Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration

Minlan Yu†, Yung Yi‡, Jennifer Rexford†, Mung Chiang‡
† Computer Science, Princeton University, Email: {minlanyu,jrex}@cs.princeton.edu
‡ Electrical Engineering, Princeton University, Email: {yyi,chiangm}@princeton.edu

ABSTRACT

Network virtualization is a powerful way to run multiple architectures or experiments simultaneously on a shared infrastructure. However, making efficient use of the underlying resources requires effective techniques for *virtual network embedding*—mapping each virtual network to specific nodes and links in the substrate network. Since the general embedding problem is computationally intractable, past research has focused on two main approaches: (i) significantly restricting the problem space to allow efficient solutions or (ii) proposing heuristic algorithms that do not use the substrate resources efficiently. In this paper, we advocate a different approach: rethinking the design of the substrate network to enable simpler embedding algorithms and more efficient use of resources, without restricting the problem space. In particular, we simplify virtual link embedding by: i) allowing the substrate network to split a virtual link over *multiple* substrate paths and ii) employing *path migration* to periodically re-optimize the utilization of the substrate network. We also explore node-mapping algorithms that are *customized* to common classes of virtual-network topologies. Our simulation experiments show that path splitting, path migration, and customized embedding algorithms enable a substrate network to satisfy a much larger mix of virtual networks.

1. INTRODUCTION

Network virtualization has emerged as a powerful way to allow multiple network architectures, each customized to a particular application or user community, to run on a common substrate. For example, virtualization can enable multiple researchers to evaluate new network protocols simultaneously on a shared experimental facility [3, 7, 2, 10]. In addition, network virtualization could serve as the foundation of a future Internet that allows multiple service providers to offer customized end-to-end services over a common physical infrastructure [14, 27]. For example, Voice over IP (VoIP) could run on a virtual network that provides predictable performance (by allocating dedicated resources and employing routing protocols that ensure fast recovery from equipment failures), whereas online banking could run on a virtual network that provides security guarantees (through self-certifying addresses and secure routing protocols).

Making efficient use of the substrate resources requires effective techniques for *virtual network (VN) embedding*—mapping a new virtual network, with constraints on the virtual nodes and links, on to specific physical nodes and links in the substrate network. However, the VN embedding problem is extremely challenging, for four main practical reasons:

Node and link constraints. Each VN request has resource constraints, such as processing resources on the nodes and bandwidth resources on the links, that the embedding must satisfy. For example, to run a controlled experiment, a researcher may need 1 GHz of CPU for each virtual node and 10 Mbps for each virtual link. In addition, the VN may impose additional constraints on node location or link propagation delay. For example, a commercial gaming service may need virtual nodes in several major cities, as well as virtual links with propagation delays less than 50 msec. The combination of node and link constraints make the embedding problem computationally difficult to solve.

Admission control. Since the substrate resources are limited, some VN requests must be rejected or postponed to avoid violating the resource guarantees for existing virtual networks. That is, the substrate must reserve node and link resources, and perform admission control on new requests to ensure that sufficient resources are available. For example, a virtual network that requires 1 GHz of CPU for each virtual node may be rejected if no physical nodes have enough unallocated processing capacity. Once accepted, the virtual networks receive their guaranteed resources through scheduling techniques for sharing the node and link resources.

Online requests. The VN requests are not known in advance, and may arrive dynamically and stay in the network for an arbitrary period of time before departing. For example, a researcher may start a new experiment at any time, to run for some duration based on the needs of the experiment. Similarly, a service provider may deploy a new service at any time, and continue supporting the service indefinitely, possibly discontinuing the service when it is no longer profitable. To be practical, the embedding algorithm must handle VN requests as they arrive, rather than handling a large collection of requests at once. Online problems are typically much more difficult to solve, because the embedding algorithm has little (if any) visibility into the future request arrivals.

Diverse topologies. The virtual networks may have diverse topologies. For example, researchers may run experiments under a variety of topologies to explore how their protocol performs in different settings. Also, a service provider may tailor the virtual-network topology to the application, such as a hub-and-spoke to provide customers with access to a centralized server, or a tree to distribute streaming video to a group of receivers. Although virtual networks may have a wide variety of structures, certain kinds of topologies—such as a hub-and-spoke or a tree—may be especially common in practice. Handling arbitrary topologies, while efficiently supporting the most common topologies, introduces an additional challenge for the embedding algorithm.

These four properties make the VN embedding problem very difficult. In fact, the problem is computationally intractable, even if some of these four properties are ignored. Due to the combination of node and link constraints, the VN embedding problem is NP-hard, even in the offline case. (For example, assigning virtual nodes to the substrate network without violating bandwidth constraints can be reduced to the *multiway separator problem*, which is NP-hard [6].) Even if the locations of the virtual nodes are pre-determined, embedding the virtual links with bandwidth constraints is still NP-hard, as discussed in more detail in Section 3. The online problem is even more difficult to solve. Traditional techniques for solving online problems (e.g., dynamic programming) are impractical here because the properties of incoming VN requests are generally unpredictable and the search space is prohibitively huge when the substrate network is large.

Previous research has addressed these computational challenges by restricting the problem space in one or more dimensions to enable efficient heuristics [31, 13, 22, 25], at the expense of limiting the practical applicability of the solutions. For example, the papers either solve an offline variant of the problem [31, 22], consider only bandwidth constraints [13, 25], or do not perform admission control [31, 22, 13]. In this paper, we take a different approach—we reconsider the capabilities of the underlying substrate network, to *make the substrate network more supportive of the VN embedding problem*. This allows us to create simpler embedding algorithms that make more efficient use of the substrate resources, without compromising on the four challenges listed above. Since network virtualization is still in its infancy, we believe it is important to explore how to design the substrate to best satisfy its goals.

In particular, we investigate how to simplify the problem of virtual-link embedding by allowing the substrate network to map a virtual link to *multiple* substrate paths with a flexible path-splitting ratio. In addition, for efficient handling of online requests, we allow the substrate to periodically re-optimize the mapping of existing virtual links, either by selecting new underlying paths or adapting the splitting ratios for the existing paths. Flexible path splitting is realizable in practice without disrupting the basic properties of a link—such as predictable bandwidth, propagation delay, and in-order packet delivery, as discussed in more detail in Section 3.3. Flexible path splitting allows us to map virtual links to the substrate in polynomial time, while making much more efficient use of substrate bandwidth and increasing robustness to substrate failures. This feature allows us to satisfy the first three of the four challenges listed above. To address the fourth challenge, we introduce customized node-embedding algorithms for common topologies like a hub-and-spoke.

The remainder of this paper is organized as follows. In Section 2, we define the VN embedding problem and present a simple embedding algorithm that does not assume any special capabilities from the substrate. This algorithm is similar to the techniques proposed in previous research, extended to support admission control and online requests. The algorithm serves as a basis of comparison for our new algorithm for substrates that support path splitting and migration, as discussed in Section 3. Section 4 presents simulations that evaluate our algorithm and quantify the benefits of a more flexible substrate. In Section 5, we present our customized node-embedding algorithm for hub-and-spoke topologies, and promising initial simulation results. Section 6 compares our algorithms with related work, and Section 7 concludes the paper with a discussion of future research directions.

2. VIRTUAL NETWORK EMBEDDING

In this section, we first describe the general VN embedding problem. Then, we present an original solution to this problem without assuming the substrate can split a virtual link over multiple underlying paths.

2.1 Virtual Network Embedding Problem

Substrate network. We denote the substrate network by an undirected graph $G^s = (N^s, L^s, A^s_N, A^s_L)$, where N^s and L^s refer to the set of nodes and links, respectively. We use superscript to refer to substrate or virtual network, and use subscript to refer to nodes or links, unless otherwise specified. Substrate nodes and links are associated with their attributes, denoted by A^s_N and A^s_L, respectively. In this paper, we consider CPU capacity and location for node attributes, and bandwidth capacity for link attributes. We also denote by $\mathcal{P}^s$ the set of all loop-free paths in the substrate network.

Table 1: Notations of VN Embedding Problem

G^s	Substrate network
N^s	Nodes of substrate network
L^s	Links of substrate network
A^s_N	Node attribute of substrate network
A^s_L	Link attribute of substrate network
$\mathcal{P}^s$	Paths on substrate network
G^v	Virtual network
N^v	Nodes of virtual network
L^v	Links of virtual network
C^v_N	Node constraint of substrate network
C^v_L	Link constraint of substrate network
R_N	Resources allocated for virtual network nodes
R_L	Resource allocated for virtual network links

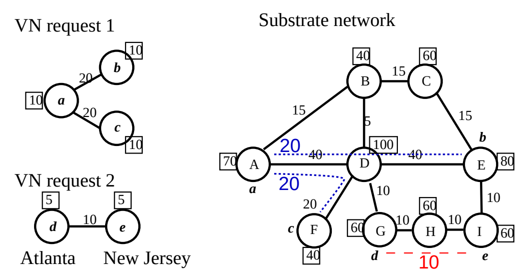

Figure 1: An example of VN embedding

The right side of Figure 1 shows a substrate network, where the numbers over the links represent available bandwidths and the rectangles are the CPU resources available at the nodes.

Virtual network request. We denote by an undirected graph $G^v = (N^v, L^v, C^v_N, C^v_L)$ a virtual network request. A VN request typically has link and node constraints that are specified in terms of attributes of the substrate network. We denote by C^v_L and C^v_N the set of link and node constraints, respectively. Figure 1 depicts two VN requests. For example, the VN request 1 requires the bandwidth 20 over the links (a, b) and (a, c), and the CPU resource 10 at all nodes, a, b, and c. The VN request 2 is: "connect two nodes $d, e \in N^v$ with constraints that node d should be in Atlanta (where substrate nodes D and G are located), and node e should be in New Jersey (where substrate nodes E and I are located), with ten units of bandwidth on the virtual link between them."

VN embedding. A virtual network embedding for a VN request is defined as a mapping $\mathcal{M}$ from G^v to a subset of G^s, such that the constraints in G^v are satisfied, i.e.,

$$\mathcal{M} : G^v \mapsto (N', \mathcal{P}', R_N, R_L),$$

where $N' \subset N^s$ and $\mathcal{P}' \subset \mathcal{P}^s$, and R_N and R_L are the node and link resources allocated for the VN requests. The VN network embedding can be naturally decomposed into node and link mapping as follows:

$$\begin{aligned} \text{Node Mapping:} \quad & \mathcal{M}^N : (N^v, C^v_N) \mapsto (N', R_N), \\ \text{Link Mapping:} \quad & \mathcal{M}^L : (L^v, C^v_L) \mapsto (\mathcal{P}', R_L). \end{aligned}$$

The right side of Figure 1 shows the VN embedding solutions for the two VN requests. For example, the nodes a, b, and

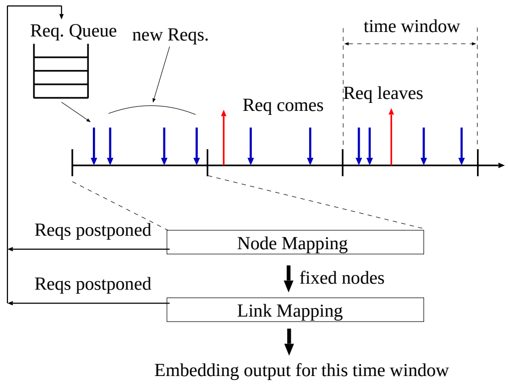

Figure 2: The general algorithm overview

c in VN request 1 are mapped to the substrate nodes A, E, and F, and the virtual links (a, b) and (a, c) are mapped to the substrate paths (A,D,E) and (A,D,F) with the CPU and bandwidth constraints all satisfied. A similar mapping occurs for VN request 2.

Objectives. Our main interest is to propose an efficient embedding algorithm for the online problem, where VN requests arrive and depart over time. From the substrate network provider's point of view, a natural objective of an online embedding algorithm would be to maximize the *revenue*. We introduce the notion of *revenue* that corresponds to the economic benefit of accepting VN requests. We denote by $R(G^v(t))$ the revenue of serving the VN request at time t. Then, our objective is to *maximize the long-term average revenue,* given by the following:

$$\lim_{T\to\infty} \frac{\sum_{t=0}^{T} R(G^v(t))}{T}. \tag{1}$$

The revenue can be defined in various ways according to economic models. In this paper, we focus on bandwidth and CPU as the main substrate network resources. Then, a natural choice of the revenue for a VN request would be to the weighted sum of revenues for bandwidth and CPU, each of which is proportional to the amount of the requested resources. Similar to the work in [31], we introduce a tunable weight α that allows the substrate provider to strike a balance between the relative costs of the two classes of resources.

Thus, for a VN request G^v, we define its revenue $R(G^v(t))$ at any particular time t that the virtual network G^v is running as:

$$R(G^v(t)) = \sum_{l^v\in L^v} \mathrm{bw}(l^v) \;+\; \alpha \sum_{n^v\in N^v} \mathrm{CPU}(n^v), \tag{2}$$

where $\mathrm{bw}(l^v)$ and $\mathrm{CPU}(n^v)$ are the bandwidth and CPU requirements for the virtual link l^v and the virtual node n^v, respectively. We note that the bandwidth revenue (i.e., the first term in Equation (2)) is not affected by the substrate paths that the virtual links are mapped to, in particular, physical distance or the number of hops of the mapped paths. This seems to be reasonable, since VN requests are interested in just satisfiability of their constraints in the substrate network and will not pay for longer distance.

To achieve the goal in Equation (1), it is crucial to embed incoming VN requests efficiently, such that the substrate resource is minimally occupied. This is because an inefficient embedding of a virtual network at time t may restrict the substrate's ability to accept future requests.

2.2 General VN Embedding Algorithm

In this subsection, we propose a simple embedding algorithm that does not exploit any special capabilities from the substrate network. The algorithm is motivated by the techniques proposed in related work (e.g., [31, 25]) with extensions to perform *admission control* and handle *online requests*. Figure 2 depicts our algorithm.

Algorithm 1 Greedy Node Mapping Algorithm

Step 1 Sort the requests according to their revenues.
Step 2 If no requests left, stop.
Step 3 Take one request with the largest revenue.
Step 4 Find the subset S of substrate nodes that satisfy restrictions and available CPU capacity (larger than that specified by the request.) If $S == \emptyset$, store this request in the queue, and GOTO **Step 2**.
Step 5 For each virtual node, find the substrate node in S with the "maximum available resources" H (defined in Equation (3)), and GOTO **Step 2**.

Our algorithm collects a group of incoming requests during a *time window* and then tries to allocate substrate resources to satisfy the constraints required by the requests. Some requests may be deferred due to lack of bandwidth or CPU resources in the substrate network, and returned to the *request queue*. The requests in the queue are dropped if they cannot be served within some *delay*, which, specified by a request, corresponds to the time that a request is willing to wait. The requests in the request queue are processed again in the subsequent time windows.

We process all VN requests arriving within the time window as well as in the request queue, in decreasing order of their revenues. We map virtual nodes onto the substrate for all the considered VN requests, and then map the virtual links for the requests that successfully finish the node mapping stage. An alternative way, which we do not choose, is to map the nodes and links of one request first, before mapping the other requests. In the general VN embedding algorithm, both methods produce similar mapping results, but our method is more efficient because of batch processing in the node/link mapping stage.

The optimal embedding algorithm is computationally intractable as discussed in Section 1. Our general VN embedding algorithm heuristically tries to achieve the goal in Equation (1) over each time window. Indeed, the algorithm contributes to instantaneous revenue maximization by giving higher priority to the requests with more revenue and accepting as many requests as possible in the node mapping. Additionally, the algorithm tends to make efficient utilization of the substrate bandwidth resources by mapping virtual links to shortest paths in the substrate network, leaving more resources for future requests.

2.2.1 *Node Mapping Algorithm*

We employ a "greedy" node mapping algorithm, since it is computationally too expensive to employ other strategies, such as iterative methods [22] and simulated annealing [13, 19]. The motivation of the greedy algorithm is to map the virtual nodes to the substrate nodes with the *maximum* substrate resources so as to minimize the use of the resources at the bottleneck nodes/links [31]. This is beneficial to future requests which require specific substrate nodes with scarce resources.

In our algorithm, we collect all outstanding requests, and then map all the virtual nodes in these requests to the substrate nodes. VN requests sometimes impose some *restrictions* on their nodes. The examples of node restrictions include geographic location and special functionality at the substrate node. These node restrictions are quite common in practice, e.g., servers near their customers in content-delivery service, programmable routers, and a node with Internet-2 network connectivity. Requests with restrictions reduce the search space for placing the virtual nodes (**Step 4**). For example, location-specific requests usually limit their virtual nodes to particular

Algorithm 2 Link Mapping Algorithm

Step 1 Sort the requests that successfully completed the node-mapping stage by their revenues.
Step 2 If no requests left, stop.
Step 3 Take one request with the largest revenue.
Step 4 For each virtual link of the request, we search the k-shortest paths for increasing k, and stop the search if we can find one with enough bandwidth capacity.
Step 5 If fail in **Step 3** for some virtual link, then defer this request, and store it in the request queue.
Step 6 GOTO **Step 2**.

geographic regions.

Then, we keep track of the available node/link resources of the substrate network. Note that for a substrate node $n^s \in N^s$, we do not use $\mathrm{CPU}(n^s)$ alone as the metric of available resource, because we not only want to make sure that there is enough CPU capacity available, but also consider bandwidth capacity to prepare for the subsequent link mapping stage. Therefore, we define the amount of available resources for a substrate node n^s by:

$$H(n^s) = \mathrm{CPU}(n^s) \sum_{l^s \in L(n^s)} \mathrm{bw}(l^s), \qquad (3)$$

where $L(n^s)$ is the set of all adjacent substrate links of n^s, $\mathrm{CPU}(n^s)$ is the remaining CPU resource of n^s, and $\mathrm{bw}(l^s)$ is the unoccupied bandwidth resource for the substrate link l^s. With this definition, for a virtual node, we find the substrate node with the maximum available resources (**Step 5**).

2.2.2 *Link Mapping Algorithm*

When the substrate nodes are selected for mapping, we map the virtual links to specific substrate links. Finding an optimal mapping from a virtual link to a single substrate path with fixed node mapping reduces to the Unsplittable Flow Problem (UFP), which is NP-hard [20, 21]. Therefore, we use the *k-shortest path algorithm* as an approximation approach in order to minimize bandwidth consumption by the virtual network.

We search the k-shortest paths for increasing values of k, until we find a path which has enough bandwidth to map the corresponding virtual link. Our k-shortest-path link-mapping algorithm can be solved in $O(N \log N + kN)$ time in a substrate network with N nodes [12]. Both for computational efficiency and efficient use of substrate resources, k should be kept small.

3. PATH SPLITTING AND MIGRATION

Restricting each virtual link to a single substrate path makes the link-embedding problem computationally intractable, and the resulting embeddings inefficient. In this section, we first argue that the substrate network should support flexible splitting of virtual links over multiple substrate paths, and present a new link-embedding algorithm that capitalizes on the flexibility. Next, we describe how to periodically re-optimize the mapping of existing virtual links to allow the substrate network to accept more new requests. Finally, we explain how substrate support for path splitting and migration can be implemented in practice.

3.1 Path Splitting

3.1.1 *Motivation for Flexible Path Splitting*

To motivate substrate support for path splitting, consider the example in Figure 3. Initially the substrate network runs a single virtual network with three virtual nodes and two virtual links that each require 20 units of bandwidth. The virtual nodes are mapped to physical nodes A, E, and F, and the two virtual links are mapped to the paths (A,D,E) and (A,D,F),

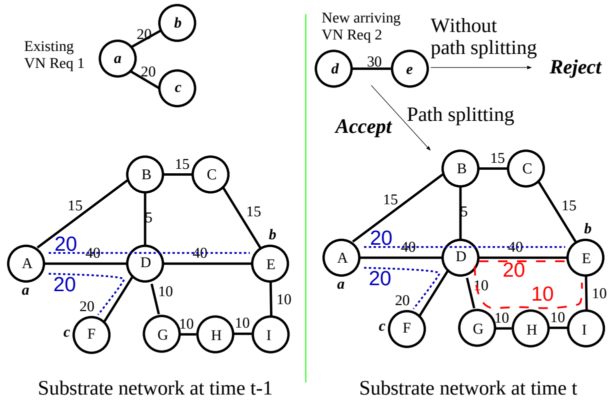

Figure 3: Illustration of the benefit of path splitting

as shown in the lower left part of the figure. Now, suppose a new VN request arrives with a single virtual link that requires 30 units of bandwidth. Unfortunately, no one path in the substrate network can accommodate the new request. However, the new VN could be mapped to nodes D and E, if the substrate could allocate 20 units of bandwidth on the path (D, E) and 10 on the path (D, G, H, I, E). That is, directing two-thirds of the traffic over the (D, E) path and one-third over the (D, G, H, I, E) path would allow the substrate to accept the second request.

Path splitting enables better resource utilization by harnessing the small pieces of available bandwidth, allowing the substrate to accept more VN requests. In addition, flexible path splitting makes the link-embedding problem computationally tractable. A virtual link l with some capacity constraint, say C_l, is mapped into multiple paths in the substrate network, such that the sum of reserved end-to-end bandwidth along the multiple paths is equal to C_l. The division of traffic over the substrate paths is specified as a *splitting ratio*, such as a ratio of 2:1 in the example in Figure 3. Under flexible splitting over multiple paths, the link-embedding problem can be reduced to the Multicommodity Flow Problem (MFP) [5], which can be solved in polynomial time.

The benefits of having multiple paths have been established in other contexts, such as load balancing and reliability. In some problem formulations, even having just *two* paths can significantly reduce the maximum load on a network, compared to solutions that limit the traffic flow to a single path [23, 18]. Having multiple paths also enables faster recovery from network failures. For example, if a link or node fails, the network can quickly switch the affected traffic to other paths simply by changing the splitting ratios. In contrast, in a single-path setting, a failure requires establishing a new end-to-end path, leading to a more severe service disruption. Due to the computational, performance, and reliability benefits, we believe flexible path splitting should be a key feature in future virtualized network infrastructures, and the rest of this paper will provide the algorithmic and simulation-based evidence to support this view.

3.1.2 *Link Mapping Algorithm with Path Splitting*

We describe the link mapping algorithm supporting path splitting to enable efficient solutions in Algorithm 3. In **Step 1**, We first construct linear constraints for the virtual links. For simplicity, consider a request with only one link l^v with the capacity constraint C, where two end nodes of l^v are denoted by n_1^v and n_2^v. We denote by $\mathcal{M}^N(n_1^v) = n_1^s$ and $\mathcal{M}^N(n_2^v) = n_2^s$ the substrate nodes chosen for n_1^v and n_2^v, respectively, by the node-mapping algorithm in Section 2.2. The pair of substrate nodes (n_1^s, n_2^s) is a commodity, and finding multiple substrate paths for l^v is equivalent to finding flows from the source n_1^s

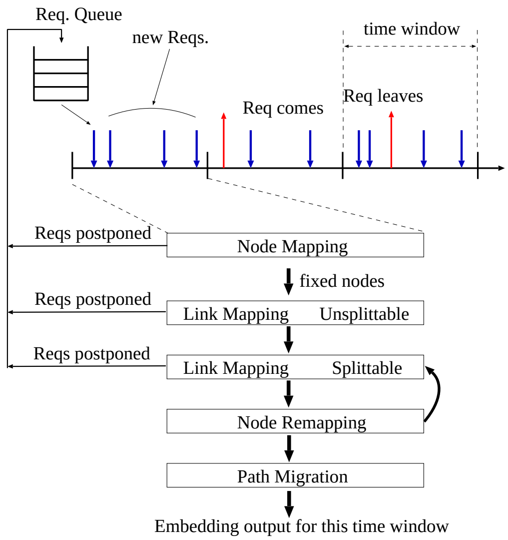

Figure 4: Algorithm for path splitting and migration

to the destination n_2^s in the substrate network with available bandwidth on the substrate links.

Thus, a group of, say r, virtual links generates a group of r commodities. The algorithm tries to find all the paths for r commodities based on the following linear constraints:

$$\forall l^s \in L^s, \quad \sum_{i=1}^{r} f(c_i, l^s) \leq \text{bw}(l^s) \tag{4}$$

where $f(c_i, l^s)$ is the bandwidth on the substrate link l^s that we allocate to commodity c_i, or its corresponding virtual link. After generating the r commodities and the linear constraints, we solve the resulting multicommodity flow problem (**Step 2**).

Even with flexible path splitting, the MFP problem may not have a feasible solution because one or more substrate links do not have enough available capacity. The algorithm revisits the node-mapping decisions for these virtual links (**Steps 4, 5, and 6**). The failure in the MFP computation implies that one of more substrate links violate the linear constraints in Equation (4). Fortunately, the MFP algorithm can easily output the substrate links that violate the constraint, as well as the extent of the violation. The node-remapping stage focuses its attention on the substrate link with the largest violation, i.e., the $l^s \in L^s$ with the highest value of $\sum_{i=1}^{r} f(c_i, l^s) - \text{bw}(l^s)$.

We randomly choose one virtual link that is originally mapped to the path including the bottleneck link, and map one end of this virtual link to another substrate node with maximum remaining resource H (defined in Equation (3)), in order to avoid occupying this bottleneck link. We try this node remapping for a pre-defined number of times T_{try} and make sure each time we choose a different bottleneck link. If the MFP is still infeasible after T_{try} trials, we defer the request that requires the most bandwidth on the bottleneck substrate link and return it to the request queue, and then try to solve the MFP with the remaining requests again. Larger values of T_{try} increase the computational overhead but improve the likelihood of finding a successful embedding.

In practice, some virtual networks may have strict requirements that preclude path splitting[1]. As such, we envision our algorithm would handle a mix of both kinds of VN requests. As illustrated in Figure 4, we first apply Algorithm 2 for requests that do not allow path splitting, before applying Algorithm 3

[1]The substrate provider may charge an extra fee to these virtual networks, since non-splittable virtual links are more difficult to support.

Algorithm 3 Link Mapping Algorithm for Requests with Path Splitting

MFP Computation:

Step 1 For all requests with splittability, construct linear constraint on the commodities for each substrate link.

Step 2 Solve MFP (Multicommodity Flow Problem).

Step 3 If feasible, stop.

Node Remapping:

Step 4 If infeasible, find the "bottleneck" substrate link.

Step 5 Randomly choose one virtual link that is originally mapped at the bottleneck link, pick one end of the virtual link and map it to another substrate node with maximum remaining resource H (defined in Equation (3)). Then GOTO **Step 2** with new linear constraints.

Step 6 If remapping of virtual nodes for T_{try} times does not produce a feasible solution, eliminate one of the VN requests having the "largest" impact on infeasibility. Then, construct the linear constrains only with the remaining requests, and GOTO **Step 2.**

---

for requests that allow path splitting. Unfortunately, node-remapping is difficult to perform for the unsplittable virtual links, since the embedding algorithm processes one virtual link at a time. For the collection of virtual links that fail to find a suitable path, we cannot easily identify the most congested substrate link in a computationally efficient manner. To maintain computational simplicity, we do not consider node-remapping for these requests. In the evaluation, we quantify the benefits of path splitting, with and without the node-remapping step.

3.2 Migration for Online Requests

To deal with the online nature of the VN embedding problem, we introduce the idea of path migration, i.e., changing the route or splitting ratio of a virtual link. This turns out to be another advantage offered by the availability of multipath in the substrate network.

3.2.1 *Motivation for Path Migration*

Since VN requests arrive and depart over time, the substrate network can easily drift into an inefficient configuration, where resources are increasingly fragmented, forcing the substrate to reject future requests or route new virtual links over more expensive (longer) paths. Theoretically, one could try to address these challenges with predictive models of future requests, coupled with mathematical techniques like dynamic programming. However, the arrival and departure of requests is unpredictable and the underlying search space is too large for dynamic programming to be practical. Instead, we argue the substrate network should be able to "rebalance" the mapping of virtual networks to make more efficient use of the substrate resources and to maximize the chance of accepting future requests. In particular, the ability to migrate virtual links to different substrate paths—while keeping the node mapping intact—can further improve the substrate's ability to accept future requests.

To motivate substrate support for path migration, consider the example in Figure 5. Initially the substrate networks runs a single virtual network with three nodes and two virtual links that each require 20 units of bandwidth. Now, suppose a new VN request arrives with a single virtual link that requires 30 units of bandwidth. Unfortunately, no pair of nodes in the substrate network can accommodate the new request, even if path splitting is permitted. For example, in the left side of Figure 5, nodes D and E have a path (D,E) with 20 units of bandwidth, a path (D,B,C,E) with 5 units of bandwidth, and a path (D, A, B, C, E) with 0 units of bandwidth—not enough to support a virtual link requiring 30 units of bandwidth. However, migrating some of the traffic for the first virtual network to a different path would enable the substrate to accept the new request. In

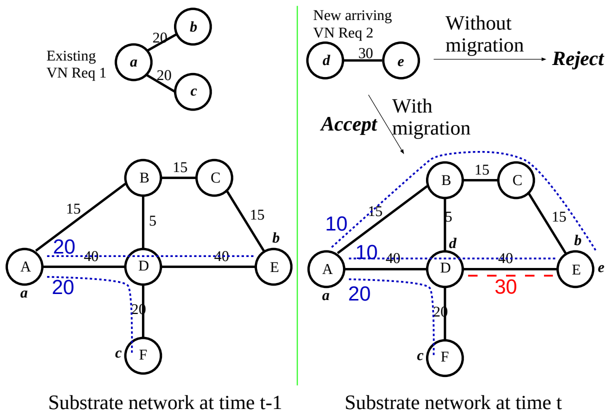

Figure 5: Illustration of the benefit of migration

particular, the substrate could carry half of the traffic for virtual link (a,b) on a new path (A, B, C, E) to free up additional capacity on the substrate link (D, E). Then, the second virtual network can have link (d,e) mapped to substrate path (D,E).

3.2.2 Migration Algorithm

In the migration algorithm described in Algorithm 4, we fix the node mapping of the virtual networks already running on the substrate. We perform path migration by rerunning the link-mapping algorithm with requests that allow path splitting (Algorithm 3). Path migration is performed by either changing the splitting ratios for the existing paths or selecting new underlying paths.

If only adjusting the splitting ratios is allowable rather than setting up any new paths, we have to make sure that the flows coming from a commodity only traverse the substrate paths originally taken in the link mapping stage. Thus, we add the following linear constraints to the constraints in Equation (4) (**Step 2**):

$$f(c_i, l^s) = 0, \quad \forall l^s \in L^s, \forall c_i, l^s \notin P^s(c_i), \tag{5}$$

where $P^s(c_i)$ is the set of original substrate paths the virtual link (or corresponding commodity c_i) was mapped to. Then we solve the MFP problem again with the new constraints of both Equations (4) and (5). If we are allowed to select new underlying paths, we rerun the link-mapping algorithm with only the constraints in Equation (4).

Path migration allows us to (periodically) treat the online embedding problem as an offline problem, to capitalize on the efficiency gains that are possible when handling a large collection of requests together. As such, we expect the benefits of path migration to be highest when the *time window* (for grouping requests) is small, and less significant as the window grows larger.

In practice, migrating paths introduces overhead to establish new paths, switch the traffic onto the new paths, and tear down the old paths. As such, the benefits of path migration should be weighed against the overheads. To illustrate this, we expect that VN requests would be quite diverse in their durations, which corresponds to their running time in the substrate network, ranging from a few months to several hours. As an example, a content distribution network like Akamai [1] may run indefinitely, whereas an impromptu conference or video game may last for a few hours. The algorithm should not migrate short-lived virtual networks that are likely to exit the system soon after the migration completes. Thus, our algorithm only considers the requests whose durations are larger than some threshold T_{dur} (**Step 1**). Fortunately, migrating long-running virtual networks should offer ample benefits in practice, since many short-lived virtual networks will come and go while they run. Virtual-network requests would indicate their likely duration, or we can infer that a virtual network that has run for a long time is likely to continue running for a long time, analogous to previous research on migration in the context of job scheduling [17].

Algorithm 4 Path Migration Algorithm

For all the served requests,
Step 1 Select the request set S whose durations are larger than a threshold T_{dur}.
Step 2 If only changing splitting ratio is allowed, add linear constraints (Equation (5)), so that each virtual link is forced to be mapped to the paths it originally take in the link mapping step. If setting up new path is also allowed, skip this step.
Step 3 Rerun the link mapping algorithm with path splitting, and migrate the related paths.

3.3 Implementation Issues

Path splitting can be easily implemented in the substrate network without significant overhead. When the virtual node directs a packet over the virtual link, the substrate sends the packet over one of the paths based on the target splitting ratio. The virtual network is largely oblivious to the splitting of the traffic, as long as care is taken to prevent out-of-order packet delivery. The substrate can employ a variety of techniques to prevent performance disruptions:

1) *Hash-based splitting.* Out-of-order delivery is a primary concern for packets in the same *flow*—a group of packets between the same end hosts or part of the same transport-level connection. Hash-based splitting prevents out-of-order delivery by directing all packets from the same flow to the same path. The substrate router first divides the hash space into weighted partitions that each correspond to one substrate path. Then, we apply hashing to the packets based on their header bits and forward the packets to the corresponding substrate path. This hash-based scheme is efficient and, in fact, is widely used in IP networks to split traffic evenly over equal-cost multipath [15, 8]. The substrate router can use *consistent hashing* to minimize the fraction of flows that must change paths when the splitting ratio changes [9].
2) *Adding artificial delay.* Another solution is to equalize the delays along the multiple paths. This is possible because all substrate nodes and links belong to a single party—the substrate provider. The substrate provider can add small artificial delay to overcome variable propagation delay, e.g., by using Dummynet [26]. Moreover, we do not need to be concerned about congestion-related delay, since unlike in a conventional best-effort network (e.g., the Internet), these virtual networks are allocated bandwidth resources in advance.
3) *Tagging the packets.* Since the substrate network is under the control of a single party, each packet can be tagged with a sequence number or timestamp. Then the remote end-point of the virtual link can reorder the packets based on the tags before delivering the packets to the receiving virtual node.

Path migration relates closely to path splitting, and is easily implemented either by selecting new underlying paths or adapting the splitting ratios for the existing paths. In addition, path migration will not cause significant service disruptions for two reasons: (i) we need just a slight change of flow splitting ratio for the already-existing paths; (ii) we can create the new path in advance before moving the traffic to avoid service disruption. Therefore, path migration should not unduly influence the performance experienced by the virtual network.

In our current work, we have focused on path migration while keeping the node-mapping intact, to minimize the disruption experienced by the virtual networks. However, in ongoing work we plan to explore node migration to provide even greater flexibility in handling new VN requests. We believe node migration should be feasible for several reasons. First, long-running services usually have their own maintenance windows, where they drain traffic off a server to upgrade the software. These maintenance windows can be used for node migration. Second, with ample warning and prior planning, we can minimize the negative effects of node migration on an ongoing service. Node migration can be done quite quickly in practice, e.g., within a few seconds [29, 28], and the virtual node can continue running in the old location until the migration completes.

4. PERFORMANCE EVALUATION

In this section, we first describe the performance evaluation environment, and then present our main evaluation results. Our evaluation focuses primarily on quantifying the benefits of substrate support for flexible path splitting and migration in the VN embedding problem.

4.1 Evaluation Environment

We implemented a VN embedding simulator (publicly available at [4]) to evaluate our embedding algorithm and the advantages of flexible support for path splitting and migration.

The actual characteristics of substrate and virtual networks are not well understood since network virtualization is still an open field. Therefore, we use synthetic networks to study the trends and quantify the benefits of path splitting and migration.

Substrate network. We use the GT-ITM tool [30] to generate a substrate network topology. The GT-ITM tool has been popularly used in research that requires practical network topology generation. The substrate network is configured to have 100 nodes, a scale that corresponds to a medium-sized ISP, and at every experiment, 500 links are generated on average. The CPU resources at nodes and the link bandwidths at links follow a uniform distribution from 0 to 100 units.

Virtual network request. In one VN request, the number of VN nodes is randomly determined by a uniform distribution between 2 and 10, following a similar approach as previous work [13].

Each pair of virtual nodes are randomly connected with probability 0.5. This means that for a n-node virtual network, we have $n(n-1)/4$ links on average. The arrivals of VN requests are modeled by a Poisson process with mean five requests per time window. The duration of the requests follows an exponential distribution with 10 time windows on average. We run all of our simulations for 500 time windows, which corresponds to about 2500 requests on average in one instance of simulation.

The parameters and their symbols that we vary in all our simulations are summarized in the following table:

E[CPU]	average CPU requirement on a virtual node
E[BW]	average bandwidth requirement on a virtual link
RPS(%)	percentage of the requests allowing path splitting
DELAY	time a request is willing to wait (see Section 2.2)
α	weight constant in revenue function (Equation (2))
T_{try}	number of rounds in node remapping

Comparison method. Comparing our algorithm with previous work is difficult because these earlier embedding algorithms do not start with the same problem formulation. They do not handle one or more of the first three challenges in Section 1 (i.e., combined node and link constraints, admission control, or online requests). Instead, we use the algorithm in Section 2.2, which embodies many of the key ideas from prior work, as a baseline for comparison.

4.2 Evaluation Results

Our evaluation results quantify the benefits of path splitting and migration in various environments. We present our simulation results by summarizing the key observations.

(1) More requests allowing path splitting leads to larger revenues, which is further improved by path migration. Figures 6 shows the long-term average revenue with increasing percentages of the requests permitting path splitting for different average link bandwidth requirements. In these experiments, we remove the influence of CPU in the constraint and the revenue (i.e., E[CPU]=0 and α=0). Each request which cannot be served immediately will wait for at most 3 time windows in the queue (DELAY=3) and node remapping in Algorithm 3 is tried just once, i.e., $T_{try} = 1$.

In Figure 6, the performance of the general algorithm in Section 2 does not depend on RPS(%), since the general algorithm maps each virtual link into *a single path* in the substrate network. However, with more requests allowing path splitting, the substrate network resources are efficiently utilized at current time window, which enables the system to accept more requests, leading to an increase in the average revenue. When all the requests allow path splitting, our algorithm achieves about 120% revenue increase over the general algorithm. Even with half of the requests permitting path splitting, we still gain about 65% revenue increase.

Figure 6 also shows that path migration further increases the revenue. For example, when RPS(%)=100, our algorithm with path migration achieves additional 15% revenue increase over the algorithm only with path splitting. This implies that path splitting is a dominant factor in the revenue increase, and path migration further builds on path splitting to adapt to the online VN embedding problem more flexibly. More benefits are expected to be obtained by node migration, at the expense of more service disruption.

When the bandwidth requirement is low (E[BW]=25) and substrate resources are ample, we can accept all the requests for both algorithms. Naturally, the revenue remains the same, whether the requests allow path splitting or not. We will show later in Figure 11 that in this case, our algorithm reduces cost more than the general algorithm. Note that in Figure 6, the revenues differ when E = 25 and E =50 due to its dependence on the amount of required (average) bandwidth in the requests.

(2) Path splitting still increases revenue when CPU requirements are considered. Figure 9 shows the long-term revenues with both CPU and bandwidth requirements, where the average CPU requirement is set to be 25 and other parameters are the same as those in Figure 6.

We observe a similar increase in revenue from path splitting. We achieve more than 100% revenue increase over the general algorithm, when RPS(%)=100; and about 50% when RPS(%)=50. However, the benefits from migration are less dramatic. This is due to the fact that we only employ path migration, which does not offer any benefits when the node CPU resource is the bottleneck. Note that revenue increase with the CPU requirement is less than that without the CPU requirement. This is anticipated, because when CPU requirements tends to reduce the number of accepted requests.

To further evaluate effects of CPU resource constraints, in Figure 10, we have tested different weight constants α in the revenue definition (Equation (2)) while keeping all the other parameters the same. The benefits of path splitting over the general algorithm decrease as α increases, since path splitting and migration only improve the bandwidth resource utilization in the link mapping stage. For example, when CPU and bandwidth are almost equally evaluated in the revenue function, we achieve 90% of revenue increase over the general algorithm; when CPU resource becomes the main factor (α=1), our al-

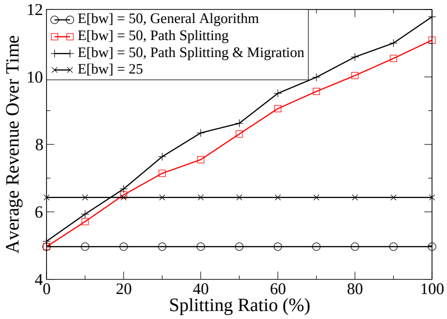

Figure 6: Revenue changes with RPS(%) **(**E[CPU]**:0,** DELAY**:3,** α**:0,** T_{try}**:1)**

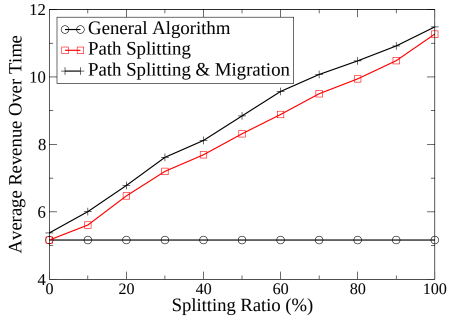

Figure 7: Revenue changes with RPS(%) **(**E[BW]**:50,** E[CPU]**:0,** DELAY**:6,**

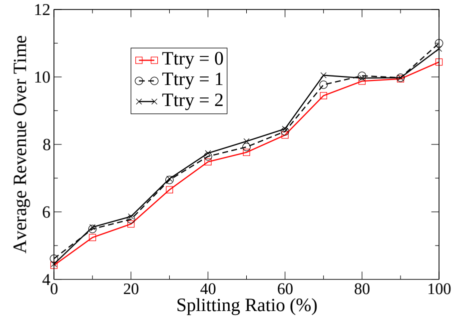

Figure 8: Influence of Node Remapping (Revenue changes with RPS(%)

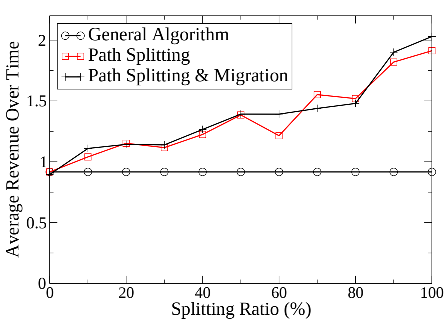

Figure 9: Effect of CPU requirement (E[BW]**:50,** E[CPU]**:25,** DELAY**:3,** α**:0,** T_{try}**:1)**

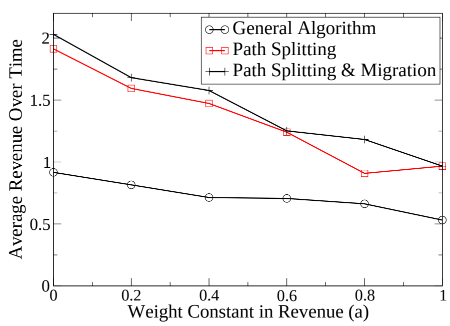

Figure 10: Effect of α **(**RPS(%)**:100,** E[BW]**:50,** E[CPU]**:25,** DELAY**:3,** T_{try}**:1)**

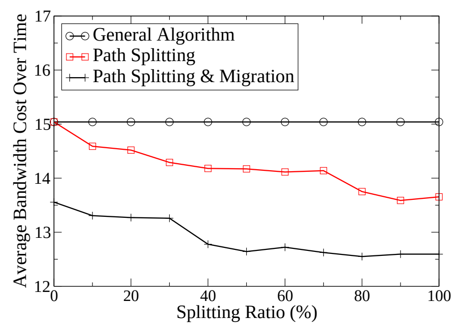

Figure 11: Effect on cost (E[BW]**:25,** E[CPU]**:0,** DELAY**:3,** α**:0,** T_{try}**:1)**

gorithm with path splitting still achieves around 80% more revenue than the general algorithm.

(3) Node remapping contributes modestly to revenue increase. Revenue is not significantly influenced by the delay we choose. All previous experiments were made with $T_{try} = 1$, i.e., we ran one round of node remapping in Algorithm 3. In Figure 8, we show the result of the path splitting algorithm without node remapping (i.e., $T_{try} = 0$), where the revenue only decreases by 4%. This implies that the revenue increase shown in earlier simulations mainly comes from path splitting itself. With increasing values of T_{try}, we could achieve more substantial increases in revenue at the expense of computation time, because we must rerun the link mapping stage for T_{try} times more than the path splitting solution without node remapping.

Figure 7 shows that our benefits of path splitting and migration are not influenced by delay we choose (DELAY=3), since the result for DELAY=6 is similar to Figure 6 where DELAY=3. This is because the substrate resources are almost fully used with requests coming and departing over time, so that the deferred requests cannot be accepted even if it waits for more time.

(4) Without admission control, path splitting and migration reduces cost. In case when the substrate network resources are sufficient, but the number of incoming VN requests per time-window is small, we can probably service all the requests, irrespective of using of path splitting or not, i.e., the long-term revenue achieved will be the same for the general algorithm and the algorithm with path splitting. However, the algorithm allowing path splittability saves substrate network resources.

First, we discuss notions of cost to quantify efficiency in resource utilization. The bandwidth cost for a VN request should be defined to reflect the entire amount of bandwidth used to map the request to the physical substrate network. For a single virtual link in the request, it would be natural to use its required bandwidth multiplied by the length of the substrate path that the virtual link is mapped to. With path splitting, we count the bandwidth allocated on each path of the virtual link and sum them up. Thus, we define the cost of virtual network G^v by:

$$C_{\mathrm{bw}}(G^v) = \sum_{l^v \in L^v} \sum_{p \in P^s(l^v)} \mathrm{bw}(p, l^v), \tag{6}$$

where $P^s(l^v)$ is the path(s) the virtual link l^v is mapped on, $\mathrm{bw}(p, l^v)$ is the amount of bandwidth allocated to that virtual link. Similarly, we define the CPU cost of virtual network G^v by:

$$C_{\mathrm{CPU}}(G^v) = \sum_{n^v \in N^v} \mathrm{CPU}(n^v), \tag{7}$$

where $\mathrm{CPU}(n^v)$ is the amount of CPU virtual node n^v requires.

In Figure 11, we have simulated the case where the resource requirement of requests is low (E[BW]=25, E[CPU]=0), where other parameters are the same as before, i.e., (DELAY=3, α=1, T_{try}=1).

We observe that with the increase in percentage of requests permitting path splitting, we reduce the bandwidth cost C_{bw} over the general algorithm by making more efficient use of the network. When RPS(%)=100, we reduce 10% cost than the general algorithm. Path migration further reduces the bandwidth cost by 7%. The CPU cost C_{CPU} remains the same ($C_{\mathrm{CPU}} = 3.1$) with the increase of RPS(%), since all the requests are accepted, whether they allow path splitting or not.

Algorithm 5 Customized Node Mapping Algorithm for Requests with Hub-and-spoke

Steps 1, 2, and 3: Same as in Greedy Node Mapping (Algorithm 1).
Step 4 If the request has hub-and-spoke topology,
  4.1 For each *hub* node, find the substrate node with the maximum available resource in S.
  4.2 For each *spoke* node, find the shortest path between a substrate node in S and the substrate node mapped to the corresponding hub node.
  else, apply **Step 4** in General Node Mapping.
Step 5 Same as in Greedy Node Mapping.

5. CUSTOMIZED NODE MAPPING

Although virtual networks may have arbitrary topologies, we expect some classes of topologies to be relatively common, since they meet the needs of the key applications in network virtualization. For example, a hub-and-spoke topology is commonly used to connect multiple sites to a centralized server, e.g., gaming and CDN (Content Distribution Network), and a tree topology is commonly used to distribute content efficiently to a large collection of receivers, e.g. multicast distribution of IPTV.

The popularity of a small set of topological structures can be leveraged for better solutions to the VN embedding problem. In our ongoing work, we present node-mapping techniques that are customized to specific topologies, starting with the simple hub-and-spoke topology.

As an example, we propose a customized node mapping (which is extended from the Greedy Node Mapping in Algorithm 1) with hub-and-spoke topologies, as summarized in Algorithm 5. The customized node mapping algorithm differs from the greedy node mapping in that we choose the substrate nodes differently for hub and spokes nodes. The maximum available resource is allocated only for *the hub nodes* (**Step 4.1**), and the spoke nodes are mapped into the substrate nodes that have the *shortest path* to the substrate node hosting the virtual hub node (**Step 4.2**). This is motivated by the fact that the hub node handles much more traffic than the individual spokes. In **Step 4.2**, we also achieve significant cost reduction, since cost is generally proportional to the distance (i.e., number of hops), whereas the greedy algorithm allocates large substrate resource to "unimportant nodes" (i.e., the spokes). The wasted resources keep the greedy node mapping algorithm from leaving enou

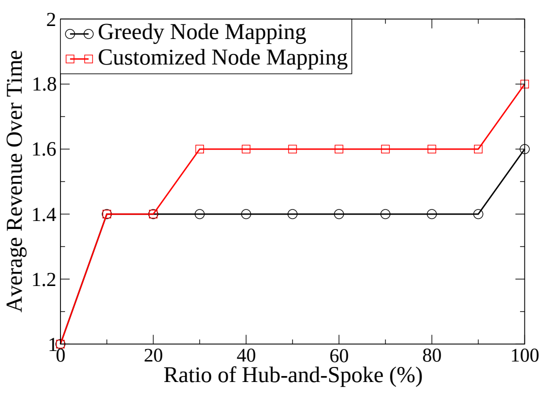

Figure 12: Effect of customized node mapping for hup-and-spoke topology (RPS(%):0, E[BW]:10, E[CPU]:10, α:0, T_{try}:1)

To evaluate the customized node mapping algorithm, we compare the greedy node mapping and our customized node mapping with the requests of hub-and-spoke topologies in Figure 12. Our preliminary experiments evaluate a small substrate network handling virtual networks with small resource requirement of requests correspondingly (E[BW]=10, E[CPU]=10). We also keep RPS(%) as zero to evaluate the benefits of customized node mapping in the absence of path splitting. We observe that our customized algorithm performs better than the greedy algorithm when the percentage of hub-and-spoke topologies among all the requests increases. By taking advantage of the topology information, our customized algorithm allocates the hub-and-spoke request more efficiently than the greedy algorithm. Thus, our algorithm can allow more requests and achieve a higher average revenue over time. We are currently exploring and evaluating other algorithms that are customized for different common topologies, like trees.

6. RELATED WORK

Previous research has explored how to embed Virtual Private Networks (VPNs) in a shared provider topology [11, 16]. However, several of the properties in Section 1 make the VN embedding problem different, and more difficult, than the VPN design problem. In particular, the VN embedding problem must deal with both node and link constraints for arbitrary topologies, whereas VPNs usually have a standard topology, such as full mesh and hub-and-spoke [24]. Moreover, the resource constraints in a VPN are typically just bandwidth requirements, specified by a traffic matrix (i.e., the traffic volume for each pair of nodes), rather than node constraints (e.g., processing resources).

Related work on VN embedding addresses the hardness of the problem by *relaxing one or more of the key properties* of the problem (see Table 2). These key properties include (i) whether requests are processed online or not, (ii) whether the requests have link constraints, node constraints, or both, (iii) whether admission control is performed to reject requests when resources are insufficient, and (iv) what virtual topologies are supported.

Several of the previous studies focus on the *offline* problem, where all VN requests are known in advance. Zhu and Ammar [31] assume that *the substrate network resources are unlimited, and aim at achieving load balancing in the substrate network, obviating the need for admission control.* The VN-embedding problem for the requests with general topology is solved by subdividing the requests into multiple star topologies to allocate more substrate resource to the center of each decomposed star topology. Lu and Turner [22] also consider an offline problem for only a *single* virtual network with a backbone-star topology, where their goal is to minimize the cost. They assume that only bandwidth constraints are imposed, and the substrate network resources are unlimited with no admission control needed.

In regard to the online problem, Fan and Ammar [13] consider dynamic topology reconfiguration policies for virtual networks with dynamic communication requirements, but no consideration of the node constraints such as CPU. They also assume that substrate network resources are unlimited to accept all requests (i.e., no admission control) and try to find a strategy to minimize cost. Zhu and Ammar [31] also solve the online problem by recalculating the whole embedding solution periodically, which is computationally complex. The Emulab testbed [25] considers the online embedding problem with the bandwidth constraint. The node constraint in Emulab is provided as the exclusive use of nodes, i.e., different virtual networks cannot share a substrate node. Admission control is not explicitly addressed in [25], but it can be inferred that an admission control that rejects request if the bandwidth/node resources are insufficient.

As we have described so far, past research on VN embedding algorithms tries to come up with an efficient solution by looking at only a subset of the key challenges outlined in Section 1.

Table 2: Comparison of VN Embedding Algorithms

	on/offline	CPU/BW	admission control	topology	objective
NetFinder [31]	offline	both	no	general	load balancing
[22]	offline	both	no	backbone-star	cost
[13]	online	BW	no	traffic matrix	cost
Emulab [25]	online	BW	primitive	general	cost
This paper	online	both	yes	general	revenue (& cost)

This clearly limits the applicability of the algorithms. In this work, we deal with the most general cases by considering all aspects of the four properties in the embedding algorithm design, tackling the online VN embedding problem with both node and link constraints. In particular, we modify the algorithms in previous work [31, 25] to allow admission control and further improve them by *making the substrate network more flexible and supportive to this problem, in conjunction with customization of the algorithms for different types of requests.*. We also improve the link mapping algorithm through substrate support for path splitting and migration. We are currently exploring the method to improve node mapping algorithm by customizing it for diverse topologies. In terms of the objective function in the VN embedding problem, our work is practical, because our goal is to maximize revenue, which is natural for online problem with admission control.

7. CONCLUSION

A key problem in the current study of network virtualization, the VN embedding problem, has various constraints that make it computationally intractable. In this paper, rather than significantly restrict the problem space to make the problem tractable, we rethink the VN embedding problem by proposing a more flexible substrate network to better support virtual network embedding. This flexibility includes path splitting and migration. From both the theoretical and engineering perspective, we show that allowing substrate path splitting and migration would help us to attain better resource utilization. Through our publicly available simulator, we demonstrate the benefits of these approaches in making the embedding problem computationally easier, and the resulting embeddings more efficient. We also propose that customized node mapping algorithms would improve the VN embedding performance and show preliminary results. We are currently exploring more customized node mapping algorithms dealing with common classes of virtual-network topologies.

8. REFERENCES

[1] Akamai content distribution network. `http://www.akamai.com/`.

[2] GENI. `http://www.geni.net/`.

[3] Planetlab. `https://www.planet-lab.org/`.

[4] Virtual Network Embedding Simulator. `http://www.cs.princeton.edu/~minlanyu/embed.tar.gz`.

[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[6] D. G. Andersen. Theoretical approaches to node assignment. Unpublished Manuscript, 2002.

[7] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer Magazine*, 38(4):34–41, 2005.

[8] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proc. Internet Measurement Conference*, 2006.

[9] I. Avramopoulos, D. Syrivelis, J. Rexford, and S. Lalis. Secure availability monitoring using stealth probes. Technical Report TR-769-06, Princeton University, October 2006.

[10] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI veritas: Realistic and controlled network experimentation. In *Proc. ACM SIGCOMM*, September 2006.

[11] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merwe. Resource management with hoses: Point-to-cloud services for virtual private networks. *IEEE/ACM Trans. Networking*, 10(5):679–692, 2002.

[12] D. Eppstein. Finding the k shortest paths. In *Proc. IEEE Symposium on Foundations of Computer Science*, 1994.

[13] J. Fan and M. Ammar. Dynamic topology configuration in service overlay networks: A study of reconfiguration policies. In *Proc. IEEE INFOCOM*, 2006.

[14] N. Feamster, L. Gao, and J. Rexford. How to lease the Internet in your spare time. *ACM Computer Communication Review*, 37(1):61–64, 2007.

[15] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford. NetScope: traffic engineering for IP networks. *IEEE Network Magazine*, 14(2):11–19, March 2000.

[16] A. Gupta, J. Kleinberg, A. Kumar, R. Rastogi, and B. Yener. Provisioning a virtual private network: A network design problem for multicommodity flow. In *Proc. ACM Symposium on Theory of Computing*, 2001.

[17] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[18] P. Key, L. Massoulie, and D. Towsley. Path selection and multipath congestion control. In *Proc. IEEE INFOCOM*, 2007.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[20] J. Kleinberg. *Approximation algorithms for disjoint paths problems*. PhD thesis, MIT, 1996.

[21] S. G. Kolliopoulos and C. Stein. Improved approximation algorithms for unsplittable flow problems. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 426–435, 1997.

[22] J. Lu and J. Turner. Efficient mapping of virtual networks onto a shared substrate. Technical Report WUCSE-2006-35, Washington University, 2006.

[23] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[24] S. Raghunath, K. K. Ramakrishnan, S. Kalyanaraman, and C. Chase. Measurement based characterization and provisioning of IP VPNs. In *Proc. Internet Measurement Conference*, 2004.

[25] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM Computer Communication Review*, 33(2):65–81, 2003.

[26] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[27] J. S. Turner and D. E. Taylor. Diversifying the Internet. In *Proc. IEEE GLOBECOM*, 2005.

[28] Y. Wang, J. van der Merwe, and J. Rexford. VROOM: Virtual routers on the move. In *Proc. ACM SIGCOMM HotNets Workshop*, 2007.

[29] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proc. Networked Systems Design and Implementation*, 2007.

[30] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. IEEE INFOCOM*, 1996.

[31] Y. Zhu and M. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *Proc. IEEE INFOCOM*, 2006.