Dynamic Service Chaining with M2M

Xuan Kelvin Zou, Amy Tai, Ronaldo Ferreira, Jennifer Rexford, Pamela Zave

ABSTRACT

Middleboxes are at once an abomination (a violation of the end-to-end argument) and a necessity (for deploying higher-level services between endpoints). Conventional techniques for getting middleboxes "on path", so called network service chaining, rely on manipulating how the network switches route traffic. However, tweaking the routing configuration is clumsy and inefficient, especially when middleboxes need to serve mobile endpoints or to support dynamic flow migration. Instead, we propose an architecture for Mobility and Migration Management (M2M). M2M presents a clean separation of data, control, and management planes for scalable and flexible policy management with low overhead. We introduce a Middlebox-to-Middlebox (M-to-M) protocol that leverages M2M for dynamic service chaining. The M-to-M protocol deals with middleboxes as an explicit part of the end-to-end path, and adds, removes or replaces middleboxes dynamically as needed. We also develop update algorithms to address packet reordering and race conditions during a flow migration. Our high-performance prototype implementation can sustain 40 Gbps on a commodity server, and establish a session and manage flow migrations within the order of milliseconds.

1. INTRODUCTION

Middleboxes are ubiquitous and elastic: Network administrators use middleboxes to apply services on the traffic exchanged between pairs of communicating endpoints. Today, they are routinely used to improve security (firewall, packet scrubber), protect user privacy (encryption, anonymization), share a set of IP addresses (network address translator), spread traffic over multiple backend servers (load balancing), reduce bandwidth consumption (compression, video transcoding, caching), monitor traffic and perform application-specific tasks. Recent trends of network function virtualization (NFV) implement middleboxes as virtual machines (VMs) or user-space applications separate from the physical host. Virtualization enables running network functions (NFs) at many different locations in the network or even in public clouds, and the NF instance can spin up (or down) or even migrate as load demands.

Endpoints move: The growing cellular network and WiFi hotspot coverage has pushed the term "mobile" to a new era. Mobility support has become a key consideration in terms of network infrastructure design. More and more network applications and services are run in virtualized environments; the server VMs can run anywhere and the VM migration can happen at any moment due to load balancing, infrastructure maintenance, etc.

To support a new Internet era where client mobility, and server and middlebox migration can take place anywhere and anytime, we propose a M2M(definition goes here).

The existence of middleboxes already breaks the end-to-end principle, so we capitalize on this violation to create a new abstraction for a connection. We identify each connection by a supersession and a list of subsessions, where a supersession represents the original end-to-end communication. NF traffic steering simply stitches the subsessions together, and mobility and migration apply the same building block operation: replacing an old subsession with a new one. Hence, the subsession abstraction exists to isolate middlebox operations in a supersession to individual subsessions.

Although previous works consider some of the problems we address, none provides a comprehensive solution for endpoint mobility, server migration and middlebox service chain and migrations. TCP Migrate [39], Mobile IP [3], Serval [30] and ROAM [43] support endpoint mobility, but none of them explicitly deals with the existence of middleboxes. DoA [41] proposes a delegation architecture for middlebox service chain, however it neither supports flow migration across NF nor server migration. SIMPLE [31] and OpenNF [16] take advantage of modern switches that offer fine-grained control over routing (e.g., OpenFlow enabled switch), to steer traffic selectively through one or more middleboxes. However, the routing solutions fail to support endpoint mobility and suffer from scalability and flexibility of fine-grained policy control due to the limited size of ternary content-addressable memory (TCAM) on switches. More fundamentally, we argue middle-

boxes should be addressed explicitly instead of treated as second-class citizens.

M2Machieves:

- **Unified solution:** It supports client mobility, server migration, middlebox service chain and flow migration during middlebox migration.
- **Incrementally deployable:** The system can be deployed between client and server, client and middleboxes, or even just between middleboxes.
- **Same socket abstraction:** It does not change transport layer, and thus does not require any change to the higher-level application if the application is using socket abstraction.
- **No routing change:** Since we address the subsessions explicitly by the network layer, simple solutions like shortest path routing still work.
- **High performance:** An in-kernel prototype shows that a shim translate layer can sustain up to 14.2 Gbps per CPU core.

The paper is organized as follows: Section 2 gives motivating examples of dynamic service chaining and endhost mobility and how current solutions fail to meet the requirement. The architecture of M2Mis described in Section 3, and the protocol of supersession – subsession establishment and migration is described in Section 4. Section 5 describes crucial data plane properties provided by M2M. Implementation is described in Section 7 and evaluation in Section 7. Section 8 presents a discussion about deployment, security and fault tolerance. We cover related work in Section 9 and conclude in Section 10.

2. MOTIVATING EXAMPLES

In this section, we first present a few scenarios where a system may require network function (NF) insertion, removal, or migration, and host mobility. We also discuss how the current solutions fail to address our requirements.

2.1 Dynamic NF Policy

Enterprise networks deploy various network functions for better performance and security. Being able to modify dynamically NF types/instances in a service chain improves the efficiency and flexibility of the network system. NF insertion is necessary if a flow is marked as suspicious by a coarse-grained intrusion detection and prevention system (IDPSs [36]) and requires a fine-grained deep packet inspection (DPI). NF removal is preferred if a connection goes through a cache proxy, but the proxy has a cache-miss and the content is uncacheable; the cache proxy may remove itself from the chain. NF load balancing is necessary if the network operators want to distribute a flow evenly among multiple instances of the same NF. Moreover, NF instance migration can happen in a network function virtualization (NFV) setting, and the flows that use the NF instance also need to migrate along with the instance.

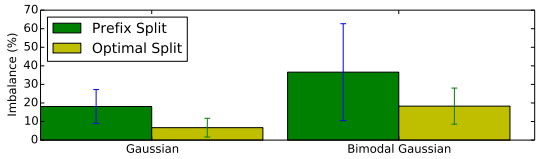

Figure 1: The distribution of flow size for different prefix can be measured using a weight[4]. We assume two different weight distributions for IP prefix: Gaussian, Bimodal Gaussian with the parameters from[4]. We show that in the case of distributing flow over two NF instances, a routing solution, while doubling rules, fails to balance the load across middleboxes in the scenario above. (Prefix split uses the most significant bit, and optimal split uses arbitrary one bit wildcard matching).

Efficient, dynamic NF policies cannot be implemented on conventional switches due to coarse-grained routing, e.g., the switch may simply tunnel all the traffic to IDS for further inspection and remove the IDS once it finds the proverbial "needle in the haystack". Some recent techniques leverage fine-grained routing switches (e.g., OpenFlow [25] based) for dynamically changing NF policies. However, these solutions are neither scalable due to TCAM rule size, nor flexible due to a complex dependency between routing rules. The SDN controller plays an excessive role in data plane, while still fails to fulfill certain network order-preserving properties [34, 16]. Figure 1 gives an example of the imbalance caused by routing solution when trying to distribute flows across two instances of the same NF type.

2.2 Mobility and NF Policy

A cellular network can be divided into user equipment (UE), local access network (LAN) and core network. LANs communicate to UEs through its base station and communicate to the Internet through the core network. Cellular networks rely on a wide range of NFs (e.g., firewall [22], load balancer [1], cache proxy [7]) to improve performance and enhance security. As mobility and network function virtualization become ubiquitous, we may ask for (i) seamless mobility and (ii) dynamic service chaining.

Many mobility solutions have been proposed over the past years [3, 39, 43, 30], yet none of them consider the existence of network functions. Worse, NFs can even be a hindrance for these protocols since they may interfere with the protocol control logic (e.g., TCP split [11]). Routing-based solutions have been proposed for service chaining in cellular networks(e.g., SoftCell [18]). However, in order to support host mobility, it places NFs only in the core network and does not support dynamic

service chaining.

3. ARCHITECTURE

We now give an overview of M2Marchitecture. We discuss the design goals that motivate a new architecture, which separates functionality into three different planes, architecture components, and the mechanisms to achieve the design goals.

3.1 Design Goals

Scalable policy management. Correctness conditions dictate that traffic must be correctly forwarded through a middlebox chain according to network policies. Relying on routing for traffic steering is inefficient, because distributed routing protocols are slow to converge during topology reconfiguration and provide only coarse-grained control over flows. SDN-based solutions avoid this problem by installing per-flow rules on network switches, but these do not scale well since the SDN controller must be involved in per-flow decisions, and switches have limited memory to store flow rules; in the worst case, each switch must install one rule per TCP flow.

In M2M, we avoid both of these scalability impediments. The protocol is designed such that the destination address of each packet identifies the next middlebox or endpoint in the path and the source address identifies the sending middlebox or endpoint. Doing so avoids the need of routing tweaks when network topology changes or endpoints move. Policy management in M2Mis controlled by a logically centralized controller, but packets on the data plane are never redirected to the policy server. This is a key difference from SDN solutions that queue packets on the controller during flow migration between middleboxes [16]. Moreover, the policy server can offload per-flow decisions to the middleboxes, for instance allowing a middlebox to remove itself from a session path. Off-loading further relaxes any dependence on the policy server.

Low performance overhead. Recall that we identify each connection with a supersession and its associated subsessions. A simple approach for decomposing a supersession into subsessions can be implemented by establishing tunnels between middeboxes. However, by adding a new header to each packet, this approach introduces overhead in the form of MTU increase and consequently packet fragmentation. Packet fragmentation can be solved by increasing the MTU of switches and routers inside a single network administration, but this solution is not viable when packets have to cross network domains, which has become common in virtualized network functions that are outsourced to public clouds. Moreover, solutions that rely on tunnels generally add extra overheads such as encryption and compression [38], features that can be redundant or unnecessary to all flows. M2Mrelies on network address translation for explicitly addressing subsession endpoints, hence the only overhead is incurred by port remapping. Also, supersession IDs in M2Mare not carried on each packet, as in [41]. They are stored on the middlebox agents during supersession setup and reconfiguration.

Endpoint mobility and NF migration. The growing trend in network function virtualization can cause dynamic migration of flows between middleboxes. Along with frequent end host location changes due to device mobility or VM migration, these new possibilities complicate traffic-steering solutions relying on administrative control over network routing. In M2Marchitecture, dynamic policy changes are centrally abstracted by a separate policy server, which decides (i) the sequence of middleboxes traffic should traverse and (ii) informs the hosts — endpoints or middleboxes — of these decisions by pushing policies ahead of time to avoid increasing connection set-up delay. After fulfilling these two responsibilities, the policy server steps back to allow hosts to establish sessions and handle migrations as needed.

3.2 Components

An overview of the M2Marchitecture with its main components and the interactions between the modules is depicted in Figure 2. The architecture presents a clear separation of management, control, and data plane functionalities. The management plane is composed of a policy server that is responsible for coordinating the agents on the control plane based on high-level network policies provided by the network administrator. The control plane implements the protocol in §4 for session initiation, flow migration, and network function insertion and removal. The data plane is responsible for translating supersession IDs into local IP addresses and ports, delivering packets to the network functions, and forwarding packets between middleboxes. To simplify the presentation, we first assume a homogeneous scenario in which endpoints and all middleboxes run M2Magents, and defer to §8 a discussion of a more realistic scenario where M2Mcan be deployed incrementally. Also, we assume that network functions do not change the packet 5-tuple, i.e. — source and destination IP addresses and port numbers, and protocol type; the case where packets are changed will be discussed in §6.2.

Policy server: The M2Mpolicy server is a logically centralized server that receives high-level policy specification from the network administrator and reliably delivers subpolicies to the agents, i.e. — it makes sure that an agent receives a policy consistent with its location. For instance, a client agent should receive only policies related to the network it is connected.

The policy server makes global decisions based on: (i) configuration changes or (ii) network state changes. Configuration changes are triggered by the network administrator, and network state changes are triggered by monitoring information, which the policy server receives from the M2Magents. Based on the monitoring information, the policy server may decide to migrate flows from an overloaded network function to a different instance, insert new network functions in a supersession, e.g. — introduce a DPI after a light IDS flag packets for deeper inspection, or remove a network function out of a supersession path because it is no longer needed. To further extend system flexibility, network administrators can allow the policy server to offload decisions to the M2Magents running on the middleboxes. A common decision that can be easily offloaded to the agents is the selection of a network function instance when the network administrator configures multiple instances of the same type.

In M2M, a policy specifies a predicate, which defines a set of affected packets, and a sequence of network function types. This sequence defines the service chain to be traversed by the packets satisfying the predicate, e.g. — packets from subnet 10.0.1.0/24 should be forwarded through the chain {Load Balancer → IDS → Firewall → Proxy}. The general form of a policy is:

$$\texttt{match(predicate)} \texttt{ >> } \{\texttt{NFT}_1 \rightarrow \cdots \rightarrow \texttt{NFT}_n\}$$

A predicate is specified with source and destination IP addresses and ports, and protocol types. Complex predicates can be specified using conjunction (&), disjunction (|), and negation (~) operators like in Pyretic [26]. A packet that satisfies the predicate in the match statement is forwarded through the chain of network function types specified on the right-hand side of the >> operator. A network function type ($\texttt{NFT}_i$) specifies a set of network function instances of the same type. For example the network administrator can specify such a set of proxy servers, and the policy server will either choose an instance for each client or delegate this decision to the M2Magents.

M2MAgents: M2Magents are key enforcers of network policies. They must ensure that packets are either correctly forwarded to the next middlebox on the service chain or dropped if they do not comply with the network policy. Whenever a client application starts a connection, the M2Magent running on the same machine intercepts the first packet of the connection and matches it with the client's network policies to determine the sequence of network function types the packet must traverse. If a match is found, the agent starts the session setup protocol in §4.1 that will create the subsessions between middleboxes and the corresponding mappings as described below. The 5-tuple of the client connection is used to identify the supersession packets when they are processed by the network functions. Once the supersession is setup, all subsequent packets from the client's connection are forwarded to the first middlebox on the service chain, and from there they are sequentially forwarded to the remaining middleboxes and to the server.

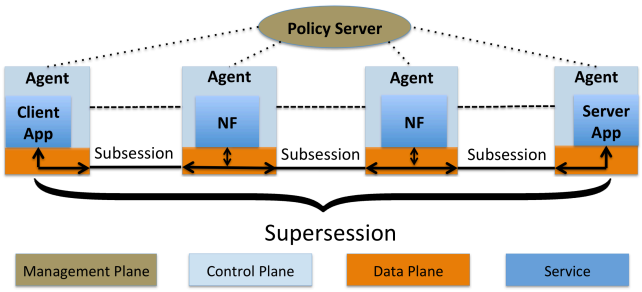

Figure 2: M2Marchitecture

In each middlebox, a M2Magent must create two mappings for each session. One, which we call horizontal NAT, translates the 5-tuple of an incoming packet to the corresponding 5-tuple that is used in the next subsession. The other one, which we call vertical NAT, translates the 5-tuple of each incoming packet to the supersession 5-tuple before the packet is delivered to the network function or to the applications on the endhosts. Keeping the supersession 5-tuple invariant on the middleboxes has several advantages. First, it simplifies policy specification by abstracting the subsessions from the network administrator. Second, it simplifies network function state migration, because network functions always receive packets with the supersession 5-tuple regardless of flow migrations or middlebox insertions or removals. Third, network functions that do not change the packet 5-tuple do not need to be changed to work with M2M.

M2Magents are also heavily involved in service chain maintenance and supersession reconfigurations. The agents report management information, such as resource utilization in the middleboxes, to the policy server, so it can make global management decisions about service chains. If allowed by network policy, reconfigurations can also be triggered by a network function. A common case of a reconfiguration triggered by a network function is when a cache proxy detects that the content of a session is not cacheable and signals the M2Magent to remove the proxy from the service chain. Once a reconfiguration is initiated, either by the policy server or by the network function, M2Magents execute the protocols in §4.2.

4. PROTOCOL

Considering middleboxes as explicit components of the end-to-end between two endpoints is the crux of our protocol. Only by doing so can we achieve the de-

sired scalability and flexibility for both endpoints and middleboxes. We discuss session setup in §4.1 and flow migration control in §4.2.

4.1 Session Setup

In M2Mprotocol, each endpoint or middlebox sends packets whose destination is the next middlebox or endpoint in the session path. This obviates the need for special support in the switch or router to direct packets through the chosen chain of network functions (service chain), despite changes in network topology or host movement.

The list of middleboxes, L, that a flow has to traverse is provided by the policy server and can be pulled from the server or pushed to the client. When the client initiates the connection, the control plane uses a three-way handshake to establish the supersession and its associated subsessions. More specifically, the client's control plane sends a SYN message that includes the supersession header and L to the first middlebox. The middlebox strips itself from the head of L, gets the address of the next middlebox from L, and relays the rest of the message to the next middlebox. The SYN message is thus passed recursively through the elements of L before reaching the server. Upon receiving the SYN, the server sends a SYNACK back to the client along $reverse(L)$. Upon receiving the SYNACK, the client immediately sends an ACK to the server via the same mechanism. Once these three control messages are exchanged, the supersession and subsessions are established, and data packets are explicitly addressed to the subsession IPs.

If we simply rewrite the source and destination IPs as described in our "horizontal NAT", we lose supersession information and introduce ambiguity. Consider the case where flow a and b have the same source port and destination IP and port, but different source IPs. If a and b share the same first hop middlebox, the two flows may become indistinguishable upon arrival at the first hop middlebox. To address this issue, we modify the port numbers to identify the flow, a standard technique in NAT [13]. We integrate this port allocation into the three-way handshake: when a middlebox receives a SYN, it assigns port mappings and initiates the next subsession with the rewritten port numbers. If we rewrite both source and destination ports, M2Mcan support four billion unique flows per middlebox pair. See Figure 3 for the complete session setup. (AT: Can we insert some words over the port remapping in Figure 3, to indicate that the → represents port remapping?)

4.2 Migration and Mobility Control

Supporting dynamic middlebox modification for a flow improves the efficiency of the network and NF use, e.g. — removing a cache proxy if the content is not cacheable, inserting an IDS upon detecting suspicious flows,

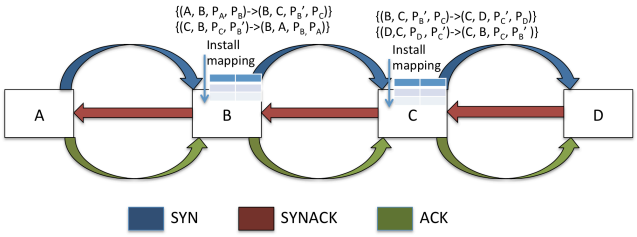

Figure 3: Session setup

or switching from a heavily loaded transcoder to a lightly loaded one. As a natural extension of general middlebox migration, we include endhost mobility in the design of M2M; supporting mobility is also critical in cellular networks.

4.2.1 *"Make before break"*

To find the right mechanism to support flow migration, we investigated existing mobility protocols [39, 43, 3, 30, 28, 29] under the session-location mobility framework [42]. However, the key distinction between host mobility and NF flow migration is that a move in host mobility is unexpected, whereas flow migration among middleboxes is planned.

In fact, this type of reconfiguration is quite common in circuit design and addressed via the "make before break" philosophy. Namely, we stitch together subsessions on the new path before closing the subsessions on the old path. To achieve this, we treat the two neighbors of the moving middlebox as two *signaling endpoints* during a flow migration. We stress that the resulting three nodes that are involved in the migration are *consecutive*. Because we offload some policy decisions to the middleboxes, this property ensures that a middlebox cannot decide the fate of other on-path middleboxes that are not directly affected by the migration.

For middlebox insertion, we first deterministically identify and notify one of the two signaling endpoints as an initiating point. Starting at the initiating point, we set the two signaling endpoints to a suspend state to prevent data transfer on the new path and complete a three-way handshake (UPDATE-SYN, UPDATE-SYNACK and UPDATE-ACK) on the new path, consisting of the two signaling endpoints and the new middlebox. Once the new path is established, we commence data transfer on the new path and remove the old path. We extend the same mechanism to handle middlebox removal and replacement.

4.2.2 *Handling Concurrent Migration*

For correctness, we must properly deal with cases where two middleboxes initiate simultaneous move operations. In particular, if two neighbor middleboxes both initiate removal, we may lose the supersession connection. Strawman solutions include: (1) using a two

phase commit to allow the one with the highest ID to move first; (2) a central controller that assigns token to the middleboxes. However, strong serialization via a two phase commit is not necessary, and a central controller suffers from scalability, one of our initial design goals.

To address concurrent migration, we rely on the following properties: (i) migration is per flow, and (ii) at most three nodes on the current service chain are involved in each flow migration. The two assumptions help us design an efficient algorithm in which migration can happen simultaneously for different flows, and concurrent migrations can happen at different points of the service chain if they involve disjoint sets of nodes; each node has a `pending` counter to ensure that it participates in at most one concurrent update.

The proposed algorithm only allows one concurrent operation for all the nodes involved. When a migration is initiated at a certain node, the node postpones the update if its `pending` counter is not zero. Otherwise it increments the `pending` counter and sends the request to the node immediately to its left if the migration is removal or replacement, or connects to the new middlebox if it is an insertion. Middleboxes for a flow have a strict ordering, which is simply the order in which the middleboxes are traversed by the path from the client to the server. We define two comparators on this ordering, which we term "left" and "right". M_1 is left of M_2 iff the path from the client to M_2 passes through M_1; M_1 is right of M_2 iff the path from the client to M_1 passes through M_2. If the left node responds with a reject, the node backs off, otherwise it receives an approval and proceeds with the update. A node always accepts UPDATE-SYN requests to establish a new path. Algorithm 1 describes the algorithm details and Figure 4 depicts the steps for migrations.

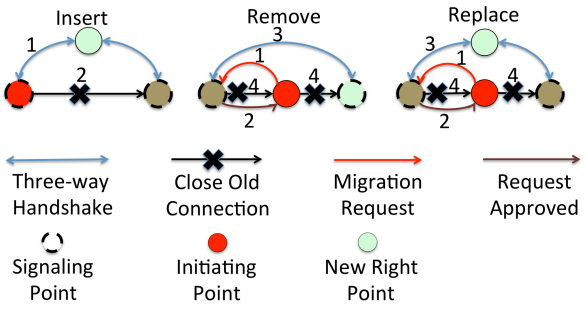

Figure 4: Flow migration during insert, remove and replace

4.2.3 *"Break before make"*

(AT: Do you mean "supersession" in the following bolded words?) When a client moves, it may drop the old **subsession** before establishing a new **subsession**. Consider when a UE moves across a cell boundary, upon which the UE may suffer from transient connection loss,

Algorithm 1: Concurrent Flow Migration

```
Function Trigger_Migration()
    if recv(migrate) then
        if pending == 0 then
            pending++;
            if migration == insert then
                sendto(New right, UPDATE-SYN);
            else
                sendto(Left, request);
        else
            Exponentially backoff and retry;
Function Msg_Handler ()
    if recv(request) then
        if pending> 0 then
            sendback(reject);
        else
            pending++ ;
            sendto(New right, UPDATE-SYN);
            sendback(approve);
    if recv(reject) then
        Exponentially backoff and retry sendto(Left, request);
    if recv(approve) then
        //do nothing, to avoid request re-transmission
    if recv(UPDATE-SYN) then
        pending++;
        if (migration==insert or replace) and (current ==
        not signaling point) then
            forward(UPDATE-SYN);
        else
            sendback(UPDATE-SYNACK);
    if recv(close) then
        //clean old flow state;
        pending−−;
    if recv(UPDATE-SYNACK) then
        pending−−;
        if (migration==insert or replace) and (current ==
        not signaling point) then
            forward(UPDATE-SYNACK);
        else
            sendto(OldMBox, close);
            sendback(UPDATE-ACK);
            //clean old flow state
    if recv(UPDATE-ACK) then
        pending−−;
        if (migration==insert or replace) and (current ==
        not signaling point) then
            forward(UPDATE-ACK);
        else
            //clean old flow state
```

since it is out of the old cell's range.

After losing the old **subsession**, the client needs to rebind to the first hop middlebox. If we use the client's physical IP as part of the supersession identification, the first hop middlebox will fail to identify the supersession if the client changes its IP during mobility. To solve this problem, we can either put the old connection's information in the rebinding message sent by the client or, in a single domain case, administrators can assign a non-routeable IP to each device as a unique ID and use this ID to help identify the supersession.

5. DATA PLANE PROPERTIES

In this section, we discuss three data plane properties and the mechanisms that support them atop the control logic. (AT: the data plane is technically below the control logic. Maybe find another word for "atop")

5.1 Loss-Free Update

At the data plane, there is a translation table stored in each middlebox. The M2Magent accepts flows for which it has established state, rewrites the header, and forwards the flow based on the supersession-subsession mapping stored in the translation table. Moving a flow from one path to another is equivalent to updating the translation tables at the data plane to accept a flow from the new subsession(s) and reject the same flow from the old subsessions. Since there are multiple middleboxes involved in the update procedure, if the update happens in the wrong order, the translation layer may drop packets, e.g. — if the old subsession is removed before the new subsession is fully established.

Finding the right sequence of updates for a general network is proven to be NP-complete [17, 20], but for a special topology, linear in our setting (we only abstract the topology between middleboxes), we can apply the concept from network consistent update [35, 21]. We first ensure that all new translation rules are pushed before the egress applies the new rule, and the old rules are not removed until all the new rules are installed via a complete handshake. In particular, when a migration is initialized from the middlebox (a signaling point), the middlebox notifies its neighbor through the control plane, which inserts a new rule for incoming traffic. Then this neighbor notifies the other side of the connection with an UPDATE-SYN control message. Every hop that receives UPDATE-SYN updates its own translation table. Hence once the other signaling endpoint receives the notification, a new rule for the flow has been installed at every hop for one direction of traffic; it is thus safe to apply the new rule for the egress. The opposite direction is set up in the same way with UPDATE-SYNACK messages. Once the new bidirectional path is built, we tear down the old path by removing the old rules. This loss-free update mechanism mirrors the control plane "make before break" philosophy and is in fact facilitated by the control plane design.

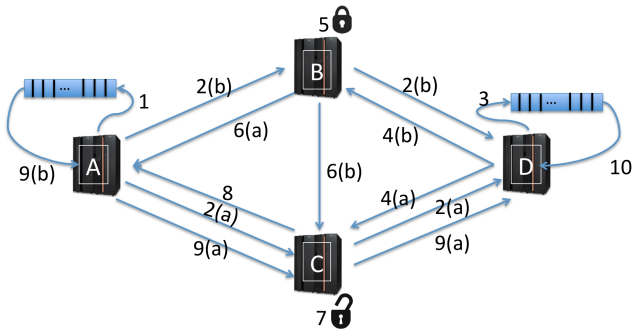

Figure 5: Flow Migration in Packet Order Preserving ((a) and (b) can happen in parallel.)

5.2 Packet Order Preserving

The previously described loss-free update is insufficient if the NF instance (state) also needs to migrate. More specifically, the NF state cannot be migrated since it is being continuously updated as packets are coming in via the old path. To lock and migrate the middlebox state, M2Mmust stop sending traffic during the new path setup. Since changing the protocols' (e.g., TCP) flow control is undesirable, we choose to buffer the traffic and do not release it until the network function state has been replicated at the new path middlebox. Since NF state replication and migration is a well solved problem [16, 34, 33], we do not address this problem here.

A complete flow and state migration takes 10 steps as depicted in Figure 5: 1) lock outgoing traffic; 2(a) send UPDATE-SYN via new path; 2(b) send UPDATE-FIN via old path; 3) lock the reverse direction traffic; 4(a) send UPDATE-SYNACK via new path; 4(b) send UPDATE-FINACK via old path; 5) lock middlebox states; 6(a) send UPDATE-FINACK via old path; 6(b) migrate states; 7) unlock middlebox states; 8) send UPDATE-SYNACK via new path; 9(a) send ACK packets; 9(b) release buffered traffic; 10) release buffered traffic for the other direction.
(AT: is this newline intentional)

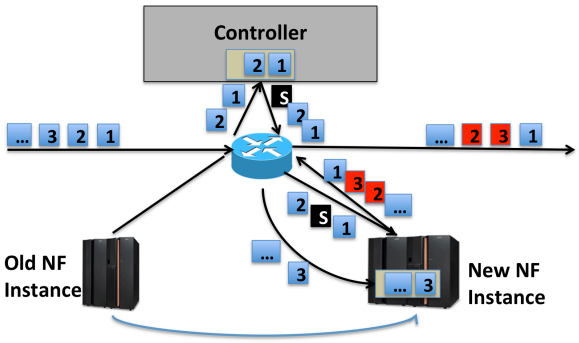

Figure 6: Order-preserving problem in OpenNF without FIFO assumption, the signal packet **s** and the data packets 1 and 2 maybe reordered and thus create reordering. Note the switch is one big switch abstraction.

This update mechanism also satisfies the same **packet order-preserving** property as defined in OpenNF: assuming the path with the NF instance is **FIFO**, i.e., the control message sent after the last data packet is always received after the last data packet, *all packets should be processed in the order they were forwarded to the NF instance by the switch (network)* [16]. We will now show how we achieve the same property without the FIFO assumption.

5.3 Substream Order Preserving

During migration it is useful to create two separate substreams of a byte stream (e.g., TCP), one each for the old and new path. For example, when migrating a flow from one IDS to another, we may want to ensure

that all the SYNs and corresponding ACKs go through the same IDS in order to avoid an alert like "ACK before SYN". When the flow passes through a deep packet inspection (DPI) NF, both string matching and regexp matching build a Deterministic Finite Automaton (DFA), which is traversed from the root based on the byte stream [10, 12]. If we want to use different DPI instances for a byte stream, but the first instance has not seen a complete substream, the first DPI can only check the largest continuous byte stream before it has to buffer the remaining bytes and send them to the second DPI.

In order to divide substreams cleanly, the left neighbor of the moving node leverages TCP sequence numbers. Suppose a migration is initiated at time t. The left neighbor uses the maximum seen TCP sequence number at time t as a *checkpoint*: it buffers packets with a higher sequence number than the checkpoint and forwards packets with a lower sequence number. M2Mdoes not lock and migrate the old NF instance state until it sees a TCP ACK with sequence number *checkpoint*. The ACK guarantees the delivery of all the packets in the old substream. Note that although the buffering may create a reordering at the left neighbor middlebox, this protocol results in the correct order for the two substreams from the perspective of endpoints. See Algorithm 2 for details.

Algorithm 2: Substream Order Preserving

```
Event_Handler recv_SYN(TCP_packet p)
    checkpoint = hash_lookup(p);
    if p.seq > checkpoint then
        Buffer (p);
    else
        Forward(p);
Event_Handler recv_ACK(TCP_packet p)
    if p.ack > checkpoint then
        Migrate NF state;
        //wait until migration finishes;
        sendto(left neighbor, release_buffer);
Event_Handler release_queue()
    while !buffer.empty() do
        Forward(buffer.dequeue());
    Reset(hash_table);
```

A combination of order preserving and substream separation provides us with a stronger **substream order preserving** property during an update: *substreams should be processed by different NF instances in the order they were sent from the sender*. OpenNF cannot guarantee order preserving with respect to a byte stream when network links are not FIFO [1]. Furthermore, because OpenNF is by design router-based, it simply *cannot* achieve substream order preserving; see Figure 6. Our protocol is able to provide this property precisely because the system architecture is designed to be aware of and rely on transport protocols.

Note that we may need to split the packets in the case of SYN-ACK piggyback and packet coalescing. In the first case, both directions asking for substream separation can result in deadlock since M2Mmay choose the new path for a substream with higher sequence number in one direction and the old path for ACK as it is ACK-ing a substream with lower sequence number in the reverse direction. In the second case, the TCP client may coalesce packets, causing the payload to cross the *checkpoint* boundary in the byte stream if retransmission occurs.

[1]OpenNF tech report relies on this assumption in its proof.

6. IMPLEMENTATION

6.1 Prototype

There are three options for an in-kernel implementation of the data plane: (i) OS native network stack; (ii) NIC driver (e.g., DPDK [2]); and (iii) customized in-kernel software switch (e.g., openvswitch [19]). We chose option (i) because (a) M2Mhas to choose the right interface before sending to the driver when having multiple interfaces; and (b) a customized in-kernel software switch has high overhead. We implement a kernel module data plane and user space control/management plane via TCP/UDP sockets, with a total of about 4000 lines of code in C and C++. We install kernel modules to register callback functions with Linux `netfilter` for data plane operations. The control/management plane has extensible, complex control logic, and the data plane does specific, simpler actions such as header rewriting, queuing packets to user space via `netfilter_queue` and updating the kernel hash table, which holds the flow → port mapping. The user and the kernel agent communicate via `netlink`, a native Linux Inter Process Communication (IPC) function. Currently the policy server is a simple TCP server that proactively pushes policies to the agents.

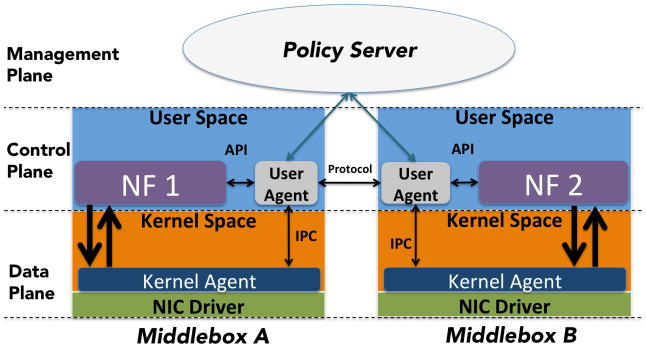

Figure 7: Layout of the implementation blocks

6.2 Network Function Support

We categorize NFs into two types — active and passive functions — based on whether the NF acts on the original packet.

Name	Type	Key Functions	Binding Library
PRADS [5] (P)	Monitoring	got_packet()	libpcap
Bro [8] (P)	IDS	DumpPacket()	libpcap
Snort [6] (P)	IDS	PQ_Show()	libpcap
Balance [1] (A)	Load Balancer	recv(), writen()	user socket
Squid [7] (A)	Proxy	getsockopt()	user socket
Traffic-Squeezer [9] (A)	WAN-Optimizer	net_receive _skb()	Linux skbuff

Table 1: Commonly-used Network Functions; (A) means active and (P) means passive NF.

Many passive NFs make decisions based on a clone of the original packet. Based on our survey in Table 1, we found that most passive NFs use libpcap to capture cloned packets from a raw socket. As described in §3, we must restore the supersession header of the copies before delivering them to the NF. To implement this so-called vertical NAT, we modified the `pcap_handle_packet_mmap()` function in `pcap-linux.c` in libpcap [24] to restore the supersession.

For active NFs, there are two cases. If the NFs only act on the payload or MAC layer, e.g.— TrafficSqueezer, supersession restoring also works. However, if the NF acts on the 5-tuple, we have to extend NF functions to notify the M2Magent of its header mapping. For example, a transparent cache proxy [7] gets the packets from its listening port and sets up a new TCP socket to the final destination. M2Mwill break the supersession into two if it is unaware of the mapping between two sessions. On the other hand, if the NF informs the M2Magent of the mapping between its listening and sending TCP sockets, the M2Magent can stitch the two subsessions into the same supersession.

7. EVALUATION

In this section we show that the separation of the data plane from control and management operations in the M2Marchitecture results in a system where network functions can be deployed with very small overhead. Compared with a conventional deployment that relies on routing for service chaining, M2Madds a network round trip time for session setup, which can be easily and safely eliminated in an enterprise network if the session setup information is piggybacked in TCP syn packets, we describe this optimization in §7.2.1. Moreover, we show that M2Mcan sustain throughput at line speed in a 40 Gbps testbed and present microbenchmark results for the overheads introduced for table lookup, rule insertion, and header rewrite. We also show measurement results for latencies introduced during session setup when multiple network functions are on the session path and latencies for flow migrations under different order preserving semantics.

We used two dedicated testbeds for the performance evaluation, with the following configuration: (i) one L2 switch with 16 1Gbps ports, and five *mid-range* workstations, each with one Intel Quad-Core Xeon 3.7GHz processor, 32GB of memory, and two 1Gbps NICs; and (ii) one L2 switch with four 40Gbps ports, and four *high-end* servers, each with two Intel 8-Core Xeon 2.2GHz processors, 64GB of memory, and one 40Gbps NIC. We conducted throughput stress tests in the four high-end servers, and unless specified, all the other experiments were performed in setting (i).

7.1 Throughput

We first show that M2Mcan sustain throughput at line speed in 40Gbps links using a vanilla Linux kernel and 1500-byte packets. The throughput numbers were obtained from `iperf` [40] executions on three *high-end* machines connected on a line topology client–middlebox–server. Since the high-end machines have only one NIC each, we configured virtual interfaces on the NICs and connected them to different IP subnets. As a baseline for comparison, we enabled IP routing on the middlebox on the same topology and removed the M2Mkernel modules on the three machines. We also show the throughput results for three other configurations: first, a VPN tunnel between the middlebox and the server; second, the same VPN tunnel with compression enabled (VPN+OP); and third, a software switch (OVS) performing network address and port translations. VPN tunnels have been proposed as a solution for outsourcing virtualized network functions to the cloud [38]. The OVS results are included to show the throughput gains of having a dedicated kernel implementation for M2M.

Figure 8 shows the throughput measurements for the five different scenarios: M2M, baseline, VPN, VPN+OP, and OVS. The results show the M2Mkernel-module data plane implementation can sustain 14.2 Gbps on a single core, and it scales linearly to the number of cores, reaching 37.1 Gbps at its peak. The difference betweek the peak throughput and 40Gbps is because `iperf` measures the throughput at the application layer, i.e., packet headers (TCP+IP+Ethernet) are not included in the computation. The baseline case yields a throughput of 14.6 Gbps on a single core, so the kernel module overhead is under 3% in the worst case. As the number of flows increases, more cores are involved in packet processing and the gap between M2Mand the baseline shrinks until it becomes negligible. The reason is that interrupts per core happen less frequently and the link bandwidth becomes the bottleneck instead of the CPU. We envision if we increase the NIC bandwidth to 100 Gbps or higher, the gap will stay the same and the throughput will grow linearly before the link bandwidth or the PCI bus become the bottleneck.

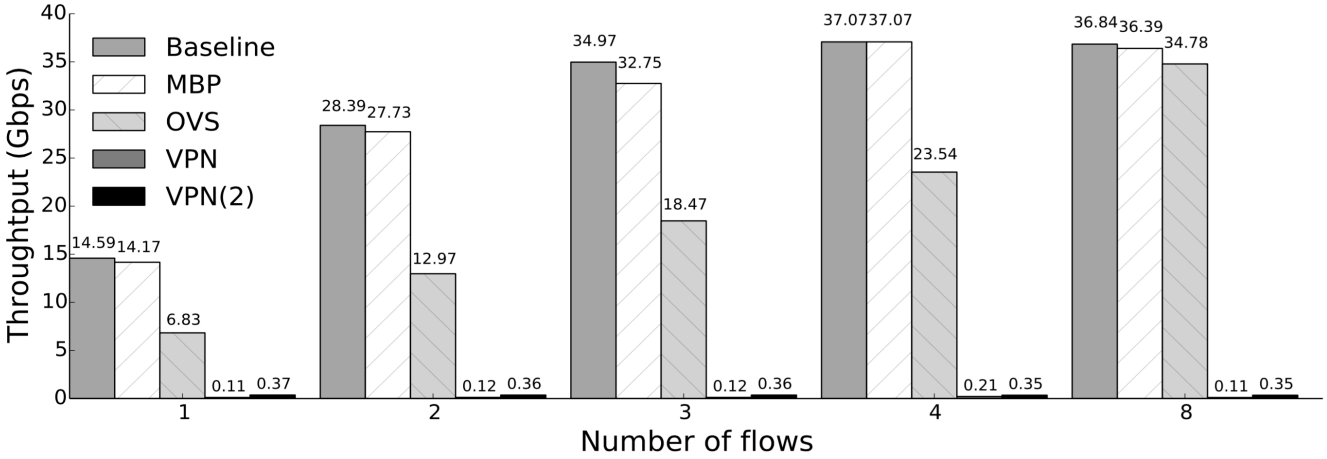

Figure 8: Throughput of different approaches.

We initially observed inconsistent results for four flows because the default hash function (Toeplitz hash) of the NIC driver was not spreading well the flows into different cores. We had to apply a minor tweak on the NIC driver to get a better distribution of flows into cores. When the number of flows is small, the driver manual recommends the change of the Toeplitz hash function to a simple XOR function to avoid flow colliding on the same core [**?**]. After we changed the hash function, we observed consistent results. The same configuration was used for the five scenarios.

The throughput results for the other three scenarios are also depcited in Figure 8. We compare our approach with off-the-shelf VPN-encapsulation mechanism, which is used in APLOMB [38] to achieve its redirection. Not surprisingly, VPN uses encryption and thus offers much lower throughput. OpenVPN [15], which is used in APLOMB, achieves less than 400 Mbps on a single core, even with the compression optimization. Current OpenVPN does not support multi-threading and is not able to take advantate of the multiple cores on the machines. Even if multi-threading is incorporated in OpenVPN in the future and assuming it will scale linearly to 16 cores, it will be able to offer only 17% of the maximum throughput.

OVS results.

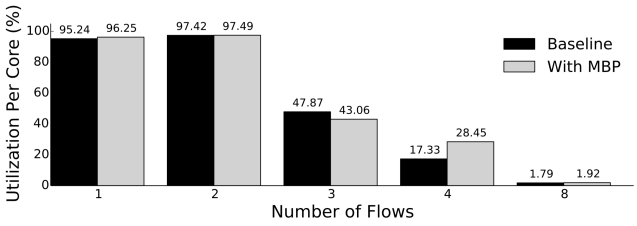

Figure 9: Average CPU utilization in stress test. Note we only include the cores that are interrupted by the NIC.

Figure 9 shows the CPU utilization of the middlebox in the same experiments. Note that since the middlebox acts as a software router without the kernel module, it also consumes a certain amount of CPU cycles. The highest difference between the baseline and the system lies in the setting of 4 flows, the CPU utilization for M2Mis 10% higher than that of the baseline (27.6% versus 17.2%), but under most circumstances the gap is within 2%. One thing worth noting is that as the number of flows increases, the CPU utilization per core plummets, our explanation is that since the testbed machine has 16 cores and 32 hyper-threads, it spreads out quite evenly, and the CPU utilization scales linearly at high load but it is not the case at low load (e.g., 1M packets/s causes much higher CPU utilization than 500K packets/s).

The experimental results are promising and prove that the kernel-module data plane implementation can offer very high throughput at the cost of modest increase of CPU utilization.

7.1.1 Microbenchmarks

The kernel-module implementation of the data plane includes three main functions: (i) table lookup for vertical and horizontal NATs; (ii) header rewriting and re-checksum; and (iii) rule installation.

We also have a few optimizations in the data-plane implementation (e.g., partial checksum to reduce CPU cycle, a small hash table to fit in L3 cache) result in an extremely efficient system with a very low overhead.

We conducted a microbenchmark to see the delay different functions add to the system. We insert 100K rules in the kernel hash table, and conducted 100K lookup. We also microbenchmarked the header rewriting and partial checksum of the data packets. We had the two following methods to eliminate Linux timer's overhead: (i) batch and time the insertion and lookup for every one hundred rules; (ii) get the pure timer lookup time and then subtract that base number. The result shows

that lookup, insertion and header rewriting takes on average 131, 251, and 214 ns, this gives us ≈ 2.8M packets per seconds with lookup and header rewriting, this is equivalent to 33.6 Gbps per core with MTU of 1500 Bytes.

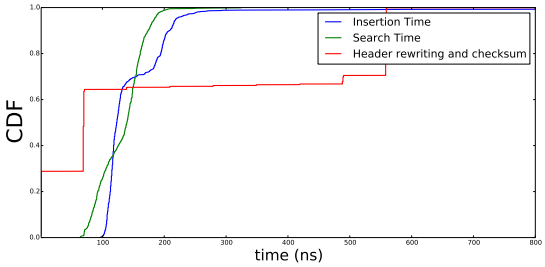

Figure 10: Time CDF for different functions

7.2 Latency

We evaluate our system's latency in the following order: (i) how much overhead it has for a super-session setup? (ii) how much overhead it has during an update while preserving different properties. We choose setting (i) testbed since it is closer to a real enterprise network w.r.t. round trip time and switch buffer size (queuing delay). Since M2Mis per-flow based operation, the number of flows does not affect per-flow latency.

7.2.1 Three-Way handshake

We first measure the latency for session establishment, in a topology of three hops (client - middlebox - server). We have two different implementation, (i) out of band UDP ahead to TCP handshake, (ii) piggyback TCP handshake at the payload. Not surprisingly, TCP piggyback can greatly reduce the extra latency due to propagation delay, with a modest increase less than 40μs. On the other hand, out-of-band UDP incurs 688μs delay ahead of TCP handshake.

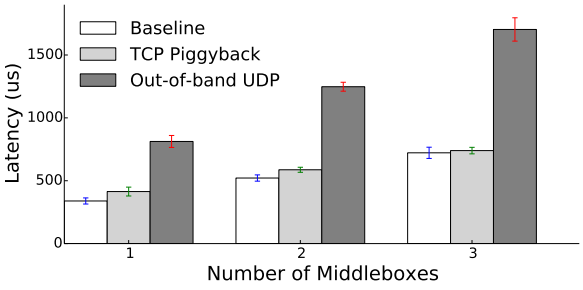

Figure 11: Latency for session establishment

7.2.2 Flow Migration

M2Msupports flow migrations with various properties. We measure the time interval for such an update with a loss-free, packet order-preserving, and substream order preserving. We also created artificial packet reordering in the case of byte stream order preserving during an update.

We run our experiments in a topology with four machines (A,B,C,D) with two paths (A - B - D) and (A - C - D) that each consists of three hops. Client and server are at A and D. The goal of the flow migration is to replace middlebox B with D. We measure the latency for migrations with (a) loss-free (LF), (b) order-preserving (OP), and (c) substream separation and order preserving (SS+OP) properties. The latency are evaluated under a full load on a network with all 1Gbps links. All tests are run in TCP Cubic.

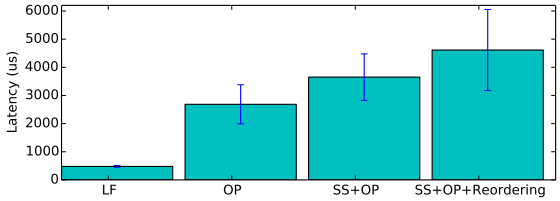

Figure 12: Latency of flow migration with loss-free (LF), packet order-preserving (OP), substream-separation and order-preserving (SS+OP), and substream-separation and order-preserving (SS+OP) with an artificial reordering.

We summarize the latency result on Figure 12. During a loss-free migration, the old path is saturated; however, the new path is idle during a subsession setup and thus has the lowest latency. During a order preserving migration, M2Mhas to send subsession setup on the new path and a marking packet right after the end of data packets on the old path; the higher latency is mostly due to the queuing delay on the old path. During substream separation and order preserving migration, besides the queuing delay, the ACK message may also be delayed from the receiving side, e.g., the extra time for TCP to process the byte stream and generate an ACK. Since we rarely see packet reordering in the testbed during an update, we artificially created a packet reordering via dropping a packet with a lower sequence number in the kernel, which triggers a retransmission of the packet with a lower sequence number after the packet with a higher sequence number.

The results demonstrate that the system can deal with flow migrations in the order milliseconds. This can greatly facilitate the flow migration during an NF insertion, removal and hot-standby replacement. NF state migration can happen within hundreds of milliseconds [16, 34], here results show that a flow migration should no longer be considered a bottleneck in the case of NF state and corresponding flow migration. A more detailed breakdown is in Table 2.

[2] at packet rate 37.5 kpps, time are consumed across flow migration and state migration

	LF	OP	SS+OP
Split/Merge [2]	500ms	no support	no support
Dionysus [3]	600-1200ms	no support	no support
OpenNF [4]	84ms	96ms	no support
M2M[5]	0.5ms	2.7ms	3.6ms

Table 2: Comparison with other dynamic middlebox and flow migration systems

8. DISCUSSION

incremental deployment
security
fault tolerance

9. RELATED WORK

Middlebox in SDN: SIMPLE [31] and FlowTags [14] take advantage of the switches with a fine-grained rule support in software-defined network (SDN), and support network function policy chaining in traditional and NFV setting respectively. Both approaches are constrained by the TCAM size, a hardware limitation in terms of fine-grained policy, and neither not support NF migration or host mobility.

OpenNF [16] and Split- Merge [34] leverage the SDN controller to manage NF's state migration and NF's flow migration. However, since the central controller is involved in both control plane (update the network rules) and data plane (buffer the packets on the fly during migration), they suffer from high latency and low scalability. They also suffer from hardware limitation for fine-grained policies.

Middlebox using naming service: DoA [41] uses a delegation-oriented global naming space architecture, that explicitly specifies intermediary middleboxes on the path. Two key distinction between DoA and M2Mis that (i) M2Mdoes not require any new naming service and (ii) DoA does not support dynamic policy service. APLOMB [38] outsources the network functions to the cloud with a naming service indirection. It uses VPN to tunnel the traffic to the cloud and use DNS-based indirection to decide which cloud to enforce the middlebox policy. It assumes the cloud can provide elastic NF service but it cannot explicitly handle dynamic policy chain.

Middlebox consolidation: CoMB [37] and click [23, 27] both consolidate network functions as an application or a VM image, and one host machine can run multiple instances of different network functions. Both approaches are mainly focused a feasibility and scalability of network functions on a single generalized servers. Both solutions consider neither scale-out across servers nor NF and flow migration.

Mobility Protocols: Mobility protocols use (i) a fixed indirection point (e.g., Mobile IP [3]), (ii) redirecting through DNS (e.g., TCP Migrate [39]), (iii) indirection infrastructure (e.g., ROAM [43]) or (iv) indirection at the link layer (e.g., cellular mobility). None of them consider the existence of middleboxes. Coexistence of network functions and new protocols is especially important for deployment, as a study in multipath TCP shows [32]. M2Mshares some similarity with many mobility protocols and has support for network functions.

[3] Large-scale networks

[4] flow update time at packet rate 2.5-10 kpps

[5] at packet rate 80 kpps (1 Gbps)

10. REFERENCES

[1] Balance. http://www.inlab.de/balance.html.

[2] DPDK. http://dpdk.org/.

[3] Mobile IP. https://tools.ietf.org/html/rfc3220.

[4] Niagara. Tech. rep. http://www.cs.princeton.edu/research/techreps/TR-973-14.

[5] PRADS. http://gamelinux.github.io/prads/.

[6] Snort. https://www.snort.org/.

[7] squid. http://www.squid-cache.org/Intro/.

[8] The Bro Network Security Monitor. https://www.bro.org/.

[9] Traffic Squeezer. http://www.trafficsqueezer.org/.

[10] Aho, A. V., and Corasick, M. J. Efficient string matching: an aid to bibliographic search. *Communications of the ACM 18*, 6 (1975), 333–340.

[11] Balakrishnan, H., Seshan, S., Amir, E., and Katz, R. H. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 1995), MobiCom '95, ACM, pp. 2–11.

[12] Bremler-Barr, A., Harchol, Y., Hay, D., and Koral, Y. Deep Packet Inspection as a Service. In *CoNEXT* (Sydney, Australia, 2-5 December 2014).

[13] Egevang, K., and Francis, P. The ip network address translator (nat). Tech. rep., RFC 1631, may, 1994.

[14] Fayazbakhsh, S. K., Chiang, L., Sekar, V., Yu, M., and Mogul, J. C. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), USENIX Association, pp. 543–546.

[15] Feilner, M. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd, 2006.

[16] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM '14, ACM, pp. 163–174.

[17] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving High Utilization with Software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 15–26.

[18] Jin, X., Li, L. E., Vanbever, L., and Rexford, J. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 163–174.

[19] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.-H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J., Thakkar, P., Wendlandt, D., Yip, A., and Zhang, R. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), USENIX Association, pp. 203–216.

	Scalable Fine-Grained NF Policy	Low Performance Overhead	NF's Flow Migration	Host Mobility	Incremental Deployment
NF Policy Enforcing [31, 14]	TCAM size, TCAM update speed		not discussed	not discussed	inter-domain middlebox
NF Dynamic Control [16, 34]	TCAM size, TCAM update speed	Handle packets at the controller		not discussed	inter-domain middlebox
Naming service based [41, 38]		VPN, or per packet encapsulate and identify	not discussed	not discussed	[some] new naming system new socket abstraction

Table 3: Reasons that different NF policy steering and mobility solutions failed to fulfill the properties

[20] Liu, H. H., Wu, X., Zhang, M., Yuan, L., Wattenhofer, R., and Maltz, D. zUpdate: Updating Data Center Networks with Zero Loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 411–422.

[21] Mahajan, R., and Wattenhofer, R. On Consistent Updates in Software Dened Networks. In *12th ACM Workshop on Hot Topics in Networks (HotNets), College Park, Maryland* (November 2013).

[22] Marmorstein, R., Kearns, P., and William, T. C. An Open Source Solution for Testing NATd and Nested iptables Firewalls. In *19th Large Installation Systems Administration Conference (LISA 05* (2005), pp. 103–112.

[23] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), USENIX Association, pp. 459–473.

[24] McCanne, S., and Jacobson, V. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (Berkeley, CA, USA, 1993), USENIX'93, USENIX Association, pp. 2–2.

[25] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[26] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing Software-defined Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 1–14.

[27] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1999), SOSP '99, ACM, pp. 217–231.

[28] Natal, A. R., Jakab, L., Portolés, M., Ermagan, V., Natarajan, P., Maino, F., Meyer, D., and Aparicio, A. C. LISP-MN: mobile networking through LISP. *Wireless personal communications 70*, 1 (2013), 253–266.

[29] Nikander, P., Gurtov, A., and Henderson, T. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *Communications Surveys Tutorials, IEEE 12*, 2 (Second 2010), 186–204.

[30] Nordström, E., Shue, D., Gopalan, P., Kiefer, R., Arye, M., Ko, S., Rexford, J., and Freedman, M. J. Serval: An End-Host Stack for Service-Centric Networking. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), USENIX, pp. 85–98.

[31] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13, ACM, pp. 27–38.

[32] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), USENIX, pp. 399–412.

[33] Rajagopalan, S., Williams, D., and Jamjoom, H. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 1:1–1:15.

[34] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), USENIX, pp. 227–240.

[35] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, ACM, pp. 323–334.

[36] Scarfone, K., and Mell, P. Guide to intrusion detection and prevention systems (idps). *NIST special publication 800*, 2007 (2007), 94.

[37] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), USENIX, pp. 323–336.

[38] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 13–24.

[39] Snoeren, A. C., and Balakrishnan, H. An End-to-end Approach to Host Mobility. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking* (2000), MobiCom '00, ACM, pp. 155–166.

[40] Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects* (2005).

[41] Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., and Shenker, S. Middleboxes No Longer Considered Harmful. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 15–15.

[42] Zave, P., and Rexford, J. Compositional network mobility. In *Verified Software: Theories, Tools, Experiments*. Springer, 2014, pp. 68–87.

[43] Zhuang, S., Lai, K., Stoica, I., Katz, R., and Shenker, S. Host Mobility Using an Internet Indirection Infrastructure. *Wirel. Netw. 11*, 6 (Nov. 2005), 741–756.