Received December 19, 2020, accepted January 3, 2021, date of publication January 11, 2021, date of current version January 22, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3050922

Distributed Virtual Network Embedding for Software-Defined Networks Using Multiagent Systems

ALI AKBAR NASIRI[1], FARNAZ DERAKHSHAN[1], AND SHAHRAM SHAH HEYDARI[2], (Senior Member, IEEE)
[1]Electrical and Computer Engineering Department, University of Tabriz, Tabriz 5166616471, Iran
[2]Faculty of Business and Information Technology, University of Ontario Institute of Technology, Oshawa, ON L1H 7K4, Canada

Corresponding author: Ali Akbar Nasiri (ali_nasiri@tabrizu.ac.ir)

ABSTRACT Virtual Network Embedding (VNE), which provides methods to assign multiple Virtual Networks (VN) to a single physical Substrate Network (SN), is an important task in network virtualization. The main problem in VNE is the efficiency of assigning customers' virtual network requests to the substrate network. This problem is known to be a Non-deterministic Polynomial-time hard (NP-hard) and heuristic solutions have been developed to solve this kind of problem. The current trend toward Software-Defined Networking (SDN) has allowed new possibilities in virtual network embedding. In this work, we propose a distributed virtual network embedding for SDNs called DVSDNE using multi-agent systems. This framework could be used to run a centralized VNE algorithm in a distributed manner to scale these algorithms with respect to network size. DVSDNE uses agents to spread the load across the substrate network. Our simulation results show the effectiveness of the proposed algorithm. Results show that DVSDNE improves execution time of embedding algorithms in large scale substrate networks, while embedding results such as acceptance ratio, revenue to cost ratio, average latency to controller, and maximum latency to controller remain comparable.

INDEX TERMS Graph partitioning, multi-agent systems, network virtualization, software-defined networking (SDN), virtual network embedding.

I. INTRODUCTION

Over the past few decades, the Internet has provided a new approach to transmit information based on the deployment of packet switching network technology and its related applications. However, the traditional architecture of the Internet was a barrier to future innovations such as cloud computing, autonomous vehicles, and the Internet of Things, specifically, because of the several Internet service providers. Therefore, introducing new network architecture would not only need changes in hosts and routers but also requires agreements among Internet Service Providers. The large size of today's Internet made the extension of new network services a difficult and time-consuming challenge [1].

Network virtualization introduces an efficient approach to address this problem. On-demand virtual network structures that might contain various protocols are embedded in shared physical infrastructure. This provides a powerful solution to expand the future Internet by running several network services concurrently on a shared physical network [2].

Based on definitions, network virtualization includes leasing and sharing of the Substrate Network (SN) or physical network infrastructure, which consists of physical nodes and links. Network virtualization aims to increase the usability of physical resources and provides extensible and flexible networks for tenants. A Virtual Network (VN) includes a set of virtual nodes connected by virtual links and forming a virtual topology. A virtual node can be assigned to a substrate node and interconnected by a collection of virtual links. Virtual links may be embedded on multiple substrate links (path) [3].

With network virtualization, applications such as multimedia streaming can make virtual networks that satisfy application-specific constraints. However, the development of efficient strategies for embedding a virtual network into a physical substrate is challenging and remains an active area of research [4].

The Virtual Network Embedding (VNE) problem focuses on how to efficiently map virtual network requests to

The associate editor coordinating the review of this manuscript and approving it for publication was Rentao Gu.

substrate resources. Therefore, efficient usage of substrate resources depends on the VNE algorithms under limitations such as node and link constraints. The node constraints include factors such as CPU processing power and storage capacity. The link constraints may include parameters such as bandwidth (capacity), packet loss, and delay. Consequently, virtual node mapping and virtual link mapping are two steps of virtual network embedding. In the virtual node mapping step, virtual nodes are mapped to substrate nodes, providing enough resources. Similarly, in the virtual link mapping step, the virtual links are mapped to substrate links, providing enough resources. In general, finding a feasible embedding is an NP-hard problem [5] and it can be reduced to a multi-way separator problem [6].

So far, many algorithms have been proposed to solve the virtual network assignment problem [7]–[9]. The virtual network embedding algorithms may be implemented in a centralized manner or a distributed manner. In the centralized approach, a central entity receives virtual network requests and maps virtual nodes to substrate nodes and virtual links to substrate links. In order to make the right decisions, the entity needs to hold up-to-date information about the physical network. However, keeping latest information in a centralized manner has serious challenges in terms of scalability, high network latency, and delays in making decisions especially when the capacity of the physical network is dynamic and changing due to the creation or deletion of new virtual nodes and networks [10]. In the distributed approach, multiple entities cooperate to map virtual networks to substrate networks.

Software-Defined Networking (SDN) is an enabling technology for virtual networking and has received significant considerations [11]. SDN is a network architecture in which the data plane and the control plane are separated in networks and a central controller is responsible for making network control decisions and transmitting them to data-forwarding switches [12]. The network controller is responsible for managing the entire network through a southbound interface such as OpenFlow [13] which was introduced to standardize the communication between the controller and the switches in an SDN architecture.

Regardless of the topology, the network delay due to the distance between the SDN controller and switches as well as the workload on each controller are the parameters that affect the performance of the controllers [14].

The industry is considering SDN as a solution to simplify network control and the roll-out of new services while reducing hardware complexity and costs [15].

In OpenFlow architecture, each controller makes a TCP connection with its switches to exchange messages. The communication between the control plane and data plane can be implemented in two ways: 1) as in-band communication, 2) out-of-band communication. In-band communication uses the same links to connect the controller to forwarding devices, while out-of-band communication uses a separate network infrastructure to connect the controller to forwarding devices. Out-of-band signaling provides better isolation between data and control messages at the extra cost of additional hardware.

We also used in-band SDN in this work for two main reasons[16]: first, feasibility and lower-cost of in-band SDN, and second, link Failure Recovery.

- *Feasibility and lower-cost of in-band SDN:* Recently, OpenFlow which is used for connection between switches and its controller have got more attention in WLANs, cellular, wireless mesh networks, etc. Therefore, the need for a dedicated control plane network is the primary reason that the Out-of-Band control plane network becomes a costly solution for such networks. Besides, it is not even practical to have another separate network in such networks which nodes spread in a broad geographic area. By using an In-Band control plane, it is simple to connect distant switches to the controller indirectly and without a separate network
- *Link Failure Recovery:* One shortcoming of the Out-of-Band control plane is that there exists a unique path from the switch to the controller paths due to using separated and dedicated cabling and networks for the control plane network. Thus, failure of the control path will effectively remove that partition of the network from the control of the SDN controller. In this situation, link failure needs to be resolved manually to reconnect the switch to the controller. On the contrary, in the In-Band control plane, any link failure may affect both control and data planes, due to the use of the same link for data plane and control plane. However, the advantage of In-Band compared to Out-of-Band control network plane is the flexibility to modify the control traffic path and the capability to apply automatic failure recovery mechanisms.

Figure 1.a represents out-of-band communication and Figure 1.b represents in-band communication. In this work, in-band communication is used for embedding virtual networks. This would require considering both control traffic and data traffic together in terms of bandwidth demand for a virtual SDN network [1].

SDN allows for better optimization of routing policies and simplifies network testing [17]. Developers can program the network without considering the forwarding and the lower level detail of packet processing in physical devices [18].

While some previous virtual network embedding algorithms can be utilized for virtual network assignment in an SDN environment, some inherent characteristics of SDN, such as the location of the controller and Controller-to-Switches traffics, require new definitions and approaches [19].

Multi-agent systems consist of agents and their environment. In [20], an agent is considered as a computer system that is placed in an environment and is able to perform autonomous actions in this environment to reach its design goals. In other words, agents are networked entities that can carry out particular tasks and have a degree of intelligence that allows them to do their tasks autonomously and to

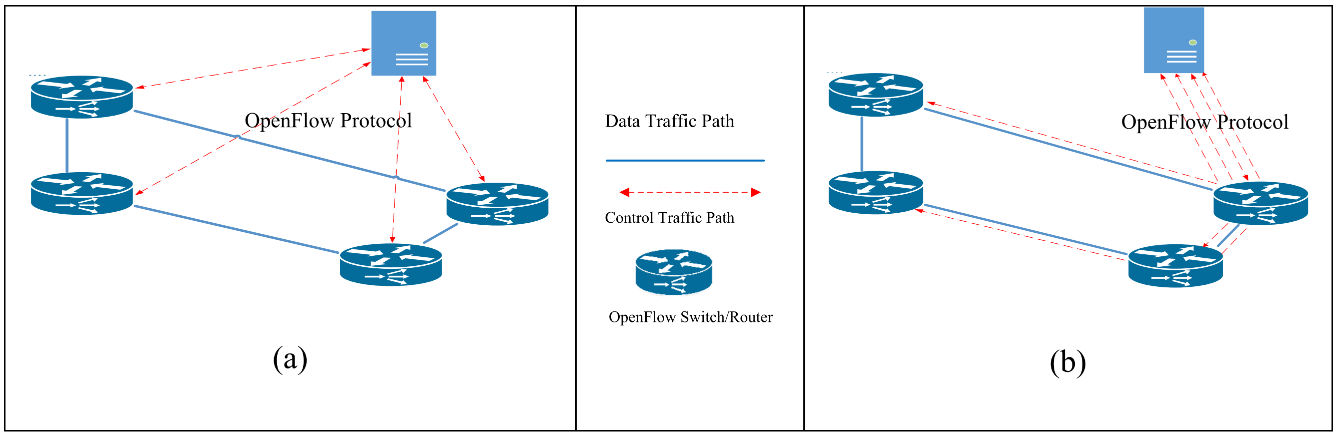


FIGURE 1. **(a) Separate network for control and data traffic (Out-of-band control network). (b) Same network for both control and data traffic (In-band control network).**

interact with their environment. In [21], a multi-agent system is considered as a system that consists of two or more agents, which cooperate with each other to reach their goals. Multi-agent systems can solve problems that are difficult for an individual agent to solve.

Solving computation tasks are becoming more complicated as the size continues to increase. As a result, it is difficult to handle these tasks in centralized manners. Although motivations to use of multi-agent systems in various areas are different, the main benefits of applying multi-agent systems include [22]: (1) the specific nature and environment of applications is considered; (2) local interactions can be studied and modeled; and (3) difficulties in computation and modeling are structured as components or parts. Therefore, multi-agent systems provide a good solution to distributed computational tasks. In addition, artificial intelligence methods can be applied.

The use of multi-agent systems on a particular area begins by partitioning the problem into smaller components. A particular agent is assigned to each of these components to perform particular tasks and reach its goals, thus making problem-solving process easier [23].

The motivation of this research was to develop a distributed framework for assigning virtual networks to substrate network resources in an SDN environment based on multi-agent systems. Each virtual SDN on the substrate network has its controller, and it needs special considerations that come with the existence of this controller. The controller is responsible for determining and sending traffic policies and routing updates. The controller needs to exchange information efficiently to all the switches that are part of the SDN, therefore, any virtual SDN embedding effort needs to concern all controller-to-switch delays.

Also, to reduce delays and improve scalability, at first, we partition a substrate network into parts. Partitioning allows the substrate network to be split into several smaller networks, resulting in VNE algorithms running on fewer nodes and links, which in turn reduces the execution time of VNE algorithms. In other words, partitioning aims to spread the computational load on several smaller parts. After partitioning, a Hypervisor virtualization tool is assigned to each part. Therefore, several Hypervisor virtualization tools are used for the main substrate network. In turn, this shortens the path between the switches and their controller, which reduces latency. A Hypervisor virtualization tool such as FlowVisor has been proposed for enabling virtualization in the SDN environment and it makes a substrate network to run multiple virtual networks. The Hypervisor allows each virtual network to run its virtual controller. Finally, we assign intelligent and autonomous agents to each part to manage its part. These agents cooperate to carry out the distributed VN mapping algorithms. A VN Mapping Protocol is designed and implemented to provide communications and facilitate message exchange among agents in a distributed manner. Agents at one level work independently of each other, and each can run a VNE algorithm simultaneously. Therefore, running VNE algorithms in this framework allows more VNs to be embedded at a given time. On the contrary, agents on different levels need to cooperate to map a VN request.

The remainder of this paper is organized as follows: Section II presents a short review of related work in the literature. In Section III, our contribution is presented. In Section IV, the virtual network embedding model and metrics are defined. Section V presents our distributed virtual network embedding for software-defined networks using a multi-agent systems framework called DVSDNE. Section VI contains results, and Section VII ends up with a conclusion and future works.

II. RELATED WORK

Existing work on assigning virtual private networks (VPNs) is similar to the virtual network embedding problem. However, a VPN request considers only bandwidth constraints without considering constraints on the nodes. As a result, VPN algorithms just try to find paths between source and destination nodes. Considering constraints on links and nodes are the subject of the virtual network embedding problem. The virtual network embedding is a complex problem and to decrease the complexity of it, current research simplifies the problem space in different dimensions, including the following [24]:

1. Considering specific topologies.
2. Considering only link or node restrictions.
3. Considering infinite resources of substrate links and nodes.

Virtual network embedding solutions can be classified into several types of underlying VNE problem as follows [24]:

1. Online or Offline. If virtual networks are mapped to the substrate network when virtual networks arrive, this class of embedding is called Online VNE [25], [26]. However, if all virtual networks remain in a queue for an arbitrary amount of time, this class of embedding is called Offline VNE 27]. This feature can be called static or dynamic as well.
2. Concise or Redundant. In the concise approach, the VNE algorithms use as many substrate resources as required to assign virtual networks to the substrate network [8]. However, in the redundant approach, the VNE algorithms reserve extra resources for virtual networks in case some substrate nodes may fail [7], [28], [29].
3. Uncoordinated or Coordinated. In the Uncoordinated approach, virtual node mapping and virtual link mapping are accomplished in two distinct phases. In other words, in the uncoordinated approach, at first, all virtual nodes are mapped one by one and then all links are mapped one by one according to some criteria. However, in the Coordinated approach, the coordination between node mapping and link mapping is considered [30].

Also, the VNE algorithm can be done in a centralized or distributed approach. In the centralized approach [31], the VNE algorithm utilizes only one substrate network for computing the embedding function. Centralized VNE algorithms suffer from scalability problems and running in parallel. From the scalability point of view, increasing network size increases the complexity of the virtual network embedding algorithm. Our solution handles scalability in a distributed manner. From a parallel point of view, in centralized VNE algorithms when multiple Virtual Network Request (VNR) arrives at the same time, embedding is performed sequentially, causing queuing delay in addition to high utilization at the embedder nodes. In other words, centralized VNE algorithms can only map a virtual network request simultaneously. Our solution solves this problem using multi autonomous agents.

SDN is an appropriate environment for network virtualizations. For the first time, the authors in [19], try to solve virtual network embedding when both virtual network and substrate network in the form of SDN. However, they do not consider the amount of CPU and bandwidth capacity of the substrate network. They only consider stress on nodes and links which is calculated based on how many virtual nodes or virtual links are assigned to substrate nodes and substrate links to solve the problem. Because the actual amount of CPU and bandwidth resources are very important for VNE algorithms, authors in [1] attempted to solve this problem by considering CPU and bandwidth resources. We use these two algorithms as centralized VNE algorithms to run in our framework and analysis of the results.

III. OUR CONTRIBUTIONS

In this section, we describe the research gap and our contributions.

Our main objective is to develop a distributed virtual network mapping in an SDN environment based on multiagent systems. Although there are some proposals and algorithms to address the VN mapping problem using multi-agent systems [10], [32], the unique characteristic of SDN requires new approaches, as discussed in Section II. In designing a solution for VSDNE, heuristics take into account the path between switches and the controller because the data plane and control plane is separated in SDN networks. However, in designing a solution for normal VNE, it does not need to consider this path because the data plane and control plane is not separated. We focus on scenarios in which the physical network and virtual networks are all software-defined. The infrastructures that we investigate may have many tenants requesting virtual networks of varying sizes. The placement of the vSDN controller is a significant factor that influences the performance of the virtual networks. Besides, efficient usage of substrate network resources, reliability, and fast response are all interesting properties. To this end, our main goal is to balance the load on the substrate network and keep delay minimum between controllers and switches in all virtual networks.

In this work, we design a distributed VNE algorithm that has the following characteristics and assumptions:

1. In this paper, we design a distributed approach for mapping virtual networks to substrate networks based on multi-agent systems, while most previous VNE algorithms are executed in a centralized approach and the virtual network assignment is performed by a central entity. This central entity keeps the latest changes in the substrate network. This approach suffers from scalability, latency, and delays in making decisions.
2. To manage cooperation between agents, we design and develop a communication protocol. Every agent is responsible for some part of the substrate network.
3. We use graph partition approaches to partition substrate network into distinct parts and assign each part a FlowVisor. Distinct parts along with its FlowVisor are managed by a single agent. Partitioning substrate network makes VNE problem space to be reduced and makes the substrate network to be scalable. Also, using multiple FlowVisor makes control traffic latency between switches and its controller to be reduced.

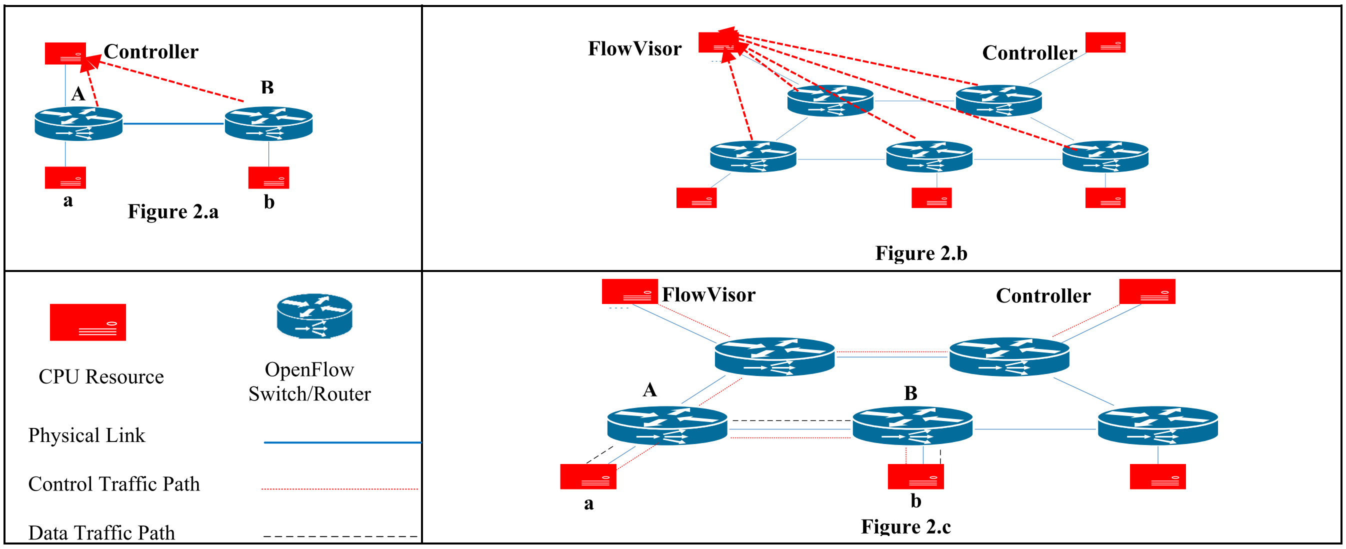


FIGURE 2. **Mapping of vSDN. FlowVisor (F) connects Controller (C) to other vSDN nodes (V).**

4. In our work, our algorithm does not reserve extra resources for control traffic between switches and their controllers. We use the same infrastructure for data and control traffic. In other words, our network uses an in-band control network.
5. In the distributed approach [33], [34], there are multiple substrate networks for computing the embedding function. Our solution is distinguished from them as it uses agents and runs in an SDN environment.
6. Our current method differs from our previous work [35], as the current work partitions the substrate network, while our previous work [30] did not partition the substrate network. Besides, in [35] agents cooperate to map a VNR at the same time, however, in this paper agents can embed multiple VNR at the same time.

IV. MODEL AND METRICS

In this section, we provide a detailed description of the problem of virtual network embedding with formula and also the main performance metrics for evaluation of this network embedding problem. In this work, we assume both virtual networks and substrate networks are in the form of SDN.

A. MODEL

Virtual network embedding is shown in Figure 2. Figure 2a shows a virtual SDN (vSDN) network request along with its controller. In our work, we use the same network for data and control traffic. Each switch uses an OpenFlow protocol to connect with its controller. The substrate network along with the position of FlowVisor is shown in Figure 2b. The embedding phase is represented in Figure 2c. The dashed lines represent virtual links assigned to physical links. During virtual SDN network operation, the control traffic is only transmitted via the physical paths, connecting the virtual SDN switches on the substrate network with its controller through the FlowVisor.

We model the substrate network as a weighted undirected graph $G_S(V_S, E_S)$, where V_S represents the set of substrate nodes, and E_S represents the set of substrate links. Also, a high-performance server hosting a hypervisor software and consequently virtual SDNs' controllers, is located on the substrate node Hypervisor, where Hypervisor $\in V_S$. Likewise, a virtual SDN (vSDN) request is modeled as a triplet $\Omega(G_R, C_R, LC)$. Here the undirected graph G_R (V_R, E_R) represents data plane of the vSDN, where V_R represents the set of virtual nodes and E_R represents the set of virtual links. There is a single software controller C_R for the vSDN. The controller C_R connects with every $v_R \in V_R$ through the virtual control links $lc(C_R, v_R) \in LC$, where LC represents the set of lc. Since the control flow is much smaller than data flow, here the bandwidth demand of $lc(C_R, v_R)$ is neglected. We suppose C_R runs on the same server with hypervisor software. As the communication latency in the internal server is much lower than network latency transmission, the controller to switch latency is equal to the network transmission delay from v_R to Hypervisor.

The VN embedding problem can be represented as a mapping from G_R to G_S:

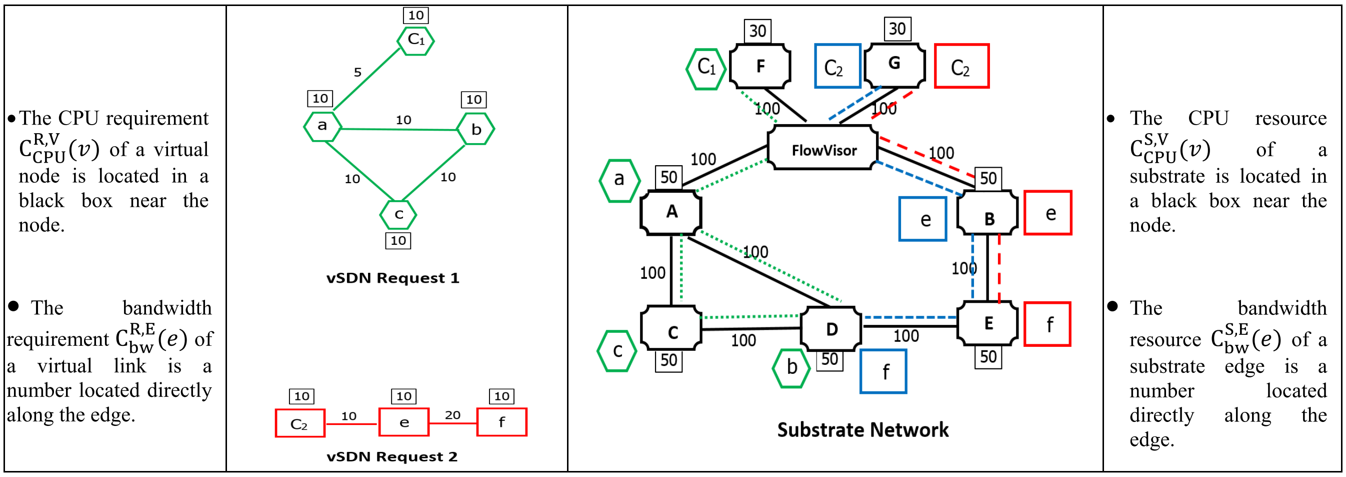
$$M(G_R) : G_R \to G_S \tag{1}$$

The mapping can be accomplished through two distinct phases: 1) node mapping 2) link mapping.

(1) Node mapping: Each virtual node is mapped to a substrate node that fulfills the node resource requirements. The node mapping function is defined by a mapping $M_N: V_R \to V_S$ from virtual nodes to substrate nodes such that,

$$\forall v_R \in V_R,\ M_N(v_R) \in V_S \tag{2}$$
$$\forall v_R, u_R \in V_R,\ M_N(v_R) = M_N(u_R), \quad \text{if } fv_R = u_R \tag{3}$$
$$\forall v_R \in V_R,\ C_{CPU}^{R,V}(v_R) \le R_{CPU}(M_N(v_R)) \tag{4}$$

(2) Link mapping: Based on the result of node mapping, each virtual link is mapped to a loop-free substrate path



- The CPU requirement $C^{R,V}_{CPU}(v)$ of a virtual node is located in a black box near the node.
- The bandwidth requirement $C^{R,E}_{bw}(e)$ of a virtual link is a number located directly along the edge.
- The CPU resource $C^{S,V}_{CPU}(v)$ of a substrate node is located in a black box near the node.
- The bandwidth resource $C^{S,E}_{bw}(e)$ of a substrate edge is a number located directly along the edge.

FIGURE 3. The assigning of two VN requests onto a substrate network.

that fulfills the link resource requirements. The link mapping function is defined by a mapping $M_L{:}E_R \rightarrow P_S$ from virtual links to substrate paths such that,

$$\forall\,(i,j) \in E_R,\ M_L\,(i,j) \in P_S\,(M_N\,(i)\,,M_N\,(j)) \qquad (5)$$

$$\forall\,(i,j) \in E_R,\ C^{R,E}_{BW}\,(i,j) \leq R_{BW}(M_L\,(i,j) \qquad (6)$$

where P_S defines the set of all loop-free paths in the substrate network and $P_S(i, j)$ defines the set of all loop-free paths from the source node i to the destination node j.

Each virtual node must be mapped to one substrate node, and a substrate node must not host more than one virtual node of the same vSDN. Eq. (2) and (3) guarantee these requirements.

Eq. (4) guarantees that the available CPU resource of the substrate node satisfies the CPU requirement of a virtual node.

Eq. (5) shows that each virtual link must be mapped to a substrate path between the substrate nodes that node i and node j are mapped on them. A substrate path consists of one or more physical links. Eq. (6) guarantees that the available bandwidth resource of the substrate path satisfies the bandwidth requirement of a virtual link. The available bandwidth resource of a substrate path P $\in P_S$ is defined as follows:

$$R_{BW}\,(P) = \min_{(u,v \in P)} R_{BW}\,(u,v) \qquad (7)$$

$R_{CPU}\,(v)$ *and* $R_{BW}\,(u,v)$ in Eq. (4) and (7) show residual CPU and Bandwidth resources respectively and is defined as follows:

$$R_{CPU}\,(v) = C^{S,V}_{CPU}\,(v) - \sum_{v' \to v : v' \in V_R : VNRaccepted} C^{R,V}_{CPU}\left(v'\right) \qquad (8)$$

where $C^{S,V}_{CPU}\,(v)$ represents CPU resource of a substrate node v and $C^{R,V}_{CPU}\left(v'\right)$ represents CPU requirement of a virtual node v'. The second part of the equation is the sum of the CPU requirements of all virtual nodes assigned to the node v of all accepted VN requests.

$$R_{BW}\,(u,v) = C^{S,E}_{BW}\,(u,v) - \sum_{\left(u',v'\right) \to (u,v) : (u',v') \in E_R : VNRaccepted} C^{R,V}_{BW}\left(u',v'\right) \qquad (9)$$

where $C^{S,E}_{BW}\,(u,v)$ represents bandwidth resource of a substrate link (u, v) and $C^{R,V}_{BW}\left(u',v'\right)$ represents the bandwidth requirement of a virtual link (u',v'). The second part of the equation is the sum of the bandwidth requirements of all virtual links assigned to the link (u, v) of all accepted VN requests.

Figure 3 shows the assigning results of vSDN for virtual network requests1 and 2. For the substrate network, the CPU resource is located in a rectangular box near the substrate node and a number located directly along the edge shows bandwidth resource. Similarly, for the virtual network, the CPU requirement is located in a rectangular box near the virtual node and a number located directly along the edge shows bandwidth requirement. For vSDN request 1 and 2, the controller is determined by C_1 and C_2 respectively.

As shown in Figure 3 for request 1, the virtual nodes a, b, c, and C1 are assigned to the substrate nodes A, D, C, and F respectively, and the virtual links (a, b), (a, c), (b, c) and (a, C1) are assigned to the substrate paths (A, D), (A, C), (D, C) and (A, HyperVisor, F) respectively. Note that the CPU resources of these substrate nodes and bandwidth resources of these substrate paths fulfill the CPU requirements of the corresponding virtual nodes and virtual links. Also, for request 2, the virtual nodes e, f, and C2 are mapped to the substrate node B, E and G respectively and virtual links (e, f) and (e, C2) are mapped to the substrate path (B, E) and (B, HyperVisor, G) respectively. Another embedding action for request 2 is that the virtual nodes e, f, and C2 are mapped to the substrate node

B, D, and G respectively and virtual links (e, f) and (e, C2) are mapped to the substrate path (B, E, D) and (B, HyperVisor, G) respectively. As shown here, each virtual request may have many assignments, and each imposes different utilization of resources. Therefore, VNE needs to apply an algorithm that minimizes the utilization of resources.

B. METRICS

The optimization goal of VNE is to use the physical network resources efficiently. Acceptance ratio and revenue to cost ratio are two evaluation measurement techniques for the effectiveness of the VNE algorithms[1]. In the following, we provide the relevant formula for the calculation of these metrics. Also, we provide the formula for another metric called Controller-to-switch delay.

The acceptance ratio shows the number of VNRs that could be mapped by a VNE algorithm and is measured by the following equation.

$$AcceptanceRatio = \frac{\sum_{t=0}^{T} VNR^S}{\sum_{t=0}^{T} VNR} \tag{10}$$

where VNR^S shows the number of the virtual network requests which are mapped successfully at time t, VNR shows the number of virtual network requests at time t, and T is total runtime.

TheRevenue tocost ratio metric is related to the acceptance ratio, as it shows the number of resources used by a VNE algorithm to map VN requests.

TheRevenue of a VN request is measured by the sum of CPU and bandwidth of the VN requests:

$$Revenue\left(VNR^i\right) = \alpha * \sum_{v \in VNR^i} C_{CPU}^{R,V}(v) + \beta * \sum_{e \in VNR^i} C_{bw}^{R,E}(e) \tag{11}$$

where $C_{CPU}^{R,V}(v)$ are the CPU requirements for the virtual node v $\in$VNR^i and $C_{bw}^{R,E}(e)$ are the bandwidth requirements for the virtual link e $\in$VNR^i. The value α and β reflect the relative importance of CPU resources and bandwidth resources, respectively, to the revenue.

TheCost is measured by the sum of the substrate resources used to map VN request:

$$Cost\left(VNR^i\right) = \alpha * \sum_{v \in VNR^i} C_{CPU}^{R,V}(v) + \beta * \sum_{e \in VNR^i} (C_{bw}^{R,E}(e) * Length(P(e))) \tag{12}$$

where P (e) is the assigned substrate path of a virtual link e $\in$ E$_R$, and Length(P(e)) is the length of that path. For example, as shown in Figure 3, virtual link (e, f) is mapped to the substrate path (B, E, D). Therefore, P (e, f) = (B, E, D) and Length (P (e, f)) = Length (B, E, D) = 2.

Therefore, the revenue to cost ratio is measured by the division of revenue and cost:

$$\frac{R}{C}\left(VNR^i\right) = \frac{Revenue\left(VNR^i\right)}{Cost\left(VNR^i\right)} \tag{13}$$

In general, for revenue to cost ratio, a higher value is better. The lower value means that a VNE algorithm uses more resources to embed VN requests. A value of 1 which would be optimal means that the amount of required resources equals the number of used resources.

Controller-to-switch delay measures the latency between the controller and the switches for each vSDN. For a network graph G (V, E) with vertex set V and edge set E, where number along edge represents propagation latencies, d(v, f, c) represents the shortest path from node v $\in$ V to controller c passing through FlowVisor (f), and n = |V | represents the number of nodes, the average propagation latency is defined as follows[36]:

$$Delay_{Avg} = \frac{1}{n} \sum_{v \in V} d\,(v, f, c) \tag{14}$$

Also, Maximum latency considers the maximum controller-to-switch delay and is defined as follows [31]:

$$Delay_{Maximum} = max_{v \in V}\, d\,(v, f, c) \tag{15}$$

V. OUR PROPOSED FRAMEWORK: DVSDNE

In this section, we introduce our method named Distributed Virtual Software-Defined Network Embedding (DVSDNE).

The algorithm works as follows: first, the substrate network is hierarchically structured into smaller partitions (Initialization Step). Second, an agent is assigned to each partition that aims to cooperate with other agents and provides virtual network embedding within that partition (Embedding Step). Partitioning the substrate network into smaller sections in the initialization step reduces the problem size and leads to better scalability and improved runtimes. In the embedding step, DVSDNE can run any centralized VNE algorithms which are designed for software-defined networks in a distributed manner. In other words, DVSDNE uses any centralized VNE algorithm within any given partition to perform actual embedding.

Unlike centralized approaches that depend on a single node to compute virtual network embedding, our framework distributes load to multiple nodes (agents) which works independently at each level in a hierarchical structure. Multiagent systems require a protocol to cooperate. This protocol provides the transfer of virtual networks and coordination among partitions.

Upon the arrival of the virtual network request, DVSDNE evaluates which partition is suitable to embed a new arriving virtual network. After determining which partition provides sufficient network resources for embedding, DVSDNE transfers the request to the agent assigned to that partition. If the specified agent cannot embed a virtual network request, DVSDNE consecutively transfers it to larger partitions until the root partition is reached or an agent can embed the virtual network request. In the case of reaching root partition, if root partition cannot embed the virtual network, our algorithm rejects embedding this virtual network.

In the following, the Initialization Step and the Embedding Step of the DVSDNE framework are explained in more detail.

A. INITIALIZATION STEP

DVSDNE runs the Initialization Step to divide the substrate network into smaller partitions. This step is performed before receiving the first virtual network embedding request and occurs only once for a substrate network.

DVSDNE organizes the substrate network into hierarchical partitions in such a way that in each layer a set of non-overlapping partitions is constructed. Many algorithms can be used to partition a graph [37]. DVSDNE uses a multilevel k-way partitioning algorithm [38]. The k-way graph partitioning problem is defined as follows: Given a graph G = (V, E) with vertex set V and edge set E, divide V into k subsets $V_1, V_2, \ldots, V_k$ such that the subsets are disjoint and cover V, i.e., $V_i \cap V_j = \varphi$ and $\cup V_i = V$. The algorithm aims to minimize the number of edges of E whose incident vertices belong to different partitions. In other words, the algorithm tries to group vertices that are highly interconnected. The basic structure of this algorithm is straightforward. At first, a sequence of smaller subgraphs $G_i = (V_i, E_i)$ is constructed from the main graph, through a process called the *Coarsening Phase*. Secondly, a k-way partitioning of the smallest graph is determined named *initial partitioning Phase* which DVSDNE uses 2-way partitioning. Finally, this partitioning is projected back toward the main graph by continuously purifying the partitioning during the *Uncoarsening Phase*.

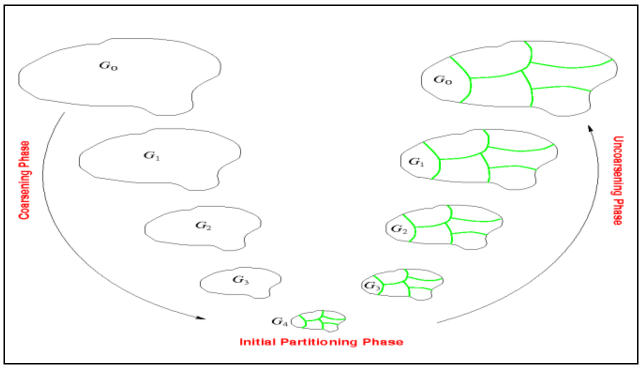


FIGURE 4. Phases involved in graph partitioning.

Figure 4 shows these phases. In this example, the Coarsening Phase consists of 5 levels named G_0, G_1, G_2, G_3, G_4. At each level, the size of the graph is reduced by collapsing edges and nodes. At Initial Partitioning Phase, the smallest graph which is G_4 is partitioned into 6 parts. At the Uncoarsening Phase, G_4 is expanded to recover partitioning for the main graph.

These partitions are used for embedding virtual networks. For each partition, DVSDNE assigns an agent named an embedder agent and a FlowVisor controller. An embedder agent performs the actual VNE algorithm. Because each agent is aware of the available resources and the topology of the allocated partition, it can perform an embedding algorithm within the allocated partition. After the partition algorithm has terminated on each level, the algorithm is continuously repeated to constitute smaller partitions on the next level.

Figure 5 shows an example, where the physical network is partitioned once. In this example, there are 4 switches and 3 FlowVisor controllers. FlowVisor 1 and switches 1, 2, 3, and 4 constitute level 0. Since at level 0 there is one partition, agent 0 will be assigned to this level. At level 1 there are two partitions and for each partition, there is one FlowVisor. FlowVisor 1 and switches No. 3 and 4 constitute partition 1 and agent 1 will be assigned to this partition. Similarly, FlowVisor 2 and switches No. 1 and 2 constitute partition 2, and agent 2 will be assigned to this partition. Since at level 1 there are two partitions, two agents (one for each partition) will be assigned to this level.

B. EMBEDDING STEP

The Embedding Step is performed after completing the Initialization Step. As we mentioned before, in the Initialization Step, the partitioning of the substrate network is set up and in the Embedding Step, the actual VNE algorithm is performed.

Upon the arrival of aVNR^i, CPU and bandwidth resource demand are calculated as follows:

$$Demand\left(VNR^i\right)_{CPU} = \sum_{v\in VNR^i} C^{R,V}_{CPU}(v) \tag{16}$$

$$Demand\left(VNR^i\right)_{BW} = \sum_{e\in VNR^i} C^{R,E}_{bw}(e) \tag{17}$$

Suppose that part of the substrate network managed by Agent_j is showed by the graph $G(V_S, E_S)$. The load balancer agent determines all agents satisfying the following equations:

$$\sum_{v\in V_S} R_{CPU}(v) - Demand\left(VNR^i\right)_{CPU} \geq 0 \tag{18}$$

$$\sum_{(u,v)\in E_S} R_{BW}(u,v) - \gamma * Demand\left(VNR^i\right)_{BW} \geq 0 \tag{19}$$

$R_{CPU}(v)$ *and* $R_{BW}(u,v)$ in Eq. (18) and Eq. (19) show residual CPU and Bandwidth resources respectively and is calculated using Eq. (8) and (9) as follows

Load balancer uses Eq. (18) and Eq. (19) to estimate which agents are sufficient resources to perform embedding. Since a virtual link may assign to multiple physical links the requested bandwidth is multiplied by a parameter γ.

Then, the merit value for those agents is determined using the following equation:

$$MeritValue\left(\mathrm{Agent_j}\right) = \left(\sum_{\mathrm{v}\in \mathrm{V_S}} \mathrm{R_{CPU}}(\mathrm{v}) - \mathrm{Demand}\left(\mathrm{VNR^i}\right)_{\mathrm{CPU}}\right) * \left(\sum_{(\mathrm{u,v})\in \mathrm{E_S}} \mathrm{R_{BW}}(\mathrm{u,v}) - \gamma * Demand\left(\mathrm{VNR^i}\right)_{\mathrm{BW}}\right) \tag{20}$$

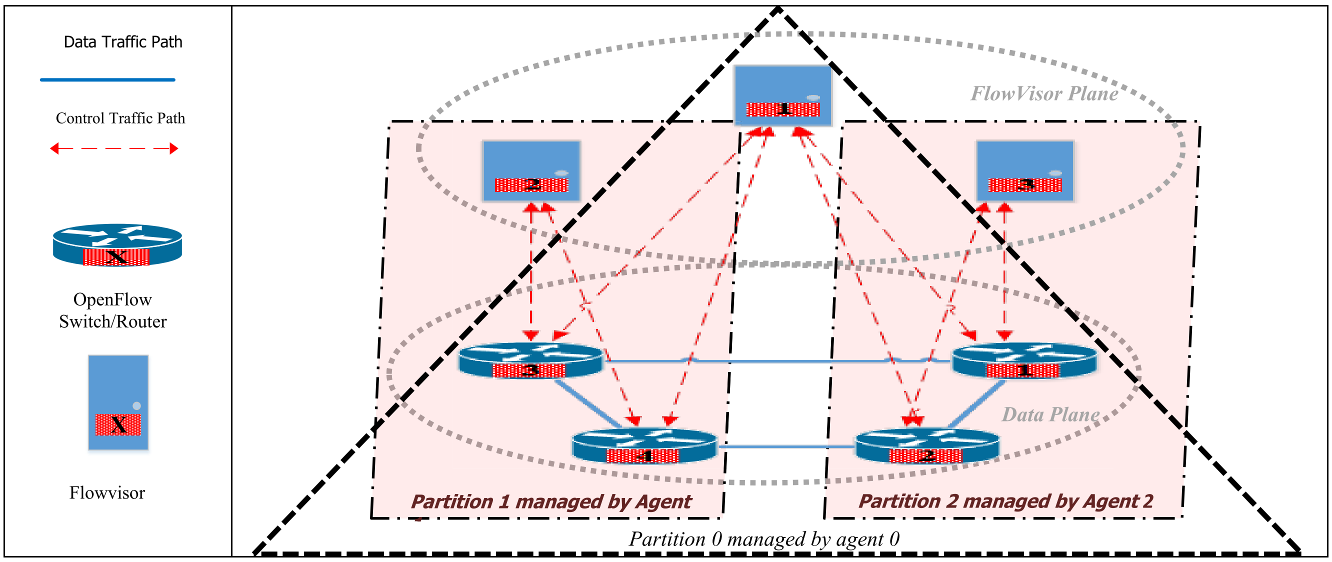


FIGURE 5. Partitioning the substrate network and assigning the agent at each level.

An agent with the minimum merit value is selected as an appropriate embedder agent. After an appropriate embedder agent is found, the load balancer agent forwards a Virtual Network Request VNR to that embedder agent. If the embedder agent cannot embed the VNR, it informs the load balancer agent that it cannot embed this VNR. After the load balancer receives this information, it sends the VNR to the parent agent, also asks it to embed VNR. This procedure is successively performed until the VNR is embedded or the agent assigned to the root partition cannot embed this VNR which means that VNR is rejected.

C. AGENT COMMUNICATIONS

Since agents act autonomously, they have to synchronize their actions with each other. Synchronization is done using a protocol by sending messages to each other. Also, a locking mechanism is used to prevent resource contention between agents that share the same parts of the substrate network. To explain resource contention between agents, we illustrate Figure 6 (FlowVisor controllers are not shown). Figure 6 shows an example of hierarchical partitioning and assigning agents to each partition. It can be seen that the substrate network is partitioned twice and partitions do not overlap at each level. Suppose the agent assigned on the right partition of level 1 (including nodes 5-8) is mapping a VNR. At the same time, the agent assigned on the rightmost partition of level 2 (including nodes 7 and 8) is running another embedding. Since these two agents share the same parts of the substrate network and these two agents work in parallel, resource contention occurs. In other words, to avoid resource contention between these two agents, when one of them is running an embedding procedure another has to wait until the embedding procedure is finished. It is worth noting that the agent assigned on the right partition of level 1 and the agent assigned on the left partition of level 1 can work in parallel without inconsistency.

To solve this problem, the load balancer agent keeps a data structure where information about the states of agents is held. This data structure includes two variables named Self-lock and Parent-lock. Each of them holds a binary data type. The variable of Self-lock can be in one of two states: Unlock or Lock. The variable of Parent-lock can be in one of two states: Waiting or Ready. The descriptions about each state are described in the following:

Unlock: Neither the agent nor its parents are running the Embedding algorithm.

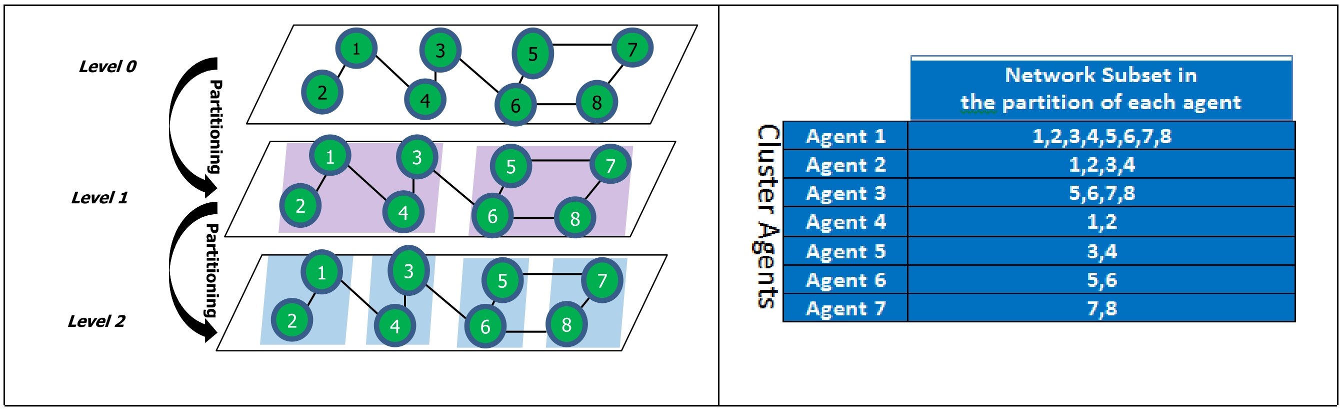
Lock: The embedder agent or one of its parents is running the Embedding algorithm. Therefore, the agent does not know up-to-date resource mapping information on all of its nodes and links. Therefore, the load balancer agent cannot send it a VNR at the moment. It needs to wait until all of its children terminate embedding.

Waiting: At least, one of the parents of the embedder agents wants to run the Embedding algorithm.

Ready: None of the parents of the embedder agent wants to run the Embedding algorithm.

In the following, communication protocol including messages communicated between agents is described in details:

START (VNR): This message is used by the load balancer agent to send a VNR to an embedder agent. When an embedder agent receives this message, firstly, it receives a resource assignment update from its children using UPDATE_GET message, then it runs the Embedding algorithm. This message is sent when the Self-lock and Parent-lock state of the embedder agent is Unlock and Ready respectively. This message makes the Self-lock state of the embedder agent and all of its children change to Lock.



	Network Subset in the partition of each agent
Agent 1	1,2,3,4,5,6,7,8
Agent 2	1,2,3,4
Agent 3	5,6,7,8
Agent 4	1,2
Agent 5	3,4
Agent 6	5,6
Agent 7	7,8

FIGURE 6. An example of hierarchical partitioning.

UPDATE_GET: This message is used by the parent of the embedder agent to receive resource assignment update from its children.

UPDATE_SET: This message is used by the parent of the embedder agent to send a resource assignment update to its children.

END (Success | Failure): This message is used by an embedder agent to inform the load balancer agent about its result. If the embedding is successful, END (Success) is sent to the load balancer agent, otherwise, END (Failure) is sent. When the load balancer agent receives END (Success), the Self-Lock state of the embedder agent and all of its children change to Unlock. Also, the embedder agent sends a resource assignment update to its children using UPDATE_SET message.

When the load balancer agent receives END (Failure), if the sibling embedder agent of the sender of this message or one of its children is running embedding (i.e. the Self-Lock state of the agent is Lock), the load balancer agent waits for them to finish embedding. The load balancer waits for agents by setting the state of them to Waiting. After all children of the parent embedder agent finish embedding, the load balancer agent wants the parent embedder agent of the sender of this message to start embedding using START (VNR) message.

Algorithm 1 shows VN embedding Algorithm for each embedder agent in pseudo-code. The algorithm consists of 3 distinct parts S_1, S_2, and S_3. Each of embedder agents, upon receiving **START** message (part S_1), first checks whether it has children or not. If it has, it sends a message **UPDATE_GET**(part S_2) to receive resource assignment update from its children. Next, after it receives the resource assignment update, it tries to embed the VNR. If the mapping result is successful, it sends a resource assignment update to its children by using the message **UPDATE_SET** (part S_3 **)**.Also, it sends **theEND (Success)** to the load balancer agent to inform that the embedding result is successful. Otherwise, the embedder agent sends **END (Failure)** to load balancer agent to send the VNR to another embedder agent.

Algorithm 2 shows VN embedding Algorithm for the load balancer agent in pseudo-code. The algorithm consists of 3

Algorithm 1 The Main VN Mapping Algorithm for Each Embedder Agent

```
(S1) Upon receiving START message do
        If Embedder Agent has children
            Receive update from all children using
             UPDATE_GET message
        Result = embed(v)
        If Result is successful
            Send an update to all children using
            UPDATE_SET message
            Send END (Success) message
        Else
            Send END (Failure) message
(S2) Upon receiving UPDATE_GETmessage do
        Send up-to-date resource assignment to sender
(S3) Upon receiving UPDATE_SET message do
        Set resource update from sender
```

distinct parts S_1, S_2, and S_3. The load balancer agent, upon receiving VNR (part S_1), first finds an appropriate agent based on the states and merit value of embedder agents. Next, after the load balancer agent finds the appropriate embedder agent, it sends the VNR to that embedder agent using **START** message and asks it to embed the VNR. Accordingly, the load balancer agent updates its data structure to prevent sending another VNR to this embedder agent. Based on the embedding result, the load balancer agent gets **END (Success)** or **END (Failure)**. If it receives **END (Success)**(part S_2), the load balancer agent updates its data structure accordingly. Otherwise (part S_3), it waits for all children of the sibling embedder agent to finish embedding. Next, the load balancer agent updates its data structure and asks the parent embedder agent to embed this VNR.

For clarification, a mapping scenario is described in the following (see Figure 7 and Table 1).

Suppose the states for each agent are at initial states according to Table 1. As Table 1 shows, the Self-lock of embedder agent C is Lock which means it is trying to embed a VNR

Algorithm 2 The Main Algorithm for Load Balancer Agent

```
(S1) Upon receiving VNR do
        Send VNR to appropriate embedder agents
        based on \
                states and merit value of agents
        Update its data structure
(S2) Upon receiving END (Success) message do
        Update its data structure
(S3) Upon receiving END (Failure) message do
        Wait until all children of sibling embedder agent
\
                finish embedding
        Update its data structure
        Send START message to parent embedder agent
```

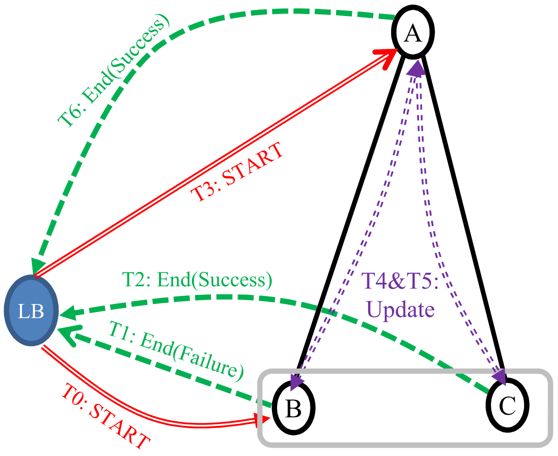


FIGURE 7. Mapping scenario described our multi-agent systems.

TABLE 1. Agent's states during the running of DVSDNE for the sample mapping scenario described by Figure 7.

Time	Agents / States	Embedder Agent A	Embedder Agent B	Embedder Agent C
Initial states	*Self-lock*	*Unlock*	*Unlock*	*Lock*
	Parent-lock	*Ready*	*Ready*	*Ready*
T0	*Self-lock*	*Unlock*	*Lock*	*Lock*
	Parent-lock	*Ready*	*Ready*	*Ready*
T1	*Self-lock*	*Unlock*	*Unlock*	*Lock*
	Parent-lock	*Ready*	*Waiting*	*Waiting*
T2	*Self-lock*	*Unlock*	*Unlock*	*Unlock*
	Parent-lock	*Ready*	*Waiting*	*Waiting*
T3	*Self-lock*	*Ready*	*Ready*	*Ready*
	Parent-lock	*Lock*	*Lock*	*Lock*
T4	*Self-lock*	*Ready*	*Ready*	*Ready*
	Parent-lock	*Lock*	*Lock*	*Lock*
T5	*Self-lock*	*Ready*	*Ready*	*Ready*
	Parent-lock	*Lock*	*Lock*	*Lock*
T6	*Self-lock*	*Unlock*	*Unlock*	*Unlock*
	Parent-lock	*Ready*	*Ready*	*Ready*

named VNR_I. Below describe the states of each agent at a different time.

T0: The load balancer agent receives VNR_1 and it is forwarded to agent B by START (VNR_1) message. Thus, the states for each agent are according to time T0 in Table 1.

T1: Agent B runs the Embedding algorithm. However, agent B cannot embed the VNR_1 and sends an END (Failure) message. Since the agent C is embedding the VNR_I, the load balancer agent waits for it to finish the embedding. The states for each agent are according to time T1 in Table 1.

T2: After agent C finishes its embedding successfully, the states for each agent are according to time T2 in Table 1.

T3: Load balancer agent sends VNR_1 to agent A which is the parent embedder agent of agent B and wants it to embed VNR_1. Thus, the states for each agent are according to time T3 in Table 1.

T4: Agent A sends UPDATE_GET message to agent B and C to receive resource assignment update from its children. The states for each agent are equal to the state at time T3.

T5: After updates are done, agent A runs embedding. This time embedding is successful and then agent A sends UPDATE_SET to agent B and C to send resource assignment update to its children. The states for each agent are equal to the state at time T3.

T6: Agent A informs the load balancer agent that embedding is successful using the END (Success) message. The states for each agent are according to time T6 in Table 1.

VI. PERFORMANCE EVALUATION

In this section, the simulation environment, the performance metrics the results, and the complexity of DVSDNE are discussed. Then, we will evaluate our performance. We apply vSDN embedding in an offline manner, considering all vSDN requests are known in advance.

A. SIMULATION TOOLS

Here, we describe the tools and packages that we used for the simulation of our method. First, we mention the tool that we used for topology and resource generation followed by referring to the tool we used for partitioning. Then, the simulation tool for implementing multiagent systems is described.

1) TOPOLOGY AND RESOURCE GENERATION

Networkx tool [39] is a Python package for creating and studying complex networks. We use this package to generate random SN and VN topologies. Random CPU resources from 50 to 100 are uniformly generated and assigned to the substrate nodes and random bandwidth resources from 250 to 500 are uniformly generated and assigned to the substrate links. Also, the latency between adjacent substrate nodes is set to be 1. Likewise, Random CPU and bandwidth resources are uniformly generated between 25 and 50 and assigned to virtual nodes and links. Moreover, we assume that control plane traffic is negligible compared to the data plane traffic. Also, we consider α and β equal to 1 in Eq. (11) and (12) for measuring both revenue and cost metrics. Besides, parameter γ is set to 10 in parts B.1 and B.2 of the result section. All experiments run on a stand-alone personal computer with core-i3 CPU running at 3.6 GHz and 4 GB main memory.

2) INITIALIZATION STEP

As described in Section V, at Initialization Step, DVSDNE partition the substrate network into hierarchical parts. DVSDNE uses the Metis tool[1] that provides an implementation of the algorithm of [38] to create a binary tree hierarchy. At each level, every part of the substrate network is partitioned into two sections to constitute another level.

3) MULTIAGENT SYSTEMS FRAMEWORK

For simulating a multiagent system, we used PADE [23] which is a framework for developing, running, and managing multiagent systems in distributed computing environments. PADE is an open-source platform implemented in Python language and uses the twisted libraries for implementing the communication between the network nodes. Although this framework provides a development solution for distributed systems and also is compliant with the specifications of the Foundation for Intelligent Physical Agents, we need to add appropriate codes to implement our desired system.

B. EVALUATION

In this section, the performance of DVSDNE is evaluated based on three items: generality of DVSDNE, scalability of DVSDNE, and Evaluation of DVSDNE's parameter γ.

Generality means that centralized VNE algorithms can run in DVSDNE and scalability means that DVSDNE scales well in large substrate networks.

Before moving onto larger topologies and to show the generality of our work, we are running our framework with two centralized VNE algorithms designed for SDN i.e., [1], [19] and compare the message overhead with a basic algorithm designed for distributed VNE [10]. Table 2 lists these methods and Table 3 shows and describes the notations used for evaluating DVSDNE.

TABLE 2. List of VNE approaches used for evaluating DVSDNE.

Algorithm	Description
SBE[19]	Stress balancing embedding heuristic
NR[1]	Node ranking embedding heuristic

TABLE 3. Notation used for comparing algorithms.

Notation	Description
SBE	Run SBE Algorithm in a centralized manner
SBE_DVSDNE	Run SBE Algorithm in our framework
NR	Run NR Algorithm in a centralized manner
NR_DVSDNE	Run NR Algorithm in our framework

To demonstrate the generality of DVSDNE, we compare the embedding results of DVSDNE with those of basic centralized algorithms.

Large scale network topologies are used to evaluate the scalability of DVSDNE. These results demonstrate DVSDNE framework scales with the size of substrate networks. Also, runtime measurements are shown, representing that DVSDNE improves runtime in large scale substrate networks.

[1] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

1) GENERALITY IN DVSDNE

This section shows and analyzes results for measuring DVSDNE's generality. To study the behavior of different algorithms, network load is varying from 2 to 12 virtual networks. The results of the DVSDNE framework are evaluated based on acceptance ratio, Revenue/Cost, average latency to the controller, maximum latency to the controller, and message overhead.

We ran our algorithm for 20 different scenarios. Each scenario consists of different VNRs with 4 virtual nodes. Then, we ran our framework for these different scenarios one by one. For each scenario, we calculate the acceptance ratio and other metrics. Finally, we calculated the average of these metrics for 20 scenarios to calculate the final metrics. Also, in this subsection to measure the performance metrics, we used a substrate network with 16 nodes.

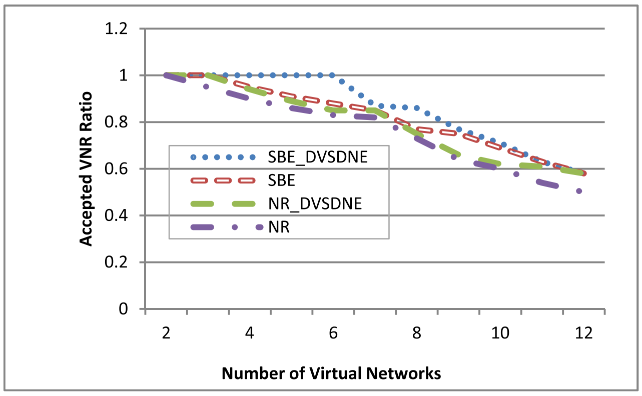


FIGURE 8. Accepted VNR ratio.

a: ACCEPTANCE RATIO AND REVENUE/COST RATIO

Figure 8 shows DVSDNE's acceptance ratio is comparable to the acceptance ratio of basic centralized algorithms. The same is obtained from the Revenue/Cost metric as shown in Figure 9. Both metrics get a little better performance when they run within in DVSDNE framework. Mapping SVNRs to partitions instead of a whole substrate network improve

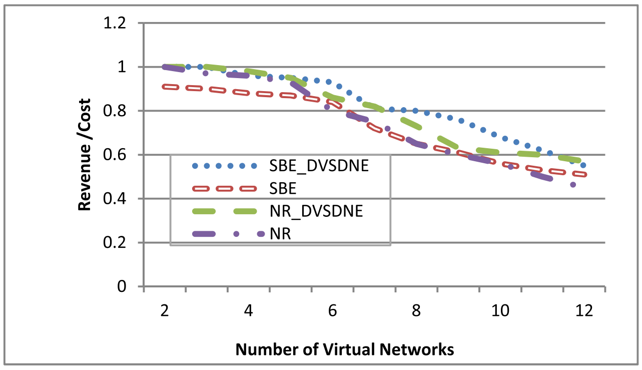


FIGURE 9. The R/C ratio for VNs embedding.

both the Acceptance ratio and Revenue/Cost. This is the consequence of partitioning substrate networks. The partitioning algorithm tries to categorize nodes that are highly interconnected.

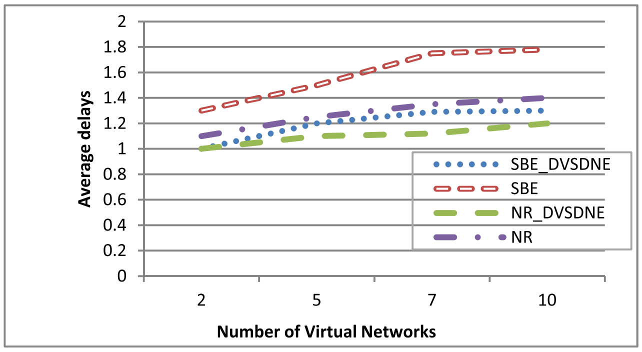


FIGURE 10. Average controller-to-switch delays.

b: AVERAGE LATENCY

Figure 10 shows DVSDNE's average latency is comparable to the average latency of a basic centralized algorithm. As the size of the virtual network gets larger, the average latency increases. That is because virtual nodes map at a far distance from the controller. Since every partition uses a dedicated FlowVisor, average latency improves when the algorithms run inside DVSDNE framework.

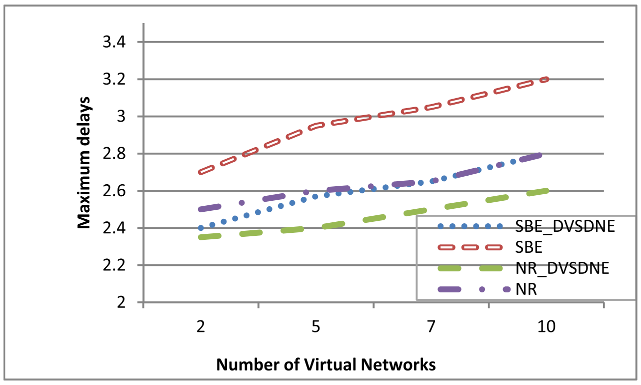


FIGURE 11. Maximum controller-to-switch delays.

c: MAXIMUM LATENCY

Figure 11 shows DVSDNE's Maximum latency is comparable to the Maximum latency of a basic centralized algorithm. Similar to average latency, as the size of the virtual network, gets larger, the Maximum latency increases. That is because virtual nodes map at a far distance from the controller. Since every partition uses a dedicated FlowVisor, the Maximum latency improves when the algorithms run inside DVSDNE framework.

d: MESSAGE OVERHEAD

Figure 12 shows the numbers of messages are transferred between agents during the mapping process within DVSDNE framework. As the figure shows, when the number of networks increases, the messages are sent increases. That is because bottom partitions cannot embed the latest VNRs and those VNRs are sent to upper partitions for embedding. Clearly, centralized VNE algorithms do not transfer any messages.

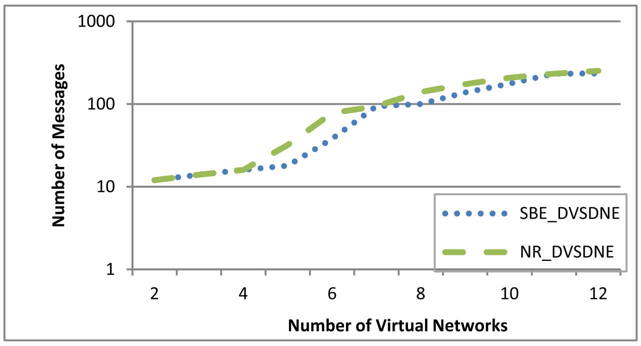


FIGURE 12. Message Overhead for large scale network.

2) SCALABILITY IN DVSDNE

To evaluate the scalability of DVSDNE, these experiments are performed for large scale networks. For evaluation, along with Revenue/Cost, Average latency to the controller, and Maximum latency to the controller, the runtime of algorithms is measured. It is shown how dramatically execution time increases when the substrate network nodes increase. In this scenario, a virtual network with 16 nodes and 33 links is successively embedded 10 times.

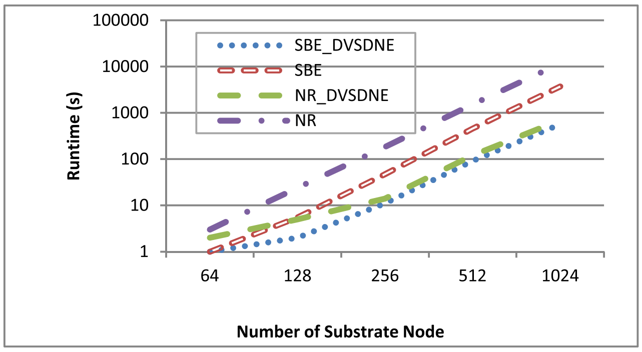


FIGURE 13. Execution time for large scale network.

a: RUNTIME

Figure 13 shows execution time when they run outside and inside DVSDNE. As Figure 13 shows, DVSDNE outperforms centralized algorithms for execution time. Since partitions used by DVSDNE are smaller than the original substrate network, it results in significant improvement.

b: REVENUE/COST RATIO

Figure 14 shows the R/C ratio for large scale networks. Similar to Figure 9, the results of centralized algorithms run inside DVSDNE are better than original centralized algorithms. As mentioned before, it is caused by pre-partitioning used by DVSDNE.

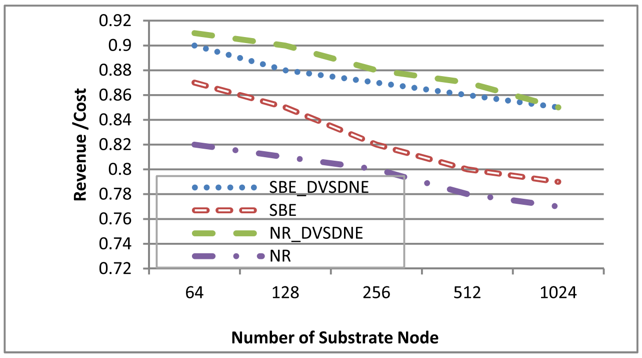


FIGURE 14. The R/C ratio for large scale network.

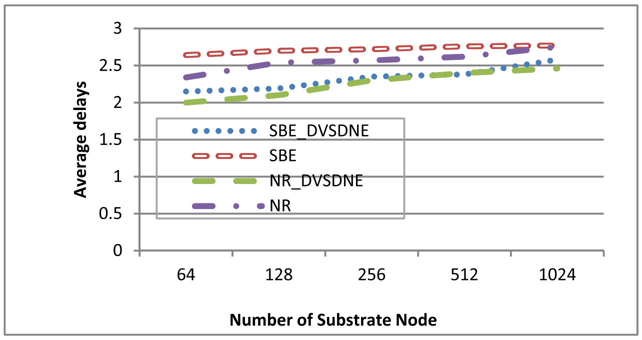


FIGURE 15. Average controller-to-switch delays for large scale network.

c: AVERAGE LATENCY

Figure 15 shows the average latency for large scale networks. Similar to Figure 10, DVSDNE outperforms centralized algorithms results concerning average latency. As mentioned before, it is caused by using a FlowVisor at each partition.

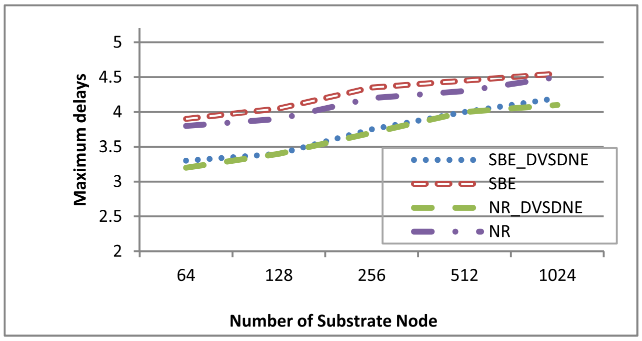


FIGURE 16. Maximum controller-to-switch delays for large scale network.

d: MAXIMUM LATENCY

Figure 16 shows the Maximum latency for large scale networks. Similar to Figure 11, DVSDNE outperforms centralized algorithms results concerning Maximum latency. As mentioned before, it is caused by using a FlowVisor at each partition.

3) EVALUATION OF DVSDNE'S PARAMETER γ

This subsection evaluates the impact of parameter γ on the performance metrics. As we stated before, the load balancer agent needs to find an appropriate embedder agent to forward the virtual network request to that agent. For each virtual network request, an appropriate embedder agent is found by using Eq. (20). This equation finds an appropriate embedder agent based on estimating the number of resources required to embed a VNR. Parameter γ in Eq. (20) tunes the estimation. If parameter γ is chosen large, DVSDNE selects an embedder agent with higher resources than what VNRs require. Besides, if parameter γ is chosen small, DVSDNE selects an embedder agent with fewer resources than what VNRs require.

Embedder agents with more resources are located at higher levels in a hierarchical structure and have more children than embedder agents with fewer resources that are located at lower levels. As Algorithm 1 depicts, embedder agents with more children should exchange more messages to update their resources assignment. Therefore, embedder agents with more resources should exchange more messages than embedder agents with fewer resources.

Also, embedder agents at higher levels in hierarchical structure manage more nodes and links than embedder agents at lower levels in the hierarchical structure. Since the virtual network embedding problem is NP-hard, the execution time increases as the number of nodes and links increases. Therefore, embedding algorithms run by embedder agents at higher levels take more time than embedder agents at a lower level in a hierarchical structure to embed VNRs.

To clarify the impact of parameter γ on our algorithm, 4 virtual networks with 5 nodes each were chosen and a substrate network with 32 nodes was generated. The parameter γ is set to 1, 10, 100, 200 respectively, and the performance metrics are calculated for those values. These parameters are chosen so that DVSDNE selects different embedder agents at different levels. For this experiment, when parameter γ is set to 1, the unsuccessful embedding attempt occurs and DVSDNE wants the parent of the current embedder agent to embed VNRs. The parent embedder agent can embed VNRs successfully. When parameter γ is set to 10, 100, 200 respectively, the unsuccessful embedding attempt does not occur, however, DVSDNE prefers the embedder agents which have more resources to run the embedding algorithm. As a result, the number of messages exchanged and the execution time increase.

a: MESSAGE OVERHEAD

Figure 17 shows the message overhead. By increasing parameter γ, at first message overhead decreases, and then it increases. This is because for lower γ values, embedder agents at lower levels are selected and for higher γ values, embedder agents at higher levels are selected.

Since an embedder agent at a lower level does not have sufficient resources to embed VNRs, an unsuccessful embedding attempt occurs. Therefore, the embedder agent uses messages by which want the load balancer to forward VNRs to its parent. Therefore, message overhead increases. As parameter γ increases, an embedder agent that has sufficient resources

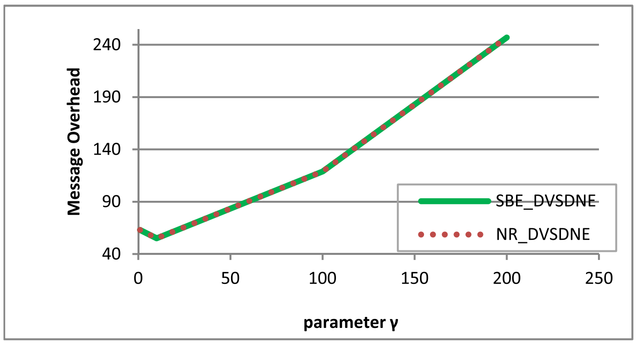


FIGURE 17. Message overhead due to change of parameter γ.

is selected. Therefore, the number of a message decreases, since the VNRs do not need to be forwarded to other embedder agents. By increasing parameter γ, an embedder agent that has more resources and is located a higher level is selected. In this situation, the embedder agent can embed the VNRs but a lot of messages are exchanged among the embedder agent and its children to update their resource assignments. Therefore, the message overhead increases.

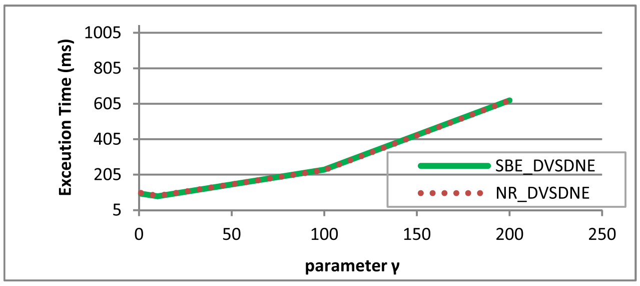


FIGURE 18. Execution time due to change of parameter γ.

b: EXECUTION TIME

Figure 18 shows the execution time. Similar to message overhead, by increasing parameter γ, at first, execution time decreases, and then it increases. This is since DVSDNE chooses embedder agents that manage a larger substrate network. When embedder agents with lower substrate resources are chosen, those embedder agents could not embed the VNRs and they want load balancer agent to forward VNRs to parent embedder agents, leading to increasing execution time. In this situation, execution time consists of time spent on unsuccessful embedding attempts and successful embedding attempts. As parameter γ increases, an embedder agent that has sufficient resources is selected. In this case, execution time is equal to the time spent on successful embedding attempt. So, the execution time is lower than when there exist unsuccessful embedding attempts. If the parameter γ continues to increase, embedder agents that manage a larger part of the substrate network are chosen. In this case, there do not exist unsuccessful embedding attempts. However, as shown in part B.2, the execution time of embedding algorithms increases in large substrate networks.

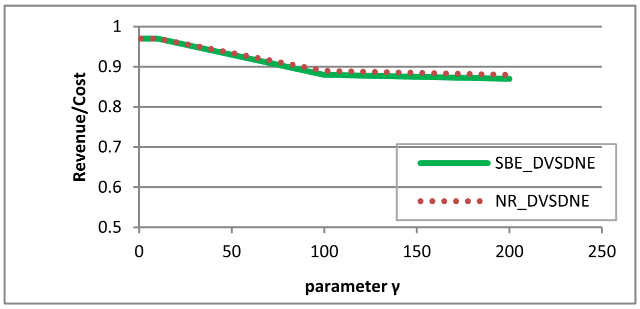


FIGURE 19. R/C ratio due to the change of parameter γ.

c: REVENUE/COST RATIO

Figure 19 shows R/C ratio. As the γ parameter increases, the R/C ratio remains constant at first and then decreases. When an unsuccessful embedding attempt occurs, the parent embedder agent of the embedder agent is selected to run the embedding algorithm. The parent of the selected embedder agent when parameter γ is set to 1 is the same as the selected embedder agent when the γ parameter is set to 10. Therefore, VNRs are mapped with the same agents when the γ parameter is set to 1 or 10. As a result, the R/C ratio does not change. If the parameter γ continues to increase, embedder agents that manage a larger part of the substrate network are chosen. As shown in part B.2, the R/C ratio of embedding algorithms decreases in large substrate networks.

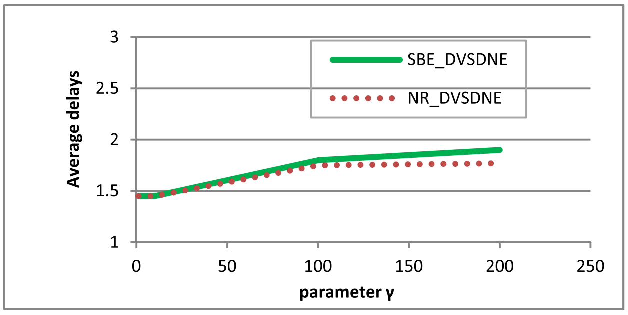
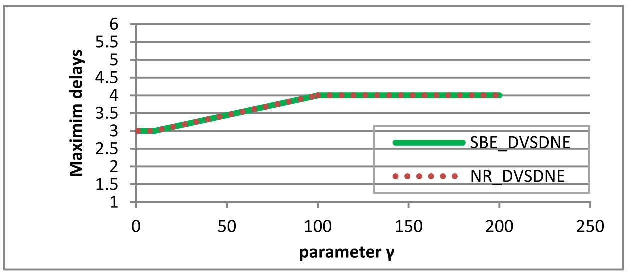


FIGURE 20. Average controller-to-switch delays due to change of parameter γ.



parameter γ.

FIGURE 21. Maximum controller-to-switch delays due to change of parameter γ.

d: AVERAGE LATENCY AND MAXIMUM LATENCY

Figure 20 and Figure 21 show average latency and maximum latency respectively. As the γ parameter increases, the average latency remains constant at first and then increases. When parameter γ is set to 1 or 10 the same agents run embedding algorithms. As a result, average latency and maximum latency does not change. If the parameter γ continues to increase, embedder agents that manage larger part of the substrate network are chosen. As shown in part B.2, the average latency and maximum latency increases in large substrate networks.

C. ANALYZING OF THE COMPLEXITY OF THE PROPOSED FRAMEWORK

In this section, we analyze that the worst-case time complexity of running VNE algorithms in our framework is not worse than the original algorithms.

To achieve that, we consider our previous VNE algorithm [1] named NR as the baseline and then show that O(DVSDNE) = O(NR). The complexity of algorithm NR is $O(V^3)$, where V denotes the number of nodes in the substrate graph. In DVSDNE method, worst-case scenario occurs when embedding starts at a leaf embedder agent and end at root embedder agent. In this case, the algorithm NR is executed by the number of levels in a hierarchical structure. For example, in Figure 6, there are 3 levels, and each of agents 4,5,6, and 7 is a leaf embedder agent and agent 1 is the root embedder agent.

Also, we consider DVSDNE uses 2-way partitioning algorithm, which halves the number of nodes at each level. Therefore, the number of subgraphs at level k is 2^k and the number of nodes of each subgraph at level k is $V/2^k$. Therefore, in the worst-case scenario, the time complexity of DVSDNE is calculated as follows:

$$O\,(\text{DVSDNE}) = O\left(NR, \frac{V}{2^0}\right) + O\left(NR, \frac{V}{2^1}\right) + O\left(NR, \frac{V}{2^2}\right) + \ldots + O\left(NR, \frac{V}{2^{n-1}}\right) \quad (21)$$

where $O\,(NR, K)$ means the complexity of algorithm NR with K nodes and n determines the number of levels in hierarchical structure. As we mentioned earlier, $O\,(NR, K) = O(K^3)$. Therefore, the Eq. (21) can be written as follows:

$$\begin{aligned} O\,(\text{DVSDNE}) &= O\left(V^3\right) + \frac{1}{2^3}*O\left(V^3\right) + \frac{1}{2^{2*3}}*O\left(V^3\right) + \ldots + \frac{1}{2^{(n-1)*3}}O\left(V^3\right) \\ &= O\left(V^3\right)\left(1 + \frac{1}{8} + \frac{1}{8^2} + \ldots + \frac{1}{8^{(n-1)}}\right) \\ &= \frac{1-\left(\frac{1}{8}\right)^n}{1-\frac{1}{8}} O\left(V^3\right) = O\left(V^3\right) \end{aligned} \quad (22)$$

This would also hold for any other VNE algorithm to be used in place of the NR algorithm. In other words, the worst-case time complexity of running algorithm NR in DVSDNE for a sufficiently large network is the same as the complexity of the underlying VNE algorithm.

Since the Initialization step of DVSDNE executes only once, we just calculate the complexity of the embedding step. In other words, we can ignore the complexity of the initializing step which uses the Metis tool to partition the substrate graph into a smaller partition because it does not need to be repeated during run-time. However, the complexity of partitioning a graph using the Metis tool is O(V+E+klogk) where V is the number of nodes, E the number of edges, and k the number of partitions. For algorithm NR, it is obvious that with considering the complexity of both the Initialization step and embedding step of DVSDNE, the worst-case complexity of running algorithm NR in DVSDNE is not worse than running algorithm NR outside of DVSDNE, because O(V+E+klogk) is smaller than $O(V^3)$.

VII. CONCLUSION AND FUTURE WORK

In this work, we propose a framework named DVSDNE for the virtual network embedding problem for the SDN environment based on multi-agent systems. In the first step of our method, the substrate network is partitioned into parts and a FlowVisor is assigned to each part. Then, an agent is assigned to each part to do VNE. Since we designed agents to work together for mapping virtual networks to substrate network in a distributed manner, we designed a protocol. This communication protocol is designed and implemented to work with agents together. Through simulations, two cost-optimizing VNE algorithms are designed for SDN run within DVSDNE.

We evaluated the generality and scalability of our methods. We presented that DVSDNE outperforms centralized algorithms in terms of execution time, average latency to the controller, and maximum latency to the controller. It is the result of partitioning the substrate network into smaller parts and assigns a FlowVisor to each part. Also, results show that the revenue-to-cost ratio and the acceptance ratio when the algorithms run inside the DVSDNE are comparable to when the algorithms run outside the DVSDNE.

For future work, we are planning to concentrate on virtual network embedding in a dynamic environment which is an important subject. In this environment, virtual network migrations and reassignment are the main steps to map virtual networks efficiently. Next, the number of partitions, the size of each partition, and the placement of FlowVisors need to be considered. These factors directly affect the cost-efficiency and average latency.

REFERENCES

[1] A. A. Nasiri and F. Derakhshan, "Assignment of virtual networks to substrate network for software defined networks," *Int. J. Cloud Appl. Comput.*, vol. 8, no. 4, pp. 29–48, Oct. 2018.

[2] W. Miao, G. Min, Y. Wu, H. Huang, Z. Zhao, H. Wang, and C. Luo, "Stochastic performance analysis of network function virtualization in future Internet," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 613–626, Mar. 2019.

[3] D.-L. Nguyen, H. Byun, N. Kim, and C.-K. Kim, "Toward efficient dynamic virtual network embedding strategy for cloud networks," *Int. J. Distrib. Sensor Netw.*, vol. 14, no. 3, Mar. 2018, Art. no. 155014771876478.

[4] H. Cao, S. Wu, Y. Hu, Y. Liu, and L. Yang, "A survey of embedding algorithm for virtual network embedding," *China Commun.*, vol. 16, no. 12, pp. 1–33, Dec. 2019.

[5] M. Rost and S. Schmid, "NP-completeness and inapproximability of the virtual network embedding problem and its variants," 2018, *arXiv:1801.03162*. [Online]. Available: http://arxiv.org/abs/1801.03162

[6] N. M. M. K. Chowdhury, M. R. Rahman, and R. Boutaba, "Virtual network embedding with coordinated node and link mapping," in *Proc. IEEE 28th Conf. Comput. Commun. (INFOCOM)*, Apr. 2009, pp. 783–791.

[7] X. Zheng, J. Tian, X. Xiao, X. Cui, and X. Yu, "A heuristic survivable virtual network mapping algorithm," *Soft Comput.*, vol. 23, no. 5, pp. 1453–1463, Mar. 2019.

[8] P. Zhang, H. Yao, M. Li, and Y. Liu, "Virtual network embedding based on modified genetic algorithm," *Peer Peer Netw. Appl.*, vol. 12, no. 2, pp. 481–492, Mar. 2019.

[9] S.-Q. Gong, J. Chen, Q.-Y. Kang, Q.-W. Meng, Q.-C. Zhu, and S.-Y. Zhao, "An efficient and coordinated mapping algorithm in virtualized SDN networks," *Frontiers Inf. Technol. Electron. Eng.*, vol. 17, no. 7, pp. 701–716, Jul. 2016.

[10] I. Houidi, W. Louati, and D. Zeghlache, "A distributed and autonomic virtual network mapping framework," in *Proc. 4th Int. Conf. Autonomic Auto. Syst. (ICAS)*, Mar. 2008, pp. 241–247.

[11] S. Dong, K. Abbas, and R. Jain, "A survey on distributed denial of service (DDoS) attacks in SDN and cloud computing environments," *IEEE Access*, vol. 7, pp. 80813–80828, 2019.

[12] T. Das, V. Sridharan, and M. Gurusamy, "A survey on controller placement in SDN," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 1, pp. 472–503, 1st Quart., 2020.

[13] A. Mondal, S. Misra, and I. Maity, "AMOPE: Performance analysis of OpenFlow systems in software-defined networks," *IEEE Syst. J.*, vol. 14, no. 1, pp. 124–131, Mar. 2020.

[14] Y.-N. Hu, W.-D. Wang, X.-Y. Gong, X.-R. Que, and S.-D. Cheng, "On the placement of controllers in software-defined networks," *J. China Univ. Posts Telecommun.*, vol. 19, pp. 92–171, Oct. 2012.

[15] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using OpenFlow: A survey," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 493–512, 1st Quart., 2014.

[16] A. Jalili, H. Nazari, S. Namvarasl, and M. Keshtgari, "A comprehensive analysis on control plane deployment in SDN: In-band versus out-of-band solutions," in *Proc. IEEE 4th Int. Conf. Knowledge-Based Eng. Innov. (KBEI)*, Dec. 2017, pp. 1025–1031.

[17] S. Abdallah, I. H. Elhajj, A. Chehab, and A. Kayssi, "A network management framework for SDN," in *Proc. 9th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Feb. 2018, pp. 1–4.

[18] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou, "DDoS attack protection in the era of cloud computing and software-defined networking," *Comput. Netw.*, vol. 81, pp. 308–319, Apr. 2015.

[19] M. Demirci and M. Ammar, "Design and analysis of techniques for mapping virtual networks to software-defined network substrates," *Comput. Commun.*, vol. 45, pp. 1–10, Jun. 2014.

[20] M. Wooldridge, *An Introduction to Multiagent Systems*. Hoboken, NJ, USA: Wiley, 2009.

[21] G. Weiss, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1999.

[22] J. Xie and C.-C. Liu, "Multi-agent systems and their applications," *J. Int. Council Elect. Eng.*, vol. 7, no. 1, pp. 188–197, 2017.

[23] L. S. Melo, R. F. Sampaio, R. P. S. Leão, G. C. Barroso, and J. R. Bezerra, "Python-based multi-agent platform for application on power grids," *Int. Trans. Electr. Energy Syst.*, vol. 29, no. 6, Jun. 2019, Art. no. e12012.

[24] H. Cao, H. Hu, Z. Qu, and L. Yang, "Heuristic solutions of virtual network embedding: A survey," *China Commun.*, vol. 15, no. 3, pp. 186–219, Mar. 2018.

[25] T. Wang and M. Hamdi, "Presto: Towards efficient online virtual network embedding in virtualized cloud data centers," *Comput. Netw.*, vol. 106, pp. 196–208, Sep. 2016.

[26] C. K. Dehury and P. K. Sahoo, "DYVINE: Fitness-based dynamic virtual network embedding in cloud computing," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1029–1045, May 2019.

[27] A. Xiao, Y. Wang, L. Meng, X. Qiu, and W. Li, "Topology-aware virtual network embedding to survive multiple node failures," in *Proc. IEEE Global Commun. Conf.*, Dec. 2014, pp. 1823–1828.

[28] A. Hmaity, F. Musumeci, and M. Tornatore, "Survivable virtual network mapping to provide content connectivity against double-link failures," in *Proc. 12th Int. Conf. Design Reliable Commun. Netw. (DRCN)*, Mar. 2016, pp. 160–166.

[29] A. A. Santos, A. Rizk, and F. Steinke, "Flexible redundancy generation for virtual network embedding with an application to smart grids," in *Proc. 11th ACM Int. Conf. Future Energy Syst.*, Jun. 2020, pp. 97–105.

[30] M. P. Gilesh, S. D. M. Kumar, and L. Jacob, "Resource availability-aware adaptive provisioning of virtual data center networks," *Int. J. Netw. Manage.*, vol. 29, no. 2, p. e2066, Mar. 2019.

[31] C. Zhao and B. Parhami, "Virtual network embedding through graph eigenspace alignment," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 2, pp. 632–646, Jun. 2019.

[32] L. Shengquan, W. Chunming, Z. Min, and J. Ming, "An efficient virtual network embedding algorithm with delay constraints," in *Proc. 16th Int. Symp. Wireless Pers. Multimedia Commun. (WPMC)*, 2013, pp. 1–6.

[33] A. Song, W.-N. Chen, T. Gu, H. Yuan, S. Kwong, and J. Zhang, "Distributed virtual network embedding system with historical archives and set-based particle swarm optimization," *IEEE Trans. Syst., Man, Cybern. Syst.*, early access, Jan. 3, 2019, doi: 10.1109/TSMC.2018.2884523.

[34] M. T. Beck, A. Fischer, J. F. Botero, C. Linnhoff-Popien, and H. de Meer, "Distributed and scalable embedding of virtual networks," *J. Netw. Comput. Appl.*, vol. 56, pp. 124–136, Oct. 2015.

[35] A. A. Nasiri and F. Derakhshan, "An agent based approach for assignment of virtual networks to substrate network for software defined networking," in *Proc. IEEE Int. Conf. Smart Energy Grid Eng. (SEGE)*, Aug. 2018, pp. 308–312.

[36] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 473–478, Sep. 2012.

[37] T. E. Kalayci and R. Battiti, "A reactive self-tuning scheme for multilevel graph partitioning," *Appl. Math. Comput.*, vol. 318, pp. 227–244, Feb. 2018.

[38] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, Jan. 1998.

[39] A. Hagberg, P. Swart, and D. S. Chult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos Nat. Lab. (LANL), Los Alamos, NM, USA, Tech. Rep. LA-UR-08-05495; LA-UR-08-5495, 2008.

ALI AKBAR NASIRI received the B.S. degree in computer engineering (hardware) from Shiraz University, Shiraz, Iran, in 2008, the M.S. degree in computer engineering (artificial intelligence) from the Iran University of Science and Technology (IUST), Tehran, Iran, in 2011. He is currently pursuing the Ph.D. degree in computer engineering (artificial intelligence) with the Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran. His major research interests include multi-agent systems and its applications, software-defined networking, and network security.

FARNAZ DERAKHSHAN received the Ph.D. degree in artificial intelligence from the University of Liverpool, U.K. She is currently an Assistant Professor and the Director of the Multi-Agent Systems Laboratory, Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran. Her main research interests include multi-agent systems and its applications, normative multi-agent systems, multi-agent learning, the Internet of Things, and swarm intelligence.

SHAHRAM SHAH HEYDARI (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in electronic engineering from the Sharif University of Technology, Iran, the M.A.Sc. degree from Concordia University, Montreal, QC, Canada, and the Ph.D. degree from the University of Ottawa, Ottawa, ON, Canada. He is currently an Associate Professor with the Faculty of Business and Information Technology, University of Ontario Institute of Technology (Ontario Tech), Oshawa, ON, Canada. Prior to joining Ontario Tech, in 2007, he was a System Engineer and a member of Scientific Staff with Nortel Networks, where he worked on element management in ultra-high-speed IP/MPLS routers, performance modeling of automatically switched optical networks (ASON), and proprietary voice-over-IP transport control protocols. His main research interests include network design and planning, software-defined networking, applications of Artificial Intelligence (AI) in network management, and network Quality of Experience (QoE).