Loss-freedom, Order-preservation and No-buffering: Pick Any Two During Flow Migration in Network Functions

Radhika Sukapuram*, Ranjan Patowary†, Gautam Barua‡
*‡ Indian Institute of Information Technology Guwahati, India † Central Institute of Technology Kokrajhar, India
Email: *radhika@iiitg.ac.in, †r.patowary@cit.ac.in, ‡gb@iiitg.ac.in

***Abstract*—Network Functions (NFs) provide security and optimization services to networks by examining and modifying packets and by collecting information. When NFs need to be scaled out to manage higher load or scaled in to conserve energy, flows need to be migrated from one instance of an NF, called the source instance, to another, called the destination instance, or from one chain of instances to another chain of instances. Before flows are migrated, the state information associated with the source instance needs to be migrated to the destination instance. Packets that arrive at the destination instance meanwhile need to be either buffered or dropped until the state information is migrated, for correct functioning of some stateful NFs, while for some others, the destination NF may continue to function. We define the properties of Loss-freedom, where the flow migration system does not drop packets, No-buffering, where it does not buffer packets, and Order-preservation, where it processes packets in the same manner as the source NF, if there was no flow migration. We formalize these properties, for the first time, and prove that it is impossible for a flow migration algorithm in stateful NFs to guarantee satisfying all three of the properties of Loss-freedom (L), Order-preservation (O) and No-buffering (N) during flow migration, even if messages or packets are not lost. We demonstrate how existing algorithms operate with regard to these properties and prove that these properties are compositional.**

***Index Terms*—Network Functions, Flow Migration, Loss-freedom, Order-preservation, No-Latency, Service Functions**

I. Introduction

Middleboxes provide security to a network and optimize network services by inspecting, modifying and forwarding packets. For optimum performance, independence from expensive and proprietary hardware, and easy scaling, there is extensive research on implementing them in software on high performing servers, as Network Functions (NFs) [1] (also called Service Functions). In addition to middleboxes, it is also convenient to implement network components such as Evolved Packet Core (EPC) or Mobility Management, in software [2], or application specific functions such as Video Decoding, Screen Rendering etc. [3], for the same reasons.

NFs are chained in a partial order, describing a sequence in which packets may traverse them. These chains are called Service Function Chains (SFCs) [1]. When an NF instance is scaled out to reduce its load, the flows existing on that NF instance are migrated to another instance of that NF [4]–[10], as early as possible. Flows may be migrated from one or more NF instances, even from an entire SFC [11], possibly simultaneously. Flows will also need to be moved when NF instances are consolidated. The NF instance from which a flow is being migrated is called a *source NF* instance, or source NF, and the NF instance to which it is being migrated is called a *destination NF* instance, or destination NF.

A *flow* is a finite sequence of packets that have the same values for a set of headers[1]. The set of values associated with one or more structures or objects internal to an NF and related to a flow (or a set of flows) forms a *state* [4]. For correct functioning, NFs are required to maintain state information on a per-flow basis or across flows. For example, a stateful firewall may be used to ensure that connections can be initiated only from inside the network that the firewall protects. When such a connection is initiated, the firewall keeps state information corresponding to this flow so that when a packet belonging to this flow comes from outside the network, it can be allowed in. Therefore when flows are migrated from one NF to another, the states associated with these flows also need to be migrated. In fact, when flows traverse a set of switches that maintain state (for example, programmable switches), the states associated with these switches also need to be migrated [12].

When flows are migrated from an NF instance to another, the destination NF receiving packets of the migrated flow may not have enough information to process the packets if associated state is not migrated before the packets are received (such as in a firewall). Packets such as these must not be dropped on account of flow migration. We call this property *Loss freedom*. It is not unusual for packets to be dropped on networks — however, a flow migration resulting in packet drops could be used to trigger attacks, besides increasing application latency. In SFCs (modeled as directed graphs), there could be situations where packets are cloned and forwarded to off-path NFs [6]. If a packet that is traversing a set of switches along its intended path is dropped, it may be re-transmitted depending on the protocol used or the applications involved. However, if the packets that are cloned and sent to off-path NFs are dropped, they are irrecoverably lost and would never reach the intended NF, potentially causing security issues [13].

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

[1]For example, the set of headers may be the 5-tuple of source IP address, destination IP address, source port, destination port and protocol

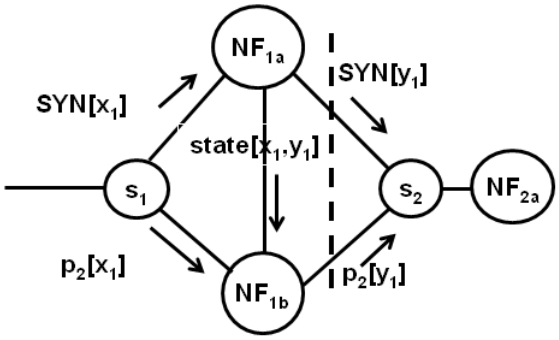

Fig. 1. Example of a Network Address Translator

Therefore preserving Loss-freedom is of interest.

Another option to deal with packets that arrive at the new NF instance before state information is migrated to it is to buffer them at the controller [4], at the NFs [14] or even at switches that have large buffers [15]. Alternatively, states are located on a remote common server [16] and they are updated by all NFs - this would also entail packet buffering. Another alternative is to implement states in a distributed manner [5], [17], also requiring packet buffering to retrieve or update states. NF instances may cache states as and when required [6]. If states are unavailable in caches, packets will need to be buffered until states are available. Packet buffering causes latency, which may cause violation of SLAs, and increased storage requirements. While the seminal work of OpenNF [4] requires packets to be buffered at the controller, TFM [14] reduces the total buffer requirements during migration and buffers packets at the NFs. For example, for 100 flows, each of 1000 pps, 139792 packets are buffered for OpenNF while for TFM, only 4770 packets are buffered [14]. Other attempts to reduce buffer sizes during migration are explained in detail in Section VII. Ideally, there should be no need to buffer packets at all. We call this property *No-buffering*.

Consider an NF such as a Network Address Translator (NAT) in Fig. 1. Assume that a NAT instance NF_{1a} receives a SYN of a TCP flow from its internal network. It updates the source address and source port number of SYN, denoted as y_1, stores these values against the original source address and port number, denoted as x_1, and sends the SYN out. After the TCP handshake is over, when a packet p_1 arrives from the internal network, NF_{1a} updates the source address and source port number to the one stored for this flow. If the flow is migrated to NF_{1b}, another NAT instance, and the order of arrival at NF_{1b} of the stored state information at NF_{1a} (denoted as $state[x_1, y_1]$), and the next packet of the flow p_2, is not maintained, then p_2 will need to be buffered or dropped. Thus, for 5-tuple-modifying (source IP address, source port, destination IP address, destination port and protocol) NFs such as NATs, the state of the destination NF must be consistent with that of the source NF before it processes the packets received. *In other words, the flow migration system containing the source and destination NFs must process packets of a flow in the same manner that the source NF would have processed them in the absence of migration*. If so, the flow migration system is said to be *Order-preserving*.

Consider another example. Deep Packet Inspectors (DPIs) such as signature-based Intrusion Detection Systems and exfiltration checkers require total ordering of packets within a flow [18] because they can be used to classify flows based on patterns found in the payloads of packets. For example, packets that contain requests for small objects in a database could be sent over low-latency paths while requests for large objects may be sent over paths with high bandwidth. A pattern that is being searched may be spread across the payloads of two or more packets. Therefore, if packet-order is not maintained, packets will need to be buffered and processed in the correct order. If deep packet inspection is being done for purposes of security, it is important to perform matches that cross packet boundaries. If that is not done, an adversary could split malicious content across packet boundaries to evade detection.

Assume that NF_{1a} and NF_{1b} in Fig. 1 are DPIs. In order to process packets in order, the state of the destination NF must be consistent with that of the source NF before it processes packets received by it and it must receive packets in order. Re-ordering packets reaching NF_{1b} can be a strategy by an adversary to increase the load on the buffer in NF_{1b} and render it dysfunctional. In addition to preserving Order, if a flow migration system requires that packets are processed by it in the order in which they arrive at its input, it is said to preserve *Strict-ordering*.

Suppose NF_{2a} in Fig. 1 is a Deep Packet Inspector while NF_{1a} and NF_{1b} are instances of a NAT. For packets to be received in order at NF_{2a}, the flow migration system containing NF_{1a} and NF_{1b} must not re-order packets. That is, the packets that cross the boundary of the system, shown by the dotted line, must be in the order of timestamps (timestamped by the flow migration system at its entry), which is another property of flow migration. We call this property *External order* (E). This property ensures that a flow migration system *outputs* packets of a flow in the same manner that the source NF would have *output* them in the absence of migration. Suppose there are packets that do not cause state updates in the input and a subset of them are processed by the source NF and the remaining by the destination NF. These packets may not be output necessarily in the order of their entry into the network, even if the flow migration is Order preserving or Strict-order preserving.

Existing solutions for flow migration achieve Loss-freedom (L) and Strict-Order-preservation (SO) [4], [6]–[10] or only Loss-freedom with reduction in the buffer size required [13] or provide workable solutions that guarantee low latency, minimum buffering and weak but sufficient consistency [5], [17]. There is no solution that achieves No-buffering (N), and no solution discusses preserving the order of packets that exit a flow migration system (E). All the properties are desirable. *This is the first work that formalizes these properties, as far as we know*. These are the questions we wish to explore in this paper: *Is it possible to have a flow migration algorithm that guarantees to preserve Loss-freedom (L), Order (O) and No-*

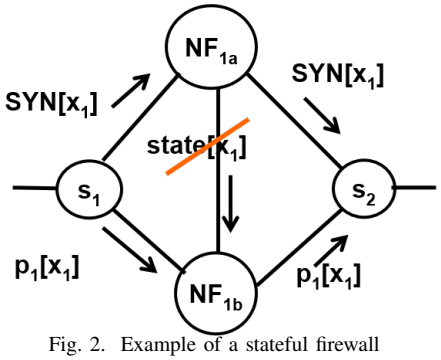

Fig. 2. Example of a stateful firewall

buffering (N) simultaneously? What are the characteristics of each of these properties? How are External-order preservation (E) and SO related to O?

We illustrate that the three properties L, O and N cannot be *guaranteed to be* satisfied during flow migration, even if packets and messages are not lost. The same is true for L, SO and N too. The properties to be preserved will depend on the NFs, as will be illustrated in the next section. Answering the questions posed is useful because the algorithm to be used for flow migration may be tailored for the properties that must be preserved, which in turn depend upon the set of NFs participating in the migration, instead of the general "one-size-fits-all" approach followed currently. We prove that all the properties are compositional. Whenever a property of flow migration is referred to, the first letter is capitalized.

II. Motivating Examples

We cannot preserve all of Loss-freedom (L), Order-preservation (O) and No-buffering (N) during migration of flows from a set of NF instances to another set of NF instances when the NFs are stateful. Consider Fig. 1, regarding flow migration from one instance of a NAT NF_{1a}, to another instance, NF_{1b}. If we forgo L at NF_{1b}, upon receiving p_2, $NF1_{1b}$ may drop p_2 and subsequent packets until it gets the required state information, thus preserving Order (O) and No-buffering (N). If we forgo N at NF_{1b}, NF_{1b} (or a switch connected to it, the controller or any other node) may buffer p_2 and subsequent packets until it receives $state[x_1, y_1]$ and then it can forward p_2, thus preserving Order of packets (O) and Loss-freedom(L). If we forgo Order-preservation (O), NF_{1b} can send p_2 immediately, preserving L and N, but it will not be meaningful for this example. For this, let us consider the example of a stateful firewall, in Fig. 2.

A stateful firewall NF_{1a} is protecting a network. Bi-directional flows can be originated only from inside the network that the firewall protects — once a flow originates from inside and the firewall updates its state, traffic can flow in both directions. Assume that a flow that has originated inside the network (SYN is already sent through the source NF instance) needs to be migrated from a firewall instance (NF_{1a}) to another instance (NF_{1b}). Assume that $state[x_1]$, where x_1 denotes the identifier of a flow, is not yet migrated to NF_{1b}. When a packet p_1 is received at NF_{1b}, it can forward p_1 as it

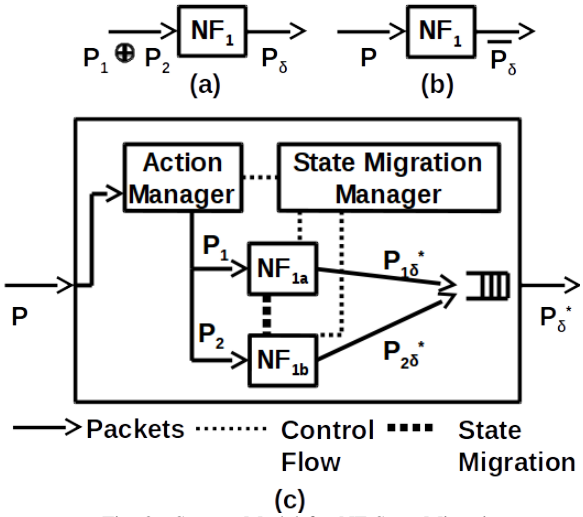

Fig. 3. System Model for NF State Migration

can check if it has originated from within the network. NF_{1b} can forward packets of this flow, destined inward, from outside the network too. If no state associated with x_1 is received from NF_{1a} after waiting for a suitable amount of time, NF_{1b} can drop further packets belonging to x_1, if it receives any. Packets can continue to be forwarded while waiting for state information. Thus L and N are preserved here, but not O.

To illustrate one of the ways in which the impossibility result may be used, consider the example of an SFC which consists of a DPI followed by a NAT. Let us assume that the load on a particular NAT instance in this chain is high and that some of the flows need to be moved to a new NAT instance such that O is not to be preserved, as described above. Let us also assume that buffer availability is low at the controller or at any of the NAT instances. For this, an algorithm that forgoes O and preserves L and N may be devised. Another application of the result is devising an algorithm that weakens one or more properties such that preserving all the weakened properties is guaranteed. How to formally specify weakening each property or set of properties such that all the weakened properties are preserved, is an open problem. How existing algorithms attempt weakening of the stated properties to achieve their goals is described in Section VII.

III. Formal Model

States that are updated exclusively by a flow are discussed in this paper and common states updated by more than one flow (for example, counters) are outside the scope of discussion. If a set of flows are migrated together because they share common states, such as proxies that consolidate multiple TCP connections into one connection, that is within the scope of discussion.

During migration of a flow, an NF is a tuple $M = (\mathcal{Q}, \mathcal{P}, f, q_0, F, M, U)$ where

- $\mathcal{Q}$ is the set of states of the NF for a particular flow (Note: a state is a set of values).

TABLE I
LIST OF SYMBOLS USED

Symbol	Meaning
fl	The flow being migrated
NF_{1a} (NF_{1b})	Source (Destination) NF instance
NF_1	An ideal NF
P	The sequence of packets entering a flow migration system
P_1 (P_2)	The sequence of packets entering the source (destination) NF
$P_1 \oplus P_2$	The actual sequence of packets entering the source and destination NFs, in the order of timestamps
p_i	A packet belonging to P
s_i	A sequence of output packets corresponding to an input packet
q_i	An NF state
$m(k, q_i)$	Message sent to NF instance k with state q_i
P_δ	Output of an ideal NF when the input is $P_1 \oplus P_2$
$\overline{P_\delta}$	Output of an ideal NF when the input is P
Q_1 (Q_2)	The finite sequence of states created on the source (destination) NF
Q_{NF_1}	The finite sequence of states created on the ideal NF when its input is $P_1 \oplus P_2$
Q_1^* (Q_2^*)	A subset of Q_1 (Q_2)
$P_{1\delta}^*$ ($P_{2\delta}^*$)	The sequence of packets output by the source (destination) NF
P_δ^*	The sequence of packets output by the flow migration system, in order of being output: $P_{1\delta}^* \boxplus P_{2\delta}^*$
$\widehat{X}$	A set representing the elements of X

- q_0 is the initial state, that is, the state of the NF before migration begins
- $\mathcal{P}$ is a set of input packets
- $\mathcal{P}_\delta$ is a set of output packets, including a $\triangledown$, indicating no packet (explained ahead).
- M is a set containing a message m, including a $\perp$, indicating no message (explained ahead). m contains the state of the NF sending the message and the identifier of the destination NF.
- $f : Q \times \mathcal{P} \rightarrow (\mathcal{P}_\delta \times Q \times M)$ is the transition function.
- $U : M \times Q \rightarrow Q$ is the function that receives the message sent by another NF and updates the current NF.

Symbols other than the ones defined above are listed in Table I.

Let fl be the flow that is being migrated. Fig. 3 depicts migration of this flow from NF_{1a}, an instance of an NF called NF_1, to another instance of the same NF, NF_{1b}. Let us assume that both NF_1 (Fig. 3 (a)) and NF_{1a} have the same starting state, q_0. The system for NF migration, referred to as the Flow Migration System (FMS), consists of an Action Manager (AM), a State Migration Manager (SMM) and the source (NF_{1a}) and destination (NF_{1b}) NFs. What is described is a logical organization of FMS — in the general case, the source and destination NFs are on different servers connected over a network. AM manages the following actions on packets: drop, buffer and nop (indicating no operation). AM also accords a logical timestamp to each packet when it is received[2]. SMM decides the NF instance to which a packet p_i of the flow fl must be sent to, and manages migration of state, if any, from the source to the destination NF. To aid this, it may instruct AM to drop or buffer packets or forward packets to an NF instance, using messages to AM, NFs, both, and/or other network nodes. AM will forward a packet without delay, unless it is asked to buffer or drop it. SMM may also instruct NFs to start and stop updating each other's states. AM and SMM may be co-located with NFs, switches or the controller, or may be independent entities.

Let $\prec$ be a total order on the *sequence* of packets $P = \langle p_1, p_2, ..., p_n \rangle$, belonging to fl, where $\prec$ denotes the order in which packets enter FMS. As packets $\langle p_1, p_2, ..., p_n \rangle$ enter the NF, they are transformed to $\langle s_1, s_2, ..., s_n \rangle$ (denoted as P_δ) , depending on the state at the NF. Transformation of packets is by a function $f()$, where $f(p_i, q_{i-1}) = \langle s_i, q_i, m(k, q_i) \rangle$, $1 \leq i \leq n$, where q_i denotes the state of the NF. $s_i = \langle p_{\delta_{i1}}, ..., p_{\delta_{ic}} \rangle$ denotes a sequence of zero, one or more output packets. An output packet may be obtained by altering the input packet (by changing its header, for example). If there is no change, and only one packet is output, $s_i = \langle p_{\delta_{i1}} \rangle = \langle p_i \rangle$. It is possible that an NF drops an input packet, in which case no packet ($\langle \triangledown \rangle$) will be output. Packets may get re-ordered after they enter the network. Suppose FMS receives two packets, p_2, p_1, which were re-ordered by the network before entering it. FMS itself does not change their order. It may output $\langle \triangledown \rangle$ corresponding to p_2 and $\langle p_{\delta_{11}}, p_{\delta_{12}} \rangle$, corresponding to p_1. In general, FMS may output no packet, or one or more altered or unaltered packets. It is possible that due to arrival of a packet, there is no state change in an NF, that is, $q_i = q_{i-1}$. P_δ[3] is totally ordered by $\prec$.

When a flow fl is to be migrated, SMM sends a message to the source and destination NFs involved in the migration, indicating *start* of flow migration. $m(k, q_i)$ denotes the message sent to the other NF k involved in the flow migration and q_i denotes the state to be updated in the NF that receives the message. An NF sends this during the start of migration to send the initial state q_0, and subsequently if only if and when a state change occurs during flow migration. Updation of the NF that receives the message is done by $U(m, q_x) = q_j$ — the NF receives m in state q_x and updates it to state q_j. SMM informs both the source and the destination NFs when a flow migration is complete, indicating *end* of flow migration, so that they can stop updating each others' states. If no packets are being sent to the source NF after a point in time, SMM can instruct the destination NF to stop updating the state of the source NF, in which case no message ($\perp$) will be sent.

The sequence of packets that are received at the source NF in Fig. 3(c), NF_{1a}, is denoted by P_1 and the sequence of packets that are received at the destination NF, NF_{1b}, is denoted by P_2. SMM may cause re-ordering of packets. Thus

[2]This is not a requirement on any flow migration algorithm and is only for purposes of explanation

[3]The output of NFs is a sequence of a sequence of packets. For simplicity, we denote it as a sequence of packets

P_1 and P_2 may not be in timestamp order. There may be packets in P that are neither present in P_1 nor in P_2, as some packets in P may be instructed to be dropped by SMM (if Loss-freedom is not preserved). $P_1 \oplus P_2$ denotes a sequence of packets that are present in P_1 or P_2 and ordered according to their timestamps.

Fig. 3(a) depicts an NF instance, called NF_1, that accepts the sequence of packets $P_1 \oplus P_2$ and emits a modified sequence of packets P_δ, while undergoing state transformations. Let the *sequence* of state transformations for this sequence of packets be Q_1. An NF that does not drop, buffer or re-order packets is called an *ideal* NF. NF_1 in Fig. 3(a) and Fig. 3(b) are ideal NFs. The sequence of states in NF_1, denoted by Q_{NF_1}, follows the total order $\prec$ of the sequence of packets $P_1 \oplus P_2$ or P, as the case may be. The reason for two different inputs, that is, $P_1 \oplus P_2$ and P, is explained further ahead.

Let Q_1 be the finite sequence of states created on NF_{1a} and Q_2 on NF_{1b}. SMM migrates Q_1^*, a subset of Q_1, from NF_{1a} to NF_{1b} and Q_2^*, a subset of Q_2, from NF_{1b} to NF_{1a}[4] during flow migration, depending on the flow migration algorithm employed. It may instruct AM to buffer or drop packets or both until the flow migration is complete. A subset of Q_2 is considered, as in some cases, only the latest state may need to be migrated to the destination NF whereas in some others, after a state is migrated, newer packets may cause state changes at the source NF, requiring further state migrations.

$P_{1\delta}^*$ denotes the sequence of packets output by NF_{1a} and $P_{2\delta}^*$ denotes the sequence of packets output by NF_{1b}. In general, a packet p_i, $1 \leq i \leq n$ is transformed to s_i by NF_{1a} or by NF_{1b}. However, it is likely that some packets were dropped by AM before reaching NF_{1a} or NF_{1b}, due to the state migration algorithm used. Hence some packets p_i, $1 \leq i \leq n$, may not be transformed to s_i by NF_{1a} or by NF_{1b}. $P_\delta^* = P_{1\delta}^* \boxplus P_{2\delta}^*$ denotes the sequence of packets output by FMS . The order of packets in P_δ^* is the order in which packets in $P_{1\delta}^*$ and $P_{2\delta}^*$ arrive at the output of FMS, represented symbolically by a queue in Fig 3(c). This may not be in the order of packets in P. In Fig 3(a), the input to NF_1 is the union of P_1 and P_2, the *actual* sequence of packets processed by either NF_{1a} or NF_{2a}, but in the order of their timestamps (this may not contain all the packets in P, as SMM may choose to drop some packets). Fig. 3(b) represents an ideal NF whose input is P. P is the set of all packets that may enter an FMS . The output then is $\overline{P_\delta}$.

The state migration model is applicable for all networks, whether synchronous, asynchronous or partially synchronous [19], as the model makes no assumptions with regard to these.

Definition III.1 (Order preservation (O))**.** Let Q_1 be the finite sequence of states created on NF_{1a} and Q_2 on NF_{1b} between the start and end of flow migration. Let Q_{NF_1} be the finite sequence of states created on the ideal NF due to $P_1 \oplus P_2$ (Fig. 3(a)). Migration of a flow fl from an NF instance NF_{1a} to NF_{1b} is Order-preserving iff

1) Q_1 is a prefix of Q_{NF_1} and
2) Q_2 is a suffix of Q_{NF_1}

Remark. Condition 1 (Condition 2) above ensures that all state updates in the source (destination) NF are the same as those in the ideal NF. The flow migration system *processes* packets of a flow in the same manner that the source NF would have processed them in the absence of migration.

Since Q_1, Q_2 and Q_{NF_1} are the sequence of states created due to various packets, they do not include the initial state q_0. During state migration, as soon as a state update is performed on NF_{1a}, the same update must be performed on NF_{1b} and vice-versa. All updates in NF_{1a} and NF_{1b} must be in the order in which states are updated in the ideal NF, NF_1.

In Order Preservation, the packets sent to NF_{1a} or NF_{1b} may not be in order, as long as the out of order packets do not cause state changes. This is useful for NFs where every packet need not result in a state change, such as certain NAT implementations. In such NATs, packets other than the control packets of a TCP flow need not cause state changes.

There is no requirement that to preserve Order packets must not be dropped before they reach the source or destination NF. The only requirement is that while comparing with the ideal NF, the ideal NF must also receive the same packets that the source and destination NFs receive.

As per this definition, it is possible that some states in the ideal NF are missing from the source or destination NFs. For example, Suppose $Q_1 = \langle q_1 \rangle$ while $Q_2 = \langle q_3 \rangle$ for a given set of packets, while the ideal NF has $Q_{NF_1} = \langle q_1, q_2, q_3 \rangle$. However, the destination NF cannot move to state q_3 unless the source NF has moved to q_3 first or unless the destination NF itself has moved to q_2 before moving to q_3. Therefore, we rule out this possibility and do not impose this condition: $\widehat{Q_1} \cup \widehat{Q_2} = \widehat{Q_{NF_1}}$ where $\widehat{X}$ denotes a set containing the members of the sequence X.

Fig. 4(a) is an example of an Order-preserving flow migration algorithm. This shows the sequence of states (in dotted boxes) created by a sequence of packets $\langle p_1,\ p_2,\ p_3,\ p_4,\ \rangle$. Fig. 4(b) shows the sequence of steps during state migration where Order is preserved and Fig. 4(c), the timeline of events.

State migration begins, with state q_0 sent to NF_{1b} (1). This is required if no packet arrives for some time and migration can end before the arrival of any packet. p_1 arrives at NF_{1a} (2). The state of NF_{1a} changes to q_1 (3), a message $m(b, q_1)$ is sent to the destination NF ($b = NF_{1b}$) (4) which updates its state (5). A packet p_2 arrives at NF_{1b} (6) which updates its state (7), sends a message $m(a, q_3)$, with a denoting NF_{1a} (8) and updates the state of NF_{1a} (9). Since state migration ends at this point (10), no new packets will reach NF_{1a} and no messages to update NF_{1a} need to be sent to it. Packet p_3 arrives at NF_{1b} (11), the state of NF_{1b} is updated to q_3 (12), then p_4 arrives (13) and the state of NF_{1b} is further updated to q_4 (14). Q_1 is $\langle q_1, q_2 \rangle$. Q_2 is $\langle q_1, q_2, q_3, q_4 \rangle$. Q_{NF_1}

[4]All flow migration algorithms, as far as we know, stop sending packets to the source NF once flow migration begins. The model allows state movement in both directions for generality.

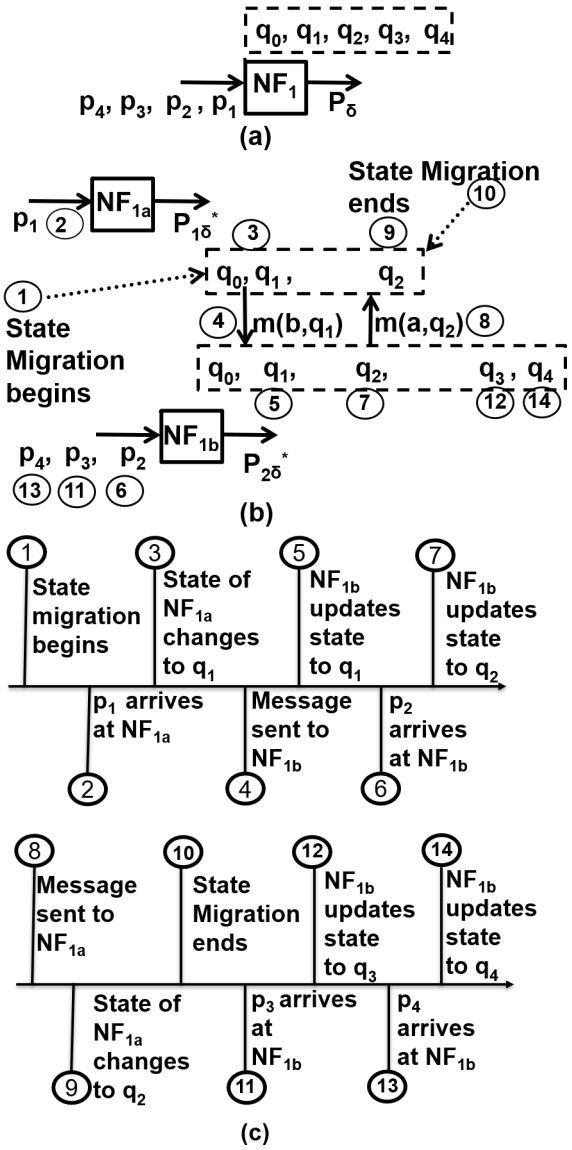

Fig. 4. Example for preserving Order: NFs and their states

is $\langle q_1, q_2, q_3, q_4 \rangle$. Thus Q_1 is a prefix of Q_{NF_1} and Q_2 is a suffix of Q_{NF_1} .

Definition III.2 (External-order-preservation (E))**.** The migration of P to NF_{1b} is External-order preserving iff: P^*_δ is a subsequence of $\overline{P_\delta}$.

Remark. P^*_δ is a *subsequence* of $\overline{P_\delta}$ in the definition of External-order-preservation to account for the fact that packets may be dropped during flow migration. Also, the fact that packets may arrive out of order at NF_{1b} needs to be accounted for. That is why $\overline{P_\delta}$ is considered instead of P_δ.

It is possible that Order is preserved, yet External-order is not preserved, as the set of packets that causes loss of order does not effect any state change. Suppose there are two packets in a flow, p_1 and p_2, entering the flow migration system in that order. p_1 enters NF_{1a} while p_2 enters NF_{1b}. They cause no state changes. NF_{1b} outputs the transformed packet first, followed by NF_{1a}. Thus the output of the flow migration system is $\langle s_2, s_1 \rangle$. This preserves Order but not External order.

As another example, assume that FMS re-orders only two packets p_1 and p_2 of fl. Let NF_{1b} receive them as $\langle p_2, p_1 \rangle$. Assuming that these two packets do not cause any state changes, Order will be preserved. However, the output of FMS will be $\langle s_2, s_1 \rangle$, not preserving External order.

In order that packets arrive at the DPI (NF_{2a}) in Fig. 1, in order, it is not sufficient that Order is preserved by the flow migration system for NAT (consisting of NF_{1a} and NF_{1b}). It is necessary for External-order to be preserved.

Order preservation ensures that a flow migration system *processes* packets of a flow in the same manner that the source NF would have processed them in the absence of migration. However, packets may not be *output* in the same manner that the source NF would have output in the absence of migration. This idea is captured in External-order preservation.

Definition III.3 (Loss-freedom)**.** Let $\widehat{P^*_\delta}$ represent a set containing the elements of P^*_δ, and $\widehat{\overline{P_\delta}}$ represent a set containing the elements of $\overline{P_\delta}$. The migration of P to NF_{1b} is Loss-free iff: $\widehat{P^*_\delta} = \widehat{\overline{P_\delta}}$.

Definition III.4 (No-buffering)**.** The migration of P to NF_{1b} is No-buffering preserving iff: no packet $p_i \in P$ is buffered by Action Manager.

Remark. If a packet is dropped by Action Manager, it preserves No-buffering as per this definition.

IV. Impossibility Result

Theorem 1. It is impossible for a flow migration algorithm in stateful NFs to guarantee satisfying all three of the properties of Loss-freedom (L), Order-preservation (O) and No-buffering (N) during flow migration, even if messages or packets are not lost.

Proof. Given any flow fl containing many packets, consider only two consecutive packets, p_1 and p_2. fl has to be migrated from NF_{1a} to NF_{1b} , as shown in Fig. 3 (c). Assume that p_1 causes state changes. If p_1 does not cause a state change, the sequence of packets before p_1, if any, contributes to state q_0, which will be migrated at the start of migration. The sequence of packets arriving after p_2, if any, will be handled by NF_{1b} after the completion of migration.

Let p_1 reach AM at t_1 and let p_2 reach AM at t_2 and let AM forward these packets instantaneously, after receiving instructions from SMM, also instantaneously. It takes a finite time T_s for state to get updated at NF_{1a} after receiving a packet (if p_1 does not cause a state change, $T_s = 0$.), since we make no assumptions about link or processing speeds. T_m indicates the time to migrate this state from one NF to another (using the message m). Let $T = T_s + T_m$.

We enumerate all possibilities below.

1) Suppose $t_1 + T \leq t_2$. In such a case, SMM will ask AM to forward p_2 to NF_{1b}, and all three of L, N, and O will be preserved. The migration will be complete and all packets of the flow, if any, will now go to NF_{1b}.

2) But, when $t_1 + T > t_2$, then when p_2 arrives at AM, it can take one of only four actions, depending on instructions from SMM:
   a) SMM asks AM to buffer the packet. In this case, N is not preserved while L and O are preserved.
   b) SMM asks AM to drop the packet. In this case L is not preserved while N and O are preserved.
   c) SMM asks AM to send the packet to NF_{1b}. In this case O is not preserved, while N and L are preserved.
   d) SMM asks AM to send the packet to NF_{1a}, in which case no migration happens. Suppose the next packet p_3, if any, arrives at AM. Consider that p_3 arrives at AM at t_3 and it takes T units of time for NF_{1a} to process the latest packet that causes state change, p_1 or p_2, and migrate the state to NF_{1b}. If $t_2 + T \leq t_3$, case 1 will apply and if $t_2 + T > t_3$, any of the cases 2a, 2b, 2c or 2d will apply. If case 2d continues to apply for subsequent packets, if any, migration will be delayed indefinitely and the flow migration algorithm will fail.

Thus no algorithm exists that can guarantee satisfying all the three properties of L, O and N.

□

Remark. The theorem states that preserving all the three properties is *not guaranteed*. For some instances, all the three properties may be preserved.

Definition IV.1 (Strict Order preservation (SO))**.** To Definition III.1, the following condition is added:

3) Both P_1 and P_2 are subsequences of P

Remark. Order preservation (O) ensures that a flow migration system *processes* packets of a flow in the same manner that the source NF would have processed them in the absence of migration. *In addition to this*, to preserve SO, Condition 3 ensures that the flow migration system does not re-order packets as they *enter* the source or destination NFs. Preserving SO is not required for NFs such as NATs and preserving O is sufficient. Preserving SO does not necessarily result in packets being *output* by the flow migration system in timestamp order. For example, in Fig. 1, NF_{1b} may not be able to prevent a packet with a later timestamp from reaching the boundary of FMS (shown by the dotted line) before a packet with an earlier timestamp output by the source NF, NF_{1a}.

Corollary 1.1. Theorem 1 is applicable if Strict Order preservation is substituted by Order preservation: It is impossible for a flow migration algorithm in stateful NFs to guarantee satisfying all three of the properties of Loss-freedom (L), Strict-Order-preservation (SO) and No-buffering (N) during flow migration, even if messages or packets are not lost.

Proof. By definition, SO is a stronger property than O. Therefore, in the proof for Theorem 1, if O is not preserved, SO is not preserved.

In cases 1, 2a and 2b of Theorem 1, SO is preserved, as O is preserved, p_1 is a subsequence of $\langle p_1, p_2 \rangle$ and p_2 is a subsequence of $\langle p_1, p_2 \rangle$.

For case 2c, either the previous cases apply, or the flow migration algorithm fails.

Thus no algorithm exists that can guarantee satisfying all the three properties of L, SO and N.

□

In the next two theorems, we examine the relationship between Strict-order (SO) and External-order (E), and Order (O) and External-order.

Theorem 2. If a flow migration algorithm is External-order-preserving (E) then it preserves Strict-Order (SO).

Proof. Let there be a flow migration algorithm A that preserves E. Let Fig. 3 depict an FMS that implements A. Let $\overline{P_\delta} = \langle s_1, s_2, ..., s_n \rangle$, corresponding to the input $P = \langle p_1, p_2, ..., p_n \rangle$. Let the corresponding states on the ideal NF be $Q = \langle q_1, q_2, ..., q_n \rangle$. Let the first packet that reaches the source NF after state migration begins be p_1.

By definition of E, P^*_δ is a subsequence of $\overline{P_\delta}$. By definition of P^*_δ, its constituents $P^*_{1\delta}$ and $P^*_{2\delta}$ are subsequences of P^*_δ.

Case 1) Let us assume that A does not satisfy SO because either of P_1 or P_2 is not a subsequence of P. Let us consider that P_1 is not a subsequence of P, because two packets are re-ordered. Let us assume that these packets do not cause state changes. If packets p_i and p_{i+1}, of P_1, are re-ordered, let the corresponding outputs be $s^{'}_{i+1}$ and $s^{'}_i$.

Case 2) Let us assume that A does not satisfy SO because Q_1 is not a prefix of Q_{NF_1}. This can happen if packets p_i and p_{i+1} of P_1 are re-ordered and they cause state changes. Let us assume that the corresponding states are $q^{'}_{i+1}$ and $q^{'}_i$ and the corresponding outputs are $s^{''}_{i+1}$ and $s^{''}_i$.

Depending on the case above, the packets $s^{'}_{i+1}$ and $s^{'}_i$ and/or $s^{''}_{i+1}$ and $s^{''}_i$, will be present in $P^*_{1\delta}$ and consequently in P^*_δ, no longer making it a subsequence of $\overline{P_\delta}$, contradicting our assumption that A preserves E. A similar argument could be made if the packets of P_2 are re-ordered or if Q_2 is not a prefix of Q_{NF_1}.

Thus, if A preserves E, it preserves SO. □

Corollary 2.1. If a flow migration algorithm is External-order-preserving (E) then it preserves Order (O).

Proof. By Theorem 2, if a flow migration algorithm preserves E, then it preserves SO. But every algorithm that preserves SO preserves O as SO is a stronger property than O. Hence the proof. □

Corollary 2.2. If a flow migration algorithm is Strict-Order-preserving (SO), then it is not necessarily External-order-preserving (E).

Proof. The implication in Theorem 2 is not in the other direction, as packets not causing state changes may not be in order. Let us assume that if a flow migration algorithm is SO-preserving, then it is always E-preserving. Consider two

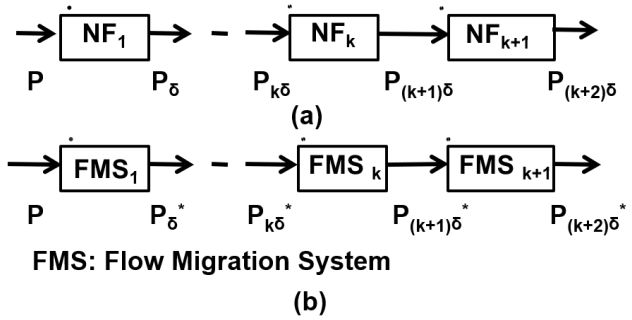

Fig. 5. A chain of NFs, where each instance pair preserves E

sequential packets p_1 and p_2 in a flow migration that preserves SO. p_1 enters NF_{1a} while p_2 enters NF_{1b}. They cause no state changes. NF_{1b} outputs the transformed packet first, followed by NF_{1a}. Thus the output of the flow migration system corresponding to $\langle p_1, p_2 \rangle$ is $\langle s_2, s_1 \rangle$. The flow migration system processes and outputs the remaining packets of fl in the order of timestamps and maintains order. Thus the flow migration system satisfies the conditions for Strict-Order-preservation, but E is violated, contradicting our assumption. □

V. Compositionality of L, O, SO and N

As stated in [20], "A correctness property P is compositional if, whenever each object in a system satisfied P, the system as a whole satisfies P". In this section, we examine whether each of L, O, SO and N is compositional.

Theorem 3. Migration of flows from a chain of NFs to another chain of NFs is External-order-preserving if for each NF instance pair it is External-order-preserving.

Proof. We prove this by induction. For an NF instance pair, when there is only 1 element in the chain (n=1, where n is the number of NFs in the chain) , as given in Fig. 3, migration of flows is External-order-preserving, that is, P^*_δ is a subsequence of P_δ, as each instance is given to be External-order-preserving (the base case). Suppose for $n = k$ this is true. Consider Fig. 5(a), illustrating a chain of ideal NFs. Suppose $P_{k\delta}$ is the input to an instance NF_k of another NF and its output is $P_{(k+1)\delta}$. Consider Fig. 5(b), illustrating a chain for flow migration systems. Suppose $P^*_{k\delta}$ is the input to another NF instance pair, the output of which is $P^*_{(k+1)\delta}$. Then $P^*_{(k+1)\delta}$ is a subsequence of $P_{(k+1)\delta}$. Assume that one more NF is added to the chain in Fig. 5(a), whose input is $P_{(k+1)\delta}$ and in Fig. 5(b), whose input is $P^*_{(k+1)\delta}$. For the instance pair newly added, the flow migration is External-order preserving, as each instance pair is order preserving. Assume that their outputs are $P_{(k+2)\delta}$ and $P^*_{(k+2)\delta}$ respectively. Since the flow migration of the last added NF instance pair is External-order-preserving, $P^*_{(k+2)\delta}$ is a subsequence of $P_{(k+2)\delta}$. If we consider a chain of $k+1$ elements, its input is P and output is $P^*_{(k+2)\delta}$ for the migration system and $P_{(k+2)\delta}$ for the chain of ideal NFs. This also implies that the chains with $k+1$ elements is External-order-preserving. Thus if a chain with k elements is External-order-preserving, a chain with $k+1$ elements is External-order-preserving. □

Note that each NF instance pair being External-order-preserving is only a sufficient condition. There could be two NF pairs where they are External-order-preserving when taken together, but not so individually.

Theorem 4. Migration of flows from a chain of NFs to another chain of NFs is (Strict-)Order-preserving if for each NF instance pair it is (Strict-)Order-preserving.

In the proof for Theorem 3, wherever each instance pair is assumed to be External-order-preserving, we can assume them to be (Strict-)Order-preserving, due to (Theorem 2) Corollary 2.1. Theorem 4 follows.

Theorem 5. Migration of flows from a chain of NFs to another chain of NFs is Loss-free if and only if for each NF instance pair it is Loss-free.

The proof is similar to that of Theorem 3 and is omitted for brevity. For the "only if" part, the proof is given below.

Proof. To prove the above, it can be proven that if migration of flows from a chain of NFs to another chain of NFs is Loss-free, then for each instance pair it is Loss-free. Consider a chain of ideal NFs, whose input is P and output is $\overline{P_\delta}$. Next, consider a chain of n source NFs from which flows need to be migrated to a chain of n destination NFs. The input to these chains is P and the output is $P^*_{n\delta}$.

Suppose for the kth instance pair flow migration is not Loss-free and j packets are dropped. For the kth pair, $\widehat{P^*_{k\delta}} \neq \widehat{\overline{P_{k\delta}}}$, where $\overline{P_{k\delta}}$ refers to the output of the corresponding ideal NF. However, $P^*_{k\delta}$ is the input to the next NF instance pair. Thus, at the end of the chain, after n instance pairs, a total of j packets are dropped and therefore $\widehat{P^*_{n\delta}} \neq \widehat{\overline{P_\delta}}$. Therefore if there exists an instance pair which is not Loss-free, migration of a flow instance to another chain is not Loss-free. □

Theorem 6. Migration of flows from a chain of NFs to another chain of NFs is No-buffering preserving if and only if for each NF instance pair it is No-buffering preserving.

The proof is similar to that of Theorem 5 and is omitted for brevity.

Thus if each of L, O, N, SO and E is satisfied within a flow migration system, each will be satisfied in a larger system composed of smaller systems. For example, if a set of flows is migrated from one SFC to another and L is preserved by each of the flow migration systems in the SFC, L is preserved by the SFC itself and there is no loss of packets while migrating the entire SFC. Thus compositionality of these properties is useful in a service chain migration.

VI. Discussion on stateless nodes

If an NF is *stateless*, no packets cause state changes. This is a general case of a subset of packets of P not causing a state change. In this case, packets may not be output in timestamp order. But O and SO are about processing being in order, not about output packets being in order. Therefore O and SO are trivially preserved in stateless NF migration and L and

TABLE II
PRESERVATION OF L,O, SO AND N IN ALGORITHMS

Work	*L*	*O*	*SO*	*N*
OpenNF [4], Buffering at source NF [7], SliM [8], SHarP [9], Controller forwarding and Tagging based solutions [10], CHC [6], TFM [14]	✓	✓	✓	✗
Packet reprocessing and P2P transfer [13]	✓	✗	✗	✗

N will be preserved if no packets are dropped or buffered, respectively. E therefore may not always be preserved as packets may be dropped.

In Software Defined Networks (SDNs), flows traversing a path can be moved to another path for reasons of traffic engineering, routing due to failures etc. The set of switches that are traversed by flows that need to be diverted form the *old path* of a flow. A subset of switches in the network need to have new rules installed or rules modified or deleted to divert flows from the old path to a *new path*. If switch updates are not performed carefully, packets may be lost or they may loop. In network updates, a per-packet consistent (PPC) update is one where a packet traverses either the old path or the new path but not a combination of both [21]. In a PPC update, packets do not get lost nor do they loop. In a per-flow consistent (PFC) update, a flow traverses either the old path or the new path, but not a combination of both [21].

In existing PPC update algorithms involving stateless nodes [22]–[27], all of L, O, SO and N are preserved. However, E is not preserved. Thus, in Fig. 1, assuming NF_{1a} and NF_{1b} are stateless, a PPC update from $s_1-NF_{1a}-s_2$ to $s_1-NF_{1b}-s_2$ will not preserve E. In fact, for these algorithms, preserving E is not a consideration. However, in a PFC update, the question of flow migration does not arise, by definition.

VII. RELATED WORK

Flow migration algorithms: Table II illustrates the properties that some of the flow migration algorithms preserve, indicated by a "✓". No algorithm guarantees to preserve all of the properties. The solutions in the first row of Table II attempt to improve migration time while reducing the amount of packets buffered. While state is being transferred from the source NF (sNF) to the destination NF (dNF) during flow migration, incoming packets are buffered at the controller in the seminal work of OpenNF [4] — thus N is not preserved. No packets are sent to the sNF after migration begins and after state transfer is completed, the packets buffered at the controller are sent in their arrival order to the dNF, preserving O and SO. As no packets are dropped, L is preserved. In another solution [7], where the goal is to improve the security of transfer, they are buffered at the sNF. To improve migration time, SliM [8] transfers state using statelets instead of packets, from the sNF to the dNF, and buffers packets at the dNF during state transfer.

In SHarP [9], packets are buffered at the controller until appropriate rules are installed in the *ingress switch* (a switch common to both the sNF and the dNF) to route the flows to be migrated, to the dNF. After that, packets are buffered at the dNF until the packets buffered at the controller are routed to the dNF, ensuring that the buffer size required at the controller is constant and is independent of the state migration time. Thus all properties except N are preserved. In [10], there are two solutions. The first requires buffering at the controller and at the dNF, and takes care of a race condition in OpenNF, while preserving properties other than N and reducing the migration time. In the second solution, the ingress switch is instructed to tag packets of the flows to be migrated and to send packets to both sNF and dNF, from a certain instance in time. The tagged packets are dropped by the sNF and are buffered by the dNF. The controller migrates state from the sNF to the dNF. After state migration is completed, the ingress switch is instructed not to tag packets and to send them only to the dNF. The dNF processes the buffered packets before it processes the packets that arrive after migration (which are not tagged). N is not preserved, while the other properties are preserved. The solution is scalable, as the controller is not a bottleneck.

In [13], the dNF imports a snapshot of state from the sNF. The sNF continues to receive packets and update its state (packet re-processing). It forwards a copy of these packets to the dNF over a P2P connection to update the state of the dNF, with a do-not-output flag, to prevent the dNF from sending out any packets. The sNF keeps updating its states until the dNF synchronizes its state with the sNF. Though the controller is not a bottleneck, the order of packets may not be preserved and it is possible that packets will need to be buffered at the dNF while the dNF continues to update its state. In TFM [14], the sNF migrates existing state to the dNF and the ingress switch forwards incoming packets directly to the dNF. The dNF buffers them, as it is yet to process those packets that reach the sNF meanwhile (the in-flight packets). The sNF is instructed to tag the in-flight packets so that the dNF can process them before processing the packets that it has buffered. TFM requires no buffering at the controller. It also reduces the buffering at the dNF due to reduced migration times and due to an overloaded sNF processing less packets, while preserving L and O.

Certain algorithms assume state to be a distributed object. In general, these algorithms are not focused on preserving the properties in Table II, with the notable exception of CHC [6], and are more focused on improving NF performance. In CHC [6], state is stored external to NFs and the retrieved states may be cached, and O, SO and L are guaranteed. S6 [5] stores states using a Distributed Hash Table and identifies an owner for each state. While packets are processed in order at a given NF instance, during migration of a flow, none of the properties in Table II is guaranteed to be preserved. Using fast memory access schemes, it is possible to use optimized shared memory systems [28] for managing distribution of states [17] to achieve fast state synchronization, thus reducing the amount of buffer required and buffering time. However, preservation of SO or O or L is not guaranteed. These solutions [5], [6], [17] aim to minimize buffer space and time delay, but buffering is not eliminated as a packet arriving at the destination NF needs

to wait for relevant state to be retrieved from where it exists, which is possibly another NF [5], [17] or an external store [6].

Properties of networks: The properties of services in distributed systems can be classified broadly as safety and liveness properties [29]. In general, there is a tradeoff between the two. The CAP theorem states that it is impossible for a web service to provide Consistency, Availability and Partition-tolerance, even when messages are not lost [30] and this is generalized and proved for both asynchronous and partially synchronous systems [31]. Later formulations state that if there is no network partition, a system can be both consistent and available and if there is a network partition, a system can be either consistent or available [32]. Abadi et al. [33] argue that there is a tradeoff between Consistency and Latency, even in the absence of partitions, and also discusses a more complex tradeoff involving Consistency and Latency in the presence of partitions and involving Consistency and Latency [33] (abbreviated as PACELC), all for distributed databases. Consistency in the above discussions is defined to be Linearizability in some formulations [31], [34] and One-copy Serializability (1SR) [35], [36] in some others.

In CAP theorem for networks [37], Consistency of networks refers to enforcement of specific policies, Availability refers to eventual delivery of packets to end hosts as long as a path exists and Partition Tolerance refers to the network continuing to operate in the presence of partitions. As per this, (for the Software Defined Network (SDN) model in the paper) all the three cannot be achieved for policies referencing two or more entities, where an entity identifier is tied to a particular host (whereas an address associated with a host may change due to migration). The control network that connects controllers is an out-of-band network that is distinct from the data network and a partition refers to a failure in the control network. To illustrate the theorem, assume that a policy references two entities A and B under two different controller domains and prevents communication between A and B. A mechanism implementing this policy would cause the controller in one domain to resolve the address of an entity in another domain, which is impossible if the control network is partitioned. However, packets will be delivered to end hosts and the network will continue to operate in the presence of partitions (satisfying A and P but not C). Our paper is on properties during flow migration, which is different from the above.

Network switches can store states and based on these states, decisions regarding packets may be taken. While SNAP [38] places such states in only one switch per flow path, in another work [39], states are replicated optimally, with the replicas maintaining eventual consistency. However, these do not address consistency issues when flows are migrated, formalization of which is what this paper is about. Swing [12] discusses movement of state from one switch to another using only the data plane. However, this does not guarantee conservation of any of the properties discussed in this paper.

Planned configuration changes in SDNs can be achieved by per-packet consistency (that guarantee that a packet is processed only by one network configuration) [21], [22], [40] and per-flow consistency (a flow is processed only by one network configuration) [21], [41]. Preserving per-packet consistency is insufficient during flow migration involving stateful nodes as these do not consider migrating states across such nodes. Waypoint enforcement during network updates [42] ensures that before and after migration, flows continue to be routed through the same set of waypoints, which may be stateful nodes. If per-flow consistency is preserved, flows are not migrated at all, by definition.

Reachability (a node is reachable from another node) [43], [44], waypoint enforcement [42], congestion-freedom and loop-freedom (packets never loop) [45] are some of the other network properties studied in the literature on network verification. In contrast to the above safety properties, a method to verify liveness properties such as "a malicious host is eventually detected" [46] is also available. A survey on verification in stateful networks has further details [47]. These do not discuss flow migration from a set of stateful nodes to another while preserving these invariants.

VIII. Conclusions

For the first time, we have formalized flow migration from a stateful source Network Function instance to a stateful destination Network Function instance and defined several properties that characterize flow migration. We have investigated the properties of Loss-freedom (L), Order (O), Strict-order (SO) and No-buffering (N) during flow migration and proved that it is impossible for a flow migration algorithm for stateful NFs to guarantee all the three properties of L,O and N, or L, SO and N simultaneously even if messages and packets are not lost. The intuition that the flow migration system itself does not re-order packets is captured by the property of External-order preservation (E). We define and prove what it means for a flow migration to preserve External-order (E) and establish that External-order-preservation is a stronger property than order preservation (O). We also prove that all the properties are compositional.

Using the above, algorithms may be designed that chooses the properties to preserve, for migrating flows from one NF to another or across chains of NFs, or in general, across a chain of stateful nodes such as programmable switches. In addition, we hope that this spurs exploration of a new line of research, such as investigating whether weakening some of these properties may be beneficial for certain NFs, and once weakened, how they may be related to each other and whether there are additional properties of interest. We plan to explore the theory of migrating flows further, in different contexts, such as replication on a large scale. We believe that properties such as availability and fault-tolerance of NFs also need to be examined in this light.

IX. Acknowledgements

We are grateful to our shepherd Dongsu Han and the anonymous reviewers for their helpful feedback.

REFERENCES

[1] B. Yi, X. Wang, K. Li, S. k. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212–262, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128618300306

[2] A. Jain, N. Sadagopan, S. K. Lohani, and M. Vutukuru, "A comparison of sdn and nfv for re-designing the lte packet core," in *NFV-SDN*. IEEE, 2016, pp. 74–80.

[3] L. Qu, M. Khabbaz, and C. Assi, "Reliability-aware service chaining in carrier-grade softwarized networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 558–573, 2018.

[4] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2015.

[5] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *NSDI 18*, 2018, pp. 299–312.

[6] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *NSDI 19*, 2019, pp. 501–516.

[7] J. Wang, S. Hao, Y. Li, C. Fan, J. Wang, L. Han, Z. Hong, and H. Hu, "Challenges towards protecting vnf with sgx," in *SDN-NFVSec*, 2018, pp. 39–42.

[8] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "Statelet-based efficient and seamless nfv state transfer," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 964–977, 2017.

[9] M. Peuster, H. Küttner, and H. Karl, "Let the state follow its flows: An sdn-based flow handover protocol to support state migration," in *NetSoft*. IEEE, 2018, pp. 97–104.

[10] W. Wang, Y. Liu, Y. Li, H. Song, Y. Wang, and J. Yuan, "Consistent state updates for virtualized network function migration," *IEEE Transactions on Services Computing*, 2017.

[11] X. Chen, Y. Chen, Q. Huang, H. Zhou, D. Zhang, C. Wu, and J. Xing, "Fastscale: Fast scaling out of network functions," in *INFOCOM WKSHPS*. IEEE, 2020, pp. 436–442.

[12] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *SOSR*, 2017, pp. 115–121.

[13] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *HotMiddleBox*, 2015, pp. 43–48.

[14] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamatian, "Transparent flow migration for nfv," in *ICNP*. IEEE, 2016, pp. 1–10.

[15] Y. Lin, U. C. Kozat, J. Kaippallimalil, M. Moradi, A. C. Soong, and Z. M. Mao, "Pausing and resuming network flows using programmable buffers," in *SOSR*, 2018, pp. 1–14.

[16] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *NSDI 17*, 2017, pp. 97–112.

[17] M. Szalay, M. Nagy, D. Géhberger, Z. Kiss, P. Mátray, F. Németh, G. Pongrácz, G. Rétvári, and L. Toka, "Industrial-scale stateless network functions," in *CLOUD*. IEEE, 2019, pp. 383–390.

[18] J. Hypolite, J. Sonchack, S. Hershkop, N. Dautenhahn, A. DeHon, and J. M. Smith, "Deepmatch: practical deep packet inspection in the data plane using network processors," in *CoNEXT*, 2020, pp. 336–350.

[19] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[20] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[21] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM.*, 2012, pp. 323–334.

[22] R. Sukapuram and G. Barua, "Ppcu: Proportional per-packet consistent updates for sdns using data plane time stamps," *Computer Networks*, vol. 155, pp. 72–86, 2019.

[23] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *NSDI 15*, 2015, pp. 73–85.

[24] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *HotSDN*, 2013, pp. 49–54.

[25] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 539–550, 2014.

[26] S. Luo, H. Yu, and L. Li, "Consistency is not easy: How to use two-phase update for wildcard rules?" *IEEE Communications Letters*, vol. 19, no. 3, pp. 347–350, 2015.

[27] R. Sukapuram and G. Barua, "Ccu: Algorithm for concurrent consistent updates for a software defined network," in *NCC*. IEEE, 2016, pp. 1–6.

[28] G. Németh, D. Géhberger, and P. Mátray, "{DAL}: A locality-optimizing distributed shared memory system," in *HotCloud 17*, 2017.

[29] B. Alpern and F. B. Schneider, "Defining liveness," *Information processing letters*, vol. 21, no. 4, pp. 181–185, 1985.

[30] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[31] ——, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[32] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[33] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[34] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[35] A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems," in *HotOS*. IEEE, 1999, pp. 174–178.

[36] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[37] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "Cap for networks," in *HotSDN*, 2013, pp. 91–96.

[38] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," *SIGCOMM*, 2016.

[39] A. S. Muqaddas, G. Sviridov, P. Giaccone, and A. Bianco, "Optimal state replication in stateful data planes," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, pp. 1388–1400, 2020.

[40] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1435–1461, 2018.

[41] R. Sukapuram and G. Barua, "Proflow: Proportional per-bidirectional-flow consistent updates," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 675–689, 2019.

[42] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014, pp. 1–7.

[43] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *NSDI 13*, 2013, pp. 15–27.

[44] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *NSDI 17)*, 2017, pp. 699–718.

[45] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," in *SIGCOMM*, 2013, pp. 411–422.

[46] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *NSDI 20*, 2020, pp. 257–272.

[47] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang, "A survey on network verification and testing with formal methods: Approaches and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 940–969, 2018.