A self-stabilizing framework for dynamic bandwidth allocation in virtual networks

Houda Jmila, Kaouther Drira, Djamal Zeghlache

A self-stabilizing framework for dynamic bandwidth allocation in virtual networks

Article · April 2016

CITATIONS
0

READS
106

3 authors, including:

Houda Jmila
Institut Mines-Télécom
7 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Djamal Zeghlache
Institut Mines-Télécom
207 PUBLICATIONS **1,508** CITATIONS
SEE PROFILE

A self-stabilizing framework for dynamic bandwidth allocation in virtual networks

Houda Jmila*, Kaouther Drira*, Djamal Zeghlache*
* Institut Mines Telecom, Telecom SudParis and UMR5157 of CNRS, Evry, France
{houda.jmila, kaouther.drira, djamal.zeghlache}@telecom-sudparis.eu

***Abstract*—This paper addresses dynamic bandwidth allocation for virtual network (VN) resources to respond to increasing or decreasing applications requirements in cloud environments. A *distributed* and *local-view* framework, composed of a controller and three algorithms running in substrate nodes, is proposed to deal with *all types* of bandwidth demand fluctuations in embedded virtual networks. The framework is based on the *Self-Stabilization* concept to drive the system back to a "stable state" when new bandwidth demands drift the system away into an "unstable state". Performance evaluation results demonstrate the effectiveness of our proposal in handling bandwidth demand fluctuations in convergence speed and cost.**

***Index Terms*—Virtual network embedding, dynamic bandwidth allocation, elasticity, self stabilization, local view, cloud, distributed and parallel algorithms;**

I. Introduction

Network virtualization allows multiple virtual networks to coexist on a shared physical infrastructure. A critical issue in network virtualization is virtual network resource provisioning. This deals with the optimal allocation of the VN onto the substrate network (SN), and the management of the allocated resources throughout the life-cycle of the VN.

The current state of the art has extensively addressed the *static* provisioning where a *fixed* amount of resources is allocated to the VN for its *entire* lifetime. But cloud services and environment require the *elastic* allocation of resources according to increasing/decreasing applications demands. Moreover, traditional resource provisioning solutions employ *centralized and total-view based* techniques that face serious limitations when applied to large and highly dynamic cloud environments. In centralized solutions, maintaining dynamically up-to-date description of real-time substrate resources states induces high latency during analysis and enforcement of decisions because of the management traffic overhead with the central entity. This leads to low responsiveness to VNs evolution and affects in fine the cloud user satisfaction.

In this paper, we propose a *distributed and parallel* framework to manage bandwidth demand fluctuation *without needing* a *global view* of available resources to avoid costly updating and monitoring. The proposal is based on the *self-stabilization* [1] concept that guaranties the convergence of the system to a stable state in a finite time regardless of its initial state. In fact, we view bandwidth requirements fluctuations as events that drift the system (the substrate network) into an unstable state, and our self stabilizing algorithms, executed locally on each substrate node, as actions that take it back to a stable state where all new bandwidth demands are satisfied.

Section II of this paper presents the background and related work. Section III defines the dynamic bandwidth allocation problem in the context of network virtualization while section IV briefly introduces the concept of Self-Stabilization. Section V describes our model and proposal. The effectiveness of our algorithms is assessed in section VI where performance evaluation is reported.

978-1-5090-0223-8/16/$31.00

II. Related work

To the best of our knowledge, only [2], [3], [4] and [5] address the VN bandwidth demand fluctuation in a distributed manner. In [2], authors propose a decentralized multi-agent resource management system based on Reinforcement Learning. Each agent (substrate node/link) dynamically adjusts the allocated resources to avoid under utilization of the substrate network and to satisfy new demands. The agents choose an action among a pre-defined set of actions (Decrease/increase allocated resources by 50/37/25/12.5 percent or maintain the currently allocated resources). The choice is made according to the results of a decentralized Q-learning based algorithm that iteratively approximates the state action values. The main drawbacks of this proposal are i) a limited set of actions that affects the algorithm efficiency and action granularity, and ii) the need for a high number of learning episodes to determine an optimal decision policy.

Authors of [3] improve the efficiency of the previous system by conceiving an autonomous system based on Artificial Neural Networks (ANN). They represent each substrate resource as an ANN whose input is the network resource usage status and the output is an allocation action. An error function is used to evaluate the desirability of ANN outputs and hence perform online training. Note that even if neural networks are important for their learning and generalization capabilities, they do not have a clearly defined way on how the number of layers as well as the number of neurons in each layer are determined.

Work in [4] is an extension of the previous model using an adaptive hybrid system composed of Neural Networks, Fuzzy systems and Reinforcement Learning. The system dynamically adjusts both the substrate network usage and structure by adding or removing substrate resources. The SN is modeled as a distributed system of autonomous agents. For each agent, an initial knowledge base is defined using supervised learning subsequently improved with Reinforcement Learning.

Unfortunately, authors still limit the action granularity of the algorithm.

Authors of [5] use a distributed controller system to adapt bandwidth allocations to the VN dynamic workloads. The system predicts the optimal request for each VN and adjusts the offered bandwidth accordingly.

All the previous work scales up and down the allocated resources but *fixes* the VN topology. This prior art does not address bandwidth demand fluctuation *with topological changes:* i.e. when adding/ deleting virtual links to/from the VN. The authors do not take advantage of virtual resources migration (re-allocation) either. We devise a self-stabilizing framework to deal with *all types* of bandwidth fluctuations without the need for a learning phase, unlike [2]–[4]. To the best of our knowledge, this is the first *distributed and local-view based* framework that manages *all types of bandwidth demand fluctuations*; i.e. increasing/diminishing bandwidth needs of embedded virtual links, new bandwidth requests to connect nodes or the need to free link bandwidth no longer needed between nodes.

III. Problem formulation

Virtual network embedding (VNE) can be decomposed into two stages: the initial VNE and a dynamic resource management phase. The first stage provides an efficient mapping of virtual resources (nodes and links) onto the physical hosts (nodes and substrate paths), while the second manages the resource demand fluctuation of already embedded VNs.

A. Initial VNE

Let us represent the substrate network by a weighted undirected graph $\hat{G} = (\hat{N}, \hat{L})$. $\hat{N}$ (resp. $\hat{L}$) is the set of *substrate nodes (resp. links)*. Each substrate node $n \in \hat{N}$ is characterized by an amount of available resources $\hat{n}_{av}$ (typically CPU and memory) and a unit cost $cost(\hat{n})$. Each substrate link $\hat{l} \in \hat{L}$ is associated with an available bandwidth $\hat{l}_{av}$ and a unit cost $cost(\hat{l})$.

Similarly, Let $G = (N, L)$ be a weighted undirected graph that represents the VN request topology. N (resp. L) is the set of *required* virtual nodes (resp. links). Let req_n (resp.req_l) denote the *minimum required capacity (resp. bandwidth)* of the virtual node $n \in N$ (resp. link $l \in L$).

Figure 1 shows an example of VN and SN. The numbers in rectangles next to the substrate (resp. virtual) nodes represent the amount of available (resp. requested) node resources and the numbers next to the substrate (resp. virtual) edges represent the available (resp. required) bandwidth.

The initial VNE stage is out of the scope of this paper, diverse approaches can be found in the literature (centralized [6], [7] and distributed [8], [9]). We focus exclusively on the second stage and more specifically on managing bandwidth demand fluctuations by adapting links and network topology.

B. Management of resource demand fluctuation

The first stage, successful initial VNE, results in multiple VNs running simultaneously over the SN. This is the starting point of the second stage, the focus of our study. To represent this situation, we denote by $N_{\hat{n}}$ the set of virtual nodes hosted by the substrate node $\hat{n}$, and $L_{\hat{n}}$ the set of virtual links *incident to or passing through* the substrate node $\hat{n}$. For example, in figure 1, $N_{\hat{a}} = \{a\}$ and $L_{\hat{a}} = \{(a, b)\}$.

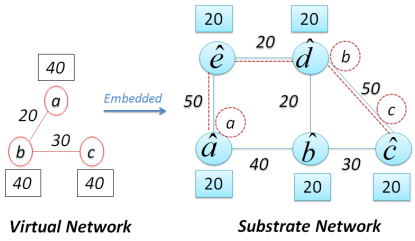


Fig. 1. Example of initial VN embedding

Over time, the VN end user requirements can change, for instance when staring/ completing a new task/ application, or when their required resources change, consequently, the corresponding VN characteristics (topology and resources requirements) will change.

We enumerate four bandwidth demand fluctuation scenarios: i) partially releasing no more required bandwidth ii) completely removing virtual links iii) adding new virtual links and iv) allocating more bandwidth to embedded links. To cope with scenarios i) and ii) the substrate network provider should release some bandwidth. For cases iii) and iv), the provider should allocate more bandwidth on the hosting paths *when feasible*, otherwise find new paths to support the requested bandwidth.

To model a virtual link l bandwidth demand fluctuation we define $allo_l$, the amount of bandwidth *allocated* to l. Note that $req_l = allo_l$ when the bandwidth demand is met. Else, we formulate the four bandwidth demand fluctuation scenarios as follows:

- i) Decrease in Bandwidth Requirement **(DBR)**: $allo_l > req_l > 0$ (the required bandwidth is lower then the allocated).
- ii) Link Removal **(LR)**: $0 < allo_l$ and $req_l = 0$ (link l does not require the allocated bandwidth any more and the link should be removed).
- iii) Link Addition **(LA)**: $allo_l = 0$ and $0 < req_l$ (link l requires an amount of bandwidth not allocated yet, it should be added to the VN topology).
- iv) Increase in Bandwidth Requirement (**IBR**: $0 < allo_l < req_l$ (the required bandwidth is higher then the allocated).

We opt for a self-stabilizing approach to manage bandwidth demand fluctuation. We will first introduce the Self-stabilization concept and motivate this choice, then, we will describe our solution.

IV. Self-Stabilization

A. Introduction to Self-Stabilization

Self-Stabilization is related to Autonomic Computing [10], which entails several "Self-*" attributes: self-management, self-configuration, self-healing, etc. It is the property of an autonomous process to reach a "stable state" *no matter what initial state is given.* Hence a self-stabilizing system will eventually correct itself automatically without the need for an outside intervention.

In a self stabilizing system, each of the individual entities (nodes) composing it maintains local variables determining its state that can be either *stable* or *active*. A node can execute *an action* to make *a move* and change its state. The system is said to be stable when all the nodes are stable. Hence all the active nodes should make moves to reach the system stability. The number of moves (or rounds if nodes make moves simultaneously) required to reach the stable state is often used to measure the efficiency of the algorithm.

Central to the theory of self-stabilization is the notion of ***daemon*** [11], it is the entity responsible for setting the actions/moves execution order. In fact, in each round, it selects some of/ or all active nodes to make a move, and the process continues until there are no more active nodes, i.e. until reaching stable state. In the self stabilization literature, a daemon is often viewed as an *adversary* to the system that tries to *prevent* stabilization by scheduling the *worst* possible nodes for execution.

However, in our work, this definition seems unnecessarily restrictive. In fact, most resource management systems ([5], [12]) employ a *manager* (or controller) to supervise and *harmonize* the distributed entities behavior. In the same perspective, we replace the daemon by a scheduler that *helps* the system to converge by setting the *best* nodes execution scheme. In the following, this scheduler is called *controller*.

Unlike resource allocation traditional central entities, our controller *does not require* the dynamic description of physical resources to make scheduling decisions. *Only* the substrate network topology (generally static) and the list of the physical nodes scheduled to execute are needed. In fact, the nodes are *the only responsible* of choosing the actions they need to perform, the controller simply sets the *actions execution order* using an elementary scheduling algorithm that sorts the nodes according to their action *priority*.

B. Motivation for Self-stabilization

A self stabilizing system $i)$ is *local-view*, hence maintaining a global up-to-date description of the physical resources is not required, $ii)$ allows simultaneous nodes execution, hence favors managing *multiple and different* demand fluctuations *simultaneously*, $iii)$ is suitable to the *topology changing networks* as modification of the code when the system topology evolves *during operation* is not required and $vi)$ does not require initialization as the system system converges irrespective of its original state. All this reasons motivated us to explore the self-stabilization concept.

V. A self-stabilizing model for bandwidth allocation

A. System model

This section extends the network model proposed in III with *self stabilizing elements*. Namely, we expand the *virtual link* and *substrate node* models.

1) ***Virtual link description****:* Let src_l and dst_l denote the source and destination virtual nodes of virtual link l, and let $\vec{P_l}$ denote the ordered list of *substrate* nodes composing l's hosting path. In Fig. 1, for $l = (a, c)$, we have $src_l = a$, $dst_l = c$ and $\vec{P_l} = \left\{\hat{a}, \hat{b}, \hat{c}, \hat{d}, \hat{e}\right\}$.

2) ***Substrate node description****:* We suppose that each substrate node knows the local variables of its neighbors (through periodic message passing or a shared memory ([13])), yet it can only modify *its* local variables. We model each substrate node *local-view* of its environment as follows:

Local view of mapping: for each virtual link l passing through or incident to substrate node $\hat{n}$, let $\hat{n}_x$ denote the feature x describing l and saved in (seen by) $\hat{n}$ ($x \in \left\{l, allo_l, src_l, dst_l, \vec{P_l}\right\}$). For example, $\hat{n}_{allo_l}$ describes the amount of bandwidth allocated to l as seen by $\hat{n}$. In other terms, each substrate node saves "a copy" of the parameters describing the virtual links $l \in L_{\hat{n}}$. When the system is in the stable state, all the substrate nodes in $\vec{P_l}$ share the same l's description, else end to end update is required.

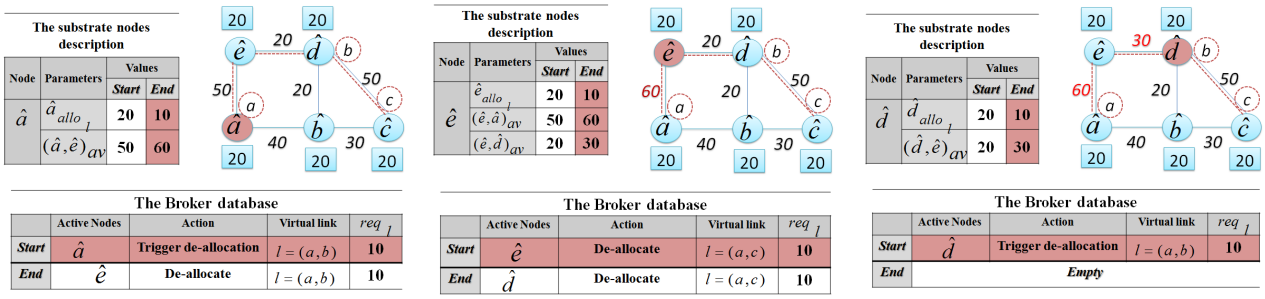
Local view of available resources: let $Neigh(\hat{n})$ denote the set of $\hat{n}$ neighbors (adjacent substrate nodes). Let $\hat{m} \in Neigh(\hat{n})$ be such a neighbor. To describe the substrate link $(\hat{m}, \hat{n})$ connecting $\hat{n}$ and $\hat{m}$ in a distributed manner, node $\hat{n}$ (resp. $\hat{m}$) holds a variable $(\hat{n}, \hat{m})_{av}$ (resp $(\hat{m}, \hat{n})_{av}$) defining the available bandwidth on $(\hat{m}, \hat{n})$. When the SN is stable, $(\hat{n}, \hat{m})_{av} = (\hat{m}, \hat{n})_{av}$ i.e. the substrate nodes $\hat{n}$ and $\hat{m}$ share the same view on substrate link $(\hat{n}, \hat{m})$ available bandwidth, otherwise an update is required.

B. The Self stabilizing framework

We propose a framework to deal with bandwidth demand fluctuation of embedded virtual networks. The framework contains ***algorithms***, composed of a set of *actions* executed locally by the substrate nodes, and a ***controller*** defining the actions execution scheme. We propose three different algorithms to deal with the four bandwidth fluctuation types described previously. We will first describe the controller role and the general execution plan, then we will describe the algorithms.

1) ***Controller description****:* The controller is responsible of setting the actions execution plan. It holds a *local database* containing a subset of substrate nodes, called *active nodes*, scheduled to execute one or many actions. An active node requires *the controller permission* to perform. Each action A is associated with a *"priority"* p_A defining its precedence.

The framework execution plan is organized into *rounds*. In each round, the controller examines the list of *active nodes* in its database and allows the nodes scheduled with the most urgent action/task to run *simultaneously*. At the end of the



(a) Round 1 (b) Round 2 (c) Round 3

Fig. 2. A step by step example of Algorithm 1 execution

round, the controller updates its database (it removes the executed actions and keeps the others for the next round). This process continues until reaching system stability.

Note that the local views of substrate nodes are combined with the controller to reduce load on this central point of failure that usually has a back up for fault tolerance. One of the substrate nodes can be elected as a central node as needed.

Two reasons can activate a substrate node: either a new bandwidth request is submitted, or the node is *solicited* by a neighboring node. When a virtual link l requires an amount of bandwidth req_l ($\neq allo_l$), the substrate node hosting src_l ($host(src_l)$) is activated: an entry of the form ($host(src_l)$, A, l, req_l) is added to the controller database to define i) the activated node, ii) the task to run, iii) the concerned virtual link and vi) the new request. Substrate nodes cooperate to reach stability, to do so a node $\hat{n}$ can activate its neighbor $\hat{m} \in Neigh(\hat{n})$ to perform an action A (concerning a virtual link l with $req_l \neq allo_l$). Node $\hat{n}$ asks the controller to add ($\hat{m}$, A, l, req_l) in its database. The different possible actions are described in the following.

*2) **Algorithms description**:* Three algorithms manage the four bandwidth fluctuation types. Each algorithm is composed of *actions*. The first algorithm concerns scenarios i) Decrease in Bandwidth Requirement and ii) Link Removal. The second manages the Link addition (scenario iii). The third handles the Increase in Bandwidth Requirement (scenario iv). Due to lack of space, only the two first algorithms will be detailed.

Algorithm1, Decrease in Bandwidth Requirement or Link Removal: This algorithm deals with the two following cases: i) a virtual link l requires less bandwidth or ii) should be completely removed from the VN topology. Hence, no more required bandwidth should be freed and the virtual link mapping should be updated along the hosting path.

To do so, the origin substrate node (hosting src_l) is activated upon receiving the new request. If selected by the controller, the node executes the following action: *Trigger Allocation*, to i) update its state (mapping and available resources) and ii) activate the next node in $\vec{P}_l$. If selected by the controller in the next round, the latter runs an other action: *De-allocate*; it updates its local description to *synchronize* with the soliciting node, then activates the next node in $\vec{P}_l$. This process continues until reaching the end (extremity) of the substrate path. The action *Trigger Allocation* (resp. *De-allocate*) is associated with a priority p_{TA} (resp. p_D), such that $p_{TA} > p_D$. The idea behind this choice is to favor managing multiple requests simultaneously. Details can be found in Algorithm 1.

Algorithm 1: Decrease in Bandwidth Requirement/Link Removal

Inputs:1: The executing node : $\hat{n}$
2: The concerned virtual link : l
3: The new required bandwidth : req_l

Action 1: Trigger de-allocation (Priority=$p_{TA} > p_D$)

Steps: 1: $(\hat{n}, \hat{o})_{av}$ += $(\hat{n}_{allo_l} - req_l)$ where $\hat{o} = \vec{P}_l^{\,next}$ (i.e. release no more required bandwidth in the substrate link $(\hat{n}, \hat{o})$ where $\hat{o}$ is the next node in l's hosting path)
2: $\hat{n}_{allo_l} = req_l$ (update the amount of resources allocated to l)
3: Add ($\hat{o}$, $De-allocate$, l, req_l) to the controller database (i.e. activate the next node in $\vec{P}_l$)
4: **if** $req_l = 0$ (i.e. In case of virtual link removal) **then**
5: $L_{\hat{n}}.remove(l)$ (i.e. remove l from $\hat{n}$ mapping list)
6: **end if**

Action 2: De-allocate (Priority=p_D)

Steps: 1: $(\hat{n}, \hat{m})_{av}$ += $(\hat{n}_{allo_l} - req_l)$ where $\hat{m} = \vec{P}_l^{\,previous}$ (i.e. update available bandwidth in $(\hat{n}, \hat{m})$ to synchronize with the the soliciting node $\hat{m}$, i.e. the previous node in hosting path)
2: $\hat{n}_{allo_l} = req_l$ (update the amount of resources allocated to l)
3: **if** $dst_l \notin N_{\hat{n}}$ (if $\hat{n}$ is not the last node in $\vec{P}_l$) **then**
4: $(\hat{n}, \hat{o})_{av}$ += $(\hat{o}_{allo_l} - req_l)$ where $\hat{o} = \vec{P}_l^{\,next}$ (release no more required bandwidth in $(\hat{n}, \hat{o})$, $\hat{o}$ is the next node in $\vec{P}_l$)
5: Add ($\hat{o}$, $Deallocate$, l, req_l) to the controller database to activate $\hat{o}$
6: **end if**
7: **if** $req_l = 0$ (i.e. in case of virtual link removal) **then**
8: $L_{\hat{n}}.remove(l)$ (i.e. remove l from $\hat{n}$'s mapping list)
9: **end if**

Figure 2 depicts a detailed example of the algorithm execution. We use the VN and SN of figure 1. For each round, we show two tables. The first describes the most relevant substrate node parameters and the second illustrates the controller database (at the start/ end of the round). The second table is a 4 column table defining i) the active nodes ii) the scheduled actions, iii) the concerned virtual links and vi) the new bandwidth requests. In each round, the *executing* substrate nodes and the main changes are colored in red.

Suppose that the virtual link $l = (a, b)$, initially requesting 20 bandwidth units and hosted by the substrate path $\vec{P}_l$ =

$(\hat{a}, \hat{e}, \hat{d})$, is now requiring a new amount of bandwidth = 10 units, and let us run through the algorithm.

Initially, the substrate network is stable: the required bandwidth is met for all the embedded virtual links, in particular, $\hat{a}_{allo_l} = req_l = 20$, where $\hat{a} = host(src_l)$. Moreover, $(\hat{a}, \hat{e})_{av} = (\hat{e}, \hat{a})_{av} = 50$ and the controller database is empty.

After receiving l's new request $req_l = 10$, the substrate node $\hat{a}$ is activated to run the *Trigger de-allocation* action. As there is only one node in the controller database, $\hat{a}$ is selected to run in the first round. It $i)$ updates l's mapping to have $\hat{a}_{allo_l} = 10$, $ii)$ frees the no longer required bandwidth in the substrate link $(\hat{a}, \hat{e})$ ($(\hat{a}, \hat{e})_{av} = 60$), and finally $iii)$ activated the next node in $\vec{P}_l$: $\hat{e}$ is added to the controller database.

In the second round, node $\hat{e}$ is selected by the controller to execute the action $De-allocate$. It $i)$ updates l mapping and the available bandwidth in $(\hat{e}, \hat{a})$ to synchronize with $\hat{a}$, $ii)$ releases bandwidth from $(\hat{e}, \hat{d})$ and finally $iii)$ activates $\hat{d}$. In the last round, node $\hat{d}$ runs the same steps as $\hat{e}$ except activating the the next node in $\vec{P}_l$ as the path's end is reached.

Note that the system has reached stability in 3 rounds, which corresponds to the number of substrate nodes composing $\vec{P}_l$. Moreover, all the substrate nodes in $\vec{P}_l$ have *the same new l description* at the end of execution, and the amount of available bandwidth is updated *through the entire substrate path.*

Algorithm 2, Link Addition:This is the case where a new virtual link l *should be added* to connect two embedded virtual nodes.

To do so, a substrate path connecting the substrate nodes hosting src_l and dst_l, and having enough available bandwidth ($> req_l$) should be found. We define the cost of embedding a virtual link l as the sum of costs of the substrate links hosting l, as in [14]. The aim is to find *the most cost effective* substrate path, in a distributed, and self stabilizing manner.

To achieve this, the algorithm runs in two steps: *first* it $i)$ searches all available paths (having enough bandwidth) to connect the two end nodes and $ii)$ saves the required bandwidth in each of them. *Second*, it $i)$ selects the most cost effective substrate path among the K *first arriving path proposals* (where K is a tuning variable), $ii)$ maps the virtual link l in the best path, and $iii)$ releases previously reserved bandwidth in non selected paths. Four actions compose this algorithm:

Action 1, Trigger searching and reserving bandwidth: This action concerns the source substrate node (hosting src_l). It triggers the first algorithm step. A node $\hat{n}$ executing this action checks if all its attached substrate paths are saturated, if it is the case, the new bandwidth request is immediately rejected as no path connecting $host(src_l)$ and $host(dst_l)$ can be found. Else, $\hat{n}$ starts building a path proposal by adding $\hat{n}$ to $\vec{P}_l$, initially empty. Then, for all available attached substrate links, $\hat{n}$ reserves bandwidth and activates the corresponding nodes to do so.

Note that the existence of an available substrate path to connect $host(src_l)$ and $host(dst_l)$ *is not guarantied*, hence, we risk to search infinitely for a nonexistent substrate path and *never reach stability*. To avoid this, we define a timer $Timer_l$, to limit the path search phase duration. It can be set in terms of rounds, and depends on the SN dimension. $Timer_l$ is launched when $\hat{n}$ starts searching for a path. If $Timer_l$ expires and no path is found, the request is rejected. Detailed algorithm can be found in Algorithm 2, action 1. ***Action 2: Search and reserve bandwidth***:

This action concerns the other substrate nodes searching for an available path for l. A node $\hat{n}$ executing this action runs different operations depending on the situation:

Algorithm 2: Link Addition

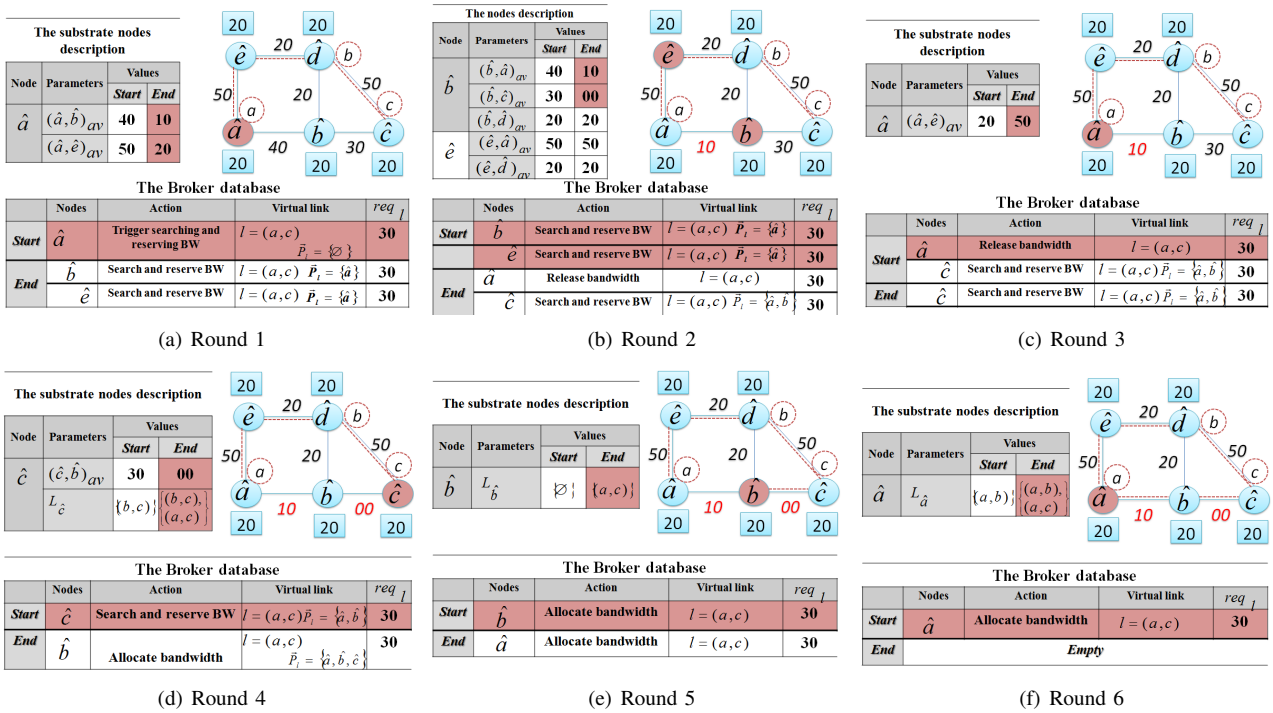
Inputs:
1: The executing node : $\hat{n}$
2: The concerned virtual link : l
3: The new required bandwidth : req_l

Action 1: Trigger searching and reserving BW (Priority=p_{TSR})

Steps:
1: Launch $Timer_l$
2: **if** all connected substrate links are saturated **then**
3: Reject the New bandwidth request
4: **else**
5: $\vec{P}_l.add(\hat{n})$ (i.e. start building a path proposal)
6: **for** all available attached links $(\hat{n}, \hat{o})$ **do**
7: $(\hat{n}, \hat{o})_{av}$ -= req_l (i.e. reserve bandwidth in $(\hat{n}, \hat{o})$)
8: Activate $\hat{o}$
9: **end for**
10: **end if**

Action 2: Search and reserve BW (Priority=p_{SR})

Steps:
1: **if** $Timer_l$ has not expired and no path is found yet for l **then**
2: **if** $dst_l \notin N_{\hat{n}}$ (i.e. if $\hat{n}$ is not the end node) **then**
3: **if** all attached substrate links are saturated, or we risk to produce a loop **then**
4: Reactivate the soliciting node to Release BW
5: **else**
6: $(\hat{n}, \hat{m})_{av}$ -= req_l, $\vec{P}_l.add(\hat{n})$ (i.e. synchronize with the soliciting node and continue building the path)
7: **for** for all available links $(\hat{n}, \hat{o})$ with no risk to produce a loop **do**
8: $(\hat{n}, \hat{o})_{av}$ -= req_l (i.e. reserve bw in the link)
9: Activate the neighbor $\hat{o}$ to Reserve BW
10: **end for**
11: **end if**
12: **else**
13: (i.e. the path end is reached and this is the Rth path proposal)
14: $(\hat{n}, \hat{m})_{av}$ -= req_l, $\vec{P}_l.add(\hat{n})$ (i.e. synchronize with the soliciting node and continue building the path)
15: **if** $R < K$ **then**
16: Save the path solution
17: **end if**
18: **if** $R = K$ **then**
19: Select the most cost effective path, called $BestPath$
20: Map l to the best path and save the new mapping
21: Activate the previous node in $BestPath$ to Allocate BW
22: **for** all non selected paths **do**
23: Release previously reserved BW
24: Activate the previous node in the path to release BW
25: **end for**
26: **end if**
27: **if** $R > K$ (i.e. this proposal is arriving late) **then**
28: $(\hat{n}, \hat{m})_{av}$ += req_l (i.e. release previously reserved bw)
29: Activate the soliciting node to Release BW
30: **end if**
31: **end if**
32: **else**
33: Add ($\hat{m}$, Release BW, l, req_l) to the controller database (i.e. activate $\hat{m}$ to Release BW)
34: **end if**



(a) Round 1 (b) Round 2 (c) Round 3

(d) Round 4 (e) Round 5 (f) Round 6

Fig. 3. A step by step example of algorithm 2 execution

IF $\hat{n}$ is not the end node, there are two cases: in the *first case*, all adjacent substrate links are unavailable *or* we risk to produce a loop if we add the neighboring node to l's path (the node is already in $\vec{P}_l$), we conclude that n is *the end of a no-through path!* and previously reserved bandwidth should be released from the saved path. To do so $\hat{n}$ re-activates the soliciting node to perform a *Release bandwidth* action, described below, that will be spread *back* through the reserved path. In the *second case:*, node $\hat{n}$ *synchronizes* with its soliciting node and goes on reserving bandwidth on available substrate links, then solicits corresponding neighbors to do so. **ELSE** the path end is reached.

Recall that $\hat{n}$ must wait K path proposals to select the most cost effective path, hence $\hat{n}$ synchronizes with soliciting node then runs the following actions depending on the rank R of the arriving proposal:

- *if* $R < K$: node $\hat{n}$ simply saves the path proposal.
- *If* $R = K$: node $\hat{n}$ i) selects the most cost effective path, ii) assigns it to l and re-activates the previous node in the selected path to spread *back* an *Allocate bandwidth* action, described below, finally iii) for all non selected paths, the reserved bandwidth is released, and the previous node in each path is activated to spread back a *Release bandwidth* action. Node $\hat{n}$ informs the controller that a path proposal was found for l to stop reserving other paths.
- *If* $R > K$: the proposal is arriving late, thus it is rejected: reserved bandwidth is released and a *Release bandwidth* action is triggered among the reserved path.

Note that if a path proposal is found or the link timer has expired, a node activated to execute this action will not run the previously described steps, but simply spreads back a *Release BW* action to free the previously reserved path.

Action 3: Allocate bandwidth: Node $\hat{n}$ updates mapping and **if** it is not the origin node, it activates the *previous* node in $\vec{P}_l$ to spread *back* the Allocate bandwidth request.

Action 4: Release bandwidth: Node $\hat{n}$ releases previously reserved bandwidth and if it is not the origin node, it releases reserved bandwidth in ($\hat{n}$,$\hat{m}$) (with $\hat{m}$ the previous node in $\vec{P}_l$) and activates $\hat{m}$ to spread *back* the Release bandwidth request.

Actions priorities: We choose the following priority order for previous actions: $p_{SR} < p_{Rel} < p_{All} < p_{TSR}$ where p_{TSR} is for action *Trigger searching and reserving BW*, p_{SR} for *Search and reserve bandwidth*, p_{All} for *Allocate bandwidth* and p_{Rel} for *Release bandwidth* action.

The motivation is threefold: first we give the highest priority to *Trigger searching and reserving bandwidth* action to favor dealing with new arriving requests and thus handle many demands simultaneously. *Allocate bandwidth action* comes next in order to allocate the virtual links as soon as an available path proposal is found. Finally, by giving *Release bandwidth* a higher priority then *Search and reserve bandwidth*, we release bandwidth before searching for available paths to increase the number of path solutions. Note that the controller selects a set of *non adjacent* active nodes to execute *Trigger searching and reserving BW* or *Search and reserve bandwidth* actions to avoid conflicts.

Figure 3 depicts a detailed example of the algorithm 2 execution. We use the same VN, SN and notations of the first example. Imagine that a new virtual link $l = (a, c)$ has to

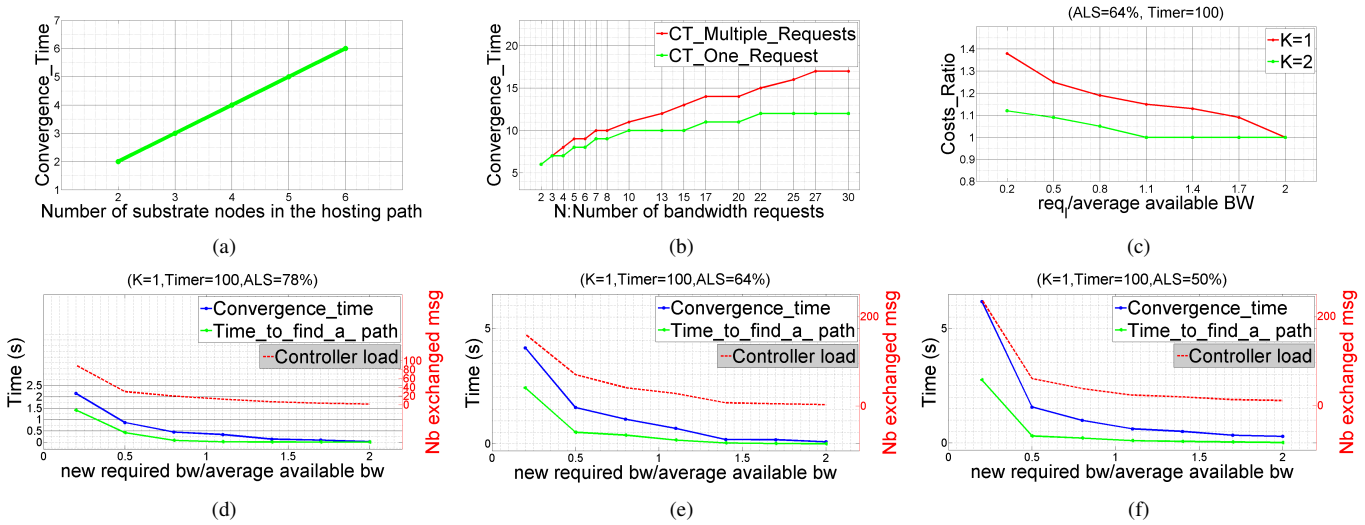


Fig. 4. Simulation results

be added to the VN to connect the two virtual nodes a and c, with $req_l = 30$. To simplify the example, we set K to 1 (the first path proposal is embedded) and $Timer_l$ to 100 (there is enough time to find a path). First, the node $\hat{a}$ is scheduled to execute a *Trigger BW research and allocation action*. When selected by the controller in round 1, $\hat{a}$ $i)$ reserves bandwidth on $(\hat{a}, \hat{b})$ and $(\hat{a}, \hat{e})$ ($(\hat{a}, \hat{b})_{av} - = 30$, $(\hat{a}, \hat{e})_{av} - = 30$), $ii)$ updates $\vec{P_l}$ ($\vec{P_l} = \{\hat{a}\}$) and $iii)$ activates $\hat{b}$ and $\hat{e}$ to continue searching for available paths. In round 2, $\hat{b}$ and $\hat{e}$ execute simultaneously: $\hat{b}$ $i)$ synchronizes with $\hat{a}$, $ii)$ updates $\vec{P_l}$ ($\vec{P_l} = \left\{\hat{a}, \hat{b}\right\}$), $iii)$ reserves bandwidth only on $(\hat{b}, \hat{c})$, as $(\hat{b}, \hat{d}) < req_l$, and $vi)$ activates $\hat{b}$. As node $\hat{e}$ is the end of a non through path ($(\hat{e}, \hat{d}) < req_l$), it re-actives $\hat{a}$ to cancel previous reservation. In round 3, two nodes are active, $\hat{a}$ is scheduled to *Release bandwidth* and $\hat{c}$ to *Search and reserve bandwidth*. Since $p_{SR} < pRel$, $\hat{a}$ is selected by the controller, it updates $(\hat{a}, \hat{e})_{av}$ to release bandwidth. Node $\hat{c}$ is selected in round 4, since it is the end node, it updates l's mapping (adds l to $L_{\hat{c}}$) and re-activates $\hat{b}$ to spread back the *Allocate bandwidth* update. During the rounds 5 and 6, l's mapping is updated along $\vec{P_l} = \left\{\hat{a}, \hat{b}, \hat{c}\right\}$.

Note that the algorithm converges in 6 rounds, but a path solution was found since the 4th round.

Algorithm3, Increase in bandwidth requirement: This is the case where an already embedded virtual link l requires more bandwidth. To handle such request, we propose a two step algorithm: **first**, the algorithm checks if there is enough bandwidth on the path hosting l to meet the new demand. If it is the case, the required bandwidth is allocated and the request is satisfied. Else we move to the **second** step that consists of $i)$ finding a new path for l, and $ii)$ de-allocating l from its old hosting path, in other terms, we perform *a virtual link migration*. To do so, a slightly modified version of already defined actions in previous Algorithms is used. Due to lack of space, these actions will not be detailed.

VI. Simulation results and evaluation

In this section, we will evaluate the effectiveness of our algorithms by conducting extensive simulations. We will describe the simulation environment and then present the results.

A. Simulation environment

We adjusted the C++ simulator used in previous work [14] to fit our scenario: substrate nodes with local view, and a round per round algorithm execution. The GT-ITM tool ([15]) is used to generate random topologies of the substrate and VN networks.

The SN size is set to 50 nodes and each pair of substrate nodes is randomly connected with probability 0.5. The node resource capacity and edge resource capacity are randomly drawn between 0 and 50 for nodes and between 0 and 100 for links. The per unit node and edge resources costs are selected randomly between 0 and 50.

The VNs requests have between 2 and 10 virtual nodes in their topologies with an average connectivity also set to 0.5.The node resource capacity is randomly selected between 0 and 20 and the edge resource capacity between 0 and 50.

In order to initialize the scenario and start the system from a typical situation we map the virtual nodes greedily and follow with the shortest path algorithm to map edges. This step leads to suboptimal embedding that can reflect the state of a SN subject to multiple virtual link evolutions. The central performance metric will be the *the number of rounds required to reach the stable state* (i.e. to converge), called $Convergence\_Time$. Other metrics will be presented later.

B. Simulation results

1) Algorithm1: Decrease in Bandwidth requirement (DBR) or Link Removal (LD): We simulate two scenarios to assess the effectiveness of this algorithm:

Scenario 1: Only one bandwidth request is managed. In accordance with the example of Fig.2, Fig. 4.a shows that $Convergence\_Time$ is equal to the number of substrate nodes supporting the substrate path hosting the evolving virtual link.

Scenario 2: Multiple bandwidth fluctuations are handled simultaneously. To create a highly dynamic environment and unpredictable states, we select randomly N virtual links among the 96 hosted by the SN as virtual links with fluctuating bandwidth demands. To each link l, we associate a Decrease in Bandwidth Requirement or Link Removal request randomly. For the DBR, we set the new bandwidth demand to $req_l = allo_l/2$.

If *multiple* bandwidth requests are managed simultaneously, $Convergence\_Time$ is at least equal to the convergence time when handling the *the longest* path among those hosting the evolving links. With this idea in mind, we evaluate the performance of our algorithm in handling N bandwidth fluctuations at the same time.

We measure both i) the $Convergence\_Time$ when handling *all the requests*, called ***CT_Multiple_Requests*** and ii) the $Convergence\_Time$ when managing *only* the virtual link with the longest supporting path, called ***CT_One_Request*** and compare.

Figure 4.b depicts the results of 200 averaged runs and shows that both convergence times increase with the number of bandwidth requests. Moreover, the gap between the two curves is small compared to the number of requests handled simultaneously ($CT\_Multiple\_Requests/CT\_One\_Request < 1.4$ for N=30). In other terms, managing multiple bandwidth fluctuations is at most 1.5 more time expensive then managing only one request. The two convergence times are even equal for low N values.

2) Algorithm 2: Link addition (LA): Recall that this is the case where a new virtual link is added to the VN topology. We will examine three metrics to evaluate the algorithm: *the virtual link embedding cost*, *the convergence time* and *the controller load*. To simulate dynamic and unpredictable LA requests, we select randomly a pair of virtual nodes (among the 133 hosted in the substrate network) as source and destination of the new virtual link. Note that the pair of nodes belong to the same VN and are non adjacent (not connected by a virtual link, in order to avoid multi-graphs).

Embedding cost Recall that the end node of a newly added virtual link should wait for K path proposals to select the most cost effective one. We measure the virtual link embedding cost for $K = 1$ and $K = 2$, for different bandwidth demands req_l, and compare to the shortest path cost, found with *a global view of the system*.

Figure 4.c shows the ratio of *the embedding cost found with our algorithm*, and that of *the shortest path algorithm*, called $Costs_Ratio$for different values of $req_l/Average\_available\_BW$ for 100 accepted requests, where $Average\_available\_BW$ is the average of available bandwidth in the substrate network.

Note that the ratio decreases with K : the most path proposals we wait, the most chance we have to find the best path. Moreover, for small values of req_l, our algorithm fails in finding the shortest path as there are *many path solutions* that can meet the demand, however, when req_l increases, the number of available paths decreases and the gap between the two algorithms costs decreases.

Convergence time Recall that this algorithm is composed of two steps: the first searches for available paths, and the second assigns the most cost effective path to the virtual link and releases bandwidth from other paths. We measure two time durations: the first phase duration, called $Time\_to\_find\_a\_path$ and the total convergence time, i.e. the two steps duration, called $Convergence\_Time$. We set K to one, and $Timer$ to 100 rounds, and make evaluation for three SN configurations: $ALS = 78\%$, 64% and 50%, where ALS is the average substrate links saturation, as defined in [14]. We measure the time in seconds for more precision.

Figures 4.d, 4.e, 4.f depict the results of 100 averaged runs and show that both times decrease when $req_l/Average\_available\_BW$ increases. In fact, when the required bandwidth is high, there are few available links, hence few substrate nodes will be activated to search for a path, and the system will reach stability more rapidly. In contrary, when req_l is small, the majority of the substrate nodes in SN will be activated to search for a path, and the system needs more time to stabilize.

Moreover, looking at Figures 4.4, 4.5, 4.6 jointly, notice that both times decrease with the average link saturation (ALS), for the same reasons explained above. Finally, note that the gap between the two curves increases with ALS, in fact, when there are more available substrate resources (ALS low), more substrate paths will be reserved during the first phase of the algorithm, hence, after finding a path solution, releasing bandwidth from reserved paths will take more time.

Controller load analysis: in each round, the controller selects and notifies nodes to execute part of the bandwidth adaptation. A *loose* upper bound on controller load (measured by the number of exchanged messages) is obtained when all nodes are involved in each round. The bound is given by the product: substrate network size $\times$ the total number of rounds. Figures 4.d, 4.e, 4.f show that the controller load curve has nearly the same shape as that of the convergence time since it is proportional to the number of activated nodes. It decreases with the new bandwidth demand for the reasons explained above.

VII. Conclusion and future work

In this paper, a self stabilizing framework was proposed to deal with bandwidth demand fluctuation in embedded virtual networks. The solution is composed of a central controller, and three parallel, distributed and local view algorithms running in each substrate node to handle all types of bandwidth demand fluctuations. Simulation results show that many requests can be managed simultaneously at low cost and in a time effective way. Future work concerns demand fluctuation in embedded virtual nodes. We aim at extending the actual framework to handle all types of resource fluctuations in virtual networks.

REFERENCES

[1] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[2] R. Mijumbi, J.-L. Gorricho, J. Serrat, M. Claeysy, F. D. Turcky, and S. Latr, "Design and evaluation of learning algorithms for dynamic resource management in virtual networks," *(NOMS 2014)*, 2014.

[3] R. Mijumbi, J.-L. Gorricho, J. Serrat, M. Claeys, J. Famaey, and F. De Turck, "Neural network-based autonomous allocation of resources in virtual networks," in *European Conference on Networks and Communications*, 2014.

[4] R. Mijumbi, J.-L. Gorricho, J. Serrat, M. Shen, K. Xu, and K. Yang, "A neuro-fuzzy approach to self-management of virtual network resources." *Expert Syst. Appl.*, 2015.

[5] M. Seddiki, B. Nefzi, Y.-Q. Song, and M. Frikha, "Automated controllers for bandwidth allocation in network virtualization," in *IEEE 32nd International Performance Computing and Communications Conference*, 2013.

[6] Y. Zhu and M. H. Ammar, "Algorithms for assigning substrate network resources to virtual network components." in *INFOCOM*, 2006.

[7] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann, "Vne-ac: Virtual network embedding algorithm based on ant colony metaheuristic," in *(ICC) 2011*, 2011.

[8] I. Houidi, W. Louati, and D. Zeghlache, "A distributed virtual network mapping algorithm," in *Communications, 2008. ICC '08. IEEE International Conference on*, May 2008, pp. 5634–5640.

[9] M. Till Beck, A. Fischer, H. de Meer, J. Botero, and X. Hesselbach, "A distributed, parallel, and generic virtual network embedding framework," in *Communications (ICC), 2013 IEEE International Conference on*, June 2013, pp. 3471–3475.

[10] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*, 2005.

[11] S. Dubois and S. Tixeuil, "A taxonomy of daemons in self-stabilization," *CoRR*, vol. abs/1110.0334, 2011.

[12] R. Carter, D. St.Louis, and J. Andert, E.P., "Resource allocation in a distributed computing environment," in *Digital Avionics Systems Conference, 1998. Proceedings.*, 1998.

[13] J. B. P. Gambette, "Introduction l'algorithmique rpartie et l'auto-stabilisation," 2006.

[14] H. Jmila and D. Zeghlache, "An adaptive load balancing scheme for evolving virtual networks," in *12th Annual IEEE Consumer Communications and Networking Conference*, 2015.

[15] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *INFOCOM*, 1996.