A Hierarchical Distributed Control Plane for Path Computation Scalability in Large Scale Software-Defined Networks

Mohammed Amine Togou, Djabir Abdeldjalil Chekired, Lyes Khoukhi, and Gabriel-Miro Muntean

***Abstract*—Given the shortcomings of traditional networks, Software-Defined Networking (SDN) is considered as the best solution to deal with the constant growth of mobile data traffic. SDN separates the data plane from the control plane, enabling network scalability and programmability. Initial SDN deployments promoted a centralized architecture with a single controller managing the entire network. This design has proven to be unsuited for nowadays large-scale networks. Though multi-controller architectures are becoming more popular, they bring new concerns. One critical challenge is how to efficiently perform path computation in large networks considering the substantial computational resources needed. This paper proposes HiDCoP, a distributed high-performance control plane for path computation in large-scale SDNs along with its related solutions. HiDCoP employs a hierarchical structure to distribute the load of path computation among different controllers, reducing therefore the transmission overhead. In addition, it uses node parallelism to accelerate the performance of path computation without generating high control overhead. Simulation results show that HiDCoP outperforms existing schemes in terms of path computation time, end-to-end delay and transmission overhead.**

***Index Terms*—SDN, hierarchical control plane, path computation, large-scale networks**

I. Introduction

With the widespread adoption of smartphones and global endorsement of 3G/4G technologies, mobile data traffic has skyrocketed. According to Cisco Visual Networking Index [1], the global mobile data traffic grew from 4.4 exabytes per month in 2015 to 7.2 exabytes per month in 2016 and is expected to reach 49 exabytes per month by 2021. Given the inherent characteristics of nowadays networks (*i.e.*, extremely expensive, manually configured and lacking dynamic scalability), handling the constantly growing data traffic while ensuring high service quality is becoming complex and very laborious.

Software-Defined Networking (SDN) is a concept that was proposed to improve network performance and management. It decouples the control plane from the data plane to enable their independent evolution [2], [3]. This separation brings about numerous advantages, including high flexibility, programmability and high scalability. The first proposed SDN systems (*e.g.*, NOX [4] and Floodlight [5]) deploy a single centralized entity, called the *controller*, responsible for managing traffic flows and monitoring switches of the entire network. Despite their simplicity and ease of implementation, these systems have failed to meet the performance requirements of nowadays large-scale networks [6]. The reason is twofold: 1) with the increase of data traffic, the centralized controller will eventually get overwhelmed, leading to performance degradation; and 2) any minor disruption to the controller's activity may jeopardize the availability of the entire network.

Various techniques have been proposed to mitigate these shortcomings. For instance, [7] and [8] used multiple cores to enhance the Input/Output performance and enabled parallelism to support large networks. [9] and [10] redistributed part of the flow requests among switches to alleviate the controller's load. Alternatively, having a physically-distributed control plane is the approach gaining ground among the SDN research community. It consists of partitioning the network into multiple areas, each of which is managed by a distinct controller. ONIX [11], ONOS [12], Google's B4 [13] and Espresso [14] are examples of SDN multi-controller architecture.

In a distributed control plane, each controller maintains topology information, including the set of switches assigned to it (*i.e.*, association information), the set of links between them (*i.e.*, link view) and the paths between all-pairs of these switches (*i.e.*, path view). Failing to synchronize this information, namely link and path views, among all controllers may lead to inconsistent path computation results. This is referred to as the *synchronization problem* [15]–[17]. Yet, solving the synchronization problem in large-scale SDNs, particularly when topology changes are frequent or even moderate, puts computational pressure on the controllers and introduces high overhead due to the large number of control messages to be exchanged [18]. This impacts the efficiency of path computation, which in turn affects the performance of various SDN applications such as traffic engineering solutions [19], [20].

This paper proposes HiDCoP, a distributed control plane, and its related solutions that address the aforementioned problems while enhancing the performance of path computation in multi-controller SDNs. The paper has five contributions:

1) a hierarchical architecture that partitions the network into multiple domains, each of which is subdivided into several areas. Each area/domain is managed by a distinct controller;

2) a path computation mechanism that takes advantage of the hierarchical architecture along with the abstracted link views to construct routing paths between hosts, based on the locations of the communicating entities. This is to ensure rapid

M. A. Togou and G-M. Muntean are with the Performance Engineering Laboratory, School of Electronic Engineering, Dublin City University, Dublin, Ireland e-mail: ({mohammedamine.togou, gabriel.muntean}@dcu.ie)

D. A. Chekired and L. Khoukhi are with ICD/ERA, University of Technology of Troyes, Troyes, France email: ({djabir.chekired,lyes.khoukhi}@utt.fr).

and scalable path computation without generating high control overhead;

3) a path update procedure that updates the pre-computed path views to reflect changes in switches/links status. This is to avoid computing the path views from scratch, alleviating the computational burden on the various controllers;

4) a failover mechanism that is deployed to reinforce the system availability in case of controllers failure, enabling service continuity; and

5) a dual theoretical and simulation-based analysis to assess the feasibility and effectiveness of the proposed solution.

The rest of this paper is organized as follows. Section II surveys some related work. Section III presents HiDCoP's overall architecture along with the control plane components. Section IV describes the proposed hierarchical mechanisms for path computation and path update. Section V presents the recovery mechanism to cope with controllers' failure. Section VI describes the simulation settings and results while section VII concludes the paper.

II. Related Work

In this section, we first survey the hierarchical control planes that were proposed to tackle the problem of scalability in SDN; then, we review the existing approaches that aim at enhancing the performance of path computation in SDN.

A. Hierarchical Control Plane

Many approaches have been proposed to address the problem of scalability in SDN for different applications and from different perspectives. For instance, Phemius *et al.* [21] proposed DISCO, a distributed SDN control plane for WANs and overlay networks where each SDN controller is in charge of an SDN domain and exchanges aggregated network-wide information with other controllers. This information includes reachability (a list of reachable hosts in controllers domain), connectivity (path view), and monitoring (the status, latency and bandwidth of peering links). The exchange of such information is carried out using an agent-based architecture where agents publish and consume messages to ensure system's consistency. De Oliveira *et al.* [22] proposed *spotled*, a distributed control plane architecture for software-defined wireless sensor networks. It consists of using cluster-head nodes as local controllers and deploying a centralized controller responsible for keeping the overall network view as well as managing the entire network (*i.e.*, with the help of the local controllers).

Genge *et al.* [23] proposed a two-tier hierarchical control plane for network traffic optimization in Industrial Control Systems (ICS). The network is partitioned into several domains, each of which is managed by a controller (*i.e.*, bottom-tier). These controllers are managed by a centralized OpenFlow controller (i.e, top-tier), which updates the forwarding rules and exposes a communication interface that can be used to implement specially-tailored network traffic control strategies. Similarly, Zhao *et al.* [24] proposed a three-tier hierarchical control plane to meet traffic engineering applications' requirements. Using this architecture, the network is partitioned into two-level domains. The low-level domains are managed by local controllers, which are supervised by top-tier domain controllers. These latter are managed by the root controller, responsible for routing coordination across domains.

Koshibe *et al.* [25] proposed a multi-tier hierarchical control plane to increase service flexibility in SDN by distributing functionalities between multiple controllers. It endorses a client-server relationship between controllers at different tiers while enabling interconnection among controllers at the same tier. Yeganeh *et al.* [26] proposed Kandoo, a two-level hierarchical control plane to enable load balancing in the control plane without modifying OpenFlow [27] switches. The bottom-layer contains local controllers which are not interconnected and are responsible for handling events received from the data plane. The top-layer contains a logically centralized controller which maintains a network-wide view and is responsible for managing local controllers. Likewise, Xu *et al.* [28] proposed a 2-tier control plan architecture for load balancing and scaling of the control plane. The bottom-tier contains local controllers responsible for processing requests received from a middle plane, which plays the role of proxy between the control plane and the data plane. The top-tier includes the global controller which is responsible for managing the local controllers. Finally, Shah *et al.* [29] proposed Cuttlefish, a 2-tier hierarchical SDN architecture that enables traffic and computation offloading from root controllers to local ones. It uses developer-specified input to identify control messages that can be correctly processed at local controllers, and makes offloading decisions based on the cost of synchronizing the offloaded data across controllers.

Although they have demonstrated a great ability in dealing with the problem of SDN scalability, most of these schemes (*i.e.*, [21]–[25], [28], [29]) enable communication between controllers in the same tier in order to share topology information, yielding high control overhead that might hinder the performance of several applications (*e.g.*, traffic engineering). This has motivated us to propose HiDCoP, a distributed three-tier hierarchical control plane that only supports communication between controllers in different tiers to reduce control overhead (*i.e.*, like [26], controllers at the same tier are not interconnected).

B. Path Computation

Few approaches have been proposed to optimize the performance of path computation in SDN. For instance, Kouicem *et al.* [30] proposed a path computation algorithm for centralized SDNs to optimize flow transmission in WAN environment. It deploys BGP-LS [31] to collect information regarding the underlying WAN (*e.g.*, link states and traffic information) and forward it to the centralized controller. Based on a client-server architecture, the centralized controller uses the received information to respond to applications' path computation requests. Synchronization between the controller and the applications is carried out via exchanging report messages. Cho *et al.* [32] described a cloud-based approach that enables SDN controllers to delegate the task of path computation to an application, which can be part of a controller or installed

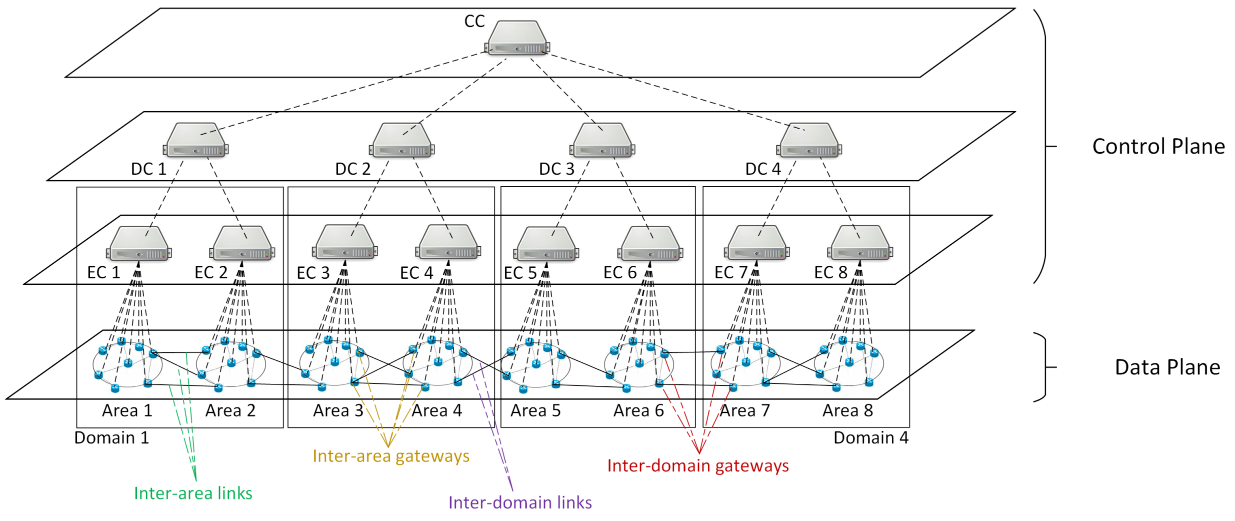


Fig. 1. HiDCoP's architecture. The control plane is made of 3-tiers: core controllers (CC), domain controllers (DC) and edge controllers (EC).

on an external server. This application collects, maintains and updates network information within its corresponding domain. When a path is to be established among multiple domains, each application computes the shortest path within its respective domain and shares it with the other applications of the involved domains. The shortest path is then computed using the backward recursive technique (*i.e.*, path computation starts from the destination and goes backwards till the source node is reached).

Qui *et al.* [15] proposed ParaCon, a parallel control plane that seeks to scale up path computation in SDN. It adopts a flat architecture and endorses a strong consistency synchronization for association and link view information (*i.e.*, all controllers must share this information immediately whenever a topology change has occurred). However, it deploys an eventual consistency policy (*i.e.*, allowing for a delay in synchronization) along with an asynchronous parallel algorithm to synchronize path view information among all controllers. This is to reduce message transmission for path computation, minimizing therefore the overall synchronization overhead and enhancing the path computation performance. Fu *et al.* [33] proposed Orion, a hybrid control plane that combines the advantages of flat and centralized hierarchical control planes to address the problems of computational complexity and path stretch. Orion is a two-tier layer architecture. The bottom layer is made of the area controllers, responsible for collecting link view information while the top layer contain the domain controllers, each of which managing several area controllers and keeping track of the global network topology. Path computation is performed by area or domain controllers, depending on where the destination switch is located, and path views are synchronized among the domain controllers. Backup paths for each pair of switches are also computed and stored at different controllers to quickly handle links' failure.

Despite their performance, these solutions have different limitations. While [30] suffers from the bottleneck and single point of failure problems, [15], [32], [33], might still incur high control overhead due to their flat architecture (*i.e.*, controllers should maintain and synchronize the global topology information), especially in large-scale networks where path requests are recurrent. In addition, Orion [33] computes path views from scratch whenever topology changes occur and no mechanism for fault tolerance was provided in case all controllers have the "Equal" role. This has motivated us to propose HiDCoP, a distributed and hierarchical control plane that aims at improving the scalability and the performance of path computation via: 1) endorsing a three-tier hierarchical architecture to reduce the control overhead due to path view synchronization; 2) enabling parallel path computation at the level of controllers in the same or different tiers to accelerate the computation process; 3) carrying incremental adjustments over pre-computed path views to reflect topology changes, further reducing the control overhead; and 4) proposing a resiliency strategy to protect the system from potential failures and attacks (*i.e.*, unlike [26], [33]).

III. Hierarchical Distributed Control Plane

HiDCoP is a distributed control plane that seeks to improve the scalability and performance of path computation in multi-controller SDNs. This section covers HiDCoP's overall architecture along with the components of the control plane.

A. The overall Architecture

Fig. 1 illustrates HiDCoP's architecture. It has two major components: the control plane and the data plane.

The *control plane* is physically distributed over three tiers, each of which is in charge of performing specific tasks. The lower tier is made of multiple Edge Controllers (EC) that are deployed near end users to enhance the quality of service of various applications. Instead of relaying application requests to remotely deployed controllers, edge controllers can process them locally, enabling therefore quick response time while reducing the computational overhead. Each edge controller is responsible for managing its own area. This includes maintaining and updating topology information and updating flow tables of its assigned switches. Domain Controllers (DC) make the middle tier of HiDCoP's control plane. They are responsible for managing their respective domains. This includes supervising edge controllers and computing inter-area paths when requested. To do so, each domain controller constructs the path view of its respective domain based on path views received from its corresponding edge controllers. It also maintains the list of inter-area gateways that keeps track of the switches connecting neighboring areas (see Fig. 1). Finally, the top tier contains one controller named the Core Controller (CC). It is responsible for supervising the domain controllers and computing inter-domain paths when requested. To do so, the core controller constructs the path view of the entire network using the path views received from all domain controllers. It also maintains the list of inter-domain gateways that keeps track of the switches connecting neighboring domains (see Fig. 1).

The *data plane* represents the physical network infrastructure (*i.e.*, switches and links). When a switch needs to forward a data packet to another switch, it sends a path request to its respective edge controller. Based on the destination's location, the edge controller can: a) compute the shortest path and update the flow table of the source and all the switches in the computed path; or b) forward the path request to the domain controller.

B. Control Plane Components

Fig. 2 depicts the various components of the control plane. There are six components, each of which is responsible for a specific task. The dashed arrows represent the communication, in steps, between the various components.

When an edge controller EC_i receives a path request via the *path request* module (step 1), it instructs the *device management* module to gather information about devices that are within its area (step 2) using the Link Layer Discovery Protocol (LLDP) (step 3). This information includes the switches' IP addresses, their type (gateway, non-gateway) and the IP addresses of the switches to which they are connected, and is used by the *link view* module to build the link view of EC_i's area (step 4). Note that whenever a switch becomes unavailable, the *device management* module informs the *link view* module in order to update the area's link view. The *path view* module in EC_i makes use of the built link view (step 5) to create and maintain the area's path view in order to select the shortest path between the source and the destination switches. In case the source and the destination are within EC_i's area, the shortest path is sent to the concerned switches through the southbound interface using the OpenFlow protocol (step 6). Otherwise, the path request is forwarded by EC_i's *path request* module to the domain controller DC_j via the *vertical communication channel* (step 7). This channel is established via a TCP connection and is used to forward device information, path requests and path views between controllers as well as to distribute rules to update flow tables.

Once DC_j receives the path request, it checks the information in the *device management* module (step 8) to verify whether the destination is within its domain. This information

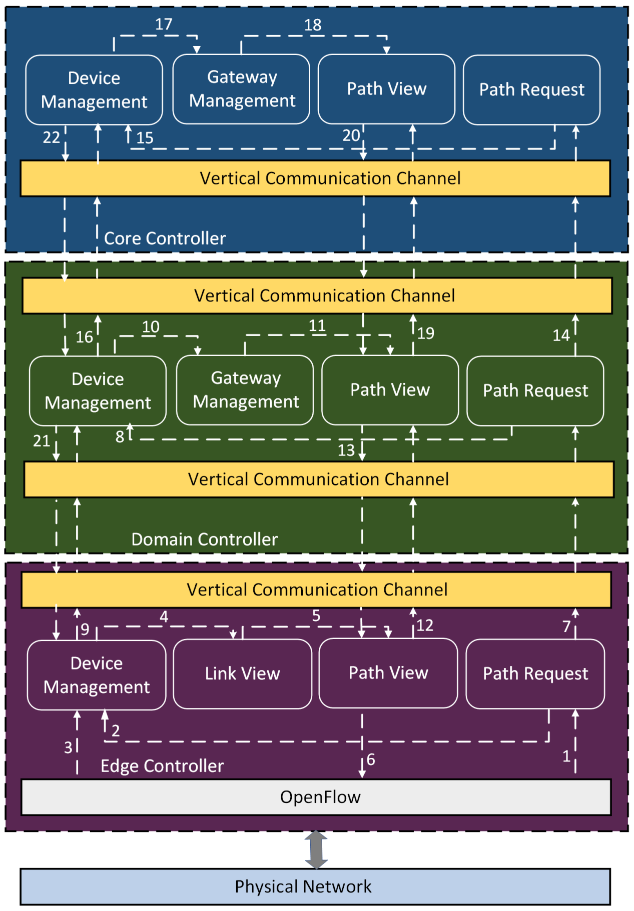


Fig. 2. Components of the control plane

is an aggregation of the device information received from all edge controllers managed by DC_j (step 9). The *gateway management* module uses this information to maintain the inter-area gateway table (IAGT) which keeps track of the switches connecting neighboring areas (step 10). Each entry in IAGT includes the switch's address, its status (*i.e.*, normal or special, described in Section V), the set of inter-area gateways to which it is connected and the address of the edge controller to which it is assigned (*i.e.*, the area). The *path view* module uses IAGT (step 11) along with the received path views from its respective edge controllers (step 12) to construct the path view of DC_j's domain. In case the source and the destination are within DC_j's domain, the shortest path is sent to the concerned switches through the appropriate edge controllers (step 13). Otherwise, the path request is forwarded to the core controller (step 14). This latter uses the information in the *device management module* (step 15), received from all domain controllers (step 16), to identify the domain where the destination is located. It then uses the inter-domain gateway table (IDGT) that contains the list of switches connecting neighboring domains (step 18) along with the path views received from the domain controllers (step 19) to construct the path view of the whole network and sends the shortest path to the concerned switches (step 20) through the corresponding domain and edge controllers. Steps 21 and 22 are carried out in case of a failure of an edge or a domain controller respectively and are covered in more details in Section V.

Note that HiDCoP's architecture discussed in this paper is designed to be deployed in a single autonomous system (AS). Nevertheless, to support routing between different autonomous systems, HiDCoP can integrate, at the level of the core controllers, the inter-SDN domain routing component [34] which implements a routing protocol that is based on BGPs most important features with some architectural differences to meet SDN requirements. Hereof, HiDCoP will first select inter-AS paths based on local preferences. In case two AS have the same local preference, path's length is then considered. To enable communication between the various instances of the inter-SDN domain routing component, TCP connections will be established between neighboring core controllers. This is similar to sessions created by BGP routers to exchange reachability information.

IV. HiDCoP's Path Computation and Update

In order to reduce the transmission overhead due to path computation and to alleviate the computational burden on the different controllers in case of topology changes, two mechanisms are proposed: Hierarchical Path Computation (HPC) and Hierarchical Path Update (HPU). This section first introduces some key notations and then describes both HPC and HPU in more details. Finally, it presents the cost analysis of both mechanisms.

A. Key Notations and Assumptions

Let S denote the set of switches with $|S|= n$ and let E designate the set of edge controllers with $|E|= m$. Each edge controller e_i maintains M_{e_i}, the set of switches within its respective area and O_{e_i}, the set of switches in its corresponding domain, excluding M_{e_i}, used for fault-tolerance purposes. In addition, let D denote the set of domain controllers with $|D|= r$. Each domain controller d_i maintains N_{d_i}, the set of edge controllers within its respective domain and N'_{d_i}, the set of switches within its respective domain.

We define the network topology as a graph $G(S, L)$ where S denotes the set of switches and L designates the set of links connecting them. Each element in L is a pair $(x, y), x, y \in S$. We represent the *link view* as the adjacency matrix L' with $L'_{x,y}$ denoting the weight of the link between x and y. If $L'_{x,y} = -1$, x and y are not connected. Moreover, we define the *path view* as the weighted graph $G'(S, W)$, where W is the weights of the best paths between the switches, and we represent it as the adjacency matrix P, with $P_{x,y}$ denoting the weight of the best path between x and y. The notion of *best path* can be defined based on the application's requirements (*e.g.*, shortest path, shortest delay, highest bandwidth). In this paper, *best path* means shortest path.

We assume that each switch can be managed by only one edge controller. Each switch has a forwarding table that contains the forwarding rules received from its corresponding edge controller. This includes the next hop of the best path to other switches as well as the IDs of the flows passing through that particular switch, represented as pairs (SrcIP, DstIP) where *SrcIP* denotes the IP address of the source switch while *DstIP* designates the IP address of the destination switch.

B. Hierarchical Path Computation (HPC)

HPC defines three path computation scenarios, depending on the location of both the source and destination switches: intra-area, inter-area and inter-domain. They are described in the following subsections.

1) Intra-area path computation: both the source and destination are within the same area. In this case, the source sends a path request to the area's edge controller. This latter gets the

Algorithm 1: Path Computation Algorithm

```
Data: $Src$, $Dst$, $M_{e_i}$, $G(M_{e_i}, L_{e_i})$
Result: $Path$ between $Src$ and $Dst$
if $Src \in M_{e_i}$ and $Dst \in M_{e_i}$ then
    // Intra-area path computation
    $G'(M_{e_i}, W_{e_i})$ = Moore($G(M_{e_i}, L_{e_i})$)
    Get Path from Src to Dst
    for $(x_i, y_i) \in Path$ do
        Install flow entry ($Src$, $Dst$)
else
    // $Src \in M_{e_i}$ and $Dst \notin M_{e_i}$
    if $Dst \in M_{e_j}$ and $M_{e_i}, M_{e_j} \in N_{d_k}$ then
        // Inter-area path computation
        $d_k \leftarrow$ path request from $e_i$
        $d_{(e_i,e_j)} \leftarrow$path view request from $d_k$
        for $e_u \in d_{e_i,e_j}$ do
            $G'(M_{e_u}, W_{e_u})$ = Moore($G(M_{e_u}, L_{e_u})$)
            $d_k \leftarrow G'(M_{e_u}, W_{e_u})$
        $M_d = M_{e_i} \cup ... \cup M_{e_j} \cup ga_{(e_i,e_j)}$
        $W_d = W_{e_i} \cup ... \cup W_{e_j} \cap W_{ga_{(e_i,e_j)}}$
        $G'(M_d, W_d)$ = Moore($G(M_d, L_d)$)
        Get Path from Src to Dst
        for $e_u \in d_{(e_i,e_j)}$ do
            for $(x_i, y_i) \in Path | (x_i, y_i) \in M_{e_u}$ do
                Install flow entry ($Src$, $Dst$)
    else
        // Inter-domain path computation
        $cc \leftarrow$ path request from $d_k$
        $c_{(d_k,d_u)} \leftarrow$ path view request from $cc$
        for $d_z \in c_{(d_k,d_u)}$ do
            for $e_v \in N_{d_z}$ do
                $G'(M_{e_v}, W_{e_v})$ = Moore($G(M_{e_v}, L_{e_v})$)
                $d_z \leftarrow G'(M_{e_v}, W_{e_v})$
            $G'(M_{d_z}, W_{d_z})$ = Moore($G(M_{d_z}, L_{d_z})$)
            $cc \leftarrow G'(M_{d_z}, W_{d_z})$
        $M_c = M_{d_i} \cup ... \cup ...M_{d_u}$
        $W_c = W_{d_i} \cup ... \cup W_{d_u} \cap W_{gd_{(d_i,d_u)}}$
        $G'(M_c, W_c)$ = Moore($G(M_c, L_c)$)
        Get Path from Src to Dst
        for $d_z \in c_{(d_i,d_u)}$ do
            for $e_v \in N_{d_z}$ do
                for $(x_i, y_i) \in Path | (x_i, y_i) \in M_{e_v}$ do
                    Install flow entry ($Src$, $Dst$)
```

source and destination IP addresses from the request message and checks M_{e_i} to make sure that the destination is within its area. Then, it uses $G(M_{e_i}, L_{e_i})$, stored in the *link view* module to construct P using an optimized version of the Bellman-Ford algorithm [35]. L_{e_i} contains all the links in the area except the inter-area links (see Fig. 1). Finally, it installs the flow entry in the flow tables of all the switches in the selected path between the source and destination.

2) Inter-area path computation: the source and destination are within the same domain, but in different areas. In this case, the source sends a path request to its respective edge controller. This latter gets the source and destination IP addresses from the request message and checks M_{e_i}. Given that the destination is not within its respective area, the edge controller forwards the path request to the domain controller. When received, the domain controller checks N'_{d_i} in the *device management* module to make sure that the destination is within its domain and to identify the area in which it is located. The domain controller then requests the set of edge controllers managing the areas between the source and the destination, denoted as $d_{(e_s,e_d)}$, to construct and transmit their path views (*i.e.*, this is done in parallel). Once received, it combines them with $ga_{(e_s,e_d)}$, the set of inter-area gateways connecting the areas between the source and the destination, to construct the path view between the source and the destination. Finally, with the support of the involved edge controllers, the domain controller updates the flow tables of all the switches in the selected path between the source and the destination.

3) Inter-domain path computation: the source and the destination are located in different domains. In this case, the source sends a path request to its respective edge controller, which forwards it to the domain controller. Given that the destination is not within its corresponding domain, the domain controller forwards the path request to the core controller. When received, the core controller gets the destination IP address from the message request and checks the *device management* module to identify the domain in which it is located. It then requests the set of domain controllers managing the domains between the source and the destination, denoted as $c_{(d_s,d_d)}$, to construct and transmit their path views (*i.e.*, by combining IAGT information with path views received from their respective edge controllers). This is also done in parallel. The core controller combines the received views with $gd_{(d_s,d_d)}$, the set of inter-domain gateways connecting the domains between the source and the destination, to construct the path view between them. Finally, with the support of the involved domain and edge controllers, the core controller updates the flow tables of all the switches in the selected path between the source and destination.

4) Proof: Algorithm 1 depicts the path computation process considering the three scenarios. The correctness of our algorithm can be proved by contradiction. Indeed, we can prove that the shortest path computed by HPC is the shortest path in the physical network. To do so, we first assert the following ***Lemma***: a shortest path is made of multiple shortest sub-paths. Let SP_{AB} be the shortest path between switches A and B computed by HPC. Assume that there is another path SP'_{AB} between A and B that is shorter than SP_{AB}. This means that

Algorithm 2: Path Update Algorithm

```
Data: Link View, $Q_u$
Result: Path view
while $Q_u$ not empty do
    $x \leftarrow Q_u.head()$
    if path weights can be optimized by including $x$ then
        Update weights in path view matrix
        $Q_u \leftarrow$ x's neighbors
    $x \leftarrow Q_u.head()$
```

SP'_{AB} contains one or more shorter sub-paths that are not included in SP_{AB}. Since SP_{AB} is the minimal length path that covers all possible links from A to B, it includes all the shortest sub-paths between A and B. This is a contradiction. Therefore, the shortest path computed by HPC corresponds to the shortest path in the physical network.

It is noteworthy to mention that for the sake of clarity, *Algorithm 1* assumes that path views are not pre-computed. However, when deploying HiDCoP, path views will be pre-computed and stored in the *path view* module. This is to reduce the transmission overhead. In case of a topology change, only incremental changes to the path views will be carried out, as will be explained in the next subsection.

C. Hierarchical Path Update (HPU)

In case of topology changes (*e.g.*, a link between two switches becomes unavailable or its weight has changed), HPU is deployed to only update path views to reflect the topology changes rather than computing them from scratch. HPU identifies three update scenarios: intra-area, inter-area and inter-domain.

1) Intra-area path update: when the link between switches $S3$ and $S4$ becomes unavailable (see Fig. 3), the edge controller $EC1$ is notified. Indeed, one of the switches (*i.e.*, in this case $S3$) sends a status message to $EC1$, informing it that the link $(S3, S4)$ is down. $EC1$ first updates the link view to reflect the change in $(S3, S4)$ status and proceeds to update the path view of its respective area. To do so, it executes *Algorithm 2*, which is based on the centralized algorithm in [15]. $EC1$ maintains a queue, labeled Q_{EC_1}, that stores the switches that need to be checked for path weight update (*i.e.*, $S3$ and $S4$). It starts by dequeuing $S3$, the node at the head of the queue, and checking whether the total weights of existing paths can be optimized if passed through $S3$. If yes, $S3$ neighbors (*i.e.*, $S1$) are added to the queue; otherwise, $EC1$ dequeues $S4$ and repeats the same process. This continues until the queue becomes empty.

Once the path view is updated, $EC1$ forwards it to the domain controller $DC1$, which will also update the path view of its domain. To do so, it first obtains the inter-area gateways connecting *Area 1* and *Area 2*. It then adds $S2, S4, S5, S7$ to the queue Q_{DC_1} and executes *Algorithm 2* as previously described. When finished, $DC1$ sends the updated path view to the core controller, which will get the inter-domain gateways and executes *Algorithm 2* by adding $S8, S9$ and $S10$ to the

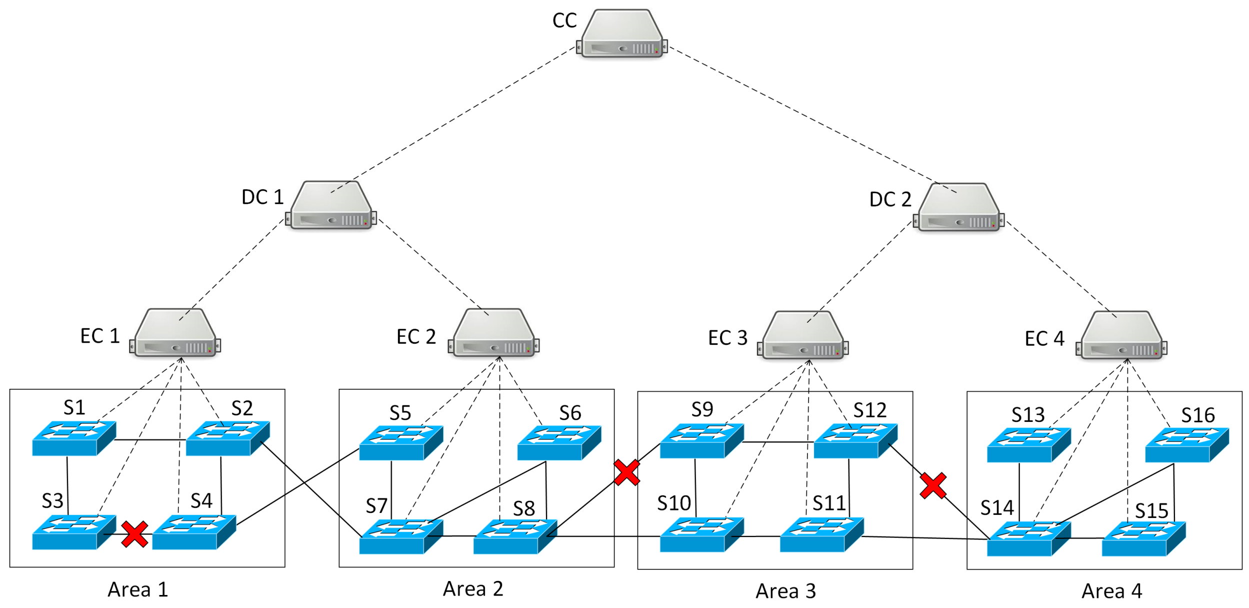


Fig. 3. Hierarchical path update in case of intra-area, inter-area and inter-domain link failure

queue Q_{CC} in order to update the path view of the whole network. Once done, CC, with the help of the domain and edge controllers, will update the flow tables of all the switches involved in the path view update process.

2) *Inter-area path update:* when the link between inter-area gateways $S12$ and $S14$ becomes unavailable (see Fig. 3), $S12$ and $S14$ notify their respective edge controllers $EC3$ and $EC4$, respectively. Both $EC3$ and $EC4$ notify $DC2$ about the change in $(S12, S14)$ status. $DC2$ adds $S12$ and $S14$ to the queue Q_{DC_2} and executes *Algorithm 2*. when finished, $DC2$ sends the updated path view to the core controller. This latter will add the inter-domain gateways $S8, S9$ and $S10$ to the queue Q_{CC} and executes *Algorithm 2* in order to update the path view of the whole network and eventually restore, with the help of the domain and edge controllers, the flow tables of all switches involved in the path view update process.

3) *Inter-domain path update:* When the link between inter-domain gateways $S8$ and $S9$ becomes unavailable (see Fig. 3), the core controller CC gets notified through $DC1$ and $DC2$ (*i.e.*, which are informed by $EC2$ and $EC3$, respectively). CC adds $S8$ and $S9$ to the queue Q_{CC} and executes *algorithm 2* to update the network path view. Once finished, it updates, with the help of the domain and edge controllers, the flow tables of all the switches involved in the path view update process.

D. Cost Analysis

Assume that there are K_d domains, each of which is made of K_a areas and each area has K_{s_i} switches (*i.e.*, for simplicity, we assume $K_{s_i} = K_s$). Let d' be the average number of links of each switch to other switches. Let M denotes the number of inter-area gateways per area and let M' be the number of inter-domain gateways per domain. Also, assume m is the number of inter-area links per area and m' is the number of inter-domain links per domain.

1) *Communication cost:* there are three types of communication costs. They are as follows:

Path view transmission cost: path views are sent in case of inter-area and inter-domain path requests. In case of an inter-area path request, edge controllers managing the areas between the source and the destination construct and transmit their path views to the domain controller. Let u be the number of areas involved. Therefore, the total path view transmission cost, T_p, in this case is:

$$T_p = u\left(K_s d' - \frac{Mm}{2}\right) \tag{1}$$

In case of an inter-domain path request, domain controllers managing domains between the source and the destination compute and send the path view of their respective domains to the core controller. This includes the transmission of path views of all the edge controllers belonging to these domains. Let u' be the number of domains involved. Therefore, the total path view transmission cost in this case is:

$$T_p = u'\left[K_a\left(K_s d' - \frac{Mm}{2}\right) - \frac{M'm'}{2}\right] \tag{2}$$

Path request transmission cost: a path request is made of four fields: source MAC (6 bytes), source IP address (4 bytes), destination MAC (6 bytes) and destination IP address (4 bytes). In case of an intra-area path computation, the path request is sent once (i.e, the source switch sends a path request to the edge controller). In case of an inter-area path computation, the path request is transmitted twice: from the source switch to the edge controller and then forwarded to the domain controller. Finally, in case of an inter-domain path computation, the path request is transmitted three times: first to the edge controller and then forwarded to the domain controller, which will forward it to the core controller.

Path update transmission cost: instead of involving all or most of the controllers whenever a topology change occurs (*i.e.*, the case of [15] and [32]), HiDCoP implicates only few controllers, depending on the link type whose status got altered. In case of an intra-area link failure, HiDCoP incurs three path update transmissions. In case of an inter-area link failure, HiDCoP incurs two path update transmissions. Finally, in case of an inter-domain link failure, HiDCoP incurs only one path update transmission.

2) Storage space: The control plane stores switch information and link information. Switch information has three fields: MAC address (6 bytes), IP address (4 bytes) and type (1 byte). Link information consists of physical and abstract links (*i.e.*, link and path views). Physical links are of three types: intra-area, inter-area and inter-domain. *Intra-area links* are twelve bytes long, including SrcSwitch IP address (4 bytes), SrcPort (2 bytes), DstSwitch IP address (4 bytes) and DstPort (2 bytes). Inter-area links are thirty-two bytes long, including SrcSwitch IP address (4 bytes), SrcPort (2 bytes), SrcEC MAC address (6 bytes), SrcEC IP address (4 bytes), DstSwitch IP address (4 bytes), DstPort (2 bytes) DstEC MAC address (6 bytes) and DstEC IP address (4 bytes). Inter-domain links are fifty-two bytes long, including SrcSwitch IP address (4 bytes), SrcPort (2 bytes), SrcEC MAC address (6 bytes), SrcEC IP address (4 bytes), SrcDC MAC address (6 bytes), SrcDC IP address (4 bytes), DstSwitch IP address (4 bytes), DstPort (2 bytes), DstEC MAC address (6 bytes), DstEC IP address (4 bytes), DstDC MAC address (6 bytes) and DstDC IP address (4 bytes). The abstract links are fourteen bytes long, including SrcSwitch IP address (4 bytes), DstSwitch IP address (4 bytes), nexthop (4 bytes) and weight (2 bytes).

The information stored in edge controllers includes: switch information, physical links, intra-area links and abstract links. Let q_i be the number of inter-area gateways in the i^{th} area with $i = 0, ..., K_a$ and $q_0 + ... + q_{K_a} = M$. Therefore, the total storage cost (in bytes) in each edge controller is:

$$C_{EC} = K_s * 11 + \frac{(K_s - q_i)d'}{2} * 12 + \frac{(K_s - q_i)}{2} * 14 \quad (3)$$

The information stored in domain controllers includes, switch information, inter-area links and abstract links. Therefore, the total storage cost (bytes) in each domain controller is:

$$C_{DC} = K_a K_s * 11 + K_a m * 32 + \frac{K_a K_s - M'}{2} * 14 \quad (4)$$

The information stored in the core controller includes, switch information, inter-domain links and abstract links. Therefore, the total storage cost (bytes) in the core controller is:

$$C_{CC} = K_d K_a K_s * 11 + K_d m' * 52 + \frac{K_d K_a K_s}{2} * 14 \quad (5)$$

3) Computation complexity: There are three types of path computations:

a) intra-area path computation, computed in $O(K_s^3)$;

b) inter-area path computation, which can be computed in $O(M^3 + K_s^3))$ as edge controllers can compute path views of their respective areas in parallel;

c) inter-domain path computation, which can be computed in $O(M'^3 + M^3 + K_s^3)$ by exploiting the parallel computation capability at the level of the domain and edge controllers.

It is noteworthy to mention that in order to ensure consistency in path views along with service continuity and network availability, HiDCoP should deal with situations where some edge or domain controllers may fail. This implies constructing link views of certain areas and updating their path views as well as the path views of their respective domains. This can incur additional transmission cost, as will be explained in the next section.

V. HiDCoP Failover

In order to mitigate the problem of isolated areas/domains due to edge/domain controllers' failure (*i.e.*, hardware faults or security attacks), HiDCoP deploys a mechanism that is fully compliant with OpenFlow. Indeed, OpenFlow enables switches to be connected to multiple controllers using different connection roles (*i.e.*, master, slave or equal). In HiDCoP, edge controllers of a particular domain are connected to all the switches within that domain, and domain controllers are connected to all switches of their neighboring domains. Switches in a specific area designate the edge controller managing that area as a master controller and the edge controllers of the same domain as slave controllers.

A. Edge Controller Failure

When the edge controller $EC2$ (see Fig. 4) becomes unavailable, the domain controller $DC1$ assigns M_{e_2}, the switches of Area 2, to either $EC1$ or $EC3$ or both. To do so, it first compares the size of both areas z_1 and z_3. If $z_1 > z_3 + M_{e_2}$, all switches in M_{e_2} will be assigned to $EC3$ whereas if $z_3 > z_1 + M_{e_2}$, all switches in M_{e_2} will be assigned to $EC1$. This is to balance the load between the two edge controllers. In case $z_1 < z_3$ or $z_3 < z_1$, $DC1$ executes *Algorithm 3* which uses I_1 and I_3, the set of inter-area gateways from Areas 1 and 3 that are connected to Area 2, to compute F_1 and F_3, the set of switches to be assigned to $EC1$ and $EC3$. It starts by adding all switches in I_1 to F_1 and all switches in I_3 to F_3. For each non-gateway switch s in M_{e_2}, it checks whether it has neighbors in F_1 or F_3. In case s has only neighbors in F_1 or in F_3, s is added to either F_1 or F_3, respectively. If s has neighbors in both F_1 and F_3, the shortest path between s and these nodes are computed and their weights are compared, based on which s is assigned to either F_1 or F_3. Once the assignment process is completed, $EC1$ and $EC2$ update their link views to accommodate the new switches and compute a new path view for their respective areas while $DC1$ updates IAGT to include the new inter-area gateways.

In order to be able to compute the shortest paths from s_1 and s_3 to s, $DC1$ needs to have the link view of area 2. HiDCoP compels inter-area gateways with the status field set to *special* to maintain the link view of their respective areas. When an edge controller becomes unavailable, the domain controller will use IAGT to reach one of these gateways to retrieve the link view of the disconnected area. Note that edge controllers send updated link views to these gateways whenever a change in topology occurs. Note also that when $DC1$ updates IAGT, it selects new inter-area gateways to keep track of the link views of their corresponding areas.

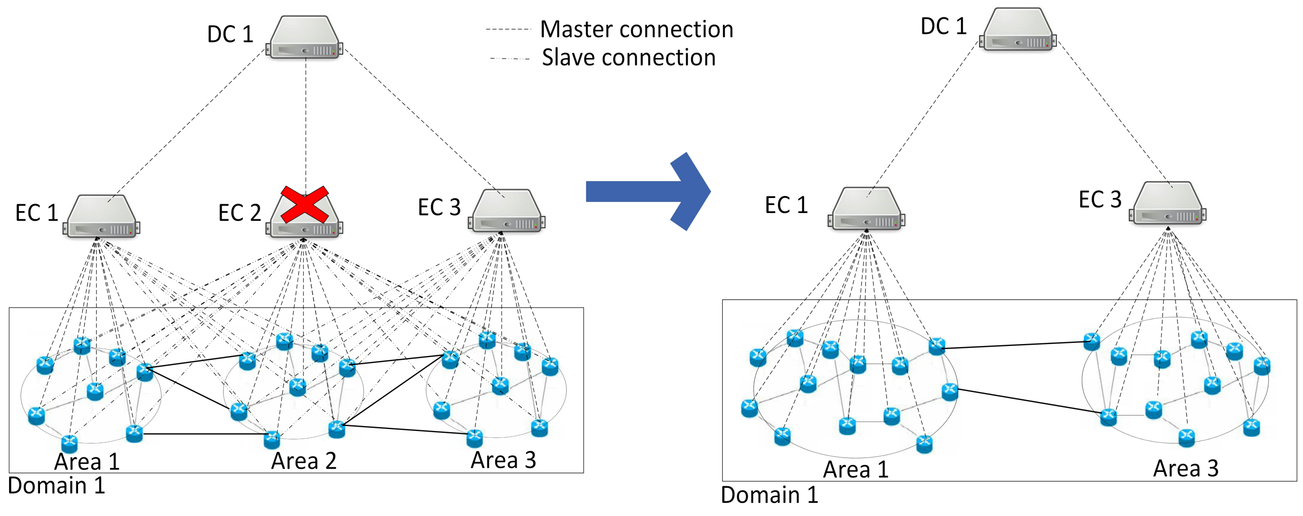


Fig. 4. HiDCoP failover mechanism when EC2 becomes unavailable

Algorithm 3: Failover Algorithm

```
Data: $M_{e_2}$, the set of inter-area gateways connecting
      areas 1 and 2 ($I_1$), and the set of inter-area
      gateways connecting areas 2 and 3 ($I_3$)
Result: The set of switches to be assigned to area 1 ($F_1$)
        and area 3 ($F_3$)
// add every element in $I_1$ to $F_1$
for $s_1 \in I_1$ do
    $F_1 \leftarrow s_1$
// add every element in $I_3$ to $F_3$
for $s_3 \in I_3$ do
    $F_3 \leftarrow s_3$
$M_{e_2} \leftarrow M_{e_2} \cap F_1 \cap F_3$
// for each non-gateway switch
for $s \in M_{e_2}$ do
    if s has neighbors from $F_1$ only then
        add s to $F_1$;
    else
        if s has neighbors from $F_3$ only then
            add s to $F_1$;
        else
            // s has neighbors $s_1$ and $s_3$
               from both $F_1$ and $F_3$, compute
               the shortest paths and
               compare their weights
            if $W_{s,s_1} < W_{s,s_3}$ then
                add s to $F_1$;
            else
                add s to $F_3$;
```

B. Domain Controller Failure

In case a domain controller becomes unavailable, the core controller uses IDGT to reach the edge controllers within the disconnected domain and to retrieve the path views of their respective areas. These areas are then assigned to one or multiple domain controllers, using *Algorithm 3*, allowing for load balancing between different domains. The core controller along with the involved domain controllers will update IDGT and IAGTs, respectively, to accommodate the new inter-domain and inter-area gateways. By doing so, HiDCoP can ensure service continuity while strengthening the system's availability.

C. Cost Analysis

Assume that we have a domain containing $K_a = 3$ areas, as illustrated in Fig. 4, each of which has K_{s_i} switches (*i.e.*, $i = 1, 2, 3$). For simplicity, let's assume that $K_{s_i} = K_s$. let d' be the average number of links of each switch to other switches in an area. Let M_i be the number of inter-area gateways in each area and let m_i be the number of inter-area links connecting areas K_i and K_{i+1}. The total failover cost in case of EC2 failure can be expressed as:

$$T_f = K_s d' + \left(2K_s d' - \frac{M_1 m_1 + M_2 m_2}{2}\right) \quad (6)$$

where $K_s d'$ represents the cost of transmitting the link view of area 2 from the inter-area gateway with status set to *special* to DC1 while the remaining component describes the transmission cost of path views from EC1 and EC3 to DC1.

Assume that the domain controller DC1 in Fig. 4 fails. In this case, area 1 and area 3 will be assigned to domain 2, managed by DC2. Therefore, the total failover cost can be expressed as:

$$\begin{aligned} T_f = &\left(2K_s d' - \frac{M_1 m_1 + M_2 m_2}{2}\right) \\ &+ \left(K_{a+2} K_s d' - \frac{M_{d_2} m_{d_2}}{2}\right) - \frac{M'_{1,2} m'_{1,2}}{2} \end{aligned} \quad (7)$$

where M_{d_2} is the number of inter-area gateways in domain 2, m_{d_2} is the number of inter-area links in domain 2, $M'_{1,2}$ is the number of inter-domain gateways between domains 1 and

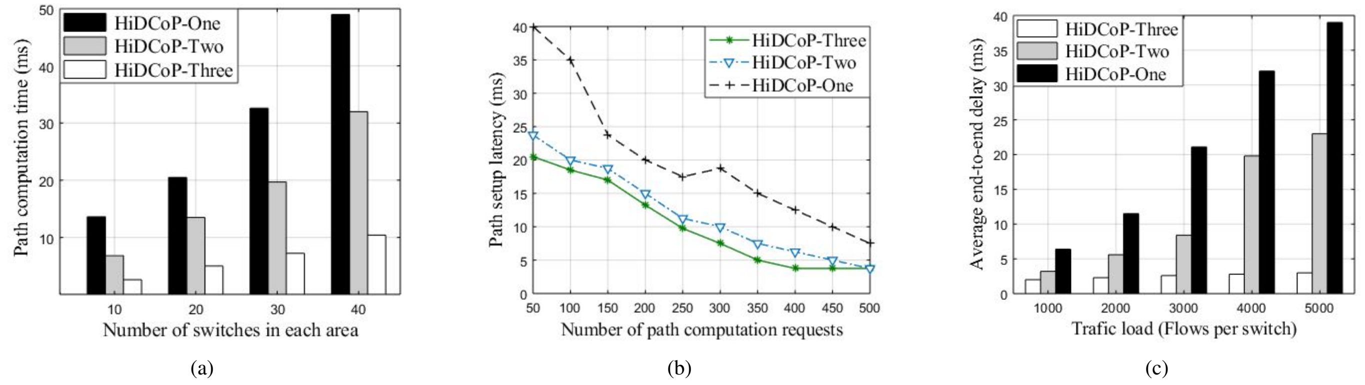


Fig. 5. Performance evaluation of the the three HiDCoP variants in terms of: a) path computation time; b) path setup latency; and c) average end-to-end delay

2 and $m'_{1,2}$ is the number of inter-domain links connecting domains 1 and 2. The first component of Eq. 7 represents the cost of transmitting path views of areas 1 and 2 to the core controller while the second component designates the cost of transmitting path views of the areas within domain 2 to the core controller.

VI. Performance Evaluation

This section presents a simulation-based evaluation of HiDCoP. HiDCoP's performance is compared to the state of the art solutions: POX [36], a widely used controller in the SDN research community that uses a centralized architecture, ONOS v1.3 [12] and ParaCon [15], which use a flat architecture, and Orion [33] which deploys a 2-tiers hierarchical architecture.

A. HiDCoP Implementation

HiDCoP was implemented by setting up virtual machines (VM) on Dell OptiPlex 7050 (Intel Core i5 CPU 2.71 GHz with 8GB RAM). There are 10 VMs, each of which is used as a controller. Three HiDCoP variants were implemented:

- HiDCoP-One: consists of one domain and eight areas.
- HiDCoP-Two: consists of two domains, each of which contains two areas. Both domains are managed by one core controller.
- HiDCoP-Three: consists of three domains. The first has one area, the second has two while the third has four areas. All domains are managed by one core controller.

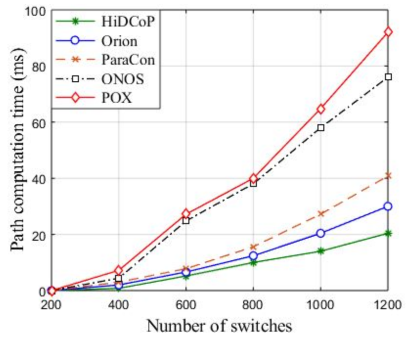
The data plane topology for each area is provided by Mininet. Each controller is based on a modified POX, where we replace the model of the path computation by our proposed path computation mechanism.

B. Simulation Results and Analysis

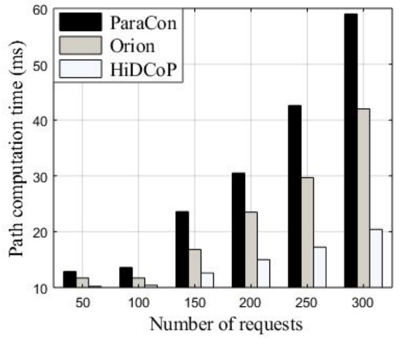
Fig. 5 depicts the performance evaluation of the three variants of HiDCoP with respect to path computation time, path setup latency and average end-to-end delay. We observe that HiDCoP-Three outperforms the remaining variants as it incurs the shortest path computation time, the shortest path setup latency and the lowest end-to-end delay. This is because HiDCoP-Three deploys more controllers than HiDCoP-One and HiDCoP-Two and balances the load efficiently among them. Unlike HiDCoP-Three, HiDCoP-One takes the longest time to compute and setup paths as the load on the domain controller is very high. Finally, as HiDCoP-two has the highest number of switches per domain, domain controllers take more time to compute the path view of their respective domains, yielding high path computation and path setup times.

In order to examine the performance of different schemes, a large-scale network with diameter 5 and made of a variable number of switches was deployed. HiDCoP in the following figures refers to HiDCoP-Three. Figs. 6(a) and 6(b) show the path computation time as a function of the number of switches and the number of requests, respectively. It can be noted that the path computation time increases with the increase in the number of both switches and requests. It can also be observed that HiDCoP outperforms all the remaining schemes as it incurs the shortest computation time. Indeed, HiDCoP achieves an average path computation time that is 27%, 55%, 116% and 122% shorter than Orion, Paracon, ONOS and POX, respectively. While ParaCon and ONOS maintain and synchronize the global topology information among all controllers, Orion synchronizes the path views among domain controllers in order to permit each one of them to build the network's path view. This makes the path computation performance of these schemes contingent to the network's diameter alongside the number of switches and links in the network. HiDCoP is not affected by this problem as it allows only the core controller to build and maintain the path view of the entire network and distributes traffic load over different controllers according to the type of the path request (*i.e.*, intra-area, inter-area and inter-domain). This enables quick path computation while reducing the transmission overhead.

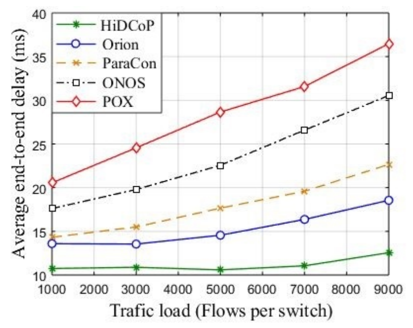
Fig. 6(c) shows the end-to-end delay as a function of traffic load. Note how the average end-to-end delay of all schemes increases with the increase in traffic load. As expected, POX generates the highest end-to-end delay since it uses a single centralized controller. Even though Orion, ONOS and ParaCon are distributed multi-controller architectures, they still incur high end-to-end delay. In fact, Orion, ParaCon and ONOS incur an average end-to-end delay that is 51%, 66% and 92% higher than HiDCoP. The reason is that HiDCoP computes paths quickly compared to the other schemes as it distributes



(a)

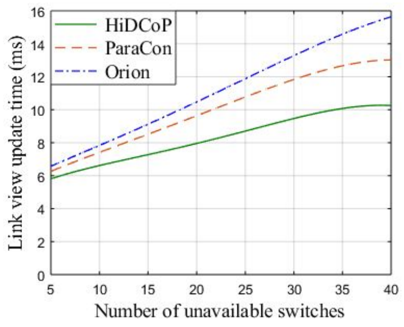


(b)

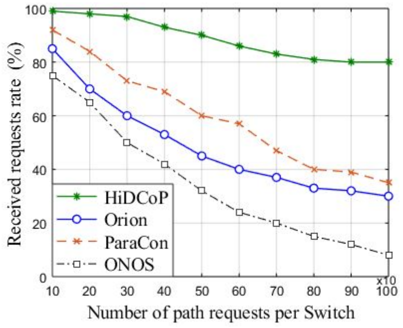


(c)

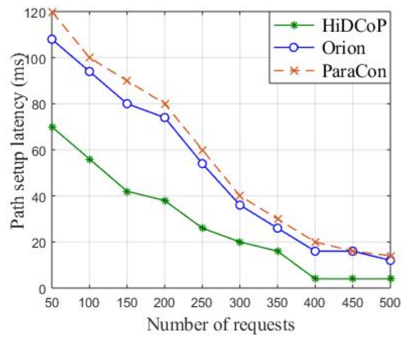
Fig. 6. Path computation time and average end-to-end delay in terms of network density and traffic load



(a)

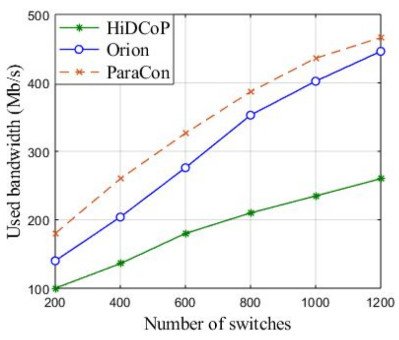


(b)

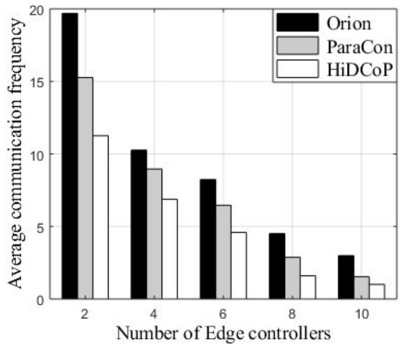


(c)

Fig. 7. Impact of topology changes on the various schemes in terms of: a) path view update time, b) received request rate and c) path setup time



(a)



(b)

Fig. 8. Control overhead: a) bandwidth used vs. number of switches in the network and b) average communication frequency vs. number of controller in the network

traffic load over different controllers according to the type of the path request and allows for the sharing of path views with upper-tier controllers only, therefore avoiding network congestion.

To simulate topology changes, we turned off random switches for a period of time and evaluated the performance of the various schemes in terms of path view update time, request delivery ratio (RDR) and path setup time. Fig. 7 depicts the results. Note how HiDCoP outperforms all other schemes as it incurs the shortest update time (*i.e.*, 19% and 31% less than ParaCon and Orion, respectively), the shortest path setup time (*i.e.*, 58% and 67% less than Orion and ParaCon, respectively) and the highest RDR (*i.e.*, 37.5% and 57% more than ParaCon and Orion, respectively). While Orion computes the path view from scratch whenever a topology change occurs, ParaCon and HiDCoP only update the pre-computed path views to reflect these changes. Still, unlike ParaCon, HiDCoP involves fewer controllers in the path view update process (*i.e.*, depending on the type of the unavailable link) and does not need to synchronize the updated path view among all the controllers in the network, yielding quicker path view update. Observe that Orion incurs lower path setup time than ParaCon (see Fig. 7(c)). This is because Orion computes backup routes for each routing path to address link failure, but it does not provide any mechanism to deal with link failure in the backup routes.

Finally, Fig. 8 depicts the communication overhead incurred at the controllers when scheduling different types of path computation requests. Fig. 8(a) portrays the control overhead in terms of bandwidth as a function of the number of switches in the network. We observe that the control overhead increases with the increase of the network density. We also observe that HiDCoP outperforms the remaining schemes as it incurs an average control overhead that is 57% and 67% less than Orion and ParaCon, respectively. Indeed, ParaCon adopts a flat architecture that requires each controller to inquire all the other controllers in order to construct and maintain the network's global topology information. HiDCoP and Orion adopt a hierarchical architecture where only the controllers that participate in the path computation process are involved. However, while Orion still synchronizes the global path view of the network among all domain controllers, HiDCoP allows only the core controller to have a network-wide view, further reducing the transmission overhead between controllers. To confirm this result, Fig. 8(b) illustrates the average communication frequency (*i.e.*, how often each controller is solicited during the path computation process) as a function of the number of controllers in the network. We observe that the communication frequency decreases with the increase in the number of controllers. This is rational since augmenting the number of controllers implies fewer switches per area/domain. We also observe that HiDCoP outperforms both Orion and ParaCon as it incurs an average communication frequency that is 29% and 56.5% lower than Orion and ParaCon, respectively.

VII. Conclusion

This paper proposes HiDCoP, a hierarchical and distributed control plane that improves the performance of path computation in large-scale SDN. HiDCoP adopts a 3-tiers architecture where the network is partitioned into domains, each of which is subdivided into areas. Each area is managed by an edge controller and each domain is supervised by a domain controller. All controllers are responsible for maintaining all or part of the topology information. HiDCoP defines three types of path computation and identifies the role of each controller in every one of them. This is to accelerate the path computation process, using parallelism, and to balance the load (*i.e.*, path computation request) among all controllers. It also describes a mechanism to update path views in case of topology changes along with a failover mechanism to address the problem of isolated areas/domains. Simulation results show that HiDCoP outperforms existing schemes in terms of path computation time, path setup latency and end-to-end delay. This makes HiDCoP a suitable candidate for large-scale software-defined networks.

Acknowledgment

This work was supported by the European Union's Horizon 2020 Research and Innovation program under Grant Agreement no. 688503 for the NEWTON project (http://newtonproject.eu).

References

[1] "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016-2021 [White paper]," https://www.cisco.com/mobile-white-paper-c11-520862.html, March 2017, accessed: 2018-04-05.

[2] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[3] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (sdn)," *Computer Networks*, vol. 112, pp. 279 – 293, 2017.

[4] N. Gude *et al.*, "NOX: Towards an Operating System for Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.

[5] "Floodlight Project," http://www.projectfloodlight.org/floodlight/, accessed: 2018-03-05.

[6] R. Trestian, K. Katrinis, and G. M. Muntean, "OFLoad: An OpenFlow-Based Dynamic Load Balancing Strategy for Datacenter Networks," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 792–803, 2017.

[7] A. Tootoonchian *et al.*, "On Controller Performance in Software-defined Networks," in *Proc. of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, ser. Hot-ICE'12, 2012, pp. 1–10.

[8] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A System for Scalable OpenFlow Control," Rice University, Tech. Rep., 2010. [Online]. Available: http://hdl.handle.net/1911/96391

[9] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 351–362, 2010.

[10] A. R. Curtis *et al.*, "Devoflow: Scaling flow management for high-performance networks," in *In ACM SIGCOMM*, 2011.

[11] T. Koponen *et al.*, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *Proc. of 9th USENIX Conf. on Operating systems design and implementation (OSDI'10)*, 2010, pp. 351–364.

[12] P. Berde *et al.*, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, 2014, pp. 1–6.

[13] S. Jain *et al.*, "B4: Experience with a Globally-deployed Software Defined WAN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[14] K.-K. Yap *et al.*, "Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, pp. 432–445.

[15] K. Qiu, S. Huang, Q. Xu, J. Zhao, X. Wang, and S. Secci, "ParaCon: A Parallel Control Plane for Scaling Up Path Computation in SDN," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 978–990, 2017.

[16] M. A. Togou, D. A. Chekired, L. Khoukhi, and G. Muntean, "A Distributed Control Plane for Path Computation Scalability in Software-Defined Networks," in *2018 IEEE Global Communications Conference (GLOBECOM)*, 2018, pp. 1–6.

[17] D. A. Chekired, M. A. Togou, and L. Khoukhi, "A Hybrid SDN Path Computation for Scaling Data Centers Networks," in *2018 IEEE Global Communications Conference (GLOBECOM)*, 2018, pp. 1–6.

[18] G. DeCandia *et al.*, "Dynamo: Amazon's Highly Available Key-value Store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[19] M. Wichtlhuber, R. Reinecke, and D. Hausheer, "An SDN-Based CDN/ISP Collaboration Architecture for Managing High-Volume Flows," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 48–60, 2015.

[20] D. A. Chekired, M. A. Togou, and L. Khoukhi, "Hierarchical Wireless Vehicular Fog Architecture: A Case Study of Scheduling Electric Vehicle Energy Demands," *IEEE Vehicular Technology Magazine*, vol. 13, no. 4, pp. 116–126, 2018.

[21] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed multi-domain SDN controllers," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–4.

[22] B. T. de Oliveira and C. B. Margi, "Distributed Control Plane Architecture for Software-Defined Wireless Sensor Networks," in *IEEE International Symposium on Consumer Electronics (ISCE)*, 2016, pp. 85–86.

[23] B. Genge and P. Haller, "A Hierarchical Control Plane for Software-Defined Networks-Based Industrial Control Systems," in *IFIP Networking Conference (IFIP Networking) and Workshops*, 2016, pp. 73–81.

[24] L. Zhao, J. Hua, X. Ge, and S. Zhong, "Traffic Engineering in Hierarchical SDN Control Plane," in *IEEE 23rd International Symposium on Quality of Service (IWQoS)*, 2015, pp. 189–194.

[25] A. Koshibe, A. Baid, and I. Seskar, "Towards Distributed Hierarchical SDN Control Plane," in *International Science and Technology Conference (Modern Networking Technologies) (MoNeTeC)*, 2014, pp. 1–5.

[26] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12, 2012, pp. 19–24.

[27] W. Li, W. Meng, and L. F. Kwok, "A survey on openflow-based software defined networks: Security challenges and countermeasures," *Journal of Network and Computer Applications*, vol. 68, pp. 126 – 139, 2016.

[28] Q. Xu, L. Li, J. Liu, and J. Zhang, "A Scalable and Hierarchical Load Balancing Model for Control Plane of SDN," in *Sixth International Conference on Advanced Cloud and Big Data (CBD)*, 2018, pp. 6–11.

[29] R. Shah, M. Vutukuru, and P. Kulkarni, "Cuttlefish: Hierarchical SDN Controllers With Adaptive Offload," in *IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, pp. 198–208.

[30] D. E. Kouicem, I. Fajjari, and N. Aitsaadi, "An Enhanced Path Computation for Wide Area Networks Based on Software Defined Networking," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017, pp. 664–667.

[31] O. G. de Dios *et al.*, "First multi-partner demonstration of BGP-LS enabled inter-domain EON control with H-PCE," in *Optical Fiber Communications Conference and Exhibition (OFC)*, 2015, pp. 1–3.

[32] H. Cho, J. Park, J.-M. Gil, Y.-S. Jeong, and J. H. Park, "An Optimal Path Computation Architecture for the Cloud-Network on Software-Defined Networking," *Sustainability*, vol. 7, no. 5, pp. 5413–5430, 2015.

[33] Y. Fu, J. Bi, Z. Chen, K. Gao, B. Zhang, G. Chen, and J. Wu, "A Hybrid Hierarchical Control Plane for Flow-Based Large-Scale Software-Defined Networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 2, pp. 117–131, 2015.

[34] R. Bennesby, E. Mota, P. Fonseca, and A. Passito, "Innovating on Interdomain Routing with an Inter-SDN Component," in *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, 2014, pp. 131–138.

[35] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[36] "SDN Hub, POX Controller Tutorials," http://sdnhub.org/tutorials/pox/, accessed: 2018-04-05.

Mohammed Amine Togou is a postdoctoral researcher with the Performance Engineering Laboratory at Dublin City University, Dublin, Ireland. He received a B.S. and M.S. degrees in computer science and computer networks from Al Akhawayn University in Ifrane, Morocco and a Ph.D. degree in computer science from the University of Montreal, Canada. His current research interests include connected vehicles, SDN-NFV, technology enhanced learning, IoT, smart cities and machine learning. Currently, he is working on the Horizon 2020 EU project NEWTON (http://newtonproject.eu).

Djabir Abdeldjalil Chekired is currently working towards a joint Ph.D. degree in computer science at the Environment and Autonomous Networks Laboratory, University of Technology of Troyes, France, and the School of Electrical Engineering and Computer Science, University of Ottawa, Ontario, Canada. His research interests include new cloud fog-computing design for software-defined networks, smart grid energy management, connected electric vehicles, and cybersecurity

Lyes Khoukhi received the Ph.D. degree in electrical and computer engineering from the University of Sherbrooke, Sherbrooke, QC, Canada, in 2006. In 2008, he was a Researcher with the Department of Computer Science and Operations Research, University of Montreal, Montreal, QC, Canada. Since 2009, he has been an Assistant Professor with the University of Technology of Troyes, Troyes, France. He has authored or co-authored over 60 publications in reputable journals, conferences, and workshops in the area of resources management in mobile and wireless networks. His research interests include mobile and wireless networks, resources management, QoS and multimedia, and communication protocols. He has participated as a General Chair, Session Chair, or Program Committee Member of many conferences.

Gabriel-Miro Muntean is an Associate Professor with the School of Electronic Engineering, Dublin City University (DCU), Ireland, and the Co-Director of the DCU Performance Engineering Laboratory. He has published over 300 papers in top-level international journals and conferences, authored three books and 18 book chapters, and edited six additional books. His research interests include quality, performance, and energy saving issues related to multimedia and multiple sensorial media delivery, technology-enhanced learning, and other data communications over heterogeneous networks. He is an Associate Editor of the IEEE TRANSACTIONS ON BROADCASTING, the Multimedia Communications Area Editor of the IEEE COMMUNICATIONS SURVEYS AND TUTORIALS, and a Reviewer for important international journals, conferences, and funding agencies. He is the Project Coordinator for the EU-funded project NEWTON http://www.newtonproject.eu.