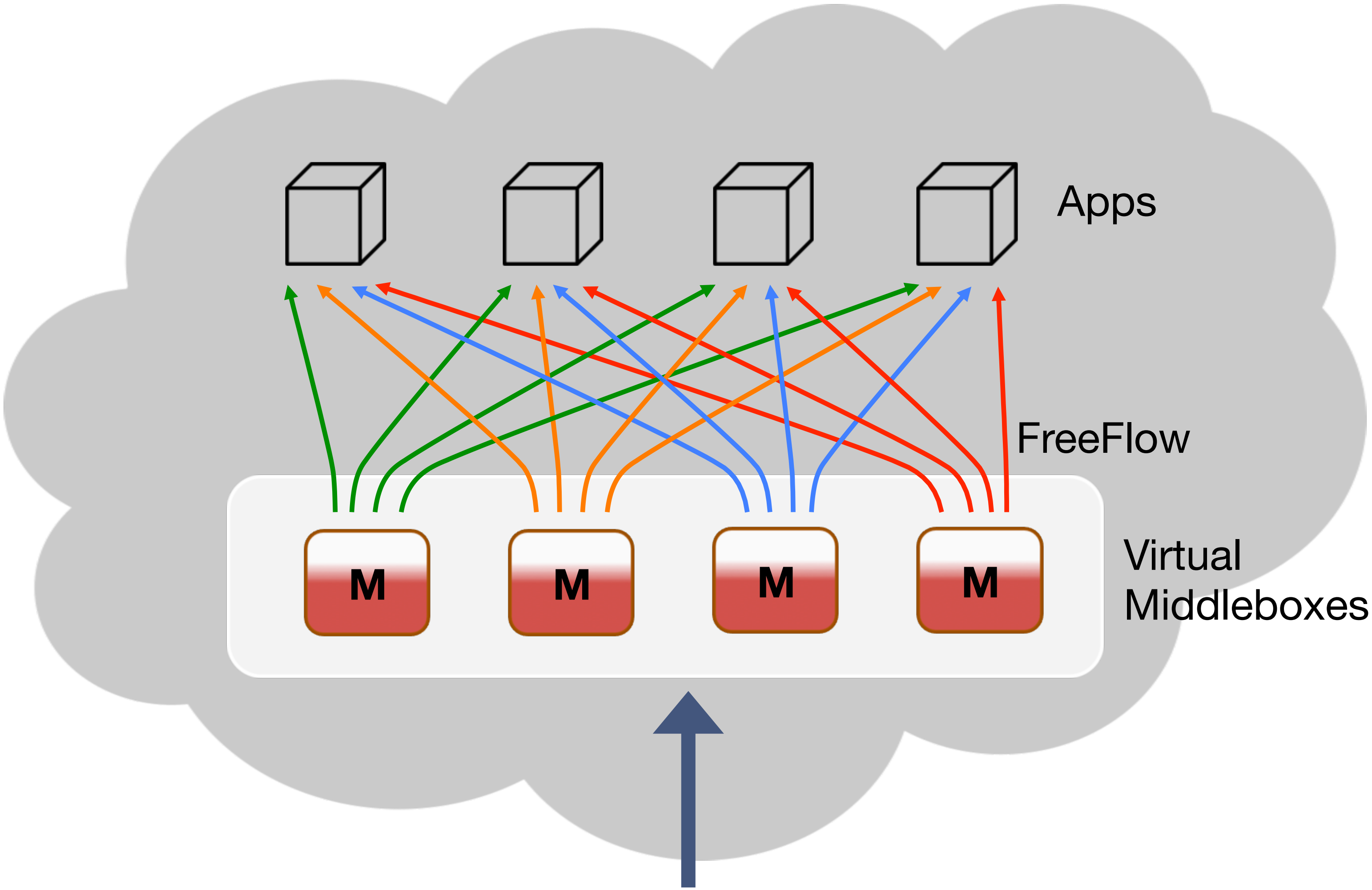


In the worst case, active flows in each instance delay scale-in

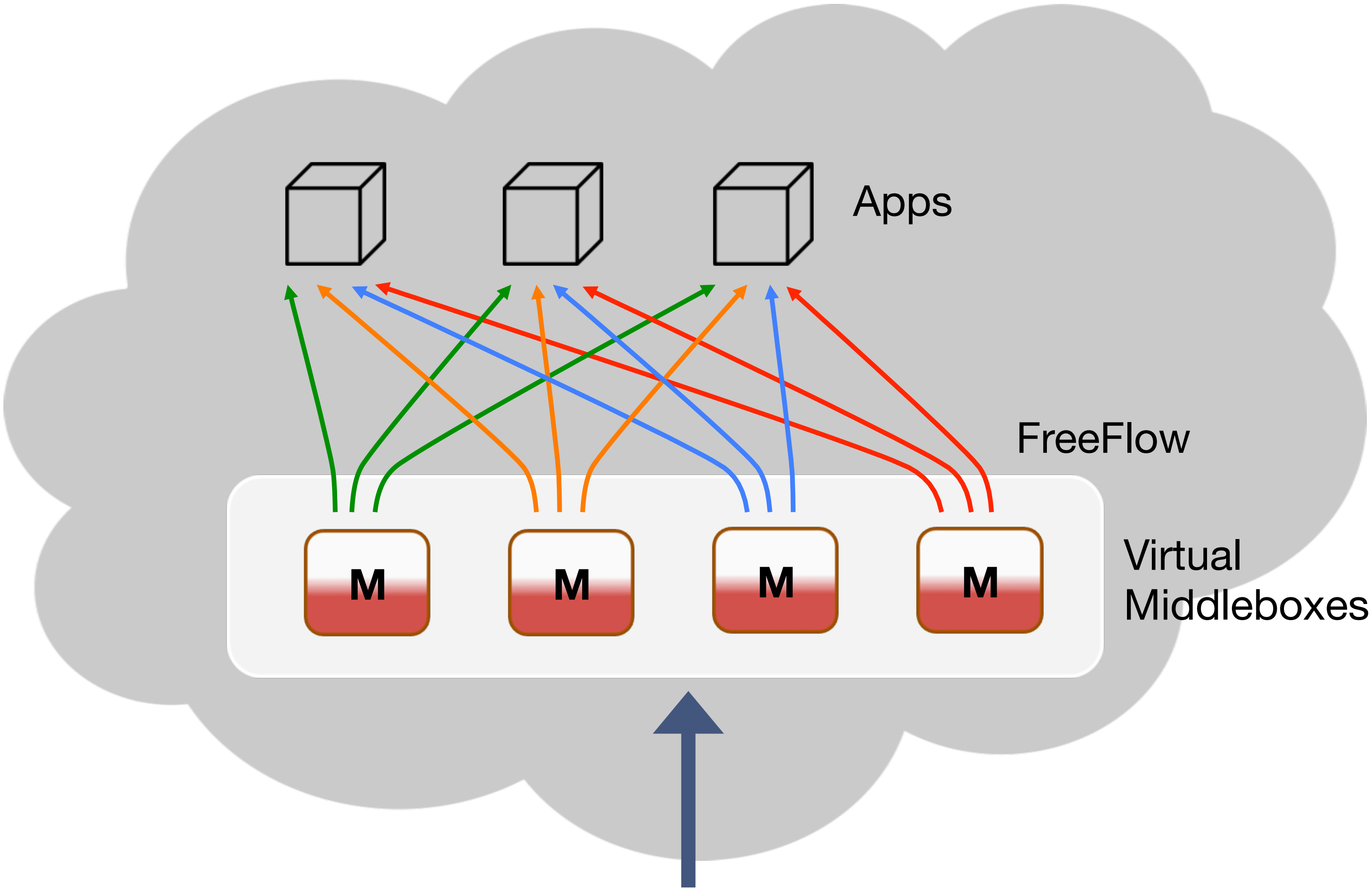
Scaling-In using *kill* : Slow & Inefficient



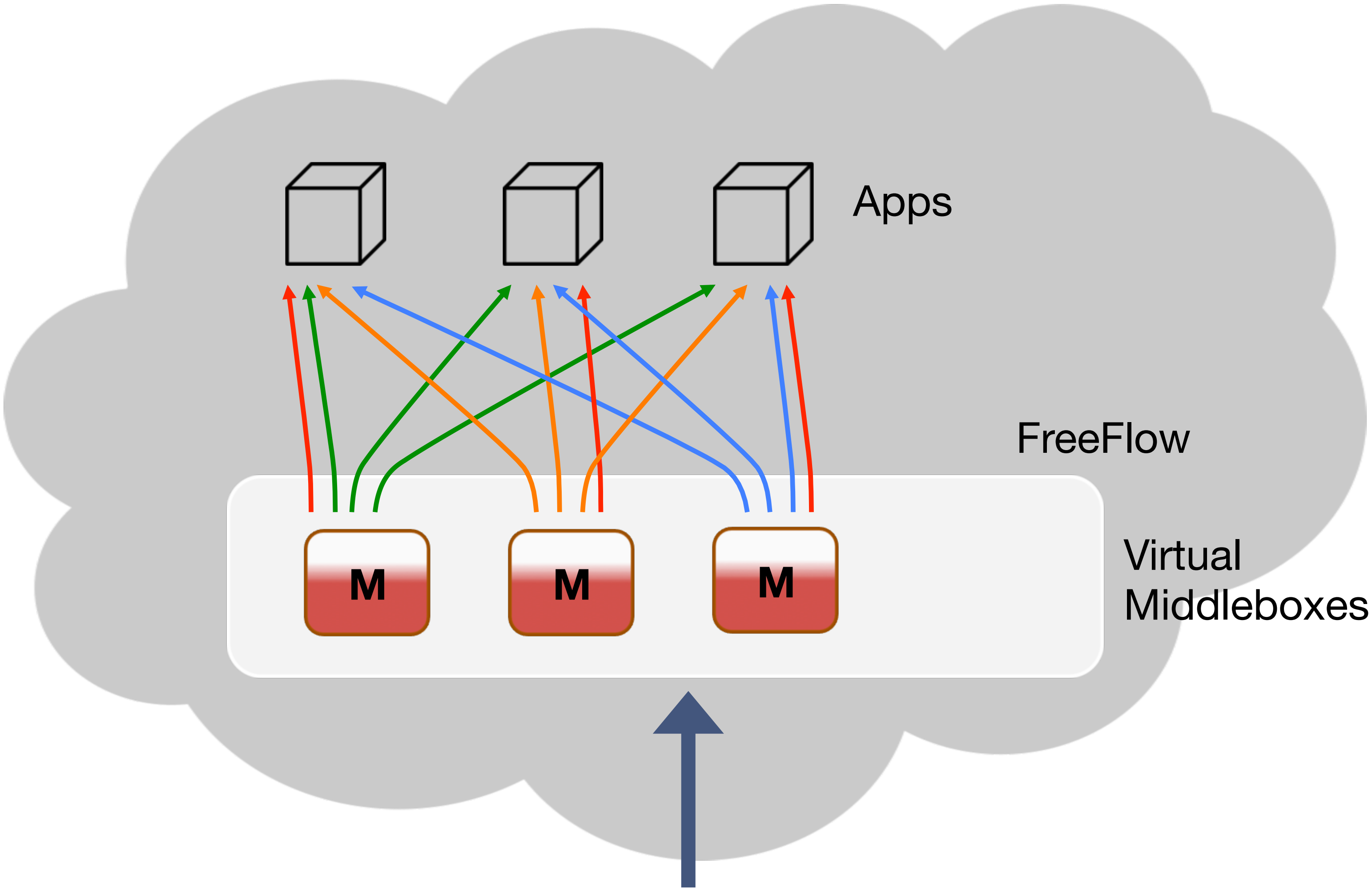
Scaling-In using *merge* : Worst Case Scenario



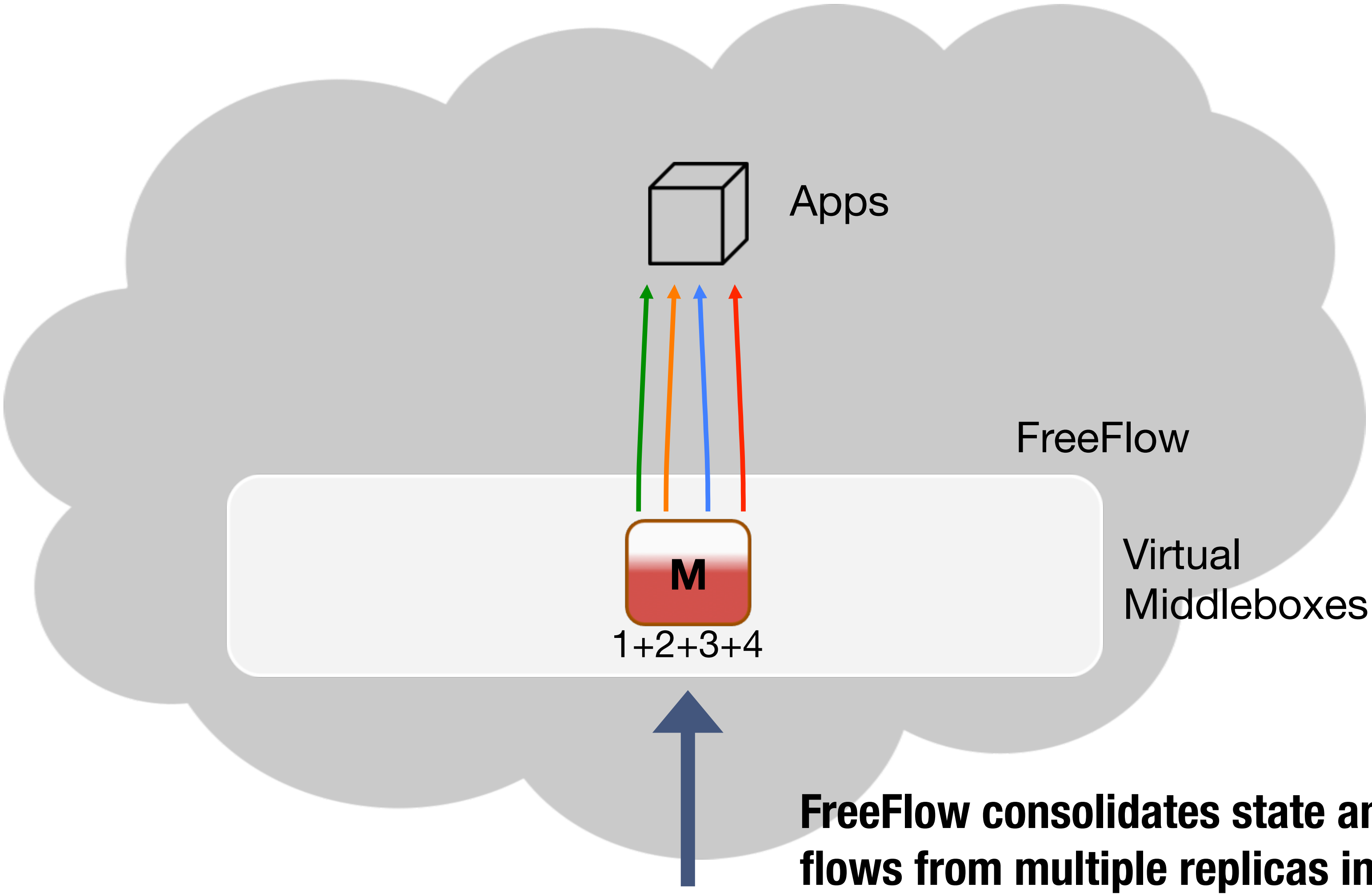
Scaling-In using *merge* : Worst Case Scenario



Scaling-In using *merge* : Worst Case Scenario

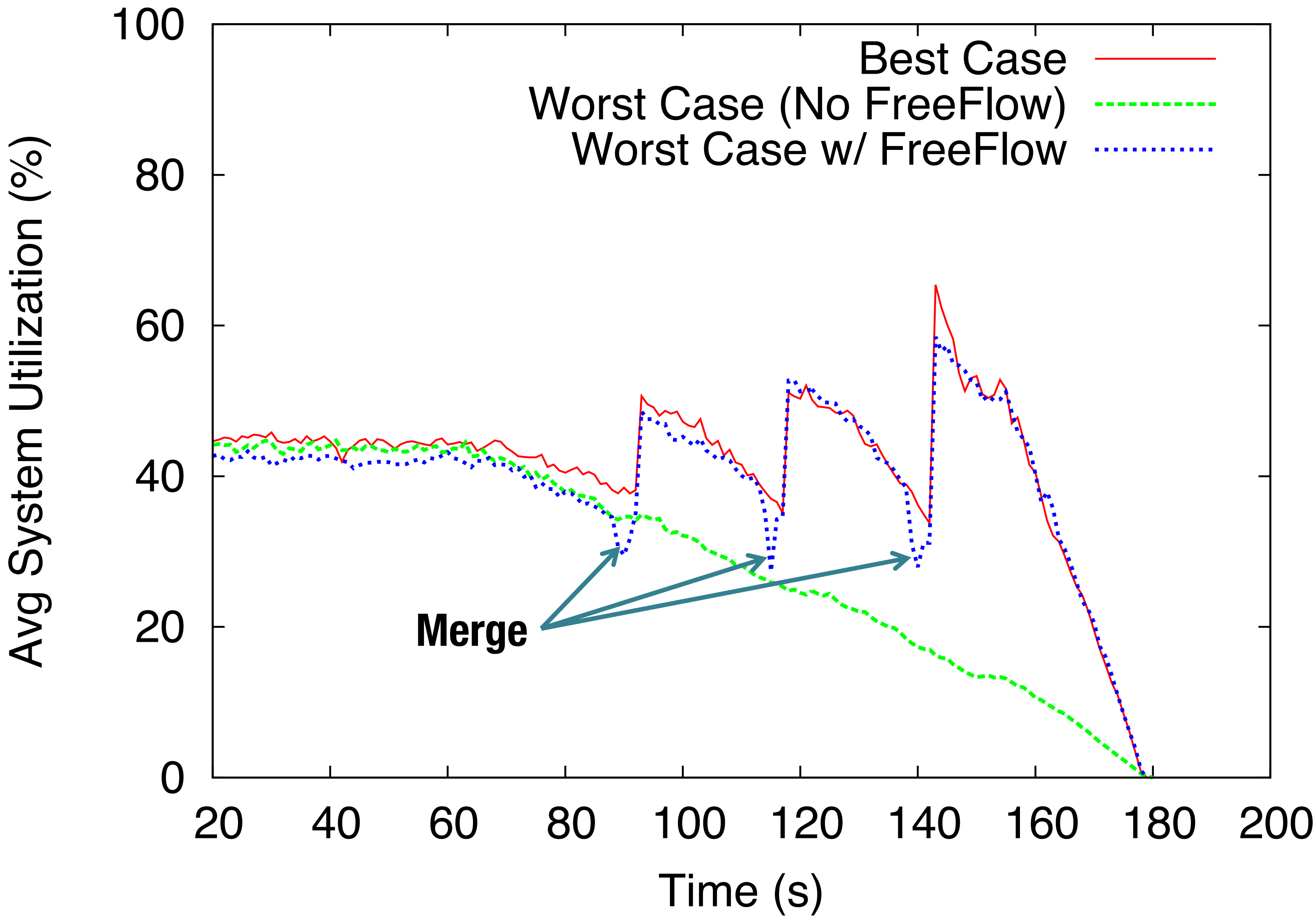


Scaling-In using *merge* : Worst Case Scenario



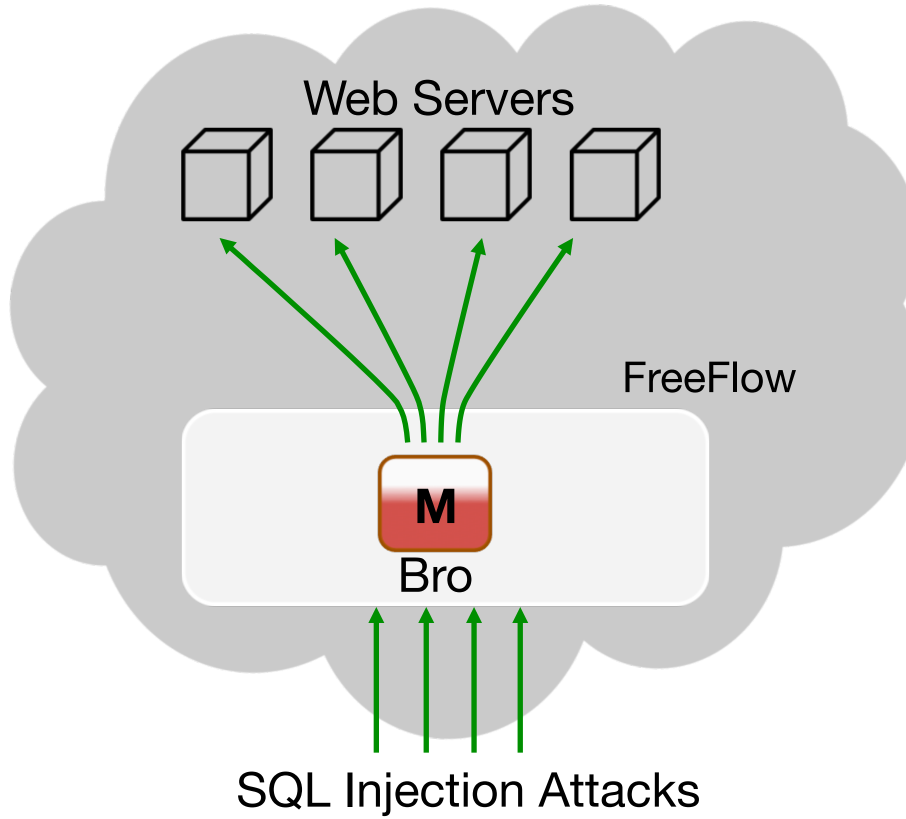
FreeFlow consolidates state and flows from multiple replicas into one

Scaling-In using *merge* : Fast & Efficient

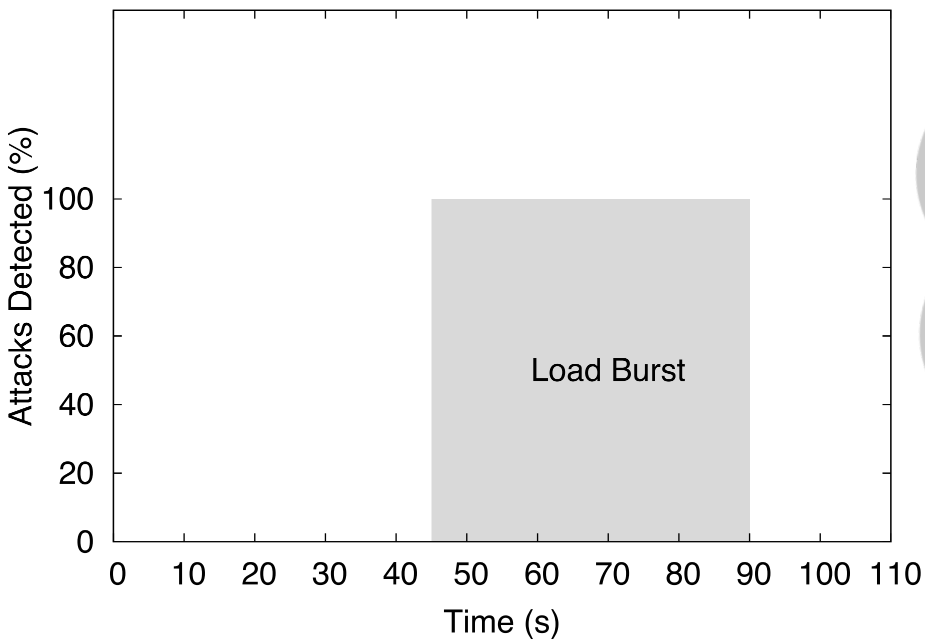


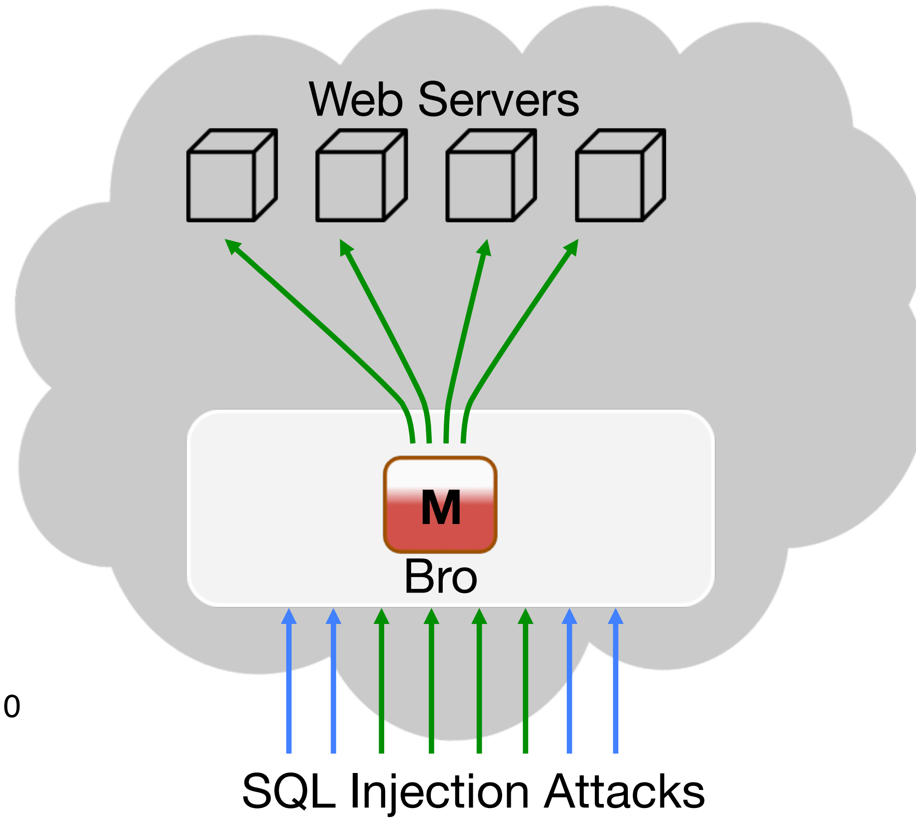
Splitting & Merging Bro IDS

- Ported the Event Engine to FreeFlow
- Support for UDP, TCP/HTTP protocols
- SQL Injection Detection plugin

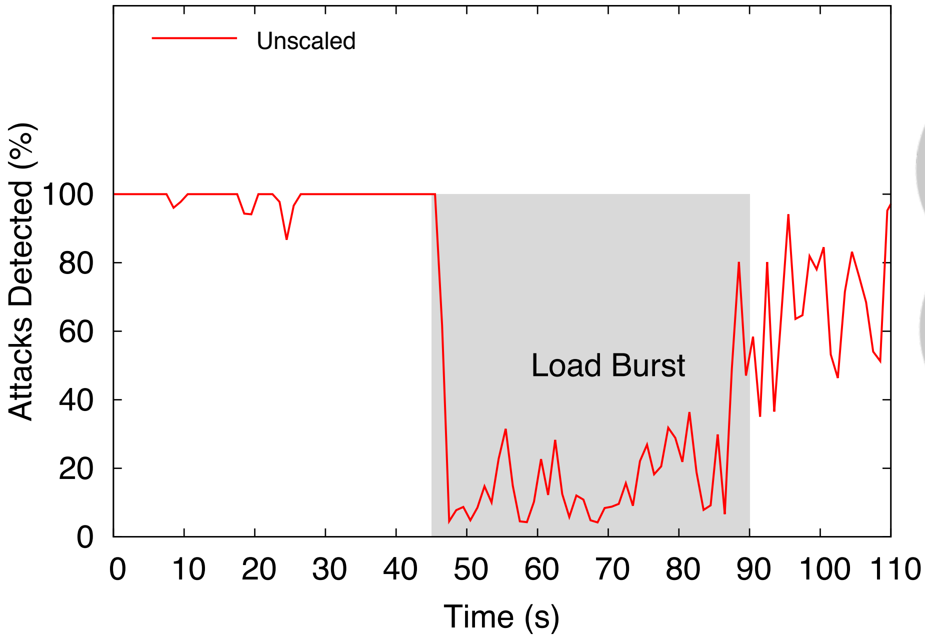


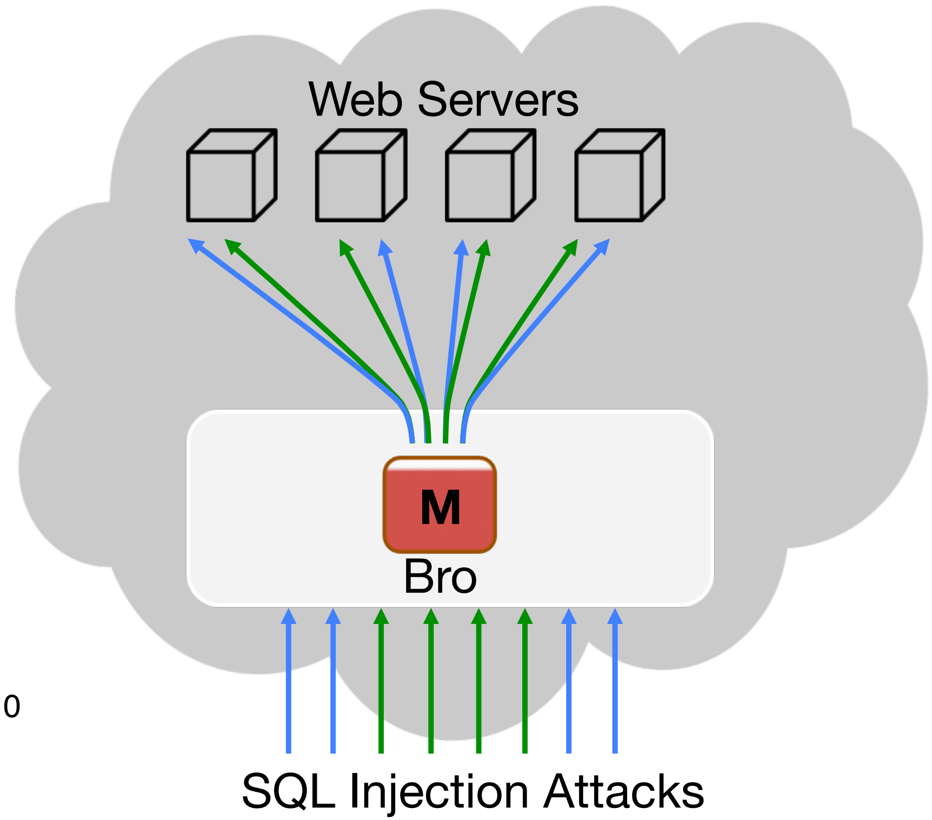
Handling a Load Burst





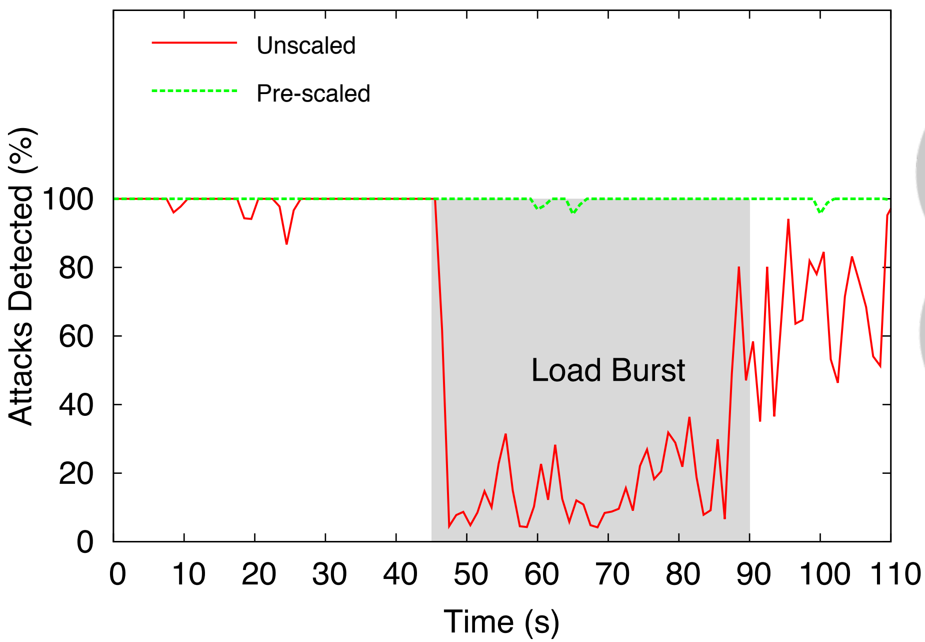
Handling a Load Burst : No Scaling

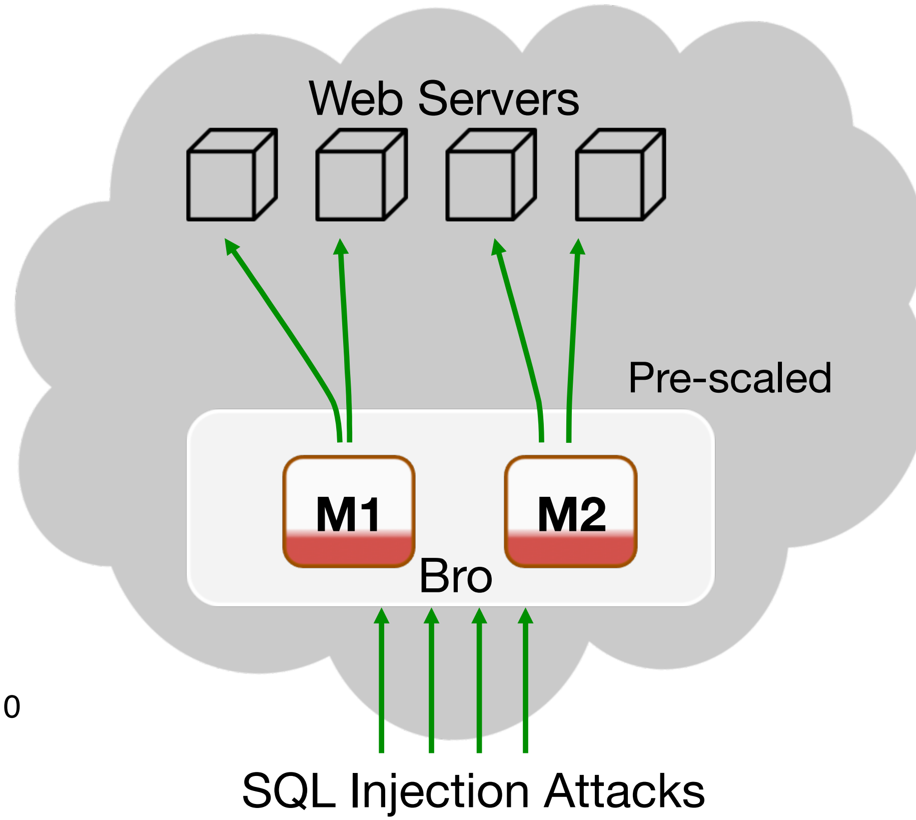




Without enough capacity to handle the load burst, the system performance degrades severely

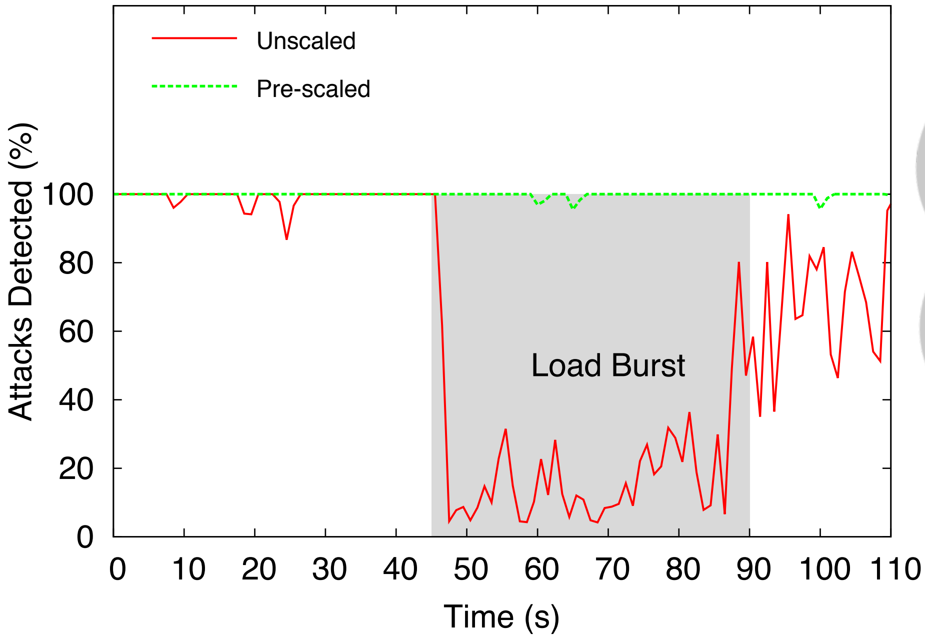
Handling a Load Burst : Pre-Scaled

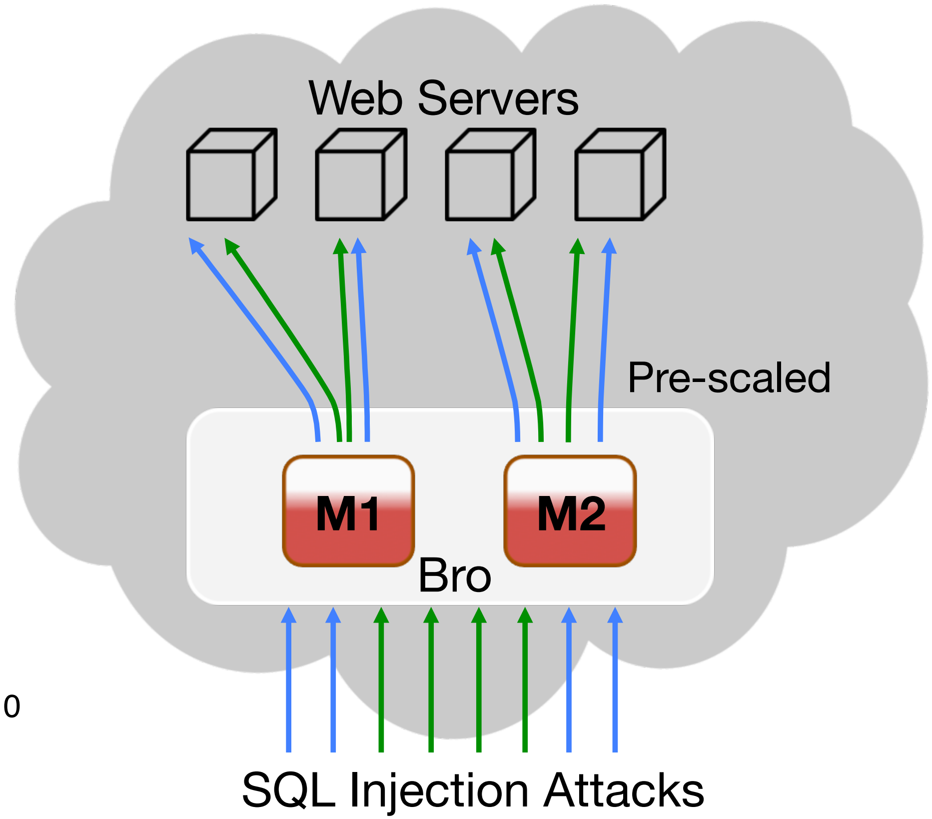




Two instances are provisioned apriori, enough to handle a load burst, if any

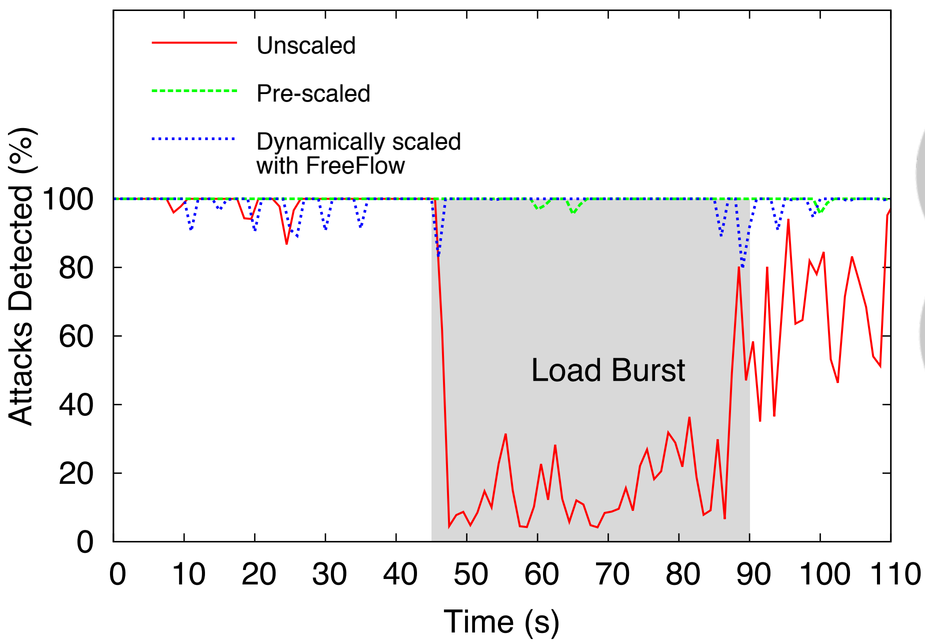
Handling a Load Burst : Pre-Scaled

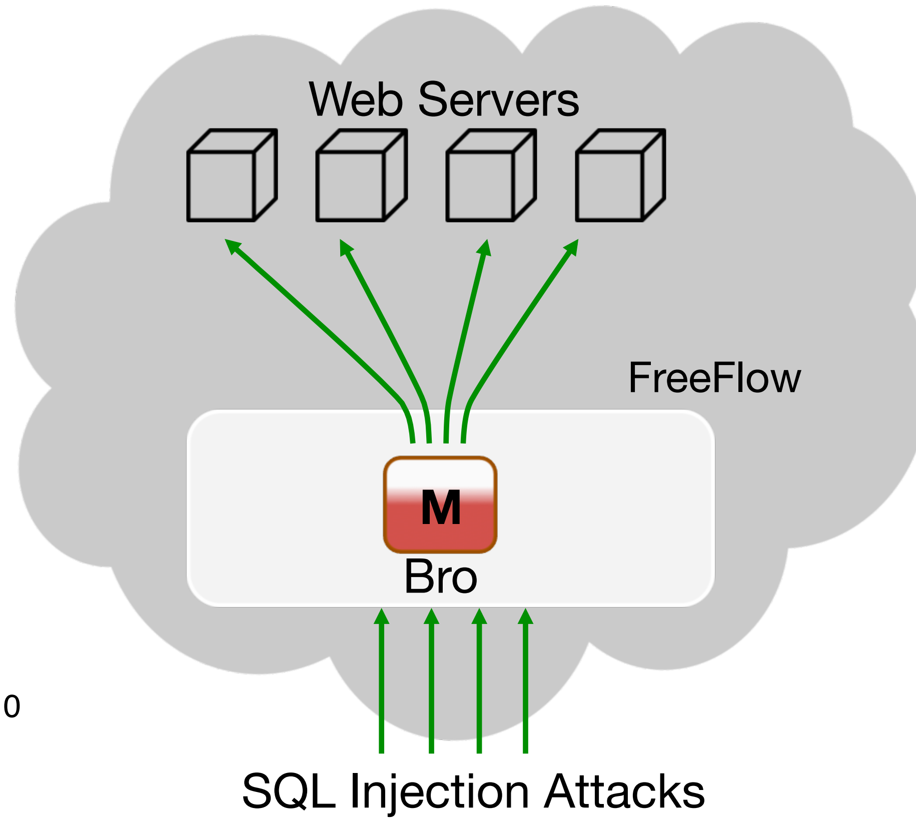




Load burst has no impact on system performance, as there is enough capacity to handle the load

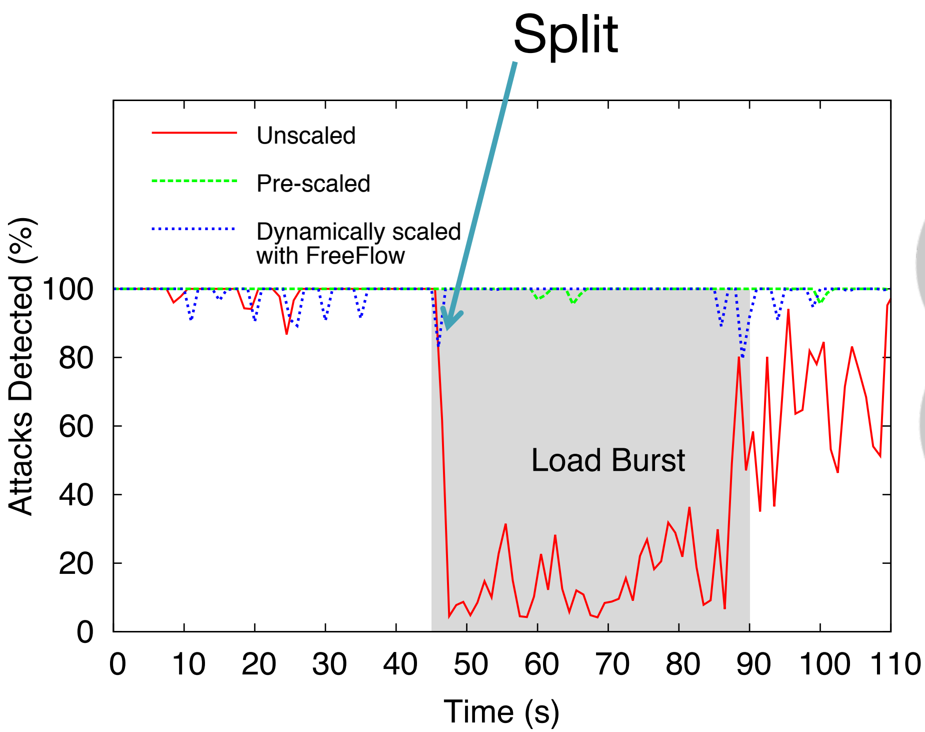
Handling a Load Burst : Split/Merge

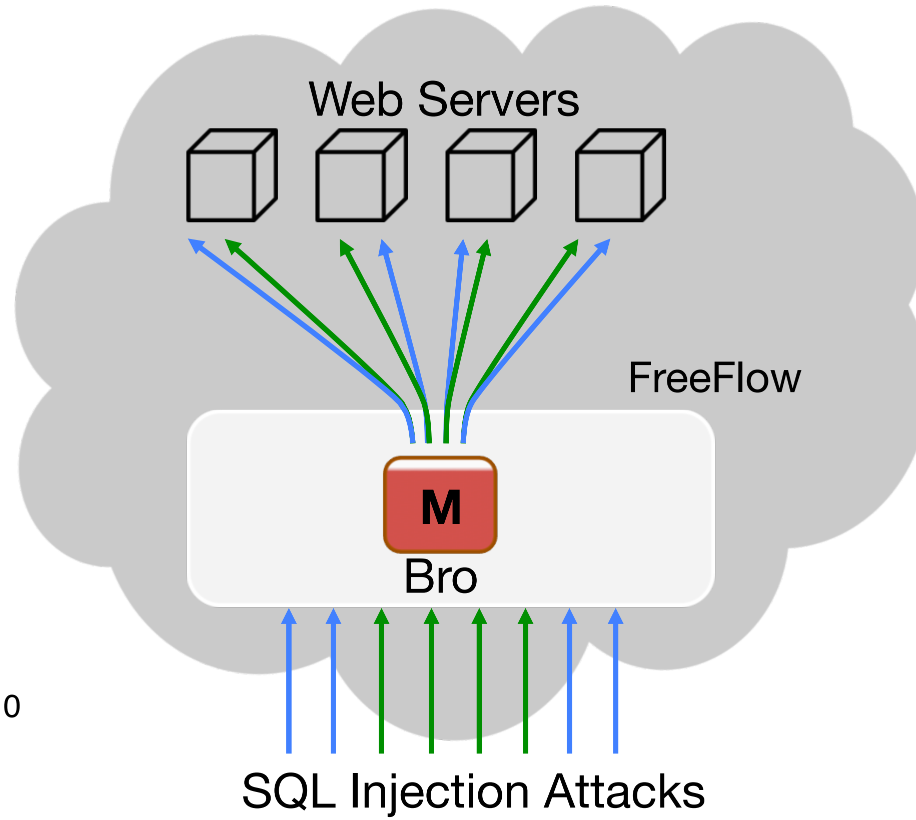




One replica handles the load well, before the load burst

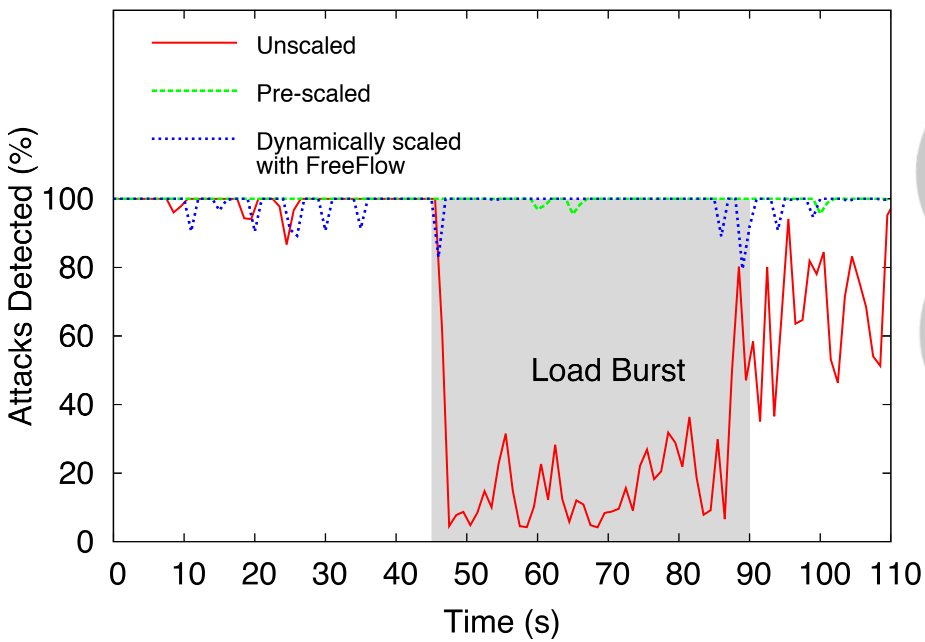
Handling a Load Burst : Split/Merge

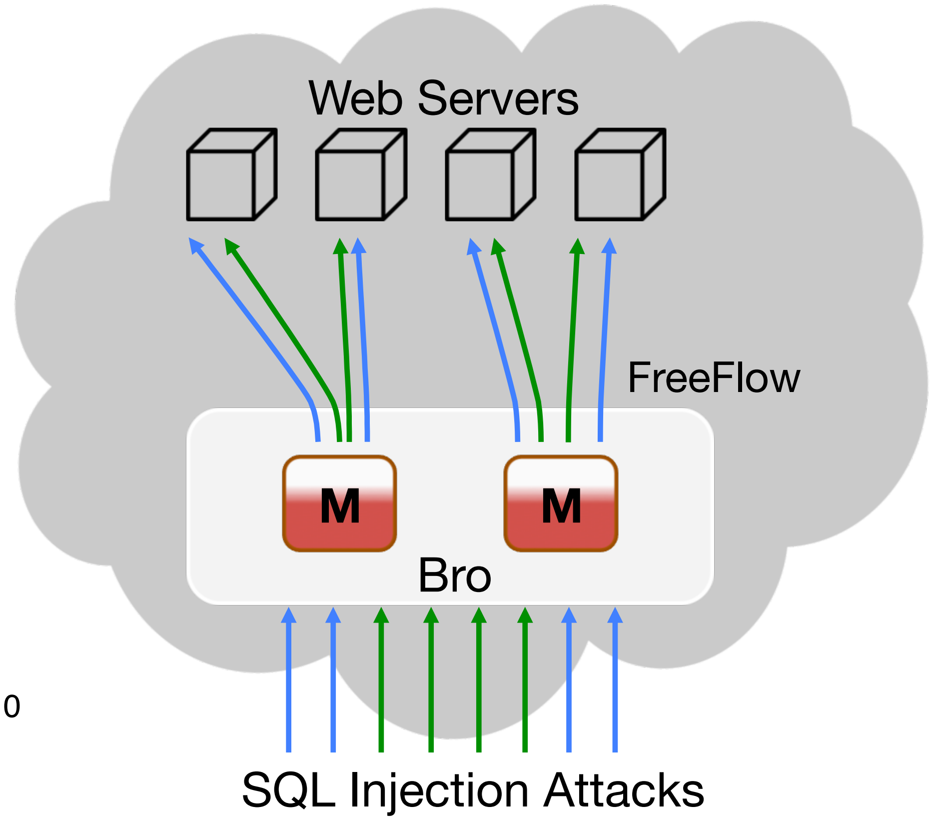




When load burst starts, the Orchestrator *splits* the replica and rebalances the load

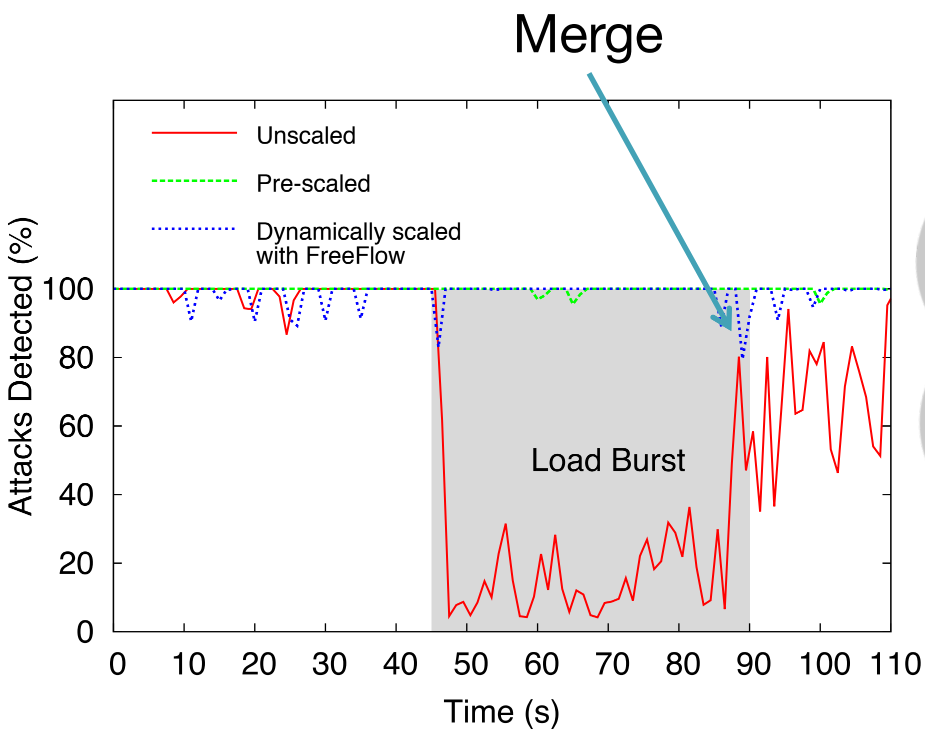
Handling a Load Burst : Split/Merge

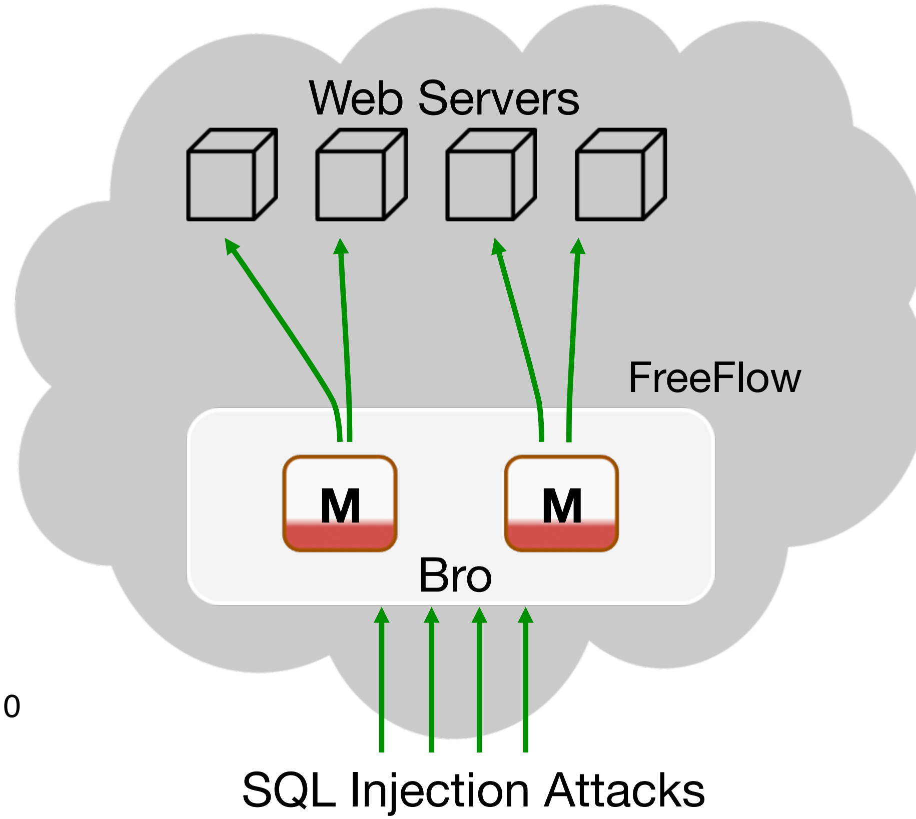




With the load rebalanced,
performance returns to normal

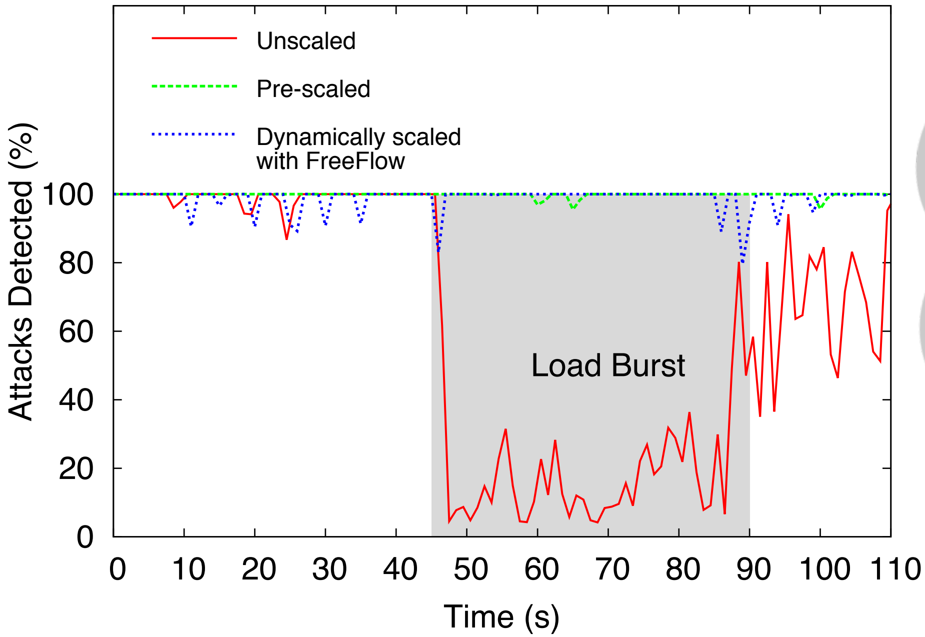
Handling a Load Burst : Split/Merge

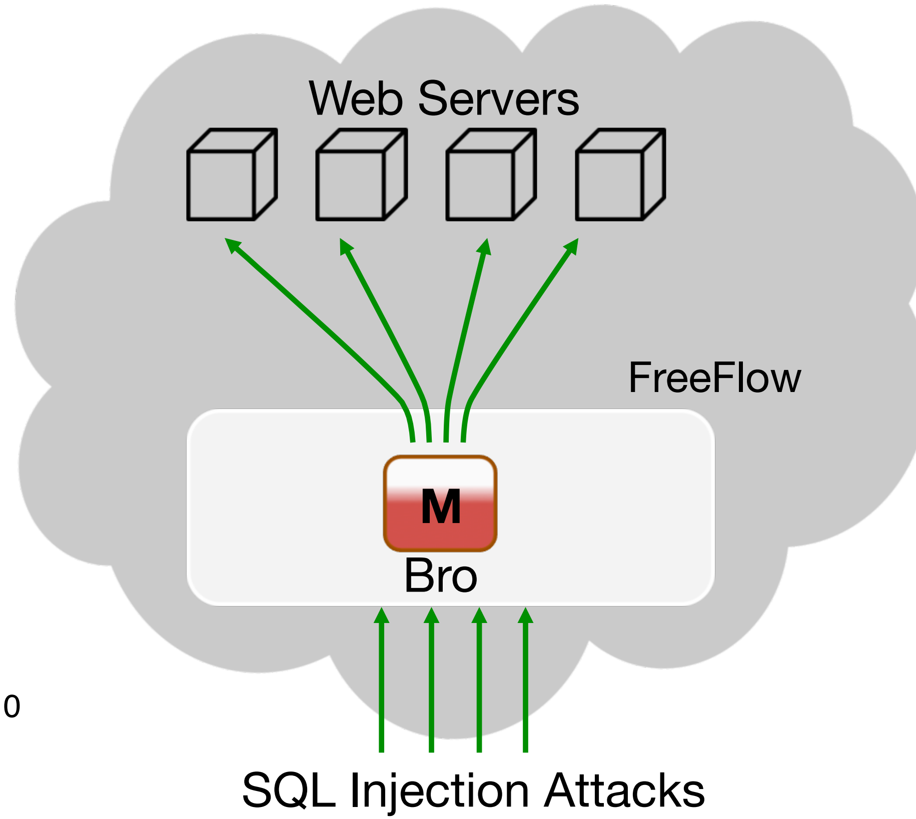




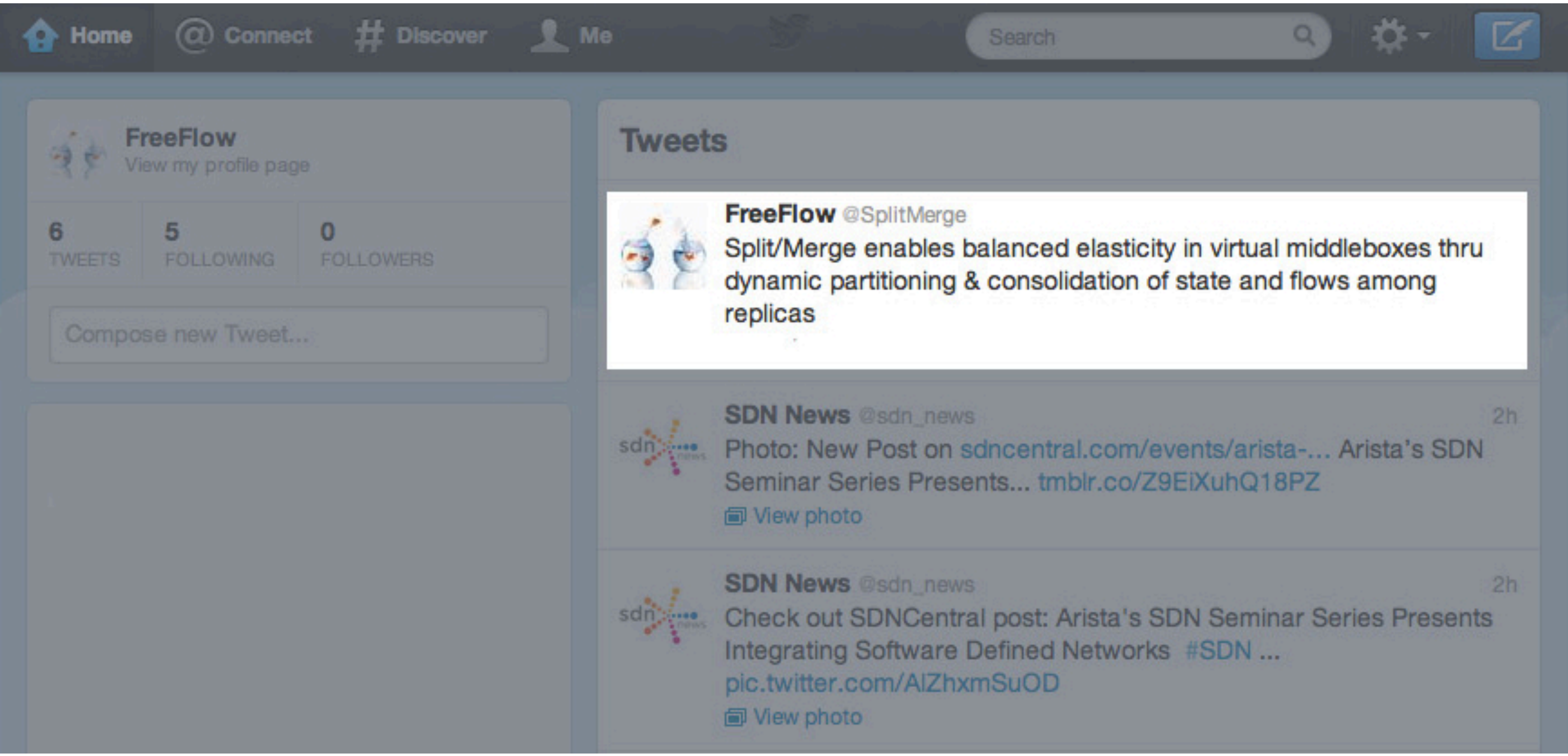
When system utilization drops after the load burst, the Orchestrator *merges* the two replicas

Handling a Load Burst : Split/Merge

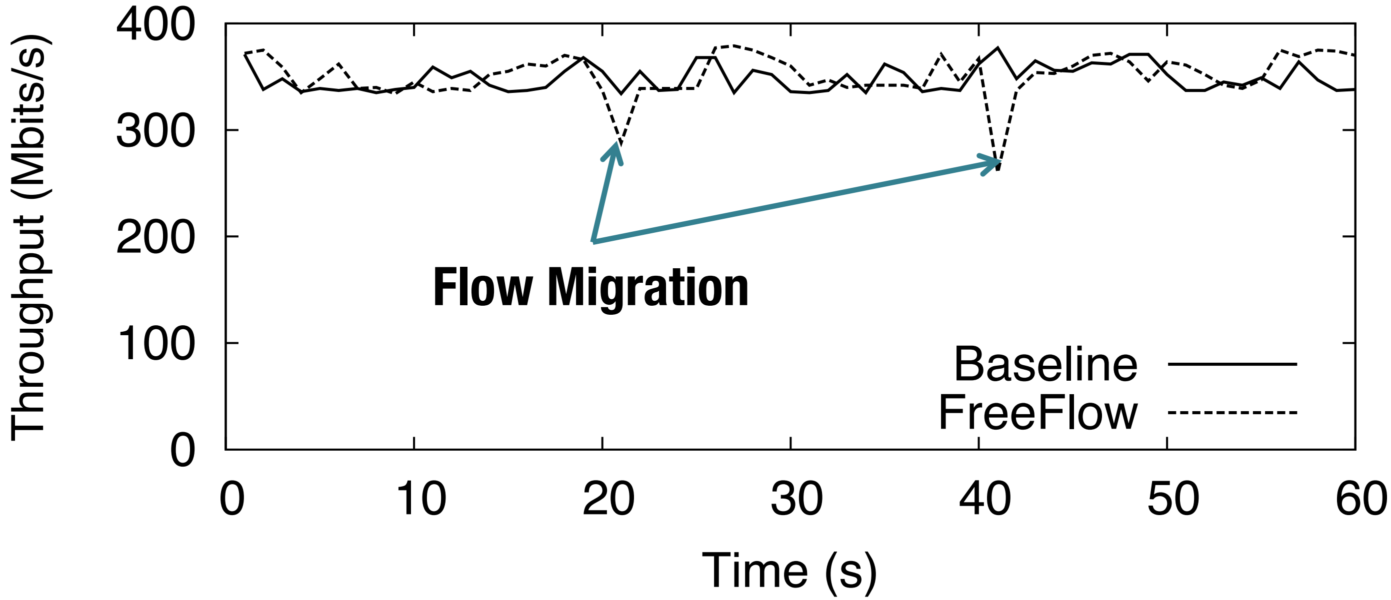




Summary


Home
Connect
Discover
Me
Search
FreeFlow
View my profile page
6 TWEETS
5 FOLLOWING
0 FOLLOWERS
Compose new Tweet...
Tweets
FreeFlow @SplitMerge
Split/Merge enables balanced elasticity in virtual middleboxes thru dynamic partitioning & consolidation of state and flows among replicas
SDN News @sdn_news 2h
Photo: New Post on sdncentral.com/events/arista-... Arista's SDN Seminar Series Presents... tmblr.co/Z9EiXuhQ18PZ
View photo
SDN News @sdn_news 2h
Check out SDNCentral post: Arista's SDN Seminar Series Presents Integrating Software Defined Networks #SDN ... pic.twitter.com/AlZhxmSuOD
View photo

Flow Migration Overhead - TCP



Flow Migration Overhead - UDP