On Minimizing Synchronization Cost in NFV-based Environments

Zakaria Alomari*, Mohamed Faten Zhani*, Moayad Aloqaily†, Ouns Bouachir‡

*École de Technologie Supérieure (ÉTS Montreal), Montreal, Quebec, Canada
†Faculty of Engineering, Al Ain University, UAE
‡Zayed University, Dubai, UAE
E-mail: *zakaria.alomari.1@ens.etsmtl.ca, *mfzhani@etsmtl.ca, †maloqaily@ieee.org, ‡ouns.bouachir@zu.ac.ae

***Abstract*—Network Function Virtualization is known for its ability to reduce deployment costs and improve the flexibility and scalability of network functions. Due to processing capacity limitation, the infrastructure provider needs to instantiate one or more instances of a particular network function when the amount of traffic increases. Most of network functions are stateful, which means that they keep a state that may be frequently read or updated (e.g., statistics like number of packets or bytes per flow). As a result, the instances of the same virtual network function should constantly share the same state to prevent incorrect operation. In this context, a major challenge is how to efficiently ensure the consistency among instances while minimizing communication cost for synchronizing their state and ensuring the synchronization delay does not exceed a certain bound set by the operator.**

In this paper, we propose a technique to identify the optimal communication pattern between the instances of the same network function in order to minimize their synchronization cost. Moreover, we propose to use a special network function named *Synchronization Function* to ensure consistency among a set of instances and to minimize the synchronization cost. We first mathematically model the problem of finding the optimal synchronization pattern and the optimal placement and number of synchronization functions as an integer linear program that minimizes the synchronization cost and ensures a bounded synchronization delay. Last, we put forward three algorithms to cope with large-scale scenarios of the problem. Extensive simulations show that the proposed algorithms efficiently find near-optimal solutions with minimal computation time.

I. INTRODUCTION

The emergence of Network Function Virtualization (NFV) technology is currently transforming the way networks are designed, deployed, and managed as it provides operators to deploy network services at low cost and high flexibility [1]–[3]. The NFV technology allows to create *Virtual Network Functions* (VNFs) (e.g., Load balancer, Firewall, IDS, NAT) and connect them to create *Service Function Chains* (SFC).

As a matter of fact, the traffic may rise occasionally because of higher demand. In this case, the infrastructure provider might be required to implement the same VNF in multiple virtual machines instances due to limited processing capacity, resource cost or location constraint [4]–[9]. However, running the same network function in a distributed manner over multiple instances is challenging.

978-3-903176-31-7 © 2020 IFIP

Indeed, network functions are either stateless or stateful. When a network function is stateless, it can operate without maintaining any state [10], [11]. Hence, all instances implementing this function can operate independently and successfully without sharing any information. In this case, no data synchronization among them is required. On the other hand, when a network function is stateful, all instances should maintain the same state (e.g., statistics like number of packets or bytes per flow, list of available ports, per-connection port mapping) that should be frequently read and updated. Ideally, all these instances should work like a single one, regardless of their number and location [12]. Thus, all instances of the network function need to have the same state to operate normally, and, hence, data synchronization between the instances becomes mandatory [11], [13], [14]. In this case, there is a compelling need to ensure a synchronization among them with minimal cost and acceptable delay.

In this context, the synchronization cost is defined as the number of messages exchanged between the instances of the same network function. Synchronization delay is the amount of time needed to exchange these messages among these instances and reach data consistency. Of course this delay is high, the performance and operation of the network function may be impacted as it may result in state inconsistency among the instances leading to late or inaccurate decisions [15].

Several studies have recently focused on the state synchronization challenge. For instance, the authors in [11], [16] suggested to set up a common repository for persistently storing and managing state shared among instances. The authors in [17], [18] have considered a different technique where each instance broadcasts its state to the other instances. Unlike previous work, our work does not only target to carry out synchronization but aims at minimizing the synchronization cost between the instances and ensuring that the synchronization delay does not exceed a certain bound to ensure normal operation of the function. To the best of our knowledge, no previous work considered minimizing the state synchronization cost and delay of VNF instances.

In this paper, we propose a technique that identifies the optimal communication pattern between the instances in order to minimize the synchronization cost and to ensure a bounded synchronization delay. We also introduce the use

a new function called *Synchronization Function* that ensures consistency among a set of instances and allows to further reduce the synchronization cost. We first formulate the problem of finding the optimal communication pattern, the optimal placement for the synchronization functions, and number of synchronization functions as an Integer Linear Program (ILP) aiming at minimizing the synchronization cost. We also devise three algorithms, called SPT, SSF, and MSF, respectively, that are able to efficiently explore the solution space for large-scale scenarios within a reasonable timescale.

The remainder of this paper is organized as follows. Section II presents some examples to better highlight the importance of data synchronization among instances. Section III provides a detailed description for the problem of finding the optimal communication pattern. Section IV presents the mathematical formulation of the addressed problem. We then describe the proposed heuristic solutions in Section V. Section VI presents the experimental results. The related work is presented in Section VII. Finally, in Section VIII, we conclude the paper and provide some future work.

II. Motivational Examples

Ideally, if a stateful network function is implemented in multiple instances, the instances should share the same state as they should operate like a single instance regardless of their number and location. However, the state synchronization may be costly and need also to be carried out in a bounded delay to ensure correct and quick operation. In the following, we provide two examples of VNFs and show how the synchronization could be costly and how the could impact the performance of the network function itself.

Example 1 : let's consider an Intrusion Detection System (IDS) as an example to demonstrate the importance of state consistency and synchronization delay among multiple instances of the same network function. Fig. 1 shows an example of an IDS implemented in two instances. For example, a Deny-of-Service attack is detected when the number of received TCP SYN packets reaches a threshold k. Each IDS instance maintains a counter for the number of received packets. The attacker may send the malicious traffic from two sources ($k/2$ from first source and $k/2$ from the second) and hence each IDS instance process only half of the traffic. If the two IDS instances collaborate to synchronize the state (i.e., the number of packets) in a timely manner, the counters will be updated to k, and hence, the attack will be detected by both instances. Any delay in the synchronization will result in a late decision,

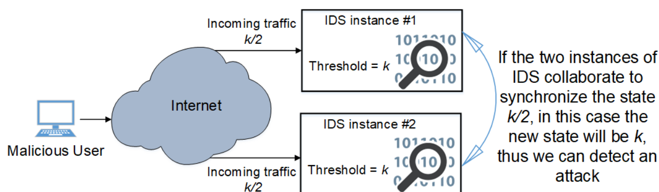


Fig. 1: State synchronization between two IDS instances.

which may impact the network and service performance. The problem is even more challenging when multiple instances are deployed and when they are located in a geographically distant locations.

Example 2 : In the context of Content Delivery Networks (CDN), multiple CDN instances may host large amounts of content. If these instances have to synchronize their content, this may consume significant amount of bandwidth. It is therefore of utmost importance to find the best synchronization pattern (e.g., paths, synchronization functions) to carry out synchronization with minimal bandwidth consumption.

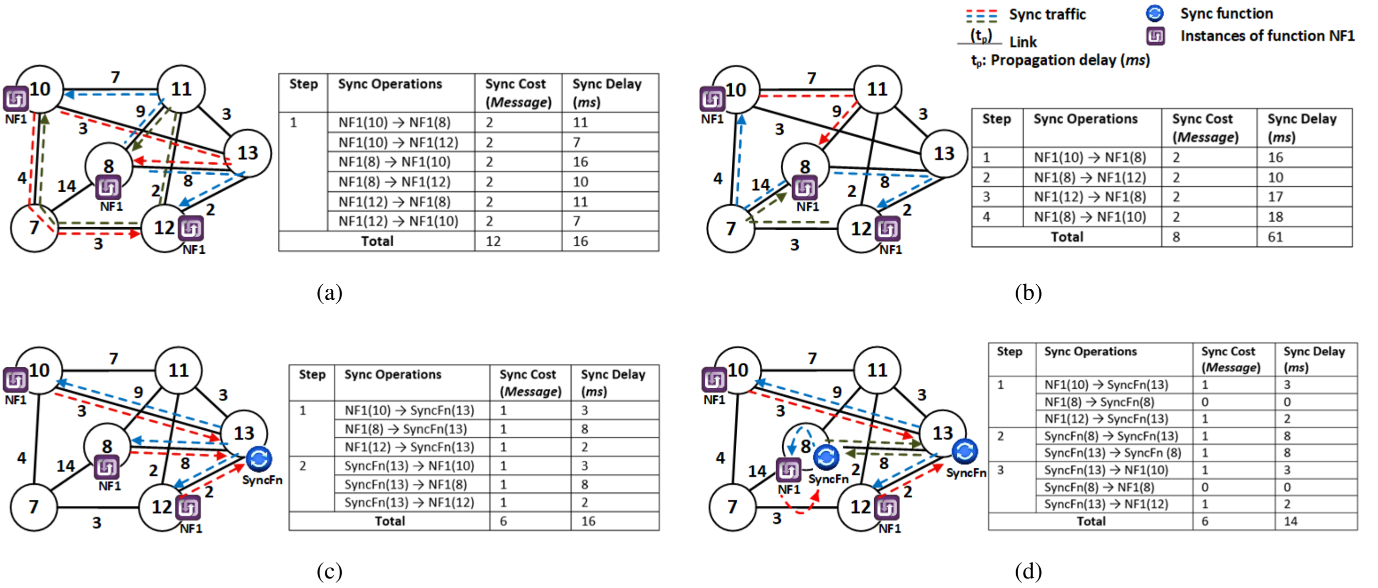
III. Problem Description

As previously mentioned, a VNF could be implemented in multiple instances. Each instance corresponds to a virtual machine or a container running on top of the hardware networking infrastructure [19]. These instances may be place in geographically distributed locations (e.g., Fig. 2) and there are different communication patterns that could be defined to ensure they can exchange data and synchronize their state. A communication pattern defines the way data is exchanged (i.e., the order at which data is sent) as well as the paths that should be taken to deliver it to the involved instances. The goal of the following example is to show how the communication pattern impact the cost and the delay of the synchronization.

Fig. 2 shows four different communication patterns (i.e., a, b, c, d) that could be used to carry out synchronization among instances of a function (NF1). We can see that there are three instances of function NF1 distributed in different physical nodes and synchronizing their state (see red, blue, and green dotted arrows).

Fig. 2(a) illustrates how instances of function NF1 synchronize the same state with each other periodically at the same time. In this communication pattern, the synchronization operations are achieved in one step. For example, the instance that is located in node 10, NF1(10), synchronizes the same state with the other instances NF1(8) and NF1(12) by using the paths $10 \rightarrow 13 \rightarrow 8$ (the sync cost is 2 $messages$ and the delay is 11 ms) and $10 \rightarrow 7 \rightarrow 12$ (the synch cost is 2 $messages$ and the delay is 7 ms). Similarly, all other instances NF1(8) and NF1(12) will follow up the same procedure. As a result, the total synchronization cost between instances is the number of messages exchanged between them which is 12 $messages$ and the delay will be represented by the path that has the highest delay which is 16 ms.

Fig. 2(b) shows that each instance synchronize the same state to the next instance sequentially. In this communication pattern, the synchronization operations are achieved in multiple steps. This means, the instances take turns sending and receiving the same state. The receiving instance reads the state addressed to it and then passes the state and any additional update to the next instance. This continues until the same state reaches the desired receiving instance. For example, the instance NF1(10) synchronizes the same state with the next instance NF1(8) by using the path $10 \rightarrow 11 \rightarrow 8$ (the sync cost is 2 $messages$ and the delay is 16 ms). Similarly,



Step	Sync Operations	Sync Cost (Message)	Sync Delay (ms)
1	NF1(10) → NF1(8)	2	11
	NF1(10) → NF1(12)	2	7
	NF1(8) → NF1(10)	2	16
	NF1(8) → NF1(12)	2	10
	NF1(12) → NF1(8)	2	11
	NF1(12) → NF1(10)	2	7
Total		12	16

Step	Sync Operations	Sync Cost (Message)	Sync Delay (ms)
1	NF1(10) → NF1(8)	2	16
2	NF1(8) → NF1(12)	2	10
3	NF1(12) → NF1(8)	2	17
4	NF1(8) → NF1(10)	2	18
Total		8	61

Step	Sync Operations	Sync Cost (Message)	Sync Delay (ms)
1	NF1(10) → SyncFn(13)	1	3
	NF1(8) → SyncFn(13)	1	8
	NF1(12) → SyncFn(13)	1	2
2	SyncFn(13) → NF1(10)	1	3
	SyncFn(13) → NF1(8)	1	8
	SyncFn(13) → NF1(12)	1	2
Total		6	16

Step	Sync Operations	Sync Cost (Message)	Sync Delay (ms)
1	NF1(10) → SyncFn(13)	1	3
	NF1(8) → SyncFn(8)	0	0
	NF1(12) → SyncFn(13)	1	2
2	SyncFn(8) → SyncFn(13)	1	8
	SyncFn(13) → SyncFn (8)	1	8
3	SyncFn(13) → NF1(10)	1	3
	SyncFn(8) → NF1(8)	0	0
	SyncFn(13) → NF1(12)	1	2
Total		6	14

Fig. 2: Different possible communication patterns to synchronize NF1 instances

all other instances will follow the same procedure. The total synchronization cost for the instance NF1(8) is 4 $messages$ and the delay is 28 ms by using the paths $8 \rightarrow 13 \rightarrow 12$ (the sync cost is 2 $messages$ and the delay is 10 ms) and $8 \rightarrow 7 \rightarrow 10$ (the sync cost is 2 $messages$ and the delay is 18 ms) respectively. The synchronization cost for the instance NF1(12) is 2 $messages$ and the delay is 17 ms. As a result, the total synchronization cost between instances is the number of messages exchanged between them which is 8 $messages$ and the delay is the amount of time needed to exchange these messages which is 61 ms.

Fig. 2(c) illustrated a communication pattern that benefits from a single synchronization function. In this communication pattern, the synchronization operations are achieved in two steps. The first step, the instances synchronize the same state to the synchronization function periodically at the same time. The second step, the synchronization function consolidates and distributes the same state to the all instances periodically at the same time. For example, the instance NF1(10), NF1(8), and NF1(12) synchronize the same state to the synchronization function that is located in node 13, SyncFn(13). Then, the synchronization function SyncFn(13) consolidates and distributes the same state to all instances by using the same procedure. The synchronization cost and the delay for both steps is depicted in Fig. 2(c). As a result, the total synchronization cost is the number of messages exchanged between instances and synchronization function which is 6 $messages$ and the delay is the amount of time needed to exchange these messages which is 16 ms.

Fig. 2(d) depicts a communication pattern that benefits from multiple synchronization functions. The synchronization function can work as follows: serve a set of instances, or serve a set of synchronization functions, or it can do both at the same time. In this communication pattern, the synchronization operations are achieved in three steps. The first step, the instances synchronize the same state to the closest synchronization function periodically at the same time. The second step, the synchronization functions synchronize the same state with each other periodically at the same time. The third step, the synchronization functions distributes the same state to all instances periodically at the same time. For example, NF1(10) and NF1(12) synchronize the same state to SyncFn(13), also NF1(8) synchronizes the same state to SyncFn(8). Furthermore, SyncFn(8) and SyncFn(13) synchronize the same state with each other by using the same procedure. Finally, SyncFn(8) and SyncFn(13) distribute the same state to their instances by using the same procedure. The synchronization cost and the delay for all steps depicted in Fig. 2(d). As a result, the total synchronization cost is the number of messages exchanged between instances and synchronization functions which is 6 $messages$ and the delay is the amount of time needed to exchange these messages which is 14 ms.

It is clear from the above-described example that the communication pattern significantly impacts the synchronization cost and delay and that synchronization functions could further reduce such cost and delay. In this paper, we focus on how to find the optimal communication pattern to synchronize the instances, and the optimal placement for the synchronization functions in a way that minimizes synchronization cost (i.e., number of exchanged messages) while ensuring that the synchronization delay does not exceed a certain bound.

IV. Problem Formulation

In this section, we formulate the problem of finding the optimal communication pattern as an Integer Linear Program (ILP) aiming at minimizing the synchronization cost while ensuring the required synchronization delay does not exceed a certain bound. Furthermore, the ILP finds the optimal com-

munication pattern to synchronize the instances, the number of the synchronization functions and their optimal placement of in the infrastructure.

The physical infrastructure consists of multiple nodes located in different geographical locations modeled by a graph $G = (N, L)$ where N indicates the set of physical nodes that are connected via physical links L. We also define w_{uv} is the value of the weight of the link (i.e., cost) between two physical nodes u and v. We define by d_{uv} the value of the synchronization delay of the link between two connected physical nodes u and v. We also denote by β the maximum synchronization delay that can be tolerated between any two VNF instances.

We define $I = \{1, ..., |I|\}$ as the set of physical nodes hosting the instances of the same network function that should be synchronized. Each node of the set I is considered as a source and a destination as it has to generate and receive data. Any subgraph that connects the nodes of the set I can be considered as a communication pattern. The weight of a subgraph is defined as the sum of the weights of the subgraph's links. This weight corresponds to the total synchronization cost of the communication pattern associated to that subgraph. As our ultimate objective is to minimize the total synchronization cost, the goal of the following ILP is to find the subgraph $\bar{G}$ extracted from the original graph G that necessarily includes all the nodes of I and having the minimum total weight. As such, the subgraph $\bar{G}$ corresponds to the optimal communication pattern.

The subgraph $\bar{G}$ can also be seen as a set of paths that connects all nodes of I while the weight of the resulting subgraph is minimal. The proposed ILP finds these paths while minimizing the weight of the generated subgraph. In the following, we define the decision variables and constraints of the ILP.

• **Decision variables:** we define two decision variables as follows:

- $x_{stuv} \in \{0, 1\}$ indicates whether the link uv ($u \in N$ and $v \in N$) is part of a path between a source $s \in I$ and a destination $t \in I$.
- $y_{uv} \in \{0, 1\}$ indicates whether the link uv is used in the path between any source $s \in I$ and any destination $t \in I$. In other words, y_{uv} is equal to 1 only if the link uv belongs to the subgraph $\bar{G}$.

• **Objective function:** our main objective is to minimize the synchronization cost among instances while ensuring the required synchronization delay does not exceed a certain bound. The objective function can be expressed as follows:

$$min\left(\sum_{u\in N}\sum_{v\in N} y_{uv}w_{uv}\right) \tag{1}$$

• **Constraints:** while finding the optimal solution, several constraints must be satisfied. For instance, we need to ensure that there is a link leaving any source $s \in I$:

$$\sum_{t\in I}\sum_{v\in N} x_{stsv} - \sum_{u\in N}\sum_{t\in I} x_{stus} = 1 \qquad \forall s \in I \tag{2}$$

Where the first term is the number of links leaving s, and the second one is the number of links entering to the node s.

We also need to ensure that there is a link entering any destination $t \in I$:

$$\sum_{u\in N}\sum_{s\in I} x_{stut} - \sum_{s\in I}\sum_{v\in N} x_{sttv} = 1 \qquad \forall t \in N \tag{3}$$

Where the first term is the number of links entering t, and the second term is the number of links leaving t.

Furthermore, there must be a path between between any source $s \in I$ and destination $t \in I$. In other words, we must ensure that for any node on the path between s and t, the number of links entering that node is equal to the number of links leaving it (as it is a path). This can be expressed as follows:

$$\sum_{u\in N} x_{stuz} - \sum_{v\in N} x_{stzv} = 0 \quad \forall s, t \in I \quad \forall z \in N\backslash\{s, t\} \tag{4}$$

Where the first term is the number of links entering a node $z \in N\backslash\{s, t\}$ that lies in the path between the source $s \in I$ and the destination $t \in I$, and the second term is the number of links leaving $z \in N\backslash\{s, t\}$ in the same path.

Furthermore, we need to ensure that, if there are common links between the paths, they should be considered once in the constructed subgraph. In other words, a link uv should be counted only once in the generated subgraph. This can be expressed as follows:

$$y_{uv} \geq x_{stuv} \quad \forall s, t \in I \quad \forall u, v \in N \tag{5}$$

Additionally, the delay of the path between any source $s \in I$ and $t \in I$ should not exceed the maximum synchronization delay β. This constraint can be written as follows:

$$\sum_{u\in N}\sum_{v\in N} x_{stuv}d_{uv} \leq \beta \qquad \forall s, t \in I \tag{6}$$

• **ILP Output:** the resulting values of the decision variable y_{uv} will define the links of subgraph $\bar{G}$ as well as its nodes. The variable x_{stuv} defines the path that should used to synchronize data between nodes $s \in I$ and $t \in I$.

Furthermore, to find the number of the synchronization function and their optimal placement, we assume that any physical node in $\bar{G}$ being connected to more than two links (i.e., the degree of this node is higher than two and such a node is called hereafter *a common node*) should host a synchronization function. This is because a node connected to only two links has only to forward the synchronized data from one link to the following. However, a node connected to more than two links is able to aggregate the data coming from two links and then forward it to the other ones. As a result, synchronization functions are placed in common nodes and have the same number as those nodes.

V. PROPOSED HEURISTIC

In this section, we describe three heuristics that could find the best communication patterns to synchronize the instances. We call the first algorithm *Shortest Path Tree* (SPT) as the algorithm does not use synchronization functions. The second algorithm is called *Single Synchronization Function* (SSF) as it uses only one synchronization function in the physical infrastructure to serve multiple instances of the same network function. The third algorithm is called *Multiple Synchronization Functions* (MSF) as it tries to place multiple synchronization functions with each serving a different set of instances of the same network function. In the following, we present the three proposed algorithms in details:

• **Algorithm SPT:** it is a modified version of the shortest path tree algorithm. Each instance synchronizes the same state with the other instances periodically at the same time using its shortest path tree.

We first consider a set of instances located in different physical nodes n. As shown in Algorithm 1, we compute the shortest paths for each instance with the other instances. Then, each instance merge the computed paths to build its own shortest path tree therefore we will have many common nodes n_{cn} (i.e., common nodes share by more than one path). However, we can consider n as a common node if there is more than one path cross it (Lines 6-9). To minimize the synchronization cost, we can configure all the common nodes n_{cn} to automatically duplicate the messages that go through them, therefore we can ensure there is no duplicate messages between multiple paths (Lines 10-12).

We compute the delay $t_{\bar{l}}$, which is the time required to exchange the messages between all the instances. This delay would be represented by the path that has the highest delay (lines 13-17). The delay $t_{\bar{l}}$ should not exceed the delay constraint β to compute the synchronization cost (lines 18-20). Finally, the whole process is repeated for instances of another network function.

Algorithm 1 SPT

1: **Inputs**
2: N : set of physical nodes
3: V : set of VNFs
4: β : delay constraint
5: **for** $v \in V$ **do**
6: **for** each instance of type $v = \{i_1, i_2, i_3, ...i_{|v|}\}$ **do**
7: $SPList(v) \leftarrow get_shortest_path_tree(i_v)$ ▷ $SPList$ is the list to store the shortest path tree for each instance of type v
8: $CNodeList(n_{cn}) \leftarrow get_CNodes(SPList(v))$ ▷ $CNodeList$ is the list to store the nodes that share by more than one path in the shortest path tree
9: **end for**
10: **for** $j \in CNodeList()$ **do**
11: $configure_Common_Nodes(j)$ ▷ Configuring the common nodes j to duplicate the messages to minimize the synchronization cost
12: **end for**
13: **for** each instance of type $v = \{i_1, i_2, i_3, ...i_{|v|}\}$ **do**
14: $i_vDelay \leftarrow compute_Delay(i_v)$ ▷ i_vDelay is variable to store the synchronization delay for each instance of type v
15: $DelayList(v) \leftarrow i_vDelay$ ▷ $DelayList(v)$ is the list to store the synchronization delay for all instances of type v one by one
16: **end for**
17: $t_{\bar{l}} = max_{i_vDelay}(DelayList(v))$
18: **if** $t_{\bar{l}} \leq \beta$ **then**
19: $compute_SyncCost(v)$
20: **end if**
21: **end for**

• **Algorithm SSF:** Algorithm 2 aims to allocate a single synchronization function for instances of the same network function. Assuming we consider a set of instances are located in different physical nodes n. We first need to get the common nodes n_{cn} between the instances. To do so, we compute the shortest paths for each instance with the other instances. Then, each instance merges the computed paths to build its own shortest path tree. Hence, we will have many common nodes (i.e., common node shares by more than one path) and each common node has different score (i.e., common node score defines the number of paths that cross this node). However, we can consider n as a common node, if its score is greater than one. Otherwise, it will not be considered as a common node (Lines 7-14).

To minimize the synchronization cost, we choose the common node n_{cn} that has the higher score $s_{n_{cn}}$, and enough resources $c_{n_{cn}}$, to be the hosting node of the synchronization function (Lines 15-20). Having no common nodes for all physical nodes mean that either is no enough resources to host the synchronization function or there is no instances of network function.

We compute the delay $t_{\bar{l}}$, which is the time required for a messages to travel from instances to a synchronization function and back again. To this end, we assume that all instances synchronize the same state to the synchronization function periodically at the same time. We compute the delay between instances and the synchronization function $t_{\bar{v},\bar{sf}}$, which represents by the instance that has the highest delay (function compute_SyncDelay in Line 21). We also compute the delay between synchronization function and all instances $t_{\bar{sf},\bar{v}}$, which represents by the highest delay between them (function compute_SyncDelay in Line 22). The $t_{\bar{l}}$ should not exceed the delay constraint β to compute the synchronization cost (Lines 23-27). Finally the whole process is repeated for instances of another network function.

• **Algorithm MSF:** In this algorithm, our goal is to maximize the number of synchronization functions that serve multiple instances of the same network function to minimize the synchronization cost.

As shown in Algorithm 3, we compute the shortest paths for each instance with the other instances. Then, each instance

Algorithm 2 SSF

1: **Inputs**
2: N: set of physical nodes
3: V: set of VNFs
4: c_n: remaining capacity in each physical node $n \in N$
5: β : delay constraint
6: **for** $v \in V$ **do**
7: **for** each instance of type $v = \{i_1, i_2, i_3, ...i_{|v|}\}$ **do**
8: $SPList(v) \leftarrow get_shortest_path_tree(i_v)$ ▷ $SPList$ is the list to store the shortest path tree for each instance of type v
9: $CNodeList(n_{cn}) \leftarrow get_CNodes(SPList(v))$ ▷ $CNodeList$ is the list to store the nodes that share by more than one path in the shortest path tree
10: **end for** ▷ find common nodes score
11: **for** $j \in CNodeList(n_{cn})$ **do**
12: $s_n \leftarrow 0$
13: $CNScoreList(n_{cn}, s_{n_{cn}}) \leftarrow get_Score(j, s_j)$ ▷ $CNScoreList$ is the list to store the common nodes n_{cn} and their scores $s_{n_{cn}}$
14: **end for**
15: n_{cn} = $max_{s_{n_{cn}}}(CNScoreList(n_{cn}, s_{n_{cn}}))$
16: **if** $c_{n_{cn}} > 0$ **then**
17: Allocate($SyncFn$, $Instances$, n_{cn})
18: **else**
19: *Move to the second highest common node*
20: **end if**
21: $t_{\bar{v},\bar{sf}} \leftarrow compute_SyncDelay(v, sf)$
22: $t_{\bar{sf},\bar{v}} \leftarrow compute_SyncDelay(sf, v)$
23: $t_{\bar{l}}$ = $t_{\bar{v},\bar{sf}} + t_{\bar{sf},\bar{v}}$
24: **if** $t_{\bar{l}} \leq \beta$ **then**
25: $compute_SyncCost(v, sf)$
26: $compute_SyncCost(sf, v)$
27: **end if**
28: **end for**

merge the computed paths to build its own shortest path tree therefore we will have a set of common nodes n_{cn} (i.e., common nodes shares by more than one path) (Lines 7-10). Then, each instance try to find the closest common node cn_{close}. To this end, we compute the number of hops h, and the delay $\bar{t}$ between each instance and all the common nodes (Line 11-16). For each instance, the algorithm selects the closest common node cn_{close} that has the lowest number of hops h, less delay $\bar{t}$, and enough resources $c_{cn_{close}}$ to be the hosting node of the synchronization function (Lines 17-26). We also assume that we can not place more than one synchronization function in the closest common node to serve multiple instances of the same network function (Lines 20-22).

Furthermore, the algorithm synchronizes the same state between multiple synchronization functions for consistency purpose. Thus, each synchronization function synchronizes the same state with the others based on the lowest number of hops h, and less delay $\bar{t}$.

We compute the delay $t_{\bar{l}}$, which is the time required for

Algorithm 3 MSF

1: **Inputs**
2: N : set of physical nodes
3: V : set of VNFs
4: c_n : remaining capacity in each physical node $n \in N$
5: β : delay constraint
6: **for** $v \in V$ **do**
7: **for** each instance of type $v = \{i_1, i_2, i_3, ...i_{|v|}\}$ **do**
8: $SPList(v) \leftarrow get_shortest_path_tree(i_v)$ ▷ $SPList$ is the list to store the shortest path tree for each instance of type v
9: $CNodeList(n_{cn}) \leftarrow get_CNodes(SPList(v))$ ▷ $CNodeList$ is the list to store the nodes that share by more than one path in the shortest path tree
10: **end for**
11: **for** each instance of type $v = \{i_1, i_2, i_3, ...i_{|v|}\}$ **do**
12: **for** $j \in CNodeList(n_{cn})$ **do**
13: $h \leftarrow compute_NumHops(i_v, j)$
14: $\bar{t} \leftarrow compute_SyncDelay(i_v, j)$
15: $CAND(v, n_{cn}, h, \bar{t}) \leftarrow \{i_v, j, h, \bar{t}\}$ ▷ $CAND$ is the list to store the number of hops and the delay between each instance of type v and all the common nodes.
16: **end for**
17: cn_{close} = $min_{h,\bar{t}}(CAND(v, n_{cn}, h, \bar{t}))$
18: **if** $c_{cn_{close}} > 0$ & $SyncFn \notin cn_{close}$ **then**
19: Allocate($SyncFn$, $Instances$, cn_{close})
20: **else if** ($SyncFn \in cn_{close}$) **then**
21: *Use existing synchronization function*
22: **else**
23: *Move to the second closest common node*
24: **end if**
25: **end for**
26: **end for**
27: $t_{\bar{v},\bar{sf}} \leftarrow compute_SyncDelay(v, sf)$
28: $t_{\bar{sf},\bar{sf}} \leftarrow compute_SyncDelay(sf, sf)$
29: $t_{\bar{sf},\bar{v}} \leftarrow compute_SyncDelay(sf, v)$
30: $t_{\bar{l}}$ = $t_{\bar{v},\bar{sf}} + t_{\bar{sf},\bar{sf}} + t_{\bar{sf},\bar{v}}$
31: **if** $t_{\bar{l}} \leq \beta$ **then**
32: $compute_SyncCost(v, sf)$
33: $compute_SyncCost(sf, sf)$
34: $compute_SyncCost(sf, v)$
35: **end if**

a messages to travel from instances to the synchronization functions and back again. To this end, we compute the delay between instances and the synchronization functions $t_{\bar{v},\bar{sf}}$, which represents by the instance that has the highest delay. We also compute the delay between synchronization functions $t_{\bar{sf},\bar{sf}}$, which represents by the synchronization function that has the highest delay. Similarly, we compute the delay between the synchronization functions and the instances $t_{\bar{sf},\bar{v}}$ (Lines 27-30). The $t_{\bar{l}}$ should not exceed the threshold β to compute the synchronization cost (Lines 31-35). Finally, the whole process is repeated for instances of another network function.

VI. Simulation and Results

In this section, we evaluate the performance of the proposed algorithms through extensive simulations. We basically compare the performance of those algorithms with the optimal solution provided by CPLEX and compare the obtained synchronization cost, delay, as well as the algorithms' execution time.

A. Simulation Setup

In order to evaluate the proposed solutions, we developed a C/C++ simulator that simulates the entire environment including the physical infrastructure, the embedded functions, and the three proposed algorithms. We considered a topology with 24 physical nodes connected through 52 physical links that were randomly generated. We assume that each physical node has different computing capacity randomly generated from 40 to 120 virtual machines. We assume that all virtual machines have the same resource capacities. We also assume that each link has different propagation delay randomly generated between 5 ms to 20 ms. In our experiment, we assume that the VNF instances in there are already embedded. Hence, we considered 8 different scenarios where the instances of the same network functions has been randomly distributed in the infrastructure. Furthermore, we fixed the required synchronization delay for the VNFs to 110 ms.

B. Synchronization cost

Fig. 3 compares the average synchronization cost found with the proposed algorithms with the optimal solution generated by CPLEX for each scenario. We can see that for all scenarios (i.e., S1-S8) SSF and MSF outperforms SPT and generate a slightly higher number of messages compared to the optimal solution provided by CPLEX. This indicate that the communication patterns provided by SSF and MSF are close to the optimal ones. To explain these results, SPT achieves the synchronization operation where each instance synchronizes the state with the other instances using its shortest path tree. In this case, the synchronization cost will be high as the number of hops between the instances is high. Therefore, the messages will travel through a high number of hops to share the state.

Unlike SPT, SSF and MSF find common nodes between the instances to provision synchronization functions therein to minimize the number of message. Indeed, synchronization functions consolidate received messages (so their number are reduced) and then distribute them to all instances at the same time.

C. Synchronization delay

Fig. 4 compares the average synchronization delay found with the proposed algorithms and the optimal solution generated by CPLEX for each scenario. It clearly shows that SSF and MSF outperform SPT and provide a near optimal results. To explain these results, the synchronization delay can be controlled by the number of hops between the instances. Therefore, the more we minimize the number of hops, the more the synchronization delay decreases. Hence, SSF and MSF succeeded in reducing the synchronization delay by provisioning synchronization functions that minimize the number of hops between instances. The results show that, for SPT, SSF and MSF algorithms as well as CPLEX, the synchronization delay is below the synchronization delay threshold β specified as input to all solutions (β is equal to 110 ms in our experiments).

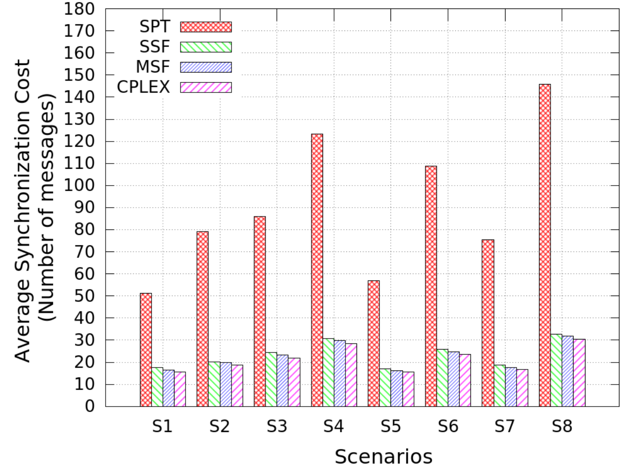


Fig. 3: Average synchronization cost

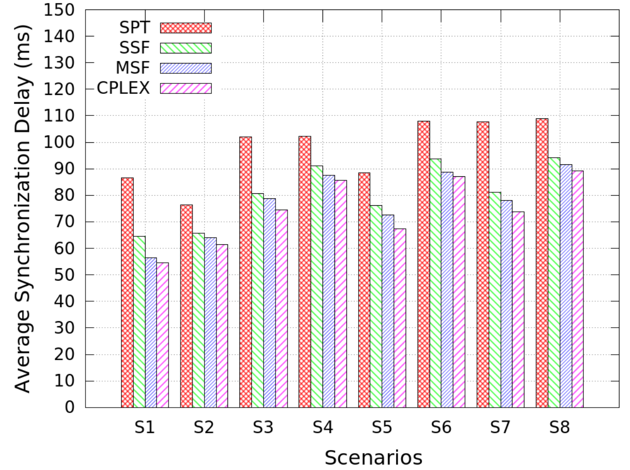


Fig. 4: Average synchronization delay

D. Execution Time

Fig. 5 depicts the execution time for the proposed algorithms compared to that of CPLEX. The figure shows that the execution time for CPLEX is between 46 seconds to 29 minutes depending on the number of instances and the distance between them. The reason behind that is that the number of variables in the ILP increase (e.g., the number of instances of the network functions, and that the number of hops between instances), which makes the problem harder to solve and takes more time to solve because of a larger solution space. However, the figure shows that the execution time for SPT,

SSF, and MSF does not significantly change for all scenarios and regardless of the number of instances and their placement.

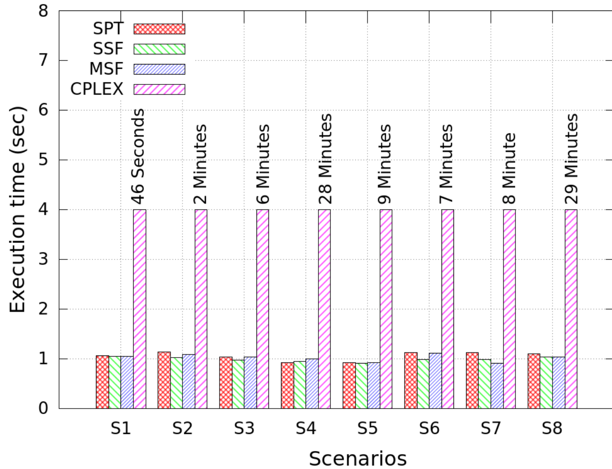


Fig. 5: Execution time.

E. Discussion

The performed simulations show that SSF and MSF are able to find near-optimal synchronization cost and delay, while SPT is so far from optimal one. Indeed, compared to the optimal solution, the average synchronization cost and delay gaps for SPT are up to 80% and 37%, respectively, which is far from the optimal one. However, the average synchronization cost and delay gaps for SSF are up to 12% and 15% respectively, which is close to the optimal one. They are even less than 5% and 7%, respectively, for MSF, which is much closer to the results of the optimal solution. Furthermore, the execution time for the three proposed algorithms are up to times faster than CPLEX used to find the optimal solution. We can conclude that MSF finds the best trade-off between complexity and performance

VII. Related Work

In this section, we briefly present recent research work addressing the state synchronization problem. For instance, Khalid and Akella [11] propose a NFV framework to support state synchronization among stateful network functions. They leverage a data store to ensure the state consistency among multiple instances of network function. Each instance records a callback function with the data store. This function is used by the data store to update the state of the instance on behalf of the others. This work does not consider synchronization cost and delay.

Satapathy et al. [16] propose a network state management system, which introduces two different solutions to ensure state synchronization. In the first solution, an instance synchronizes the state at the receipt of each single packet whereas in the second the state is synchronized after receiving multiple packets. However,the first solution obviously has a high synchronization overhead whereas the second may incur additional delay.

Peuster and Karl [20] propose E-State management framework to synchronize the state of all instances of the same network function. They use a publish/subscribe communication pattern to share the state. Subscribers are the Instances who subscribe to a same state, and Publishers are the instances who publish the state. However, this solution does not take into account the synchronization cost and delay.

Rajagopalan et al. [21] propose a state management system, which classifies state inside network function into internal and external. Internal state is required for a given instance to run and has of no effect outside that instance's execution. External state is the state that is shared across all instances of the same type. It needs to be maintained consistently between the same function's instances. This system provides a shared library used by instances to share the external state among them. This shared library supports a callback function, where each instance registers a callback function that is invoked automatically by the library when another instance updates its external state. This work also does not consider the synchronization cost and delay.

Xie et al. [22] propose a framework named *Dual* to address the synchronization problem between instances of the same network function. An approximation algorithm is designed to minimize the synchronization traffic between the instances by placing the instances close to each other in the physical infrastructure. However, this work take into consideration the synchronization delay.

Ma et al. [17] propose an algorithm for intrusion detection systems that can operate across multiple instances. This algorithm works by synchronizing processing, which means directly sharing state and waiting on other instances to complete processing as needed. This work does not use specific communication pattern for the data synchronization and does not consider the synchronization delay, which may lead to high delay in detecting potential attacks.

Unlike previous work that looked only on the way synchronization can be carried out, the novelty of our solution lies in the idea of considering how to minimize the cost of this synchronization while ensuring the synchronization delay does not exceed a certain bound in order to ensure that the network function is able to operate normally.

VIII. Conclusion

This paper addresses the problem of finding the optimal communication pattern to synchronize the instances in a way that minimize the synchronization cost while ensuring the required delay does not exceed a certain bound has been proposed. The problem was formulated as an ILP implemented in CPLEX and three heuristic algorithms (i.e., SPT, SSF, and MSF) were proposed to deal with large-scale scenarios. Though three algorithms shows much lower computational time compared to CPLEX, MSF is the one that significantly reduces execution time and provides near optimal solution.

As a future work, more optimizations techniques should be considered in order to further reduce the algorithms' complexity.

References

[1] A. Clemm, M. F. Zhani, and R. Boutaba, "Network management 2030: Operations and control of network 2030 services," *Journal of Network and Systems Management*, pp. 1–30, 2020.

[2] L. M. Vaquero, F. Cuadrado, Y. Elkhatib, J. Bernal-Bernabe, S. N. Srirama, and M. F. Zhani, "Research challenges in nextgen service orchestration," *Future Generation Computer Systems*, vol. 90, pp. 20–38, 2019.

[3] B. Varghese, P. Leitner, S. Ray, K. Chard, A. Barker, Y. Elkhatib, H. Herry, C.-H. Hong, J. Singer, F. P. Tso *et al.*, "Cloud futurology," *Computer*, vol. 52, no. 9, pp. 68–77, 2019.

[4] A. Gember, S. S. J. Anand Krishnamurthy, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud (technical report)," 2013.

[5] N. Ghrada, M. F. Zhani, and Y. Elkhatib, "Price and performance of cloud-hosted virtual network functions: Analysis and future challenges," in *IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 482–487.

[6] F. C. Chua, J. Ward, Y. Zhang, P. Sharma, and B. A. Huberman, "Stringer: Balancing latency and resource usage in service function chain provisioning," *IEEE Internet Computing*, vol. 20, no. 6, pp. 22–31, 2016.

[7] M. F. Zhani and H. ElBakoury, "FlexNGIA: A flexible Internet architecture for the next-generation tactile Internet," *Journal of Network and Systems Management*, pp. 1–45, 2020.

[8] "ETSI GS NFV-REL 001 V1.1.1 (2015-01), Network Functions Virtualisation (NFV) Resiliency Requirements," https://www.etsi.org/deliver/etsi_gs/NFV-REL/001_099/001/01.01.01_60/gs_NFV-REL001v010101p.pdf, accessed: 2020-04-10.

[9] H. B. Salameh, A. Musa, R. Outoom, R. Halloush, M. Aloqaily, and Y. Jararweh, "Batch-based power-controlled channel assignment for improved throughput in software-defined networks," in *IEEE International Multi-Conference on Systems, Signals & Devices (SSD)*, 2019, pp. 398–403.

[10] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 97–112.

[11] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions." in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 501–516.

[12] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI) 18)*, 2018, pp. 299–312.

[13] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 239–253.

[14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 163–174.

[15] M. Aloqaily, B. Kantarci, and H. T. Mouftah, "Provisioning delay effect of partaking a trusted third party in a vehicular cloud," in *IEEE Global Information Infrastructure and Networking Symposium (GIIS)*, 2014, pp. 1–3.

[16] P. Satapathy, J. Dave, P. Naik, and M. Vutukuru, "Performance comparison of state synchronization techniques in a distributed LTE EPC," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2017, pp. 1–7.

[17] J. Ma, F. Le, A. Russo, and J. Lobo, "Detecting distributed signature-based intrusion: The case of multi-path routing attacks," in *IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 558–566.

[18] T. Peng, C. Leckie, and K. Ramamohanarao, "Information sharing for distributed intrusion detection systems," *Journal of Network and Computer Applications*, vol. 30, no. 3, pp. 877–899, 2007.

[19] "ETSI GS NFV-REL 001 V1. 1.1, Network Functions Virtualisation (NFV) - Resiliency Requirements," 2015.

[20] M. Peuster and H. Karl, "E-state: Distributed state management in elastic network function deployments," in *IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 6–10.

[21] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 227–240.

[22] A. Xie, H. Huang, X. Wang, S. Guo, Z. Qian, and S. Lu, "Dual: Deploy stateful virtual network function chains by jointly allocating data-control traffic," *Computer Networks*, vol. 162, p. 106868, 2019.