Dynamic resource allocation and management in virtual networks and Clouds

Houda Jmila

THESE DE DOCTORAT CONJOINT TELECOM SUDPARIS et
L'UNIVERSITE PIERRE ET MARIE CURIE

Ecole doctorale : Informatique, Télécommunications et Electronique de Paris

Présentée par

Houda JMILA

Pour obtenir le grade de
DOCTEUR DE TELECOM SUDPARIS

Dynamic resource allocation and management in virtual networks and Clouds

---

Soutenue le 21 décembre 2015 devant le jury composé de :

Rapporteurs :		
Pascal Lorenz	Professeur	Université de Haute Alsace, France.
Luis Muñoz	Professeur	Université de Cantabrie, Santander.
Examinateurs :		
Steven Martin	Professeur	Université Paris 11, France.
Lila Boukhatem	Mdc. HDR	Université Paris 11, France.
Mourad Gueroui	Mdc. HDR	Université de Versailles, France.
Nadjib Ait Saadi	Mdc. Dr	Université Paris 12, France.
Directeur de thèse :		
Djamal Zeghlache	Professeur	Télécom SudParis. France.

Thèse N° 2015TELE0023

JOINT THESIS BETWEEN TELECOM SUDPARIS AND UNIVERSITY OF PARIS 6 (UPMC)

Doctoral School : Informatique, Télécommunications et Electronique de Paris

Presented by

Houda JMILA

For the degree of
DOCTEUR DE TELECOM SUDPARIS

Dynamic resource allocation and management in virtual networks and Clouds

---

Defence date : 21 December 2015

Jury Members :

<u>Reviewers :</u>		
Pascal Lorenz	Professor	University of Haute Alsace, France.
Luis Muñoz	Professor	University of Cantabria, Santander.
<u>Examiner :</u>		
Steven Martin	Professor	Paris 11 University, France.
Lila Boukhatem	Mdc. HDR	Paris 11 University, France.
Mourad Gueroui	Mdc. HDR	University of Versailles, France.
Nadjib Ait Saadi	Mdc. Dr	Paris 12 University, France.
<u>Thesis Supervidor :</u>		
Djamal Zeghlache	Professor	Télécom SudParis. France.

Thèse N° 2015TELE0023

In Honor of my grandfathers and grandmothers, I dedicate this work as a token of my deep love.

To my parents Khemais and Zahida,

I am particularly indebted for your sincere love, your unconditional trust and continuous support during my PhD study years. Thank you for everything!

To my dear husband Mohamed,

I am especially thankful for your love, your understanding and your continuous support. You gave me strengths on weak days and showed me the sun on rainy days. Thanks for always believing in me.

To my son Mouadh,

You are my sunshine, I hope you will be proud of your mom!

To my brothers Aladain and Daly,

Thanks for always standing by my side during difficult times and for the fun moments I have shared with you!

To all JMILA, FEHRI and IBN KHEDHER family members,

Thanks for your love, kind support and continuous encouragement!

Acknowledgement

I would like to express my deep and sincere gratitude to my supervisor, Prof. Djamal Zeghlache for his continuous support and constant guidance during my PhD study years in the SAMOVAR laboratory. Thank you for everything; it was a truly great experience working with you!

My special appreciation goes to professors Pascal Lorenz, Luis Muñoz and Steven Martin, and to Dr. Lila Boukhatem, Dr. Mourad Gueroui and Dr. Nadjib Ait Saadi.

I would like also to thank the staff of Telecom SudParis. Many thanks go to all my colleagues and friends inside and outside Telecom SudParis for the excellent and truly enjoyable ambiance.

Abstract

Cloud computing is a promising technology enabling IT resources reservation and utilization on a pay-as-you-go manner. In addition to the traditional computing resources, cloud tenants expect compete networking of their dedicated resources to easily deploy network functions and services. They need to manage an *entire Virtual Network* (VN) or infrastructure. Thus, Cloud providers should deploy dynamic and adaptive resource provisioning solutions to allocate *virtual networks that reflect the time-varying needs* of Cloud-hosted applications. Prior work on virtual network resource provisioning only focused on the problem of mapping the virtual nodes and links composing a virtual network request to the substrate network nodes and paths, known as the Virtual network embedding (VNE) problem. Little attention was paid to the resource management of the allocated resources to continuously meet the varying demands of embedded virtual networks and to ensure efficient substrate resource utilization.

The aim of this thesis is to enable dynamic and preventive virtual network resources provisioning to deal with demand fluctuation during the virtual network lifetime, and to enhance the substrate resources usage. To reach these goals, the thesis proposes adaptive resource allocation algorithms for evolving virtual network requests. First, we will study in depth the *extension of a virtual node*, i.e. an embedded virtual node requiring more resources, when the hosting substrate node does not have enough available resources. Second, we will improve the previous proposal to consider the *substrate network profitability*. And finally we will deal with the *bandwidth demand variation* in embedded virtual links.

Consequently, the first part of this thesis provides a heuristic algorithm that deals with virtual nodes demand fluctuations. The main idea of the algorithm is to re-allocate one or more co-located virtual nodes from the substrate node, hosting the evolving node, to free resources (or make room) for the evolving node. In addition to minimizing the re-allocation cost, our proposal proposal takes into account an reduces the service interruption during migration. The previous algorithm was extended to design a preventive re-configuration scheme to enhance substrate network profitability. In fact, our proposal takes advantage of the resource demand perturbation to tidy up the SN at minimum cost and disruptions. When re-allocating virtual nodes to make room for the extending node, we shift the

most congested virtual links to less saturated substrate resources to balance the load among the SN. Our proposal offers the best trade off between re-allocation cost and load balancing performance. Finally, a distributed, local-view and parallel framework was devised to handle all forms of bandwidth demand fluctuations of the embedded virtual links. It is composed of a Controller and three algorithms running in each substrate node in a distributed and parallel manner. The framework is based on the self-stabilization approach, and can manage many and different forms of bandwidth demand variations simultaneously.

Contents

List of Figures

List of Tables

Acronyms

BW	Bandwidth
DBR	Decrease in Bandwidth Requirement
LR	Link Removal
LA	Link Addition
IBR	Increase in Bandwidth Requirement

Chapter 1

Introduction

Contents

The internet is continually evolving, shifting from a mere connectivity network to a content based network. Likewise, Internet users are nowadays more demanding. In addition to communicating, they also expect to get *on demand, cheap and easily accessible* resources and computing services.

In this context, Cloud computing is a promising technology enabling Utility Computing (Buyya *et al.* (2009)) reservation and utilization on a pay-as-you-go manner according to users applications demand. Therefore, Cloud clients no longer need to buy, maintain, update and manage their infrastructure resources (Armbrust *et al.* (2009)).

Although different types of services are supplied by Cloud providers (Software/ Platform/ Infrastructure As A Service), little attention was paid to the network. Recently, the Network As A Service model (Costa *et al.* (2012)) has changed this scenario by enabling dynamic provisioning of *entire Virtual Networks (VNs)*. However, allocating a virtual network in the *active and dynamic* Cloud environment requires *flexible and adaptive* resource provisioning algorithms to deal with the *changing demands* of the virtual network during its lifetime.

In fact, most cloud-based applications are characterized by a dynamically fluctuating workload related to the nature of the offered services. In order to provision network re-

sources that support such applications, *dynamic* virtual network provisioning techniques are required. More specifically, a continuous *down or/and up- scaling* of the amount of allocated resources should be guaranteed to reflect the *time-varying needs of Cloud-hosted applications.* Past research investigated the issue of efficiently provisioning virtual network resources. However, most of the work only focused on the problem of *mapping the virtual nodes and links composing a virtual network request to the substrate network nodes and paths*, known as the **Virtual network embedding (VNE) problem** (Fischer *et al.* (2013)). Little attention was paid to the **resource management** of the allocated resources to continuously meet the varying demands of embedded virtual networks and to ensure efficient substrate resource utilization. In this thesis we will try to solve this issue by proposing new efficient adaptive resource allocation schemes.

This chapter is organized as follows. First we will introduce the Cloud Computing concept and its different service models and environment actors. Then the NaaS model and the concept of connecting the Cloud will be presented. Afterward, we will describe the problem addressed in this work and summarize the thesis contributions.

1.1 The Cloud Computing paradigm

Among the many attempts to define the *Cloud Computing concept*, we cite and rely on the definition introduced by the National Institute of Standards and Technology (NIST) (Mell & Grance (2011)):

> *"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction "*

Cloud computing associates several computing concepts: i) *Grid computing* (Krauter *et al.* (2002)), ii) *Utility computing* (Buyya *et al.* (2009)) and iii) *Virtualization* (Chowdhury & Boutaba (2009)). *Grid computing* uses the resources of different computers to handle a complex problem in a realistic time. *Utility Computing* defines a service provisioning model where computing resources are provided on demand and charged according to usage. *Virtualization* creates virtual version of hardware resources, simpler to manage. In conclusion, Cloud Computing is a grid computing which uses virtualization technologies at multiple levels to realize utility computing.

1.2 The Cloud service models

Traditionally, three service models are proposed for the Cloud (Zhang *et al.* (2010)):

- **The Software As A Service model (SaaS):** The Cloud provides software to the users by offering on demand applications over the Internet. The software is delivered and managed remotely by one or more providers. For example, Microsoft 365, Salesforce, Citrix GoToMeeting are SaaS products.

- **The Platform As A Service model (PaaS):** The Cloud provides a platform to deploy user application and software. The Cloud consumers can develop cloud services and applications directly on the PaaS cloud. Examples of PaaS providers include Amazon Web Service, Sales Force, Long Jump and Windows Azure.

- **The Infrastructure As A Service model (IaaS):** The Cloud allows users to use computing resources like processing, storage, computing hardware and so on. Users just pay resources usage. *Virtualization* is extensively used in IaaS cloud in order to decompose physical resources and offer virtual instances to customers in an isolated way. For instance, Amazon Web Service, Microsoft Azure and Google Compute Engine are IaaS products.

1.3 The Cloud environment actors

Actors in the Cloud environment are different from those of traditional Internet. In fact, thanks to virtualization, the role of the Internet Service Provider is decoupled into two independent entities; the ***Cloud Infrastructure Provider*** and the ***Cloud Service Provider***. The Cloud Infrastructure Provider owns the Cloud resources while the Cloud Service Provider creates and runs applications on these resources, to offer utility to the ***Cloud End user***. Hereafter we describe these three actors depicted in figure 1.1.

- **The Cloud Infrastructure Provider** owns and manages the Cloud resources. Relying on virtualisation tools, the rpovider creates and provides on-demand virtualised

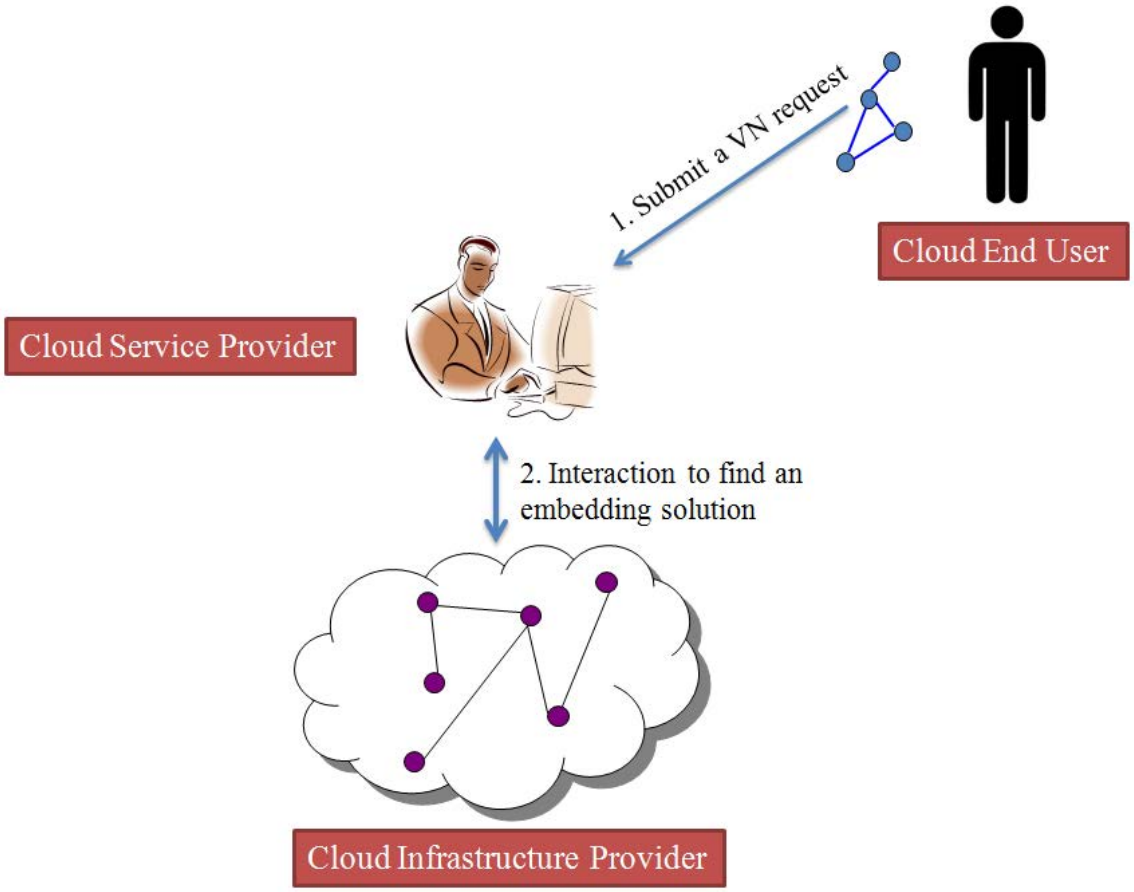


Figure 1.1: The Cloud environment actors

resources needed by the the Cloud Service Provider while meeting agreed SLA requirements. Cloud providers can be classified into *private, community, public or hybrid* according to their clients/users. *Private Clouds* are provisioned for exclusive use by a single organization/entity (e.g. university, company etc). *Community Clouds* are provisioned for a specific community of consumers (e.g. ONG, police etc.) while *public Clouds* are accessible to the general public. *Hybrid Cloud* are a composition of private, community, and public Clouds.

- **The Cloud Service Provider** is the intermediary between the Cloud provider and the Cloud End User. He negotiates, allocates and aggregates the virtual resources made available by the Cloud Infrastructure Provider. He then deploys customized mechanisms, protocols and algorithms in the allocated resources to offer end-to-end services for the Cloud End User.

- **The Cloud End User** is the actor that uses the Cloud resources. He is responsible of formulating a service request, (i.e. a Virtual Network Request in the case of NaaS) describing his needs. Once the Virtual Network is allocated, the Cloud End User can access, control and manage it. Note that even if the Cloud End User does not play a direct role in the resource provisioning process, the behavior of the workloads he generates can influence the decisions of the Cloud Infrastructure/Service Providers.

1.4 Resource provisioning in the NaaS model

Beyond the traditional Software/ Platform/ Infrastructure as a Service offers (SaaS/ PaaS/ IaaS), the NaaS has been proposed as a key technology for networking the Cloud. Networking is the ability to connect the user with Cloud services and to interconnect these services with an inter-cloud approach.

The NaaS enables customers to deploy their applications on the Cloud and to access to virtual network functions such as custom addressing, network isolation etc. Moreover, NaaS users can flexibly place their Virtual Machines (VM), they can inquire connectivity between them, and even specify the topology and characteristics of the virtual network they require (Costa *et al.* (2012)). Hence, with the NaaS model, Cloud providers offer to customers a service in the form of *a Virtual Network*. For example, a company operating video conferencing services could run on a virtual network. Likewise, a university delivering online courses for distance education may run on a virtual network.

In this context, provisioning resources in the NaaS model can be seen as a problem of Virtual Network resource Provisioning. It corresponds to mapping/embedding a virtual network, composed of a set of virtual nodes and links requiring an amount of resources (typically computing resources and memory for nodes, and bandwidth for links), into a substrate network formed of physical nodes interconnected by physical links and having limited resources, such that the VN requirements are satisfied and the used substrate resources are minimized. The allocated resources will then be released at the end of the virtual network lifetime.

But in the NaaS model, other constraints and scenarios should be investigated when allocating network resources. In fact, after satisfying the initial requirements of a VN, the cloud provider should deal with the resource requirements variations *during the VN lifetime*. Hereafter we enumerate some scenario examples of Cloud clients demand fluctuation:

- A virtual network providing office users with virtual desktop services usually experiences low-workloads at weekends, whereas another virtual network hosting online gaming services has high-workload during the weekends due to high user demands.

- A commercial gaming service needs to deal with steady and predictable increase in application traffic while maintaining a good Quality of the Service.

- A company deploying its applications on the Cloud needs to extend application delivery capabilities to new tenants (after business changes, like a merger or other event that affects the users population, for example a sudden increase of the number of users streaming a new movie, or visiting a website related to a worldwide event etc.)

When the VN user requirements vary, the VN characteristics change (in terms of topology and resource requirements). Hence, contrary to classic VNE solutions that allocate a *static* amount of resources to the VN, adaptive and dynamic techniques are needed to deal with new demands. Moreover, due to the dynamic arrival and departure of virtual networks, the Cloud infrastructure can drift into an inefficient configuration where resources are fragmented, hence strategies that continuously ensure an efficient use of the substrate resources are required.

1.5 Problem statement

In this thesis we deal with the resource requirements fluctuation during virtual networks lifetime and the efficient use of substrate resources. To describe the problem, illustrated in figure 1.2, we split the virtual network resource provisioning problem into two sub-problems: Initial VN embedding (VNE), and Dynamic Resource Management (RDM).

- **Initial VN embedding:**

The aim of this stage is to efficiently map the initial VN request onto the substrate network. First, the Cloud end user specifies his service requirements (e.g. network topology, computing resources, bandwidth, etc.) in the form of a graph with virtual nodes interconnected via virtual links, then communicates it to the service provider. Upon receiving a VN request, the Cloud Service Provider in cooperation with the Cloud Infrastructure Provider proposes a provisioning scheme for the the required VN. The Cloud Service provider identifies the

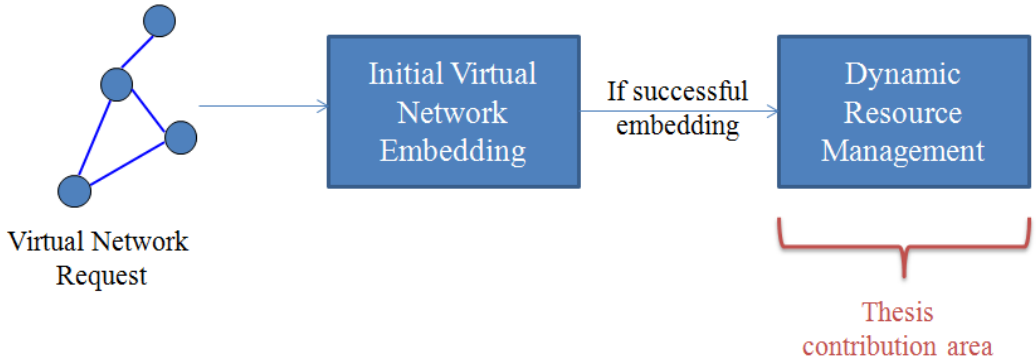


Figure 1.2: The VN resource provisioning sub-problems

best substrate resources, made available by the Infrastructure Cloud provider, that satisfy the VN request while minimizing the used substrate resources, then allocates them.

- **Dynamic resource management**:

This stage *is the focus of this thesis.* It deals with i) the resource demand fluctuation of the embedded VNs and ii) the re-optimization of the substrate network usage.

When the Cloud End User application demand vary, the VN request changes. These changes can concern *virtual nodes* and/or *virtual links*. For both, there are four general changes:

- *Increase of resource requirements of already embedded virtual node/link*
- *Decrease of resource requirements of already embedded virtual node/link*
- *Addition of a new virtual node/link to the VN topology*
- *Deletion of a virtual node/link from the VN topology*

Hence, in order to manage these demand fluctuations and deal with inefficient substrate resources use, the Cloud Service/infrastructure Providers should update the allocated resources and make mapping reconfigurations while taking into account:

- *Virtual node and link new constraints:* ensuring that new required resources and bandwidth are satisfied
- *Substrate resources limitations:* choosing the best substrate nodes and links to host new virtual elements, since substrate resources are limited.

- *Quality of Service requirements:* minimizing the service disruption and QoS degradation when migrating the allocated virtual elements.
- *Reconfiguration cost:* minimizing the amount of reallocated resources.
- *The Cloud infrastructure utilization* : maximizing the Cloud infrastructure profitability and usage.
- *Rapidity and responsiveness* : ensuring quick reaction to continuous resource requirement changes.

Note that the state of the art has extensively investigated the first stage (Fischer *et al.* (2013)) while little attention was paid to the resource management phase. Hence, in this thesis, we will focus on the latter stage. The next section summarizes the contributions of this work.

1.6 Thesis contribution

- **An overview of the virtual network resource provisioning solutions**

We will provide an overview of the most relevant virtual network resource provisioning algorithms found in the literature. As we organize this problem into two stages, we will first give a comprehensive description of the main solutions proposed for the *initial virtual network embedding problem (VNE)*, i.e. we will present the different versions of the problem definition and resolution. Second, we will give an in-depth survey of the ***substrate and virtual resource management schemes*** recently devised. We will classify the solutions in two main groups: i) Techniques dealing with the virtual network demand fluctuation, and ii) approaches enhancing the substrate network usage.

- **New reconfiguration algorithms for dynamic resources management**

To tackle the problem described above, we will first propose a solution to manage the resource demand fluctuation in *embedded virtual nodes*. Second, we will improve the previous proposal to consider *the substrate network profitability*. And finally we will deal with the bandwidth demand variation in *embedded virtual links*.

1. ***A dynamic scheme to deal with virtual nodes demand fluctuation*** (Jmila *et al.* (2014))

We will study in depth the *extension of a virtual node*, i.e. an embedded virtual node requiring more resources, when the hosting substrate node does not have enough available resources. In such situation, prior work (Sun *et al.* (2013); Zhou *et al.* (2013); Zhani *et al.* (2013)) move the virtual nodes requiring more resources to other available physical nodes. This induces a downtime or unavailability period of the service running in the migrated virtual resource, not considered. Such downtime needs to be taken into account and minimized.

In order to satisfy the extension demand while minimizing the service interruption during migration, we propose a heuristic algorithm $RSforEVN$. Its main idea is to re-allocate one or more co-located virtual nodes from the substrate node, hosting the evolving node, to free resources (or make room) for the evolving node. The virtual nodes selected for migration are those i) incurring the lowest cost and load during migration, and ii) are the most tolerant to QoS degradation. The new host is chosen with respect to a maximal allowed *downtime* during migration, and all the links associated with the selected virtual node are re-established after re-embedding.

2. ***A preventive re-configuration algorithm to enhance substrate network profitability*** (Jmila & Zeghlache (2015))

In spite of the good performance results of $RSforEVN$ (in terms of migration/ reallocation cost and convergence time), this proposal does not consider the substrate network usage/ profitability. To fill this gap, we propose a new bi-objective algorithm $Bi-RSforEVN$ that i) responds to increasing node requirements and ii) tidies up the substrate network at minimum cost and disruptions in the same time. In other terms $Bi-RSforEVN$ jointly deals with resource demand fluctuation and improves the SN profitability *in the same step*.

In more detail, like $RSforEVN$, $Bi-RSforEVN$ re-allocates virtual nodes to make room for the node requiring more resources, but the selected nodes are chosen according to their *congestion impact*. For a given node, this criterion measures the "degree of involvement" of the virtual links attached to it, in congesting their hosting substrate paths. These resources (nodes and their hanging virtual links) are then re-allocated and shifted to less saturated substrate resources to balance the load among the SN.

3. ***A self stabilizing framework for dynamic bandwidth allocation*** (Jmila *et al.* (2016))

In the third contribution, we concentrate on bandwidth demand fluctuation in virtual links. We propose a *distributed, parallel and local view* approach, based on the *Self-*

Stabilization concept. In fact, in spite of their advantages, centralized approaches are not suitable for the wide and dynamic Cloud environment, as they require a *real-time up-to-date* description of all the substrate network dynamic parameters (such as the available resources and mapping).

Our solution is a framework composed of a Controller, and *three* algorithms running locally on each substrate node to deal with *all types* of virtual links evolution: either topologically (add/delete new virtual links) or in terms of resource requirements (increase/ decrease of required bandwidth of an embedded virtual link).

Each algorithm is composed of a set of actions. The Controller is responsible of setting the actions execution scheme. Different nodes execute the algorithms *in parallel* and only a *local view* is required.

Algorithm 1 handles the case of increase of bandwidth requirements or virtual link addition (add a link to the VN topology). It manages an end-to end update among the substrate nodes of the hosting path to meet the request. Algorithm 2 is used to find the most cost-effective path, to support a new virtual link added to the topology. And finally Algorithm 3 makes a virtual link migration if needed, to handle an increase of bandwidth requirements of an embedded link.

1.7 Thesis Structure

The remainder of the this thesis is organized as follows. Chapter 2 presents the most relevant virtual network resource provisioning strategies found in the literature. Chapter 3 outlines the design of the proposed algorithm to deal with virtual node extensions. Chapter 4 presents an improvement of the previous work to deal with the substrate network profitability. Chapter 5 proposes a novel resource management framework to deal with the bandwidth demand fluctuation of embedded virtual links. Finally, chapter 6 concludes the thesis and discusses direction for future research.

Chapter 2

State of the art: Virtual Network resource provisioning

Contents

2.1 Introduction

In spite of the rapid evolution of the Cloud Computing paradigm, it is still facing many challenges (Mahmood & Hill (2011)) such as security and data confidentiality, service delivery and billing, energy and resource management etc. In this thesis we concentrate on the issue of *Dynamic resource provisioning*. In fact, given the dynamicity and impredictability of the applications running on the Cloud, most Cloud users can not estimate the amount of resources they will need in the future. To continue satisfying its clients, the Cloud provider should handle demand fluctuations during the lifetime of the allocated resources. It requires elastic resources provisioning mechanisms to adapt the mapping/embedding of supplied resources along with new demands.

To achieve such dynamic resource allocation, one of the most promising, enabling technologies is Network Virtualization (Chowdhury & Boutaba (2009)). It is the abstraction of

physical resources and their location. In more detail, computing resources (servers, applications, desktops, storage and networks) are separated from physical devices and presented as logical systems. Thanks to such abstraction, one physical resource can be shared by different logical systems in a transparent and isolated way. More specifically, thanks to network virtualization, the Cloud infrastructure can be used by various clients simultaneously. In particular, Network As A Service consumers can control *different Virtual Networks running in the same Cloud infrastructure*. Once embedded, the VN resource requirements can evolve dynamically according to the users applications demands. In this thesis, we will tackle the issue of managing such evolutions rapidly and with minimum cost.

This chapter gives an overview of the different Virtual Network resource provisioning solutions present in the literature and is organized as follows: first we will describe in detail the Network Virtualization environment. Second we will present the virtual network resource provisioning problem and describe its two sub-problems : Initial Virtual Network embedding, and Dynamic resource management. The main approaches of each sub-problem will be outlined, then a conclusion will end the chapter.

2.2 Network Virtualization

Network Virtualization (NV) was proposed as a solution to internet ossification. It provides an abstraction between computing, storage and networking hardware, and the applications running on it. Network Virtualization main merit is the ability to consolidate safely multiple networks in one physical platform. Hence, the Network Virtualization environment is made up of essentially two entities: *the Virtual Network* and *the Substrate Network*. Multiple Virtual Networks can run simultaneously over one or more substrate networks. Below, we describe in detail theses two systems.

2.2.1 Substrate Network

A Substrate Network (SN) is a physical infrastructure composed of a set of substrate nodes interconnected through substrate links. It is characterized by an amount of *limited* available resources: typically storage, CPU and memory in physical nodes, and bandwidth in substrate links, and a per unit cost of node/link resources.

A ***substrate node*** is an active electronic device, able to send, receive, or forward information to other substrate nodes. It can host one or more virtual nodes of different networks. The virtualization of a substrate node can be performed using several techniques, namely "'Operating System Virtualization"' that allows the physical node to run multiple

instances of different operating systems, hence different virtual nodes/machines can be hosted on one substrate node and use the same functionalities as a normal machine.

A ***substrate link*** is a physical medium connecting two substrate nodes. To virtualize this medium, the substrate link is split into distinct channels, and the sender and receiver are under the illusion that they own the link. Hence, the physical link can be shared by many virtual links from different virtual networks. It may host the whole virtual link or only a portion of the virtual link demand in case of path splitting (Yu *et al.* (2008)).

2.2.2 Virtual Network

Similarly to Substrate Network, a Virtual Network is composed of virtual nodes and virtual links. Its is characterized by an amount of *required* resources, that the VN user defines when formulating the VN request. This amount can change during the VN lifetime.

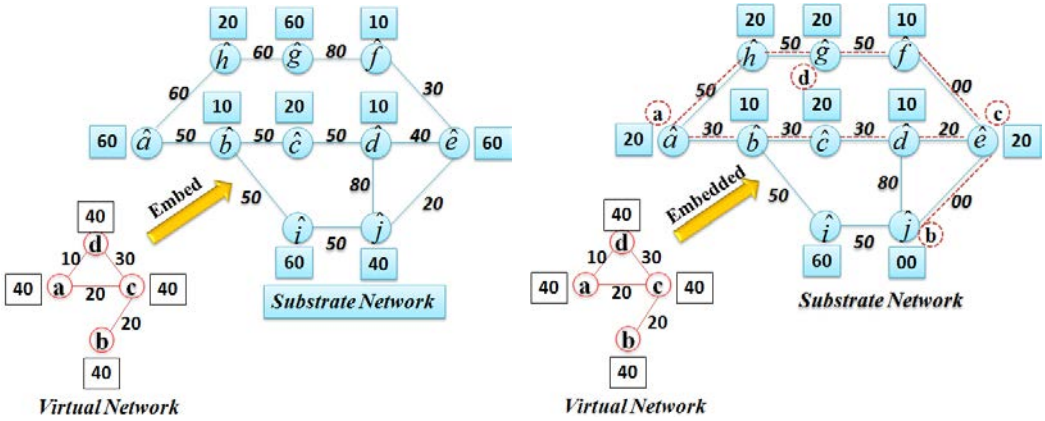
A ***virtual node*** is a software component, for example a virtual machine encapsulating CPU, memory, operating system, and network devices. In addition to the amount of required resources, other constraints can determine the virtual node *location* or *type* (router, switch) etc. A virtual node can be hosted by one and only one substrate node. Virtual nodes are interconnected through virtual links, forming the virtual network topology.

A ***virtual link*** is a logical interconnection of two virtual nodes. Usually defined by the amount of required bandwidth and the maximum allowed delay, it can span over one or more physical links that form the hosting *substrate path*.

2.2.3 Virtual Network Resource Provisioning

While many aspects of network virtualization have received attention from the research community (Wang *et al.* (2013)), a few facets remain unexplored or can be improved. Virtual Network resource provisioning is one of the areas that still require attention as it affects the physical resources utilization efficiency and the quality of service guaranties.

As stated in the first chapter, this thesis decouples the Virtual Network resource provisioning problem into two sub-problems: Initial VN embedding (VNE), and Dynamic Resource Management (DRM). Initial VN embedding provides an efficient mapping of the Virtual Network onto the Substrate Network, while Dynamic Resource Management deals with the resource demand fluctuation of the embedded VN and the re-optimization of the substrate network usage. When a VN request arrives, a VNE is attempted, if it succeeds, a dynamic resource allocation process is then initiated for the embedded VN, and continues throughout its lifetime.



(a) (b)

Figure 2.1: Initial VN embedding

Although the initial VNE is well covered in the literature, the Dynamic Resource Management is not sufficiently explored. In the following sub-sections we present the main results found in the state of the art to solve the two sub-problems of VN resource provisioning.

2.3 Virtual Network Embedding strategies

2.3.1 Initial VNE strategies

2.3.1.1 Problem formulation

The initial virtual network embedding problem consists in mapping each virtual node of the VN to one substrate node that has enough available resources, and each virtual link to one or more available substrate links connecting the source and destination virtual nodes. Other constraints may be taken into account, like the virtual path maximum delay, the geographical nodes location etc.

To explain in depth this problem, we will present a "generic" modeling of the Initial VNE problem, including a modelling for the Substrate Network, the Virtual Network, the Mapping, the Revenue and the Cost as used by most of related work:

- **The Substrate Network Model**

A substrate network is generally represented by a weighted undirected graph $G_s = (N_s, L_s)$, where N_s is the set of *substrate nodes* n_s and L_s is the set of *substrate links* l_s.

To each substrate node $n_s \in N_s$ is associated an amount of available resource capacity, denoted a_{n_s} and a per unit cost of node resource $cost(n_s)$.

Similarly, a_{l_s} denotes the *available* bandwidth on link l_s and $cost(l_s)$ is the per-unit cost of bandwidth. A variable p is used to denote a substrate path (a single or a sequence of substrate links) between two substrate nodes. P_s represents the set of loop-free substrate paths in G_s. The available bandwidth a_p associated to a substrate path p can be evaluated as the smallest available bandwidth on the links along the substrate path.

Figure 2.1(a) presents an example of a substrate network, where the numbers in rectangles next to the nodes represent the amount of available node resources at the nodes and the numbers next to the edges represent the available bandwidth in the edges.

- **The Virtual Network Model**

Like the Substrate Network, the VN request topology is represented by a weighted undirected graph $G_v = (N_v, L_v)$, where N_v is the set of required virtual nodes and L_v is the set of required virtual links. Each virtual node $n_v \in N_v$ is associated with a minimum requested capacity denoted by r_{n_v}. Each virtual link $l_v \in L_v$ is associated with a minimum required bandwidth denoted by r_{l_v}.

Figure 2.1(a) present an example of a virtual network. The numbers in rectangles next to the virtual nodes represent the amount of node resources requested by the nodes and the numbers next to the virtual edges represent the edge required bandwidth.

- **The Mapping model**

When a VN request arrives, the infrastructure provider has to perform a suitable VN embedding/mapping and allocate substrate resources to the VN such that the VN resource requirements are satisfied and the embedding cost is minimized. The allocated resources will be released when the VN request expires.

Hence, A virtual network embedding for a VN request is equivalent to the problem of finding a mapping M from G_v to G_s , with respect to the resource requirements of G_v and such that the revenue of the service provider is maximized and its embedding cost minimized.

Formally, let us decompose the Virtual Network Embedding into two sub-problems: *the node mapping problem* and *the link mapping problem*:

Node mapping: is to find a mapping $M : N_v \to N_s$, $n_v \mapsto M(n_v)$, such that

- $a_{M(n_v)} \geq r_{n_v}, \forall n_v \in N_v$ (the hosting substrate node has enough available resources)

- $M(n_v) = M(m_v)$ iff $n_v = m_v$, $\forall n_v, m_v \in N_v$ (a virtual node can be hosted by only one substrate node)

Link mapping: is to find a mapping $M : L_v \to P_s$, $l_v \mapsto M(l_v)$, such that

- $M(l_v) = p$, $\exists p \in P_s$ $\forall l_v \in L_v$(a hosting path exists)
- $a_p \geq r_{l_v}$ (and has enough available bandwidth)

Figure 2.1(b) shows an example of embedding result.

- **The Revenue Model**

The revenue of a cloud service provider when embedding a VN can be defined according to different economic models, but most of the proposals (Lu & Turner (2006); Chowdhury *et al.* (2012); Wei *et al.* (2010)) use the revenue model defined as the sum of amounts of computing and bandwidth resources requested by the VN. Formally:

$$Revenue(G_v) = \sum_{n_v \in N_v} r_{n_v} + \sum_{l_v \in L_v} r_{l_v} \tag{2.1}$$

- **The Cost Model**

When the VN is allocated, it consumes physical resources such as electricity, software and hardware etc. which incur the embedding cost. Thus, the cost of embedding a virtual network G_v is the sum of the resources allocated to/consumed by this virtual network (per unit cost). Formally:

$$Cost(G_v) = \sum_{n_v \in N_v} cost(M(n_v)) * r_{n_v} + \sum_{l_v \in L_v} \sum_{l_s \in M(l_v)} cost(l_s) * r_{l_v} \tag{2.2}$$

Finally, finding an initial embedding to a VN G_v can be formulated as:

- **Finding node and link mapping solutions** ($M : N_v \to N_s$ and $M : L_v \to P_s$, with respect to resource requirements).
- **such that** $Revenue(G_v)$ **is maximized and** $Cost(G_v)$ **minimized**

2.3.1.2 Overview of existing approaches

Solving the above problem is NP-hard, as it is related to the multi-way separator problem (Andersen (2002)). Even when all virtual nodes are mapped, mapping each virtual link to a single substrate paths is an unsplittable flow problem (Baveja & Srinivasan (2000); Kolliopoulos & Stein (1997)) which is also NP-hard. Therefore, most solutions proposed for the VNE are based on heuristics.

Moreover, by varying the defined constraints and objectives, different versions of the problem can be proposed. Depending on the scenario, diverse solutions were proposed. Hereafter we present the main categories of the approaches found in the literature.

a **Offline Vs Online**: Depending on the arrival of virtual network requests, the embedding problem can be tackled as online or offline problem. In fact, in most real situations, VNE has to be tackled **online**. In fact, as the VN requests arrive to the system dynamically, and demands are not known in advance, the VNE algorithm has to handle each VN request as it arrives without waiting for future requests. Examples of such approaches can be found in (Fajjari *et al.* (2011a); Di *et al.* (2012)). In contrary, the **offline** scenario (Lu & Turner (2006); Houidi *et al.* (2011)) assumes that all all VN requests and demands are known in advance, and the system can handle all the VNs at once. Note that the online scenario is more realistic, but more difficult to solve.

b **Multi-domain Vs Single-domain**: A VN request can be provided by a single or multiple infrastructure providers. Hence, two scenarios: single domain and multi-domain can be distinguished. Although multi-domain is more realistic, it is not well investigated in the literature. In consists in mapping a VN request over a set of substrate networks managed by different infrastructure providers, each offering a part of the virtual network. These networks are interconnected with external links and generally coordinated by a service provider that splits the request in several sub-requests and maps each of them to the most convenient SN to minimize to total embedding cost. (Houidi *et al.* (2011)) is an example of VNE across multiple infrastructure providers problem. The authors proposed an exact and heuristic virtual network graph splitting algorithms to divide the VN request among different providers.

c **Mapping coordination**: As stated above, the VNE problem can be decomposed in two sub-problems: the node mapping problem and the link mapping problem.

One alternative to handle these subs-problems is called **uncoordinated VNE** and it consists on solving each sub-problem in an isolated and independent way. In this case, the node mapping is performed *first* to provide the input for the link mapping. An example of uncoordinated mapping was proposed and evaluated in (Zhu & Ammar (2006)) where the VNE is solved in two steps. First virtual nodes are mapped in a greedy way (assign the virtual nodes with biggest demands to the substrate nodes with most available resources). Second, the virtual links are mapped using the k-shortest path algorithm (Eppstein (1999)) for increasing k.

However, the lack of coordination between the two stages might result in inefficient virtual link mapping as the solution space will be reduced after the node mapping. In fact, neighboring virtual nodes can be widely separated in the substrate network which increases the cost of mapping the virtual links connecting them.

Therefore, coordinated VNE approaches were proposed. Two versions exist: either the embedding is achieved in two coordinated stages, or performed in one stage/one shot.

In the **two stages coordinated VNE**, the node mapping is performed while taking into account the VN topology (virtual links). An example of this approach is proposed in (Chowdhury *et al.* (2012)). Authors take into account a new node constraint measuring how far a virtual node of the VN request can be from its requested substrate location. They propose that, each time a Virtual Network (VN) request is received, the substrate network graph is augmented with meta-nodes (representing virtual nodes). These meta-nodes are connected to all substrate nodes within a given distance from the requested location of the corresponding virtual node. Over this augmented graph, a relaxed Mixed Integer Programming algorithm is performed to find a node mapping solution. Thereafter, link mapping is achieved following the same solution.

In the **one stage coordinated VNE**, virtual links are mapped at the same time as virtual nodes. When the first virtual node pair is mapped, the virtual link between them is also mapped and, when a virtual node is mapped, the virtual links connecting it with already embedded virtual nodes are also mapped. An example of this variant is proposed in (Cheng *et al.* (2011)). The authors propose a new parameter of a network node (substrate or virtual) to describe its position inside the topology, this parameter measures the quality of links connection around the node. Then, inspired by the Google PageRank algorithm, nodes are ranked according to their available resources and their topological position. When such topology attributes are incorporated in

node mapping, the acceptance ratio and the link mapping efficiency are improved.

d **Splittable Vs Unsplittable link mapping**: Depending on the requirements of the substrate network, two different ways can be used for link mapping: Unsplittable link mapping and splittable link mapping. In the first case, each virtual link is mapped to one and only one substrate path. The shortest path and k-shortest path algorithms (Eppstein (1999)) can be used to solve the problem. When the substrate network supports path splitting, a virtual link can be mapped over multiple substrate paths. Each supporting a part of the virtual link requirements. This concept, introduced by (Yu *et al.* (2008)), improves the SN usage and the success rate of virtual network mapping, but can face the problem of out-of order packet arrival. In (Chowdhury *et al.* (2012); Lu & Turner (2006); Houidi *et al.* (2011)), path splitting based embedding VNE solutions are proposed using linear programming algorithms.

e **Centralized Vs Distributed**: In a **centralized VNE system**, one central entity is responsible of performing the embedding. It has a global view of the SN and takes decisions according to the up-to-date description of the available substrate resources. The majority of VNE solutions present in the literature are centralized. The advantage of such approach is that the mapping is performed while the entity is aware of the *overall SN situation*, which makes the embedding more optimal. However, it faces scalability problems in large networks, and presents a single point of failure (if the central entity fails, the entire mapping process fails). Examples of centralized VNE approaches can be found in (Fajjari *et al.* (2011a); Zhu & Ammar (2006); Razzaq & Rathore (2010); Cheng *et al.* (2011); Di *et al.* (2012)). On the contrary, in a **distributed VNE system**, multiple entities compute the embeddings. The principal advantage of such approach is scalability, but communication cost and synchronization overhead need to be minimized. (Houidi *et al.* (2008); Till Beck *et al.* (2013); Esposito *et al.* (2014)) are distributed VNE approaches.

2.3.2 Dynamic Resource Management strategies

During their lifetime, VN resource requirements can evolve according to end users fluctuating demands. Hence reserving a fixed amount of resources is inefficient to satisfy them. Moreover, the dynamic arrival and departure of VNs can drift the Substrate Network into an inefficient configuration where resources are fragmented. To cope with theses problems,

some dynamic resource management strategies where proposed. They can be classified in two main groups: i) Management of VN resource demand fluctuation and ii) Re-optimization of the SN usage. Hereafter we present the main solutions present in the literature, for each category.

2.3.2.1 Management of Virtual Networks resource demand fluctuation

Most cloud-based applications are characterized by a dynamically fluctuating workload due to the nature of the offered services and/or other external events that can influence their use (sudden increase of the number of users streaming a new movie, or visiting a website related to a worldwide event etc).

In order to provision network resources that support such applications, dynamic VN provisioning techniques are required. In fact, a continuous down or/and up- scaling of the amount of allocated resources should be guaranteed to reflect the time-varying needs of Cloud-hosted applications.

- In (Mijumbi *et al.* (2014b)), authors propose a decentralized multi-agent resource management system based on Reinforcement Learning (Sutton & Barto (1998)) to deal with demand fluctuation of embedded VNs. They model the substrate network as a decentralized system with a learning algorithm in each substrate node and substrate link. The aim is to use evaluative feedback to learn an optimal policy to deal with each resource demand fluctuation in a distributed and coordinated manner. Hence each agent (substrate node/link) dynamically adjusts the allocated resources to avoid underutilization of the substrate network. To satisfy new demands, the agent should choose an action among 9 pre-defined 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. An agent learning is evaluated using a reward function that measures link delays, packet drops and network resource utilization.

  Note that, in this proposal, a limited set of actions is allowed (increasing or deceasing the amount of allocated resources by *a fixed percentage*), this comes at a cost of efficiency, as the learning algorithm is constrained in terms of perception and action granularity. Moreover, authors assume that the VN topology does not change during its lifetime, and did not investigate the case where a new link or node is added to

the VN topology. Besides, virtual resources are always hosted in the same physical nodes/links and authors do not take advantage of moving resources to other free substrate resources.

- (Mijumbi *et al.* (2014a)) propose to improve the efficiency of the previous system by conceiving an autonomous system based on artificial neural networks (ANN) to achieve an adaptable allocation of resources to virtual networks. They first represent each substrate node and link as an ANN whose input is the network resource usage status and the output is an allocation action. Then, an error function is used to evaluate the desirability of ANN outputs, and hence perform online training of the ANN.

  For each agent (substrate node or link), a 3-layer ANN is used; i) *the input layer* consists of 3 neurons describing the a) percentage of the virtual resource demand currently allocated, b) the percentage of allocated resources currently unused, and c) the percentage of total substrate resources currently unused, ii) *the output layer* consists of one neuron representing the action that should be taken to change the resource allocation for a given virtual resource and iii) *the hidden layer* is composed of a number N of neurons, where N is an optimal number determined by experimentation.

  An error function is used to measure the deviation of an agent actual action from a target action, with the aim of encouraging high virtual resource utilization while punishing the agents for QoS degradation (packets drop for nodes, and delay for links). To do so, the degree of desirability or undesirability of an agent action is measured according to resources allocated to virtual resources, substrate resources utilization and QoS degradation.

  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. Moreover, like the previous proposal, this algorithm does not not investigate topological changes of a VN during its lifetime.

- (Mijumbi *et al.* (2015)) This work is an extension of the previous one and proposes an adaptive hybrid neurofuzzy (Nürnberger (2001)) system composed of neural networks, fuzzy systems and reinforcement learning to achieve dynamic resource allocation in

virtualized networks. The system dynamically adjusts both the substrate network usage, and the substrate network structure by adding or removing links, in form of rules. To do so, the substrate network is first modeled as a distributed system of autonomous, adaptive and cooperative agents. Then, an initial knowledge base for each agent is defined using supervised learning. To achieve this, a base with maximum possible rules was defined, then pruned using examples from a training data set. This knowledge base is continuously improved using a Reinforcement Learning evaluative feedback mechanism. Finally, authors devise a procedure for agents to cooperate and coordinate their resource allocation actions to prevent conflicting actions and share their knowledge to enhance their respective performances and ensure faster convergence of the system. We note the same criticism of ignoring the VN topological evolution.

- In (Zhou *et al.* (2013)), authors propose an incremental re-embedding scheme for evolving VNs requirements relying on the notion of physical resource migration on nodes reported in (Zhou *et al.* (2010b)) that distributes virtual resources across multiple interconnected physical resources. Considering that for each VN, re-embedding a virtual node is one kind of operation cost; their objective is to reduce the number of virtual nodes or resources that need to be re-embedded when the resources demand fluctuates.

  To do so, they select a neighboring node to provide the newly required resources then allocate bandwidth resource to provision necessary bandwidth between the two substrate nodes hosting the shared virtual node. If this is not possible, the virtual node and its hanging virtual links are re-embedded into other substrate resources greedily.

  Note that the proposed algorithm leads to increased bandwidth usage to connect the substrate nodes supporting the same virtual node, which limits the acceptance ratio of new requests. Moreover, authors do not minimize the per-node reallocation. They minimize only the *number* of reallocated nodes.

- Authors of (Sun *et al.* (2013, 2012)) addressed the problem of evolving resource requests in VN embedding and listed four VN evolution cases: i) adding new nodes and links to an ongoing VN allocation ii) deleting no longer needed resources when services end iii) releasing resources when a task requires less resources to run and finally iv)

requesting more resources when VN nodes or/and links require more resources at specific stages of an application lifetime. The Authors optimally reconfigure the evolving VNs using a Mixed Integer Problem formulation with the objective of minimizing the reconfiguration cost. Since the problem is NP-hard, they suggest heuristic algorithms to deal with each case to avoid exponential explosion.

They unfortunately consider exhaustively all mapping combinations to adapt virtual resources by evaluating the cost for each substrate node and select finally the most effective one. This strategy is not suitable for large physical networks and can not meet the *swift and rapid* adaptation required by dynamic cloud applications and services. Moreover, note that only one demand fluctuation can be handled at a given time.

- (Xu *et al.* (2014b)): Motivated by the fact that most enterprise virtual networks have periodic resource demands, authors propose a virtual network embedding algorithm that periodically explores these resource demands, if known, else predicts them using the VN requests history. For each virtual node/link, they consider i) the maximal demand, ii) the periodic demand and iii) the actual demand. The VN user pays for the *maximal* resource demand, and the VN embedding cost is defined according to the amount of *used* resources.

  The proposed algorithms aim at maximizing the Cloud service provider revenue while keeping its service cost (the used resources) minimized, such that i) the resource demands of each VN are met, *if* the periodic demand is known *else* ii) the resource violation of each VN is controlled, where the "resource violation" is defined as the ratio of the amount of violated (not met, not allocated) resources to the amount of total resource demands.

  To do so, authors first propose an embedding metric to model the dynamic workloads of substrate resource usage. In fact, they suggest that the "embedding ability" of a substrate resource in admitting a virtual resource is jointly determined by the amount of available resources and their utilization ratio, since the more the substrate resources are utilized, the higher is the risk of SLA violations they face.

  Second they propose an algorithm to embed a VN with static resource demand in two coordinated stages: the virtual nodes are first mapped into substrate nodes chosen *as closely as possible* in the substrate network, then the virtual links are mapped later with lowest cost using the shortest path algorithm.

For VNs with known periodic resource demand, graphs with different resource demands at different time slots are constructed then allocated using the first algorithm. For VNs with unknown periodic demand, each VN is initially allocated with its *maximal* resource demand, and then the embedding is reconfigured at each time interval to take into account the *predicted* requirements.

- (Zhang *et al.* (2014b,a)) devise an opportunistic resource sharing-based mapping framework to efficiently deploy virtual networks with time-varying resource demand. A work-conserving allocation algorithm is presented. This algorithm performs in two stages: i) in the global stage, the virtual nodes are placed in a first-fit fashion, and virtual links are split among multiple paths, ii) in the local stage, physical resource usage is optimized through opportunistic sharing among different resource demands.

- In (Dab *et al.* (2013)), the authors propose a dynamic resource reconfiguration approach to achieve high resource utilization and to increase the infrastructure providers revenue. The authors handle the bandwidth resources requirements of an expanding VN. They propose to adjust allocated resources in the SN according to the new users needs by reconfiguring the resource allocation in the SN.

  To achieve a cost-efficient reconfiguration, the proposal, based on Genetic metaheuristic, sequentially generate populations of reconfiguration solutions that minimize both the migration and mapping cost and then select the best solution. Note that virtual links are migrated simultaneously for better efficiency.

- In (Seddiki *et al.* (2013)), an automated controller is proposed to distribute the bandwidth among virtual networks. This controller adapts the resource allocation according to dynamic change of the workload of each VN. It uses a prediction-based approach to find the optimal configuration for virtual resources that meets QoS requirements. The system is composed of Service provider controllers and Infrastructure provider controllers. The former, composed of VN sub-controllers, periodically estimates and optimizes the VN bandwidth requirements. The latter, in turn, is responsible for allocating the available bandwidth on substrate links between multiple VNs in the aim of providing fair bandwidth allocation and avoiding bottlenecks.

- In (Blenk & Kellerer (2013)), authors address VN reconfiguration when the VN resource requirements change according to services traffic patterns. Considering predictable traffic patterns, they propose an embedding algorithm that reduces the number of link migrations to minimize the impact of reconfigurations while achieving an acceptable load balancing over substrate links.

  They unfortunately reallocate virtual nodes randomly and focus only on virtual link reallocation.

- (Fajjari *et al.* (2012)) deal with adaptive resource allocation in the Cloud backbone network. authors propose an Adaptive Virtual Network Embedding algorithm (Adaptive-VNE) based on the "divide and conquer" strategy that dynamically adapts the virtual links bandwidth allocation in order to take advantage of the unused reserved bandwidth. The algorithm divides the VN topology into many star topologies, then assigns each star using an approximation-algorithm for bottleneck problems (Hochbaum & Shmoys (1986)). The residual physical resources are taken into account when choosing the physical host. It is worth noting that a monitoring module is used to estimate an upper-bound of usage rate for each virtual link.

- In (Zhang *et al.* (2012)), a robust dynamic bandwidth allocation algorithm to periodically adjust bandwidth allocation of VNs is proposed. It consists of two stages: first, each VN uses a traffic model to forecast its traffic demand, second, a robust dynamic bandwidth allocation algorithm is applied to reassign the predicted bandwidth. The robust bandwidth allocation problem is formulated as a semi-infinite optimization problem based on path-flow model, then solved through a distributed algorithm using the Primal decomposition technique (Boyd *et al.* (2006)).

- Authors of (Wei *et al.* (2010)) present a dynamic bandwidth allocation algorithm for Virtual Networks. The issue is formulated as a Multi-Commodity Flow problem. In order to avoid bottlenecks, a traffic predictor is integrated into the MCF solver to predict traffic forecast on the most congested physical links, bandwidth is then allocated based on that prediction. Note that traffic forecast are made with the assumption of Poisson process traffic pattern. However, this assumption does not reflect necessarily the real traffic pattern in a network virtualization environment.

2.3.2.2 Management of the Substrate Network usage

Adaptation of already embedded VNs to dynamically optimize resource utilization has received little attention. The existing strategies can be classified into two main families: i) periodic and ii) reactive approaches. The first family periodically selects and re-allocates the entire or parts of the underlying VNs, but this induces high reconfiguration cost and network instability. The second family executes the re-allocation scheme only when a virtual network request is rejected thus affecting user satisfaction.

Periodic approaches

- The authors of (Zhu & Ammar (2006)) propose an online virtual network reconfiguration algorithm that operates on VNs using congested substrate resources. A periodic scheme first marks the set of VNs to re-allocate by checking the overloaded physical nodes and links. A VN making use of at least one overloaded physical node or link is marked. By comparing the ratio of the maximum link/node stress over the average link/node stress, authors determine the most imbalanced (overloaded) substrate resources. In a second stage, the algorithm reassigns the entire marked VN topology by re-running the initial embedding algorithm.

  Unfortunately, such *periodic* re-allocation is very costly and mapping again the *whole* VN topology disrupts more running services than needed because of the global rearrangements.

- Work in (Yu *et al.* (2008)) uses a periodic path migration algorithm to minimize used bandwidth to increase the VN acceptance ratio. To do so, the authors fix the node mapping of already embedded virtual networks then the initial link-mapping algorithm is performed again to find new underlying paths. The path migration is performed by either changing the splitting ratio for the existing paths or selecting new paths.

  Note that the authors do not take advantage of migrating traffic sources and sinks (i.e. virtual nodes) and unfortunately limit the reconfiguration problem to path migration.

- The authors of (He *et al.* (2008)) propose DaVinci, a periodic adaptive resource allo-

cation strategy to maximize the substrate network usage. In the Davinci architecture, each virtual network runs a distributed customized protocol, derived from optimization theory to maximize its own performance objective, while, at a larger timescale, each substrate link *periodically* adjusts bandwidth shares across virtual networks based on *local link loads.* Davinci supports i) multipath traffic management, ii) customized protocols for each traffic class and iii) separate resources at each edge router to isolate different traffic classes. For each VN, the aim of the optimization problem is to maximize its customized objective function under some capacity constraints, then, leveraging on primal decomposition technique, the authors derive the bandwidth share adaptation algorithm, performed by the substrate network. The system convergence is proved using optimization theoretic tools, (under some assumptions related to convexity of the optimization problem, timescale of adaptation and selection of some tuning metrics,) and numerical experiments evaluate the system efficiency under different scenarios.

The main criticism of this approach is that each substrate link needs to know the performance objective of *all* virtual networks to perform bandwidth allocation, which is not reasonable if the VNs do not belong to the same institution. Besides, DaVinci does not assume malicious and greedy behaviors of virtual networks when running their customized protocols, and this can harm the performance of the other virtual networks.

- The authors of (Botero & Hesselbach (2009)) studied the problem of bandwidth allocation among VNs especially when there are bottleneck substrate links. A substrate link is a bottleneck when the bandwidth required from it exceeds the bandwidth actually available. Authors propose to fairly distribute the bandwidth among competing VNs to avoid their strangulation. To do so, they define two types of connections for virtual links: restricted connections limited by the reserved bandwidth of a previous link, and unrestricted connections limited by the reserved bandwidth for the current link. To fairly distribute the bandwidth, first, restricted connections are allocated bandwidth based on their requirements and then the remaining bandwidth is distributed equally among unrestricted connections. However, we notice that the allocation scheme itself is static, and only supports virtual link allocation, hence virtual node allocation must be investigated. Besides, authors suppose that only one service class crosses a virtual link, thus, the work should be extended to support service differentiation.

- The authors of (Marquezan *et al.* (2009, 2010)) propose a distributed reallocation scheme for virtual network resources. It is an online algorithm based on self-organizing techniques that uses local view features to equalize the bandwidth and storage consumption on physical nodes by moving some virtual nodes. The main idea is to shorten the physical path embedding a virtual link that overloads at least one substrate link according to its incoming/outgoing traffic. To do this, either the source or the destination of traffic (i.e. virtual node) is moved. The proposed algorithm is divided in five stages. First, each physical node determines if there is some cut-through traffic to be eliminated by moving/receiving a virtual node. In the second and third stages, the physical neighbors exchange and analyze information about which and where these virtual nodes might be moved. The decision and reservation of the resources are made in the forth stage, and finally the virtual resources are moved. However, note that contracting a path may require a great deal of moving until the path length becomes equal to one hop. Moreover, the migration frequency of routers depends on the traffic load, which is actually unstable and correlated to the running applications.

- In (Zhou *et al.* (2010a)), the authors proposed a bandwidth allocation scheme based on game theory. The proposal is a non-cooperative game where each VN tries to maximize its utility function when sharing physical resources. The latter depends on i) the available bandwidth, ii) the congestion cost according to the assigned bandwidth and iii) the cost of resource. An iterative algorithm is used to find Nash Equilibrium. The convergence of the algorithm is shown using a very simple scenario (a physical topology containing two nodes and two links).

Reactive approaches

- Authors of (Farooq Butt *et al.* (2010)) propose a reactive reconfiguration scheme that operates when a VN request is rejected. The authors first introduced two new metrics when mapping. The first measures the likelihood of a resource of becoming bottleneck. The second denotes the saturation of a substrate resource. Based on these metrics, the reconfiguration algorithm first detects the unmapped virtual nodes and links causing the rejection of the VN request, and then moves the congested links and nodes to less critical hosts.

- (Fajjari *et al.* (2011b)) proposed a greedy approach for reallocating star components within individual VNs with the objective of freeing physical resources whenever a new VN request is rejected. The proposal first sorts the embedded virtual nodes according to a criterion that measures their suitability for migration. It takes into account i) the number of congested links in the paths embedding the virtual links attached to the virtual node, and ii) the VN residual lifetime. Second, the most suitable virtual node, as well as its hanging virtual links are migrated to other underutilized substrate resources. Note that the links re-assignment is based on the shortest path algorithm, where a path length is defined according to its saturation degree. Thereafter, the algorithm tries to map again the rejected VN. The next iteration of the algorithm migrates the following virtual node and the process is repeated until the VNR is mapped or until a predefined number of iterations is reached.

- Authors of (Tran *et al.* (2012); Tran & Timm-Giel (2013)) propose a reconfiguration strategy that takes into account the cost incurred by the service disruption during re-allocation. The proposal is a reactive mechanism, which reacts to any rejection of a VN request. Indeed, the algorithm reconfigures the currently-mapped networks to free physical resources to embedd the new request. The reconfiguration also minimizes the number of necessary changes in order to reduce the service disruption. The mechanism was mathematically formulated as a Linear Programming problem, which minimizes the number of affected virtual nodes and links during reconfiguration while guaranteeing that the new VN request can be cost effectively mapped. A heuristic to pre-select the VNs involved in the reconfiguration process was also introduced.

2.4 Conclusion

In this thesis, we concentrate on resource demand fluctuation on nodes and links.

First, we focus on virtual nodes, of already embedded VNs, requiring more resources. Compared to most previously cited approaches who ignore the service interruption during virtual nodes migration, the latter is taken into account and minimized. Moreover, unlike (Zhou *et al.* (2013)), both the per-node reallocation cost and the number of reallocated nodes are reduced. On the other hand, contrary to (Mijumbi *et al.* (2014b,a, 2015)) who

does not take advantage of moving virtual resources to other available locations, we re-allocate convenient virtual nodes to make room on saturated hosts.

Many resource management algorithms concentrate on the bandwidth demand fluctuation problem. Compared to (Sun *et al.* (2013); Xu *et al.* (2014a); Zhou *et al.* (2013); Fajjari *et al.* (2012)), our proposal is distributed and relies on a local view, more suitable for the large and dynamic Cloud environment. Moreover, we consider the VN topological changes, ignored by (Mijumbi *et al.* (2014b,a, 2015)). Hence, our framework is the first *decentralized* approach that deals with *all forms* of bandwidth demand changes. Besides, our system does not require a learning phase to initialize the decision making process as designed in (Mijumbi *et al.* (2014b)). Finally, we deal with online bandwidth variations, unlike (Blenk & Kellerer (2013); Zhang *et al.* (2012); Seddiki *et al.* (2013)), who try to predict the workload.

None of all previously cited work was concerned jointly by efficient SN utilization and satisfying new resource requirements. We fill this gap by combining the two objectives. We adapt resource allocation at minimum cost to meet new demands of already embedded virtual nodes, respect quality of service of all running applications and *simultaneously* maximize utilization by balancing the load on the SN links. In other terms, unlike the periodic and reactive approaches that lead to network instability and service disruption of reconfigured VNs, we propose a *preventive* solution to "tidy up " the SN *when* responding to fluctuating resource requirements at minimum cost and disruptions.

Chapter 3

Virtual Networks Adaptation: Node Reallocation

Contents

3.1 Introduction

This chapter addresses the dynamic re-allocation of virtual nodes to support cloud services according to varying applications and user resource requirements. More specifically we focus on virtual nodes of already embedded VNs *when more resources are required from the hosting physical machine or node.* The need for more resources may have multiple reasons such as increasing applications requirements, the need to maintain quality of service of multiple services sharing the same physical nodes, etc.

Reacting to these dynamic changes and growing needs may require allocation of additional resources from the hosts themselves when feasible, or the reallocation and optimal reshuffling of virtual resources across physical nodes or hosts. When hosts do not have enough resources, prior work on Virtual Networks Embedding (Zhu & Ammar (2006); Sun *et al.* (2013)), move the virtual nodes requiring more resources to other physical nodes to maintain the service. This *affects the active application or service* running in the virtual resource. The service will experience a downtime or unavailability period that needs to be taken into account and minimized (Kapil *et al.* (2013)). In real situations, VN users often impose Service Level Agreements with penalties for service disruptions caused by migration (e.g. penalty imposed to Amazon EC2 for violating VM availability SLA, Zhani *et al.* (2013)). Avoiding such disruptions and penalties are essential. The migration of the virtual resource will also induce load on the physical network links proportionally to the size of the migrated virtual resource.

In order to minimize these impacts, we propose to select the virtual nodes in the affected physical node that will incur the lowest cost and load during migration. Virtual resources that are intuitively candidates for such migration are those that are tolerant to disruptions and/or are of small size since the migration will be faster and will induce less load. When making migration decisions, the selected virtual resource connectivity has to be taken into account since it has to be maintained, actually all the links associated with the selected virtual resource have to be re-established.

This chapter is organized as follows: the next section describes and formulates the problem. Section 3.3 presents our proposed heuristic algorithm to achieve minimum cost and service interruption when additional resources are required. Performance evaluation of the proposed heuristic algorithm is presented and compared to prior art in section 3.4. Finally, section 3.5 concludes the chapter.

3.2 Problem formulation

This section presents a mathematical model to allocate additional resources to active VNs hosted by shared infrastructures (or SNs). Fulfilling the requests for more resources can be achieved by moving, out of the physical host, only the concerned virtual nodes themselves or by migrating other virtual nodes to other hosts. The goal is to derive from the model an objective function that will realize the re-allocation of virtual nodes at minimum overall adaptation cost. Remapping and migration costs, downtime and optimization performance need to be taken into account in the derivation.

3.2.1 Network Model

The cloud infrastructure (referred in this thesis as substrate network) can be represented by a weighted undirected graph $G_s = (N_s, L_s)$, where N_s is the set of *substrate nodes* n_s (e.g. physical servers) and L_s is the set of *substrate links* l_s (e.g. data center links). G_s is used to represent the substrate.

Let $a^t_{n_s}$ denote the *available* capacity of node n_s (typically CPU and memory) and $a^t_{l_s}$ denote the *available* bandwidth on link l_s at time t. Variable φ is used to denote a substrate path (a single or a sequence of substrate links) between two substrate nodes. Parameter P_φ represents the set of substrate paths. The available bandwidth a_φ associated to a substrate path φ can be evaluated as the smallest available bandwidth on the links along the substrate path.

Table 3.1: Summary of SN key notations

Notation	Description
G_s	Substrate Network
N_s	Set of substrate nodes n_s
L_s	Set of substrate links l_s
$a^t_{n_s}$	Available capacity of substrate node n_s at time t
$a^t_{l_s}$	Available bandwidth on substrate link l_s at time t
P_φ	Set of loop-free substrate paths φ
a_φ	Available bandwidth associated to a substrate path φ

3.2.2 VN resource Request Model

This section models the user expressed VN requests (supporting cloud services) that are sent to cloud providers. A VN request is a set of *virtual nodes* interconnected via *virtual links*. The VN request topology is represented by a weighted undirected graph $G_v = (N_v, L_v)$, where N_v is the set of required virtual nodes and L_v is the set of required virtual links. Each virtual node $n_v \in N_v$ is associated with a minimum required capacity denoted by $b^t_{n_v}$. Each virtual link $l_v \in L_v$ is associated with a minimum required bandwidth denoted by $b^t_{l_v}$.

The set of active VNs on G_s at time t is defined as VN^t and the evolving node (requiring more resources) of the virtual network i is represented by m^i_v with $i \in VN^t$ and with a new resource requirement $b^{t+1}_{m^i_v} > b^t_{m^i_v}$.

3.2.3 Mapping Model

For each VN request G^r_v in the substrate network, let $M^t_{N^r}$ resp. $M^t_{L^r}$) describe the node mapping resp. the link mapping of G^r_v in the substrate network at time t, such that resource

Table 3.2: Summary of VN key notations

Notation	Description
G_v^r	Virtual Network r of VN^t
N_v^r	Set of virtual nodes n_v^r of VN G_v^r
L_v^r	Set of virtual links l_v^r of VN G_v^r
$b_{n_v^r}^t$	Minimum required capacity of virtual node n_v^r
$b_{l_v^r}^t$	Minimum required bandwidth on virtual link l_v^r

constraints are respected. More precisely, $M_{N^r}^t : N^r \rightarrow N_s$ describes the node mapping and $M_{L^r}^t : L^r \rightarrow P_\varphi$ describes the link mapping.

Table 3.3: Summary of the mapping model key notations

Notation	Description
$M_{N_v^r}^t : N_v^r \rightarrow N_s$	Node mapping related to VN G_v^r
$M_{L_v^r}^t : L_v^r \rightarrow P_\varphi$	Link mapping related to VN G_v^r

3.2.4 Problem formulation

- ***A. Reallocation strategy*** When an evolving node m_v^i requiring additional resources and a substrate host h with $M_{N_v^r}^t(m_v^i) = h$ has insufficient resources a strategy for re-allocation of resources is needed to maintain the service. This may require a migration of the virtual node or other nodes in the host. A trivial and suboptimal strategy is to move the evolving node to another less loaded host. A more elaborate strategy should take into account multiple criteria such as migration and re-mapping costs. We accordingly adopt a strategy where we reorganize and redistribute virtual nodes in the initial host and its neighbors while minimizing overall re-allocation cost. Intuitively, the nodes inducing the smallest migration cost and disruptions should be selected in priority to find a good solution.

- ***B. Optimization objective*** With this strategy in mind, we implement the virtual node re-allocation in two phases: *Re-mapping and Migration*. The *Re-mapping (remap)* phase consists in finding alternative substrate resources to host the reallocated components. The virtual node would be remapped onto another substrate node found to have enough available resources. The links associated to the original (or source) virtual node will be also remapped to restore connectivity with the new hosting (destination) node. Secondly, the Migration phase (*migrate*) will move tasks or jobs previously running on the source virtual node onto the selected destination

virtual node to resume tasks. Moving tasks requires the establishment of a temporary connection between the old and new hosts to support task migration. This induces a transfer cost that we take into account in the reallocation cost assessment. The resource re-allocation incurs both a re-mapping cost $Cost_{remap}$ and a migration cost $Cost_{mig}$.

– **Re-mapping cost**: Similar to previous work in (Sun *et al.* (2013)), the mapping/re-mapping cost of a VN request is equal to the sum of the costs of allocating/re-allocating its virtual nodes and links from the data center or infrastructure resources (physical nodes and substrate paths). Let $cost\,(n_s)$ (and $cost\,(l_s)$) be the cost unit of substrate node (and substrate link) respectively.

Let $n_v^r \in N_v^r, r \in VN^t$ denote a virtual node selected to be re-allocated related to request r and let $\mathbb{S}_{n_v^r}$ represent the star topology formed by n_v^r and its connected virtual links. We define the cost of re-mapping n_v^r as the sum of total substrate resources reallocated to the node n_v^r and its attached virtual links. Formally:

$$\begin{aligned} Cost_{remap}\,(n_v^r) = b_{n_v^r}^{t+1} * cost\left(M_{N_v^r}^{t+1}\,(n^r)\right) \\ + \sum_{l_v^r \in \mathbb{S}_{n_v^r}} \sum_{l_s \in M_{L_v^r}^{t+1}(l_v^r)} b_{l_v^r}^{t+1} * cost(l_s) \end{aligned} \tag{3.1}$$

Where $b_{n_v^r}^{t+1}$ is the new resource demand of node n_v^r, $(M_{N_v^r}^{t+1}, M_{L_v^r}^{t+1})$ describes the mapping of re-allocated elements, and $cost\left(M_{N_v^r}^{t+1}\,(n^r)\right)$ is the cost of new mapping.

∗ **Migration Cost**: During the migration step, migrated tasks experience a downtime that depends on *i*) the migration technique Kapil *et al.* (2013), *ii*) the size of the migrated task and *iii*) the bandwidth allocated for task migration. Migration is a topic on its own that is beyond the scope of this thesis.

For our study, we consider that the downtime depends primarily on the size of the migrated task and the bandwidth available during the migration.

In our model, a maximum downtime for each virtual node n_v^r, $downtime_r$, is imposed by each VN end-user. To respect this condition, sufficient resources should be allocated from the target host depending on the size of the task to

migrate. Formally, we define $minBW_{n_v^r}$ as the minimum required bandwidth to migrate a virtual node n_v^r :

$$minBW_{n_v^r} = \frac{b_{n_v^r}^{t+1}}{downtime_r} \tag{3.2}$$

Where $b_{n_v^r}^{t+1}$ is the size of the re-allocated virtual node. The cost of task migration $cost_{mig}\left(n_v^r\right)$ is the sum of all resources allocated (needed) for migration.

Formally, if $p_{mig(n_v^r)} \in P_\varphi$ denotes the substrate path used for migrating the node n_v^r, the migration cost is defined as:

$$cost_{mig}\left(n_v^r\right) = \sum_{l_s \in p_{mig(n_v^r)}} minBW_{n_v^r} * cost(l_s) \tag{3.3}$$

* **Reallocation cost** Finally, the reallocation cost of a virtual node is the sum of its re-mapping cost and its migration cost:

$$Cost_{realloc}\left(n_v^r\right) = Cost_{remap}\left(n_v^r\right) + Cost_{mig}\left(n_v^r\right) \tag{3.4}$$

To satisfy the demand of an evolving node m_v^i for additional resources, the re-allocation of more than one virtual node may be required. The global re-allocation cost $RealloCost_{m_v^i}$ related to an evolving node m_v^i is the sum of all re-allocation costs:

$$RealloCost_{m_v^i} = \sum_{n_v^r \ isReallocated} Cost_{realloc}\left(n_v^r\right) \tag{3.5}$$

Our objective is to find the best re-allocation scheme in order to satisfy the evolving node additional resource request while minimizing all re-allocation costs. This leads to the following objective function:

Objective function:

$$minimize(RealloCost_{m_v^i}) \tag{3.6}$$

Table 3.4: Summary of the Cost model key notations

Notation	Description
$Cost_{remap}(n_v^r)$	Cost of re-mapping a virtual node n_v^r
$Cost_{mig}(n_v^r)$	Cost of migrating a virtual node n_v^r
$Cost_{realloc}(n_v^r)$	Cost of re-allocating a virtual node n_v^r
$ReralloCost_{m_v^i}$	Total re-allocation cost of an evolving virtual node m_v^i
$downtime_r$	Maximum downtime imposed for n_v^r
$minBW_{n_v^r}$	Minimum required bandwidth to migrate n_v^r
$p_{mig(n_v^r)}$	The substrate path used for migrating the node n_v^r

3.3 Heuristic algorithm design

Finding the optimal re-allocation for an evolving node while minimizing cost is NP-Hard (Baveja & Srinivasan (2000)). We resort to a heuristic algorithm called RSforEVN (*Re-allocation Scheme for Evolving Virtual Node request*) to reduce complexity, improve convergence times and to provide a scalable solution. The heuristic algorithm must decide which virtual nodes to reallocate and where to move them. This reorganization should incur minimum overall reallocation and migration cost.

- The heuristic algorithm proceeds in two optimization steps. It first finds the best set of virtual nodes to reallocate and migrate to free resources for the benefit of the evolving node (unless the best solution is to move the evolving node itself, in which case the objective is to find a new host for it). In this step, the heuristic algorithm selects the minimum number of less constraining virtual nodes (typically of small sizes and most tolerant to disruptions and QoS degradations).

- The second step consists of finding the best destination or target hosts for the selected virtual nodes. The heuristic algorithm will have to map efficiently nodes and links to meet the minimum reallocation and migration cost objective. The two steps are described in more detail in the sequel

3.3.1 First step: Selection of virtual nodes for reallocation

We use the following notations to describe the selection process: m_v^i identifies the evolving node asking for additional resources and $coloc_h^t$ the set of all virtual nodes hosted in the same physical node h as m_v^i (i.e. $M_{N_v^r}^t(m_v^i) = h$). The heuristic algorithm main idea is to re-allocate one or more co-located virtual nodes from the substrate node, hosting the evolving node, to free resources (or make room) for the evolving node (needing additional resources).

The virtual nodes are selected according to their size and QoS requirements. The size of a virtual node includes its intrinsic size and the aggregate bandwidth of its associated links. The QoS corresponds to the maximum acceptable downtime of the virtual node during migration:

$$Reem(n_v^r) = (b_{m_v^r}^t + \sum_{l_v^r \in \mathbb{S}_{n_v^r}} b_{l_v^r}^t) * downtime_r \tag{3.7}$$

Hence, the *Reem* expression is the product of two terms: the first term represents the "size" of the virtual node, whereas the second one is related to QoS requirements. The purpose behind considering the ranking criterion $Reem\,(n_v^r)$ is twofold.

- First favor reallocation of candidate virtual nodes and their attached links that require the smallest amount of resources to minimize re-mapping cost (3.1).

- Second re-allocate the smaller and more QoS degradation tolerant nodes to optimize the migration cost (3.3) since the amount of bandwidth required to perform task migration will be minimized.

As a result of this ranking, all virtual nodes in $coloc_h^t$ are sorted in a list $\vec{coloc}_h^t$ in increasing order of their *Reem* value.

3.3.2 Second Step: finding the best new physical hosts

The next step consists in re-allocating virtual nodes that have the lowest *Reem* values. Thus, one or more virtual nodes from the ranked list $\vec{coloc}_h^t$ should be re-allocated with their associated virtual links. The number of virtual nodes to re-allocate is dictated by the amount of requested additional resources by the evolving nodes. The sum of resources to free by migrating virtual nodes should be equal or greater than the amount of required additional resources for the evolving node. Virtual nodes will not be migrated if there are enough resources in the original physical node since the evolving node would receive additional resource directly from its host.

When remaining resources are insufficient, co-located virtual nodes will be migrated to free the needed resources for the evolving node. If virtual nodes can not be migrated for QoS reasons, the evolving node will be moved if possible otherwise the request is rejected. In fact, this will be the case each time all virtual nodes ranked ahead of the evolving node in $\vec{coloc}_h^t$ vector can not offer enough resources to satisfy the evolving node.

As presented in figure 3.1 our proposed algorithm takes into account two special cases:

1. If the amount of resources that could be freed after multiple re-allocations is not sufficient to satisfy the request, our algorithm tries to re-allocate the evolving node. If the re-allocation succeeds, the request is satisfied, otherwise it is rejected.

2. Else, our proposal remaps the first node in the ranked list. If it succeeds, the algorithm verifies if the resources released after this re-allocation are sufficient to satisfy m_v^i new demand, if it is the case, the elasticity request is satisfied. Otherwise, the next node is selected and the process is repeated until the elasticity request is satisfied or the evolving node is re-allocated, as long as $\vec{coloc}_h^t$ is not empty (The do-while loop in figure 3.1.

3.3.3 Virtual node reallocation scheme

After selecting virtual nodes for reallocation, the algorithm has to find the optimal nodes to host these selected virtual nodes and restore their connectivity with all their peers (previous neighbors) by finding new substrate paths to restore all the broken links. To reallocate a virtual node n_v^r, the star $\mathbb{S}_{n_v^r}$ (the node and its links) should be re-mapped, and task migration should be performed. To find the best new substrate hosts, our heuristic algorithm explores the nearest neighbors of the initial host h to find nodes that have enough resources and can reconstruct all the links associated with each virtual node candidate to migration. Links must also be established to ensure migration respecting the downtime constraints of each virtual node.

If $near_h^t$ is the set of potential (candidate) hosts for the re-allocated node, this neighbor set $near_h^t$ has to minimize migration cost (3.3). The shortest path algorithm is used to find the optimal substrate paths.

The Virtual node reallocation scheme is listed below

3.4 Simulation results and evaluation

We compare our algorithm with relevant prior art to assess performance with a focus on re-mapping and migration costs, execution time (or convergence time) and the acceptance rate of requests for additional resources. We also describe the settings, conditions and scenarios used to conduct the evaluation.

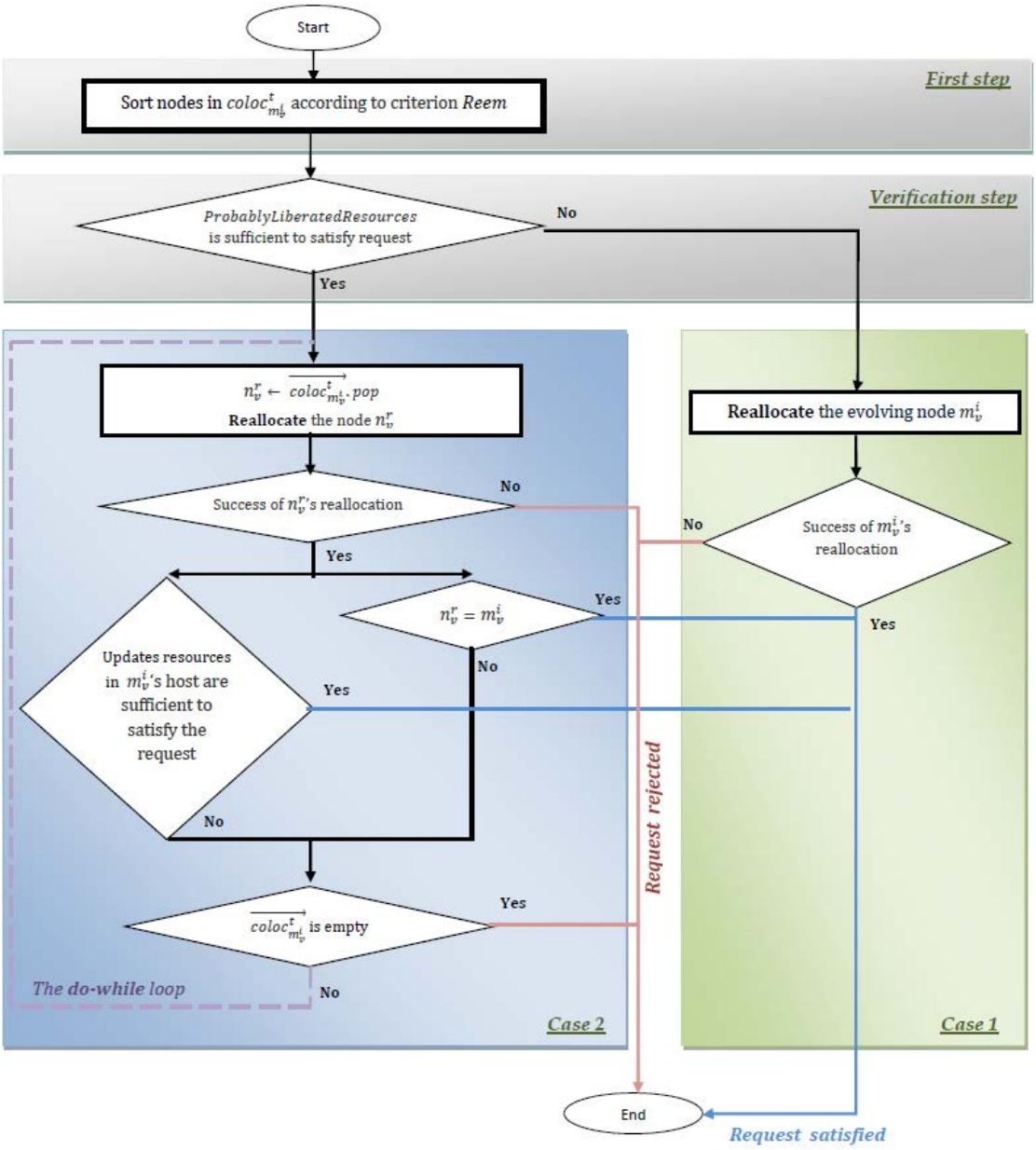


Figure 3.1: RSforEVN: Main Algorithm steps

3.4.1 Simulation environment

The GT-ITM (Zegura *et al.* (1996)) tool is used to generate random topologies of the substrate and VN networks. Similar parameter settings and simulation conditions to existing work was adopted to be able to compare in equivalent scenarios the performance of our algorithm (Chowdhury *et al.* (2012); Fajjari *et al.* (2011b)).

The SN (Substrate Network) size is set to 50 nodes and each pair of substrate nodes is randomly connected with probability 0.5 (a realistic value for typical deployed and operational networks, since they are seldom fully meshed and often have connectivity below

Algorithm 1 Node reallocation scheme

One Node Reallocation steps
Reallocate($n_v^r, RealoCost_{m_v^i}$)
$ReallocationResult \leftarrow failure$
$remapCost^{best} \leftarrow \infty$
Search $near_{n_v^r}^t$
if $near_{n_v^r}^t$ is not empty **then**
  for all $n_s \in near_{n_v^r}^t$ **do**
    map n_v^r in n_s
    for all $l_v^r \in \mathbb{S}_{n_v^r}$ **do**
      re-map virtual link l_v^r onto a substrate path φ using shortest path algorithm
    end for
    if $\mathbb{S}_{n_v^r}$ mapping succeeds **then**
      $ReallocationResult \leftarrow success$
      if $remapCost\left(\mathbb{S}_{n_v^r}\right) < remapCost^{best}$ **then**
        $remapCost^{best} \leftarrow remapCost(\mathbb{S}_{n_v^r})$
      end if
    end if
  end for
  if $ReallocationResult = Success$ **then**
    Add $cost_{mig}\left(n_v^r\right) + cost_{remap}\left(n_v^r\right)$ to $RealoCost_{m_v^i}$
  end if
end if
return $ReallocationResult$

50%).The node resource capacity and edge resource capacity are real numbers uniformly distributed between 0 and 50 in order to span reasonably the search space without making any specific assumption on the statistical characteristic of this parameter. Without loss of generality, we set the per unit node and edge resources costs to 1 (one)unit.

The requested VNs have between 2 and 10 virtual nodes in their topologies with an average connectivity also set to 50%. The node resource capacity is uniformly distributed between 0 and 20 and the edge resource capacity is uniformly distributed 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 or mimic the state of a SN subject to multiple virtual nodes evolutions.

To create a highly dynamic environment and unpredictable states or situations, we select randomly N virtual nodes among those hosted by the SN as nodes that require additional resources. The increasing resource requests are measured using the parameter "Increase Factor" (IF):

$$b_{m_v^i}^{t+1} = IF * \ b_{m_v^i}^t \tag{3.8}$$

Where $b_{m_v^i}^{t+1}$ is the new resource requirement of the evolving node m_v^i.

3.4.2 Simulation results

Only (Mijumbi *et al.* (2014b, 2015, 2014a); Sun *et al.* (2013); Zhou *et al.* (2013); Blenk & Kellerer (2013)) deal with the problem of evolving virtual nodes. Since the objective functions in (Mijumbi *et al.* (2014b, 2015, 2014a); Zhou *et al.* (2013); Blenk & Kellerer (2013)) differ and are not sufficiently close to our proposed algorithm, we do not retain them for performance comparison.

Authors of (Mijumbi *et al.* (2014b, 2015, 2014a)) do not take advantage of moving virtual resources to other available substrate hosts. Moreover, in (Zhou *et al.* (2013)) authors minimize the number of re-allocated virtual nodes, while in (Blenk & Kellerer (2013)) authors minimize the number of virtual link reconfigurations after a VN evolves.

The authors of (Sun *et al.* (2013)) considered the same objective function as that of our proposal and it is more relevant and appropriate to compare performance with their algorithm named DVNMA_NS.

The algorithms compared in our simulations are listed in the table below.

Table 3.5: Compared algorithms

Notation	Algorithm description
RSforEVN	Makes a convenient choice of virtual nodes to re-allocate and selects the most cost effective new host among nearest neighbors
DVNMA_NS	Systematically re-allocates the evolving node, and selects the most cost effective new host among *all* substrate nodes

In the simulations, the following performance metrics are used:

1. **Re-allocation Cost**, that reports *RealloCost* of all evolving nodes if their new demands are successfully satisfied.

2. **Migration Cost** measuring the amount of resource (bandwidth) required to achieve task migrations to fulfill the evolving node requests. This corresponds to the sum of all $cost_{mig}$ of re-allocated nodes.

3. **Acceptance ratio of elasticity requests** that measures the percentage of accepted additional resources requests for evolving nodes

4. **Total execution time (or convergence time)** that measure the algorithms convergence time to assess how fast the algorithms find a solution to fulfill the additional

resource requests.

All reported results are obtained by averaging the collected performance from 100 independent runs for each simulation point.

3.4.2.1 Re-allocation cost for large size evolving virtual nodes

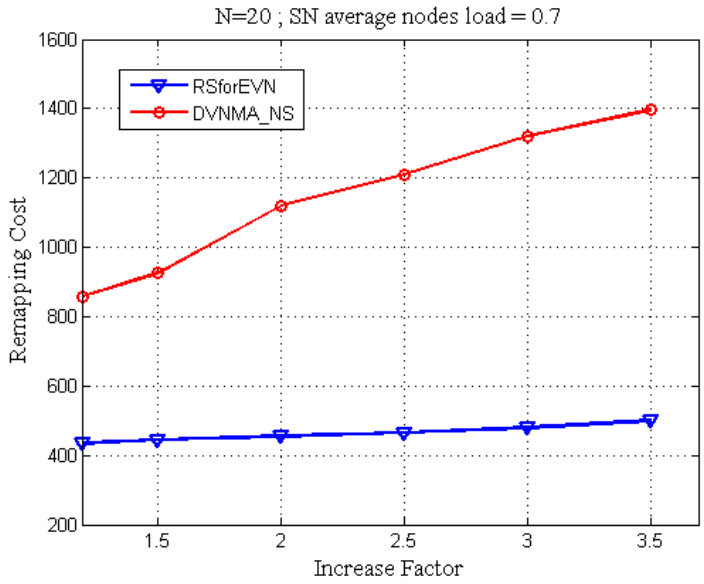


Figure 3.2: Reallocation cost

The first simulation assesses the re-allocation cost of our algorithm for evolving virtual nodes of large sizes (Equation 4.3). To produce scenarios with large virtual nodes instances to re-allocate, 20 virtual nodes are selected randomly from the top 100 largest virtual nodes currently hosted in the SN among a total of 214 nodes. The reallocation cost is measured for variable Increase Factors, representing the amount of additional resources that will be required by the 20 selected virtual nodes.

Figure 3.2 depicts the results of 100 averaged runs and indicates that our algorithm (RSforEVN) outperforms the DVNMA_NS algorithm in terms of re-allocation cost by 50%. Our algorithm reduces the re-allocation cost by selecting primarily small virtual nodes as candidates before resorting to re-mapping virtual nodes of large sizes.

This also makes our algorithm less sensitive and more robust to increasing IF values while DVNMA_NS re-allocation cost increases significantly for increasing IF values. The RSforEVN algorithm always selects the smallest virtual nodes first as opposed to the DVNMA_NS always re-allocates the evolving nodes themselves and this induces high re-mapping costs when the evolving nodes are of large size.

3.4.2.2 Migration cost

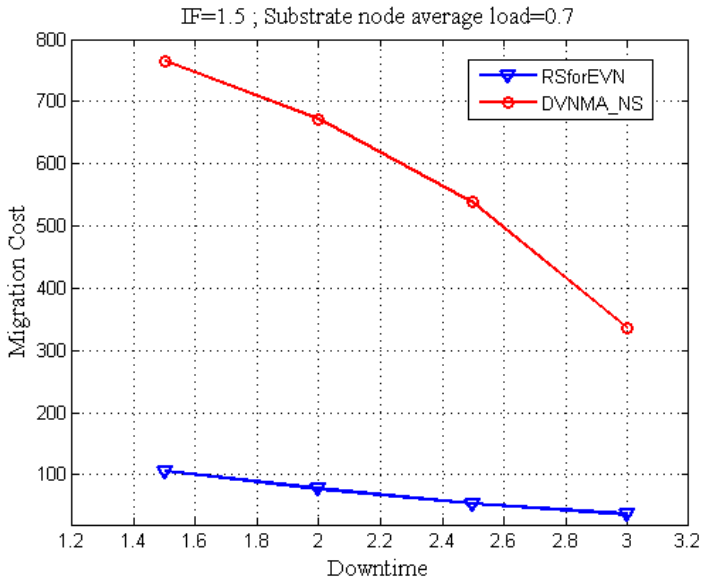


Figure 3.3: Reallocation cost

As depicted in 3.3, our algorithm, RSforEVN, performs also much better than DVMA_NS, in migration cost as a function of downtime tolerance of the virtual nodes. Without loss of generality, we assumed that all virtual nodes have the same downtime in the simulations. This is again due to the small virtual nodes selected by RSforEVN since these smaller nodes require less bandwidth for migration according to the downtime constraint.

In addition, RSforEVN selects the nearest neighbors to the substrate node hosting the evolving nodes that require more resources whereas DVNMA_NS searches for the best new hosting node in the entire substrate network and has to do so for the evolving nodes inducing high penalty and even higher cost if the evolving nodes are large. Once the best node is found, DVNMA_NS *deduces* the migration substrate path using the shortest path algorithm.

Migration cost increases for both algorithms when the downtime migration constraints become tighter as more link resources (bandwidth) are needed (is needed) to achieve faster migration.

3.4.2.3 Elasticity Request Acceptance ratio benefits for saturated SN

The next set of simulations address the performance of the algorithms with respect to the acceptance ratio of requests for additional resources and their speed in finding solutions (or execution/convergence time) to fulfill such requests for more resources. The evaluation is conducted for several scenarios as a function of the number of involved evolving nodes, the

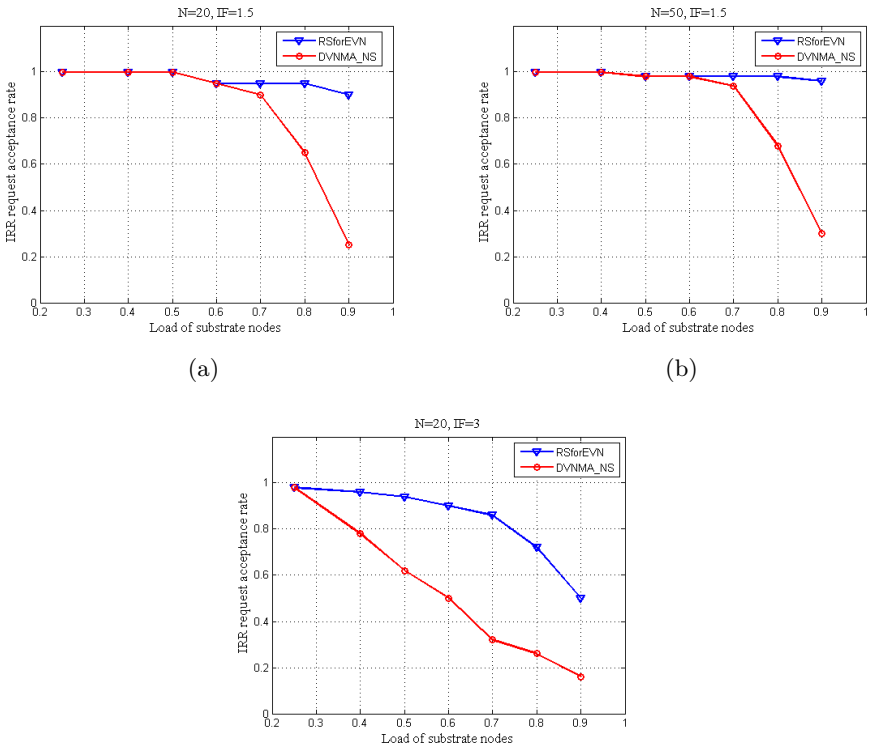


Figure 3.4: Acceptance ratio

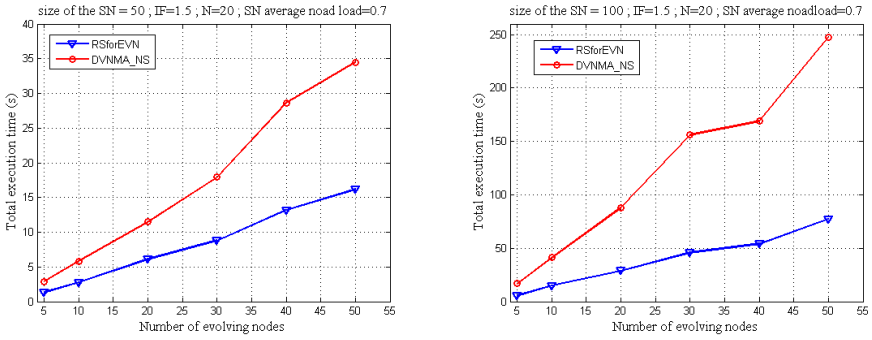
Increase factor that measures the amount of requested additional resources and the load in the substrate network or the SNs.

Figures 3.4(a) and 3.4(b) show close performance in percentage of accepted requests for both algorithms when the substrate network is not heavily loaded. However, when the substrate network is saturated our algorithm accepts 3 times more requests than DVNMA_NS that has difficulty in finding hosts available for large evolving virtual nodes. RSforEVN that moves smaller virtual nodes can instead find more easily some space available in new hosts for these small resource requests.

Figure 3.4(c) confirms that RSforEVN outperforms DVNMA_NS when the required amount of additional resources increases with the RSforEVN algorithm resisting much better to the increased stress for $IF = 3$ compared to $IF = 1.5$ (looking at Fig 2.c and Fig 2.d joinlty).

The acceptance rate for RSforEVN degrades smoothly while that of DVNMA_NS is more significant and rather abrupt. RSforEVN performs consistently better for overloaded substrate networks.

3.4.2.4 Reduced execution time, especially for large Substrate Networks



(a) (b)

Figure 3.5: Execution time

The convergence time of the algorithm also matters in terms of swift response to additional resources requests since some applications require elasticity services and high availability and can thus put very stringent requirements on extended resource allocations.

Figure 3.5(a) and 3.5(b) present the collected required time to find a solution for the resource requests for both algorithms and depict better performance in convergence time for the RSforEVN algorithm that finds solutions 2 to 3 times faster for the simulated scenarios with increasing number of substrate and evolving nodes. Figures 3.5(a) and 3.5(b) corresponding to $SN = 50$ and $SN = 100$ respectively for involved virtual nodes ranging from 5 to 50 nodes. This gap in speed performance for DVNMA_NS is expected as it searches for new hosting nodes amongst all substrate nodes while SFforEVN searches only in the vicinity or neighborhood of the host currently hosting the evolving nodes.

RSforEVN does in addition favor migration of smaller virtual nodes. In fact when analyzing all the performance results for the simulated scenarios and settings, RSforEVN performs consistently better and provides the best trade-offs in reallocation cost, migration cost, downtime and speed of convergence.

3.5 Conclusion

This chapter addressed the allocation of additional resources for extending virtual nodes and proposes an algorithm that offers the best trade-off in terms of re-mapping and migration costs, service downtime and convergence speed when compared to prior art. The performance of the proposed algorithm, RSforEVN, is compared to the DVNMA_NS and shown to be consistently superior in all reported performance metrics.

In the next chapter, we will improve RSforEVN to solve a bi-objective problem : i)

meeting evolving virtual nodes demands and *ii*) increasing the substrate resources profitability.

Chapter 4

Load balancing aware Virtual Networks adaptation

Contents

4.1 Introduction

In this chapter, we extend and enhance the algorithm proposed in chapter 3 to adapt dynamically virtual networks to additional resource requirements *while* balancing load and avoiding fragmentation in the substrate network. In fact, since VN requests arrive and depart over time, the SN can quickly drift to an inefficient configuration, where resources are progressively fragmented, leading to more VN request rejections.

To avoid such configuration, we propose to tidy up the SN when responding to fluctuating (increasing) VN resources requirements at minimum cost and disruptions. In more detail, we adapt resource allocations at minimum cost, respect quality of service of all running applications and simultaneously maximize utilization by balancing the load on the SN links while meeting new resource requirements of already embedded virtual nodes.

The next section analyzes and formulates the problem. Section 4.3 presents our proposed heuristic algorithm to achieve minimum cost and load balancing. The results of performance evaluation and a comparison to prior art are reported in section 4.4. Section 4.5 concludes the chapter.

4.2 Problem formulation and model

The problem to solve is that of allocating additional infrastructure resources, from a substrate network (SN), to virtual nodes of already embedded and active virtual networks. Our goal is to adapt previous assignments while minimizing nodes and links re-allocations costs and the average saturation of the links to ensure load balancing in the SN. We consequently propose a bi-objective function to minimize jointly i) the cost of re-allocations and ii) the average link saturation in SN to provide the required elasticity for evolving nodes while maximizing SN utilization (or profitability).

To formulate the problem, we adopt the same network/VN/Mapping model of the previous chapter. To avoid repetition, only the summary of key notations are provided below.

Table 4.1: Summary of SN/VN/Mapping key notations

Notation	Description
G_s	Substrate Network
N_s	Set of substrate nodes n_s
L_s	Set of substrate links l_s
$a^t_{n_s}$	Available capacity of substrate node n_s
$a^t_{l_s}$	Available bandwidth on substrate link l_s
P_φ	Set of loop-free substrate paths φ
a_φ	Available bandwidth associated to a substrate path φ
G^r_v	Virtual Network r of VN^t
N^r_v	Set of virtual nodes n^r_v of VN G^r_v
L^r_v	Set of virtual links l^r_v of VN G^r_v
$b^t_{n^r_v}$	Minimum required capacity of virtual node n^r_v
$b^t_{l^r_v}$	Minimum required bandwidth on virtual link l^r_v
$M^t_{N^r_v} : N^r_v \rightarrow N_s$	Node mapping related to VN G^r_v
$M^t_{L^r_v} : L^r_v \rightarrow P_\varphi$	Link mapping related to VN G^r_v

In order to quantify the amount of resources used by the substrate network to fulfill the VN requests, we use the notion of stress. As most VN request rejections are caused by bandwidth shortage (Fajjari *et al.* (2011b)), we focus on avoiding substrate links saturation by balancing the load. Similarly to (Chowdhury *et al.* (2012)) we define the link stress of a substrate link l_s as the ratio of the total amount of bandwidth allocated to the virtual links whose substrate paths pass through l_s over the amount of bandwidth initially available in

l_s. Formally:

$$s^t_{l_s} = \frac{\sum_{l_v \to l_s} b^t_{l_v}}{a^0_{l_s}} = \frac{a^0_{l_s} - a^t_{l_s}}{a^0_{l_s}} \tag{4.1}$$

where $l_v \to l_s$ indicates that the substrate path of virtual link l_v passes through the substrate link l_s, and $a^0_{l_s}$ is the initial available bandwidth in l_s. The average link stress ALS^t in SN is defined consequently as:

$$ALS^t = \frac{\sum_{l_s \in L_s} s^t_{l_s}}{|L_s|} \tag{4.2}$$

where $|L_s|$ is the total number of substrate links in SN.

4.2.1 Problem Formulation

For a running evolving node m^i_v requiring additional resources in a substrate host h (with $M^t_{N^r_v}(m^i_v) = h$) that has insufficient resources, we need a strategy to re-allocate resources to other alternate candidate nodes (those having enough resources) to maintain the service. A trivial and suboptimal strategy is to move the entire evolving node to another less loaded host (Sun *et al.* (2013)). We proposed a more elaborate strategy in the previous chapter by reorganizing virtual nodes in the initial host in neighboring hosts while minimizing overall re-allocation cost without considering SN utilization.

In this chapter we extend the work by moving some candidate virtual nodes in the affected physical node to make room for the additional needs and balance the load on the SN at the same time in order to also optimize SN utilization and profitability. This is achieved by minimizing the average link saturation of the entire SN in addition to making cost effective re-allocation and migration decisions.

Intuitively, the most congestion causing virtual nodes in the initial host should be selected in priority as candidates for re-allocation and migration.

4.2.1.1 Optimization objective

As in the previous chapter, we consider two phases for node re-allocation: *remapping* and *migration*. We briefly remind the reader that the remapping phase consists in finding alternate resources to host the candidate evolving virtual node and its associated virtual links while the migration phase tries in a second stage to migrate tasks running on the virtual node onto the selected destination node to resume these tasks. The node re-allocation cost incurring a remapping cost $Cost_{remap}$ and a migration cost $Cost_{mig}$ is the same in used in chapter 3:

- **Re-allocation cost**

$$Cost_{realloc}\left(n_v^r\right) = Cost_{remap}\left(n_v^r\right) + Cost_{mig}\left(n_v^r\right) \tag{4.3}$$

and the global re-allocation cost $ReralloCost_{m_v^i}$ related to an evolving node m_v^i is:

$$ReralloCost_{m_v^i} = \sum_{\substack{n_v^r \ is \\ re-allocated}} Cost_{realloc}\left(n_v^r\right) \tag{4.4}$$

Our objective is to find the best re-allocation scheme that satisfies the evolving node additional resource request while minimizing all re-allocation costs and the average link saturation (4.2). This leads to the following "***Objective function***":

$$minimize(ReralloCost_{m_v^i}, ALS^{t+1}) \tag{4.5}$$

4.3 Heuristic algorithm design

The problem outlined above is a multi-objective optimization problem with conflicting objectives known to be NP-Hard (). Since we are looking for practical, implementable and scalable solutions, we resort to a heuristic algorithm called Bi-RSforEVN (*Bi-objective Reallocation Scheme for Evolving Virtual Node request*) to solve it. This heuristic algorithm proceeds in two steps which consist in first selecting the virtual nodes/links should be reallocated and then finds the best new hosts for them. The sequel of the algorithm is the same as RS-forEVN ref. However, the virtual node selection criteria and the one node-reallocation algorithm differ. Below we present the main differences.

4.3.1 Virtual node selection criteria

Assume that m_v^i is the evolving node asking for additional resources and that $coloc_h^t$ is the set of all virtual nodes hosted in the same physical node h as m_v^i (i.e. $M_{N_v^r}^t(m_v^i) = h$). In order to satisfy the elasticity request for m_v^i, we will re-allocate one or more co-located virtual nodes to free resources and make room in the hosting substrate node. Recall that we also aim at simultaneously "tidy up " the substrate networks and balance the load. To do so we move (migrate) congestion causing virtual nodes to less saturated substrate nodes (hosts).

To identify the virtual nodes causing the congestion, we define a "congestion impact " metric to use as the selection criterion. In fact, we use the notion of occupancy rate $OR(l_v^r, l_s)$ of a virtual link l_v^r passing through a substrate link l_s to evaluate the congestion impact. The occupancy rate $OR(l_v^r, l_s)$ is the ratio of the virtual link l_v^r required bandwidth

$b^t_{l^r_v}$ to the total bandwidth of l_s:

$$OR(l^r_v, l_s) = \frac{b^t_{l^r_v}}{a^0_{l_s}} \tag{4.6}$$

We derive the congestion impact of l^r_v on l_s as the product of its occupancy rate and l_s stress:

$$CI^t(l^r_v, l_s) = OR(l^r_v, l_s) * s^t_{l_s} \tag{4.7}$$

CI^t measures the "degree of involvement " of l^r_v in saturating l_s. The average congestion impact of a virtual link is the average of its congestion impacts on all substrate links hosting it:

$$ACI^t(l^r_v) = \frac{1}{|M^t_{L^r_v}|} \sum_{l_s \in M^t_{L^r_v}} CI(l^r_v, l_s) \tag{4.8}$$

Where $|M^t_{L^r_v}|$ is the number of substrate links hosting l^r_v. Since the congestion impact of a virtual node n^r_v is the sum of the congestion impacts of its attached virtual links, we get:

$$CI^t(n^r_v) = \sum_{l^r_v \in \mathbb{S}_{n^r_v}} ACI^t(l^r_v) \tag{4.9}$$

Table 4.2: Summary of measurement of SN key notations

Notation	Description
$s^t_{l_s}$	Stress of substrate link l_s
ALS^t	Average link stress on SN
$OR(l^r_v, l_s)$	Occupancy rate of l^r_v on l_s
$CI^t(l^r_v, l_s)$	Congestion impact of l^r_v on l_s
$ACI^t(l^r_v)$	Average congestion impact of l^r_v
$CI^t(n^r_v)$	congestion impact of virtual node m^r_v

The virtual nodes are selected according to

- their size and QoS requirements
- their congestion impact

The size of a virtual node includes its intrinsic size and the aggregate bandwidth of its associated links. The QoS corresponds to the maximum acceptable downtime of the virtual node during migration, and the congestion impact of a virtual node n^r_v is defined by equation (12).

To select the virtual nodes for re-allocation, we use a selection metric *Reem* that ranks the nodes according to their contribution to the overall congestion in a decreasing order. The selection variable *Reem* is defined as:

$$Reem(n_v^r) = \frac{CI^t(n_v^r)}{(b^t_{m_v^r} + \sum_{l_v^r \in \mathbb{S}_{n_v^r}} b^t_{l_v^r}) * downtime_r} \tag{4.10}$$

The *Reem* expression is a fraction composed of three terms. The numerator is the virtual node congestion impact. The denominator is the product of two terms: one term represents the "size " of the virtual node and the second one is related to the QoS requirements.

The purpose behind considering the ranking criterion $Reem\,(n_v^r)$ is threefold.

- First, we favor re-allocation of candidate virtual nodes and their attached links requiring the smallest amount of resources to minimize the re-mapping cost (equation 3.1).
- Second re-allocate in priority the smaller and more QoS degradation tolerant nodes to optimize the migration cost (equation 3.3) since the amount of bandwidth required to perform task migration will be minimized.
- And finally, favor the re-allocation of the most congestion causing virtual links 4.7 by moving (migrating) them to less saturated hosts.

As a result of this ranking, all virtual nodes in $coloc_h^t$ are sorted in a list $\vec{coloc}_h^t$ in decreasing order of their *Reem* value.

4.3.2 Virtual node re-allocation scheme

To re-allocate a virtual node n_v^r, its associated star topology $\mathbb{S}_{n_v^r}$ (the node and its links) should be re-mapped, in order to maintain n_v^r connectivity with all its peers and resume tasks through migration.

In order to minimize the re-allocation cost 4.3, the new substrate host for the re-allocated node is chosen among the nearest neighbors, $near_h^t$ of the initial host h, that have enough resources and that can reconstruct all the links in $\mathbb{S}_{n_v^r}$.

In order to balance the load, these virtual links are re-allocated using the shortest path algorithm, where the weight of each physical link is defined by its stress 4.1. Among the set $near_h^t$, the selected node is the one minimizing the total re-allocation cost and the average links saturation. The Virtual node re-allocation scheme is illustrated in algorithm 2.

4.4 Simulation results and evaluation

In this section, we will study the efficiency of our proposal, Bi-RSforEVN. To achieve this, we will first describe the settings, conditions and scenarios used to conduct the evaluation.

Algorithm 2 Bi-RSforEVN: One node re-allocation Scheme

re-allocate($n_v^r, RemapCost_{m_v^i}, MigCost_{m_v^i}$)
$re-allocationResult \leftarrow failure$
$remapCost^{best} \leftarrow \infty, \quad ANS^{best} \leftarrow ANS^t$
Search $near_{n_v^r}^t$
if $near_{n_v^r}^t$ is not empty **then**
  for all $n_s \in near_{n_v^r}^t$ **do**
    map n_v^r in n_s
    for all $l_v^r \in \mathbb{S}_{n_v^r}$ **do**
      re-map virtual link l_v^r onto a substrate path φ using shortest path algorithm
    end for
    if $\mathbb{S}_{n_v^r}$ mapping succeeds **then**
      $re-allocationResult \leftarrow success$
      if $remapCost(\mathbb{S}_{n_v^r}) * ANS^{t+1} < remapCost^{best} * ANS^{best}$ **then**
        $remapCost^{best} \leftarrow remapCost(\mathbb{S}_{n_v^r})$
        $ANS^{best} \leftarrow ANS^{t+1}$
      end if
    end if
  end for
  if $re-allocationResult = Success$ **then**
    Add $cost_{mig}(n_v^r)$ to $MigCost_{m_v^i}$
    Add $cost_{remap}(n_v^r)$ to $RemapCost_{m_v^i}$
  end if
end if
return $re-allocationResult$

Then we will compare our algorithm with relevant prior art with a focus on the total re-allocation cost and the average link saturation observed after accepting requests for additional resources.

4.4.1 Simulation environment

We used the same VN embedding simulator implemented in chapter 3 and used the GT-ITM tool (Zegura *et al.* (1996)) to generate random topologies of the substrate and VN networks. We adopt the similar simulation conditions to existing work to be able to compare in equivalent scenarios the performance of our algorithm. As described in section 3.5, the SN (Substrate Network) 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 real numbers uniformly distributed between 0 and 50. Without loss of generality, we set the per unit node and edge resources costs to 1 unit. The requested VNs have between 2 and 10 virtual nodes in their topologies with an average connectivity also set to 50%. The node resource capacity is uniformly distributed between 0 and 20 and the edge resource capacity is uniformly distributed 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 k-shortest path algorithm to map edges (we choose the longest path, k=5) . This step leads to suboptimal embedding that can reflect the state of a SN subject to multiple virtual nodes evolutions.

To create a highly dynamic environment and unpredictable states or situations, we select randomly N evolving nodes among the virtual nodes hosted in SN as nodes that require additional resources. We define r, the ratio of the number of evolving nodes to the total number of virtual nodes in SN. $r = \frac{N}{|SN|}$. The increasing resource requests are expressed using the parameter "Increase Factor" (IF):

$$b_{m_v^i}^{t+1} = IF * \ b_{m_v^i}^{t} \tag{4.11}$$

Where $b_{m_v^i}^{t+1}$ is the new resource requirement of the evolving node m_v^i.

4.4.2 Simulation results

As stated in previous chapter, only the authors of (Sun *et al.* (2013)) considered the same assumptions and objective function as that of our proposal, so it is more relevant and appropriate to compare performance with their algorithm named **DVNMA_NS**.

In order to measure the effectiveness of our algorithm regarding its two main objectives,

we define two other variants of **Bi-RSforEVN**, each focusing on one of the two goals: **LB-RSforEVN** is a re-allocation scheme aiming at balancing the load over substrate links regardless of the re-allocation cost, whereas **RSforEVN** minimizes the re-allocation cost regardless of SN state, it is the algorithm proposed in our previous work (Jmila *et al.* (2014)). More details on the compared algorithms are given below.

Table 4.3: Compared algorithmsII

Notation	re-allocated virtual nodes	Chosen new host	Link re-allocation strategy
DVNMA_NS	The evolving node (systematically)	The most cost effective node among *all* substrate nodes	Shortest path (all weights=1)
RSforEVN	The smallest and more QoS degradation tolerant virtual nodes	The most cost effective node among nearest neighbors	Shortest path (all weights=1)
LB-RSforEVN	Virtual nodes with highest congestion impact	The node leading to minimum ALS among nearest neighbors	Shortest path (weight=links stress)
Bi-RSforEVN	The smallest and more QoS degradation tolerant virtual nodes with highest congestion impact	The most cost effective node leading to minimum ALS among nearest neighbors	Shortest path (weight=links stress)

4.4.2.1 Better Re-allocation cost for large size evolving virtual nodes

The first simulation assesses the re-allocation cost of our algorithm for evolving virtual nodes of large sizes (Equation 4.10). To produce scenarios with large virtual nodes instances to re-allocate, 14 ($r = 1/12$) virtual nodes are selected randomly from the top 50 largest virtual nodes currently hosted in the SN among a total of 112 nodes. The re-allocation cost is measured for variable Increase Factors, representing the amount of additional resources that will be required by the 14 selected virtual nodes.

Figure 4.1 depicts the results of 100 averaged runs and indicates that **Bi-RSforEVN** and **RSforEVN** have the lowest re-allocation cost. In fact, these algorithms reduce the re-allocation cost by selecting small virtual nodes as candidates for re-allocation. This also makes them less sensitive and more robust to increasing IF values, contrary to **DVNMA_NS** that always re-allocates the evolving nodes themselves inducing high re-mapping costs when the evolving nodes are of large size.

Note that **LB-RSforEVN** has the highest re-allocation cost. In fact, this algorithm selects the virtual nodes to re-allocate regardless of their size and only considering their

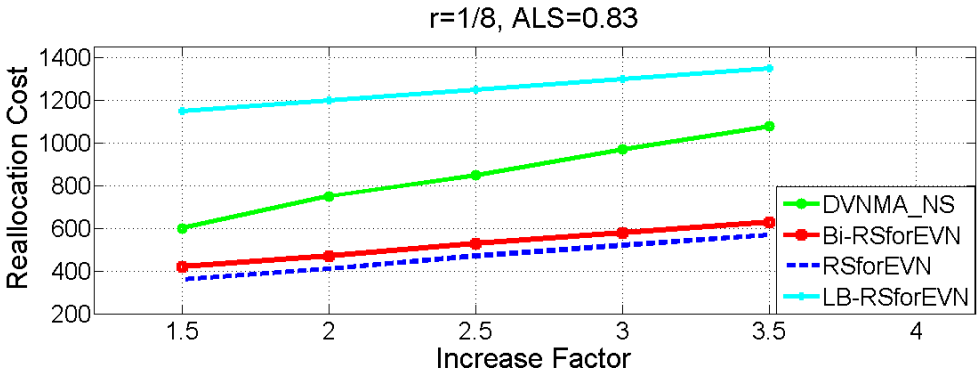


Figure 4.1: Reallocation cost (Bi-RSforEVN)

congestion impact, besides, when re-allocating $\mathbb{S}_{n_v^r}$, the new host is chosen as the one minimizing ALS in spite of the re-allocation cost. And finally, we notice that **Bi-RSforEVN** is slightly outperformed by **RSforEVN** that only focuses on minimizing re-allocation cost.

4.4.2.2 Better load balancing

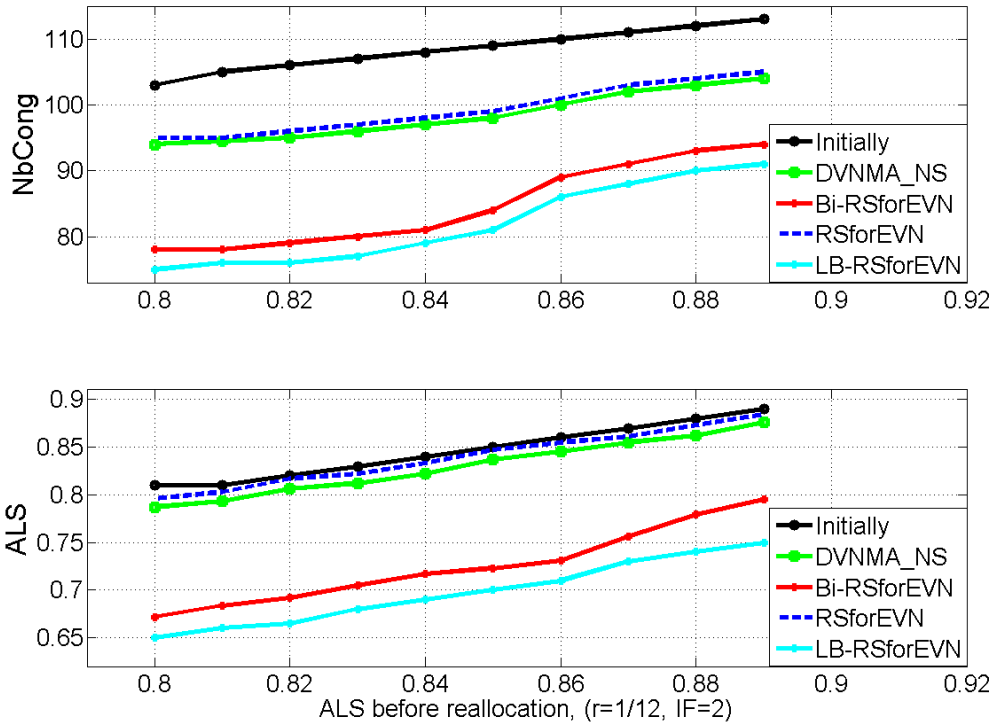


Figure 4.2: Load balancing (Bi-RSforEVN)

A substrate link is called *congested* if its is over stressed regarding the average link stress in SN ($s_{l_s}^t > ALS^t$). The number of congested substrate links is denoted $nbCongested$.

We measure *ALS* and *nbCongested* observed after re-allocating N evolving nodes (while maintaining $r = 1/12$), for different *initial ALS* values.

Figure 4.2 shows that **Bi-RSforEVN** (resp.**LB-RSforEVN**) reduces by 17% (resp. 19%) the average link saturation and 24% (resp. 27%) the number of congested substrate links, leading to a better load balancing compared to **DVNMA_NS** and **RSforEVN**.

The gap is more significant when the SN is slightly saturated, in fact, in such situation these algorithms find more easily less saturated hosts for re-allocated resources as a part of substrate resources is still available. This task is more difficult when the SN is saturated as almost all resources are congested, but they still perform well, reducing the *ALS* by 11% (resp. 13%) and *nbCongested* by 17% (resp 19%). Note that **DVNMA_NS** and **RSforEVN** minimize slightly the ALS thanks to the use of the shortest path algorithm, compared to the k-shortest path algorithm in the initial embedding.

We also notice that **Bi-RSforEVN** is slightly outperformed by **LB-RSforEVN** that only focuses on load balancing.

4.4.2.3 Load balancing Vs re-allocation cost

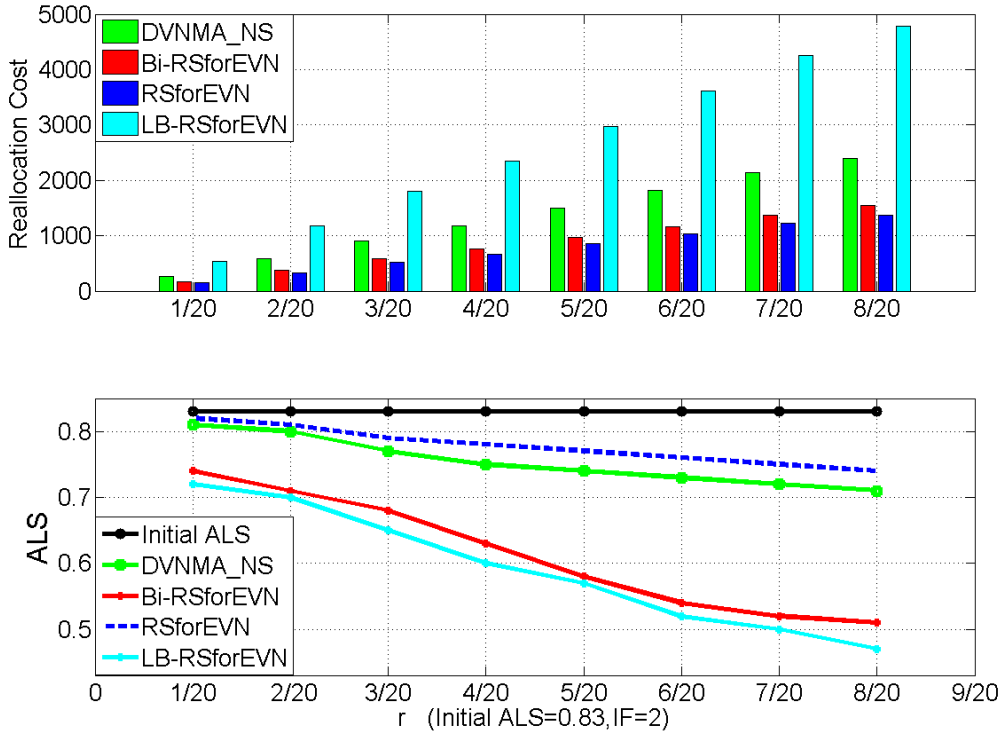


Figure 4.3: Re-allocation cost Vs Load balancing (Bi-RSforEVN)

In this simulation, we aim at measuring the effectiveness of our algorithm regarding

the two main objectives simultaneously. For different values of evolving nodes number, we measure the ALS observed after accepting all elasticity requests, while noting the total re-allocation cost.

Figure 4.3 shows that, for all algorithms, when the number of re-allocated nodes increases, the total re-allocation cost trivially increases and the ALS decreases, in fact, the more we make reconfigurations, the more we "'tidy up"' the SN and resolve eventual congestion problems.

We notice that **RSforEVN** realizes the best re-allocation cost, but it has the worst performance in term of load balancing. **LB-RSforEVN** is the best load balancing algorithm in spite of being the most costly. **DVNMA_NS** is less costly then **LB-RSforEVN**, but it is outperformed by **Bi-RSforEVN**, and does not reduce significantly the ALS. Only **Bi-RSforEVN** has good performances in both adjectives, in fact it reduces by 46% the ALS (for r=9/20) with a reasonable cost (30% less then **DVNMA_NS**).

In conclusion **Bi-RSforEVN** offers the best trade-off between re-allocation cost and load balancing strategy.

4.5 Conclusion

In this chapter we investigated two issues: i) allocating additional resources for virtual nodes in virtual networks and ii) maximizing substrate networks profitability. We proposed an algorithm that offers the best trade-off in terms of re-allocation cost and load balancing when compared to prior art. In the next chapter, we concentrate on the problem of bandwidth demand fluctuation on virtual links, and propose a distributed algorithm based on the self stabilization concept.

Chapter 5

A Self-Stabilizing framework for Dynamic Bandwidth Allocation

Contents

5.1 Introduction

Chapters 3 and 4 concentrated on node resource requirements fluctuation, and a *centralized* and *global-view* re-allocation approach. In spite of its advantages, a centralized approach is not suitable to the *wide and dynamic* Cloud environment. In fact, maintaining a global *up-to-date* description of *all dynamic network parameters* (available resources and mapping) is very costly and causes real-time monitoring overhead. Indeed, the changing demands of embedded VNs, the arrival and departure of others, the substrate resources failures,

etc. influence continuously the substrate network description. Hence, maintaining a central database containing this dynamic information induces a high latency of analysis and enforcement of changes, and produces an overhead related to the management traffic of the central entity. This leads to low responsiveness to the infrastructure evolution and affects the Cloud user satisfaction.

For these reasons, we opt for a *distributed* and *local-view* solution to address the *bandwidth demand fluctuation problem*. In fact, the proposal is an algorithm running on each substrate node to deal with the VN topological changes or variations on bandwidth requirements. The model is based on the *Self-Stabilization* concept that guaranties the convergence of the system to a stable/legitimate state in a finite time regardless of its initial situation. In our model, a bandwidth requirement fluctuation/perturbation drifts the system (the substrate network) into an "illegitimate state", while satisfying the new resource requirements takes it back to the "legitimate" state.

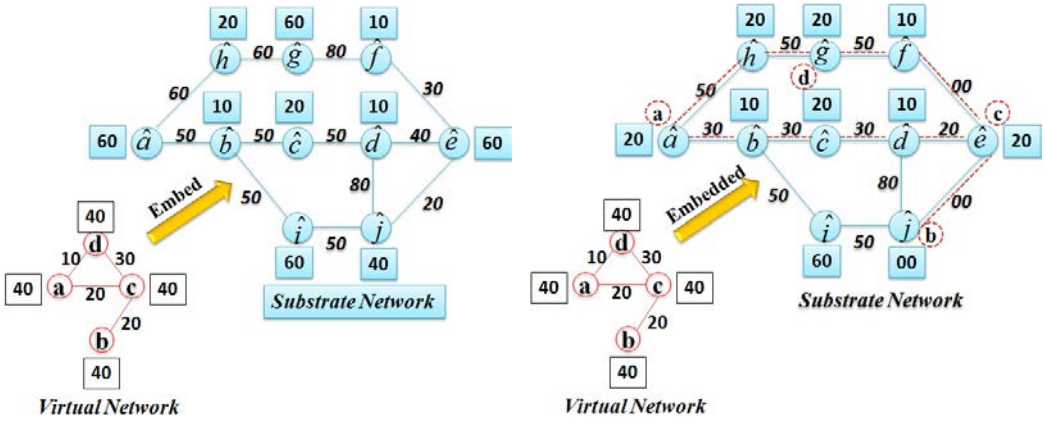
This chapter is organized as follows. In the next section, we expose the dynamic bandwidth allocation problem. Section 5.3 introduces the concept of self-stabilization, while section 5.4 describes our proposal. The performance of our algorithm is evaluated in section 5.5 and section 5.6 concludes the chapter.

5.2 Problem description

This section describes the problem of dynamic bandwidth allocation to deal with demand fluctuation during the VN lifetime. To do so, recall that the virtual network embedding is composed of two stages: *the initial VNE*, and the *dynamic resource management*. The first stage efficiently maps the initial VN request onto the substrate network, whereas the second stage deals with the resource demand fluctuation of the embedded VNs and the re-optimization of the substrate network usage. In the next section we propose a preliminary model for both stages.

5.2.1 Initial VNE

As in previous chapters, let $G_s = (N_s, L_s)$ be a weighted undirected graph that represents the substrate network, where N_s is the set of *substrate nodes* and L_s is the set of *substrate links*. To simplify the notations, a substrate node $n_s \in N_s$ will be represented by $\hat{n}$ in the rest of the document. Each substrate node $\hat{n}$ is characterized by an amount of available resources $\hat{n}_{av}$ (typically CPU and memory) and a unit cost $cost(\hat{n})$ and each substrate link $l_s \in L_s$ is associated with an available bandwidth ${l_s}^{av}$ and a unit cost $cost(l_s)$. Figure 5.1(a)



(a) (b)

Figure 5.1: Initial VN embedding

represents an example of a substrate network, where the numbers in rectangles next to the nodes represent the amount of available node resources and the numbers next to the edges represent the available bandwidth.

Similarly, the VN request topology is represented by a weighted undirected graph $G = (N, L)$, where N is the set of *required* virtual nodes and L is the set of *required* virtual links. Let req_n denote the *minimum required capacity* of the virtual node $n \in N$ and req_l the *minimum required bandwidth* on link $l \in L$. Figure 5.1(a) shows an example of a virtual network request. The numbers in rectangles next to the virtual nodes represent the amount of node requested resources and the numbers next to the virtual edges represent the required bandwidth.

Diverse approaches in the literature can be used to find an efficient embedding of the VN requests (centralized and distributed solutions cf. Chapter 2). An example of embedding result is shown in figure 5.1(b), where the SN available resources are updated. In this chapter we concentrate on the second stage of the VNE, and more precisely on the management of bandwidth demand fluctuation on virtual links.

5.2.2 Management of bandwidth demand fluctuation

The second stage follows successful initial embedding of VNs, the result is multiple VNs running simultaneously over the substrate network. 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 5.1(b), $N_{\hat{a}} = \{a\}$ and $L_{\hat{a}} = \{(a, d), (a, c)\}$.

Over time, the VN end user requirements can change, for instance when staring/ com-

pleting a new task/ application, or when their required resources change, consequently, the corresponding VN characteristics (topology and resources requirements of virtual nodes and links) will dynamically change. In such situation, adaptive techniques come into play. In this work, we concentrate on the bandwidth requirement fluctuation and we enumerate four scenarios of bandwidth demand fluctuation:

- *i*) partially release no more required bandwidth of an embedded virtual link,
- *ii*) completely remove a virtual link,
- *iii*) add a new virtual link to connect two embedded virtual nodes,
- *iv*) allocate more bandwidth to an embedded virtual link.

To cope with the scenarios *i*) and *ii*), the substrate network provider should release some bandwidth. As for *iii*) and *iv*) the provider should allocate more bandwidth to the embedded virtual link *when feasible*, otherwise find a new substrate path to support the new required bandwidth.

To model the bandwidth demand fluctuation on a virtual link l, let $allo_l$ denote the amount of bandwidth *currently* allocated to l. If the resource requirements of link l are satisfied, we have $req_l = allo_l$. Else, we formulate the four bandwidth fluctuation scenarios as follows:

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

As discussed in the introduction, for scalability reasons, we opt for a distributed and local-view solution. To do so, we will propose a "Self Stabilization" based approach. First, we will introduce the Self-stabilization theory, then explain our motivation for such concept. Afterward, we will describe our solution.

5.3 Self-Stabilization

5.3.1 Introduction to Self-Stabilization

Self-Stabilization (Dijkstra (1974)) is related to Autonomic Computing (Parashar & Hariri (2005)), which entails several "Self-*" attributes: self-management, self-configuration, self-healing, self-optimization, and self-protection. It is the property of an autonomous process to obtain *correct behavior* (reach a "legitimate 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.

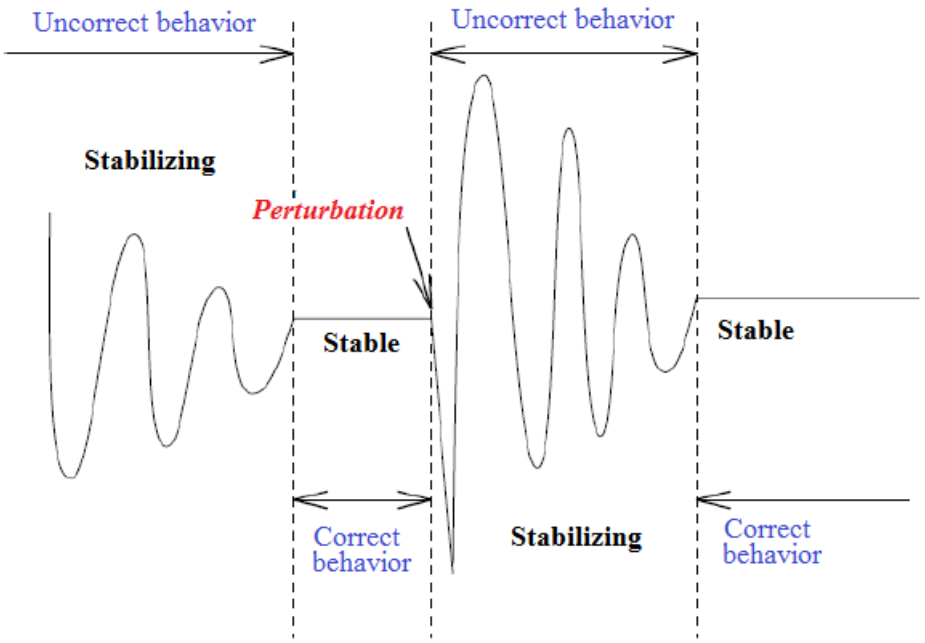


Figure 5.2: Self-stabilization according to Dijkstra.

We give the following simplified definition of a self-stabilizing system:

Definition: (Schneider (1993))
A system is self-stabilizing **if and only if**:

1. Regardless of its initial state, it has a tendency to be stable over time. (Convergence)

2. Once it is in a legitimate state, it remains stable unless perturbed by external force. (Closure)

In a self stabilizing system, each of the individual entities (nodes) composing it maintains local variables determining its local state. A node can be either *stable* or *active* and can change its local state by making *a move*. As the global state of a self-stabilizing system is the union *of all local states* of its nodes, 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 legitimate state is often used to measure the efficiency of the algorithm.

The notion of Deamon

Central to the theory of self-stabilization is the notion of the *daemon* (Dubois & Tixeuil (2011)), it is the entity responsible for setting the order of actions/moves execution, it determines if several nodes can act together or one after the other. 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 active nodes in the system, i.e. the legitimate state is reached.

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 (Seddiki *et al.* (2013); Amshavalli & Kavitha (2014); Carter *et al.* (1998)) employ a centralized *manager* (or controller, orchestrator, scheduler etc) to supervise and *harmonize* the distributed entities behavior. **In the same perspective, we rather replace the daemon by a scheduler that *helps* the system to converge by setting the *best* nodes execution scheme to take the system to stability.** In the following, this scheduler is called *Controller*.

Nevertheless, note that there is a difference between the Controller we propose and a classic central entity in charge of resource allocation (as that proposed in related work). In fact, to take scheduling decisions, our Controller *does not require any* description of the dynamic attributes related to the physical resources (like the actually available resources in nodes/links, the actual mapping etc). *Only* the 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 *execution order* using an elementary scheduling algorithm that sorts the nodes according to their action *priority*.

5.3.2 Motivation for Self-stabilization

The following advantages of self stabilization motivated us to explore this concept:

- **Local-view:** A self-stabilizing algorithm it is *local-view*, hence maintaining a global up-to-date description of the physical infrastructure (which is costly in the highly dynamic Cloud environment) is not required.
- **Parallel-processing:** Self stabilization can allow parallel processing (by allowing many nodes to make a move simultaneously), hence managing *multiple and different* demand fluctuations in the same time is possible.
- **Dynamism**: Self-stabilizing property is well suitable to the topology changing networks : the code does not need to be modified when adding or removing nodes or edges in the substrate network, topological changes are tolerated *during the system operation*.
- **Initialization:** No system initialization is required as the self-stabilization ensures the convergence of the system to the legitimate state disregarding its original state.

Hereafter we present our solution; we will extend the network model proposed in section 5.2 with *self stabilizing elements* and propose a *distributed framework* to deal with the bandwidth demand fluctuations in virtual links.

5.4 A self-stabilizing framework for dynamic bandwidth allocation

5.4.1 System model

This section extends the network model proposed in section 5.2 with *self stabilizing elements* that will be used in the framework design. Namely, we expand the *virtual link* and *substrate node* models.

5.4.1.1 Virtual link description

In order to describe a virtual link mapping, we note $\vec{P_l}$ the substrate path hosting the virtual link l, $\vec{P_l}$ is the ordered list of substrate nodes composing the path. For example in figure 5.1(b), for $l = (a, c)$, $\vec{P_l} = \left\{\hat{a}, \hat{b}, \hat{c}, \hat{d}, \hat{e}\right\}$. Moreover, let src_l and des_l be the source and destination virtual nodes connected by l (i.e. for $l = (a, c)$, $src_l = a$ and $des_l = c$). Table 5.1 summarizes the variables describing a virtual link.

Table 5.1: Summary of Virtual link key notations

Notation	Description
req_l	The amount of bandwidth *required* by l
$allo_l$	The amount of bandwidth *allocated* to l
$\vec{P_l}$	Ordered list of substrate nodes composing the path hosting l
src_l	The source virtual node l
des_l	The destination virtual node l

5.4.1.2 Substrate node description

Each substrate node has a list of local variables. We suppose that each substrate node knows the local variables of its neighbors (through periodic message passing or a shared memory (Gambette (2006))), yet it can modify only *its* local variables (but not those of other neighbors).

We model each substrate node *local-view* of its environment using the following notations:

- **Local view of mapping:** For all the virtual links passing through or incident to $\hat{n}$, i.e. for for all $l \in L_{\hat{n}}$, let $\hat{n}_x$ denote the feature x describing l and saved in (seen by) $\hat{n}$ ($x \in \left\{l, allo_l, src_l, des_l, \vec{P_l}\right\}$). 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 virtual link description, for all $l \in L_{\hat{n}}$. Ideally, i.e when the system is in the legitimate state, all the substrate nodes in $\vec{P_l}$ share the same l description. In case of bandwidth demand fluctuation, an end to end update within $\vec{P_l}$ 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 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 the substrate link $(\hat{m}, \hat{n})$. When the SN is stable, $(\hat{n}, \hat{m})_{av} = (\hat{m}, \hat{n})_{av}$, elsewhere an update is required.

Table 5.2 summarizes the variables describing a substrate node.

Table 5.2: Summary of Substrate node key notations

Notation	Description
$\hat{n}_{av}$	Available computing resources in $\hat{n}$
$N_{\hat{n}}$	The set of virtual nodes hosted by $\hat{n}$
$L_{\hat{n}}$	The set of virtual links incident to/ or passing through $\hat{n}$
$\hat{n}_x$,$x \in \left\{l, req_l, allo_l, src_l, des_l, \vec{P}_l\right\}$	The feature x describing l as viewed by $\hat{n}$
$Neigh(\hat{n})$	The set of $\hat{n}$ neighbors
$(\hat{m}, \hat{n})$, $\hat{m} \in Neigh(\hat{n})$	The substrate link connecting $\hat{n}$ to a neighbor $\hat{m}$
$(\hat{n}, \hat{m})_{av}$	The available bandwidth in the substrate link $(\hat{m}, \hat{n})$ as seen by $\hat{n}$

5.4.2 The Self stabilizing framework

We propose a framework to deal with bandwidth demand fluctuation in already embedded virtual links. 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 types of bandwidth fluctuation described in section 5.2.2, and show that theses algorithms can run *simultaneously* without conflict. In the following, we will first describe the Controller role and the general execution plan, then we will detail the three proposed algorithms.

5.4.2.1 Controller description

In order to manage the different bandwidth fluctuation types, the substrate nodes should execute different actions (reserve bandwidth, release bandwidth, allocate, de-allocate etc. cf table 5.3) to update the evolving virtual link mapping and the substrate resources description (available bandwidth).

The Controller is responsible of setting the general execution plan of these algorithms. It is an entity able to exchange messages with all the substrate nodes of the network. It holds a *local database* containing the list of *active nodes*, scheduled to execute some actions. Each action A is associated with a *"priority"*, p_A that defines the urgency of its execution. To execute a task, an active node requires the *permission* of the Controller.

In fact, the general framework execution plan is organized into *rounds*. In each round, the Controller examines the list of *active nodes* in its database. Depending on the scheduled actions/tasks *priorities*, it selects a set of nodes allowed to execute their algorithms: the

Table 5.3: Actions description

Action	Description
Reserve BW	Set aside an mount of bandwidth without assigning it to a particular virtual link
Allocate BW	Assign an amount of reserved bandwidth, to a virtual link
Release BW	Release an mount of reserved bandwidth (not assigned to a virtual link yet)
De-llocate BW	De-assign an amount of bandwidth previously allocated to a virtual link

nodes wishing to execute *the highest priority task* can perform *simultaneously*. If each of these nodes need to execute *a different number* of actions during the round, the Controller determines the *lowest number* as the number of actions that all the selected nodes should execute, hence all the nodes running simultaneously will consume the same amount of time, and the round duration is determined by these tasks execution time. After executing, the Controller updates its database (the nodes that executed are removed, the others are kept for the next round). This process continues until all the nodes of the system are stable.

Active nodes: A node is said to be *active* if it is scheduled for executing a task (i.e. if it figures in the Controller database). In our model, two reasons can activate a substrate node: either a new bandwidth request is submitted, or the node is *solicited* by another neighboring node:

- **Activation due to a new bandwidth request:** When a virtual link l requires a new amount of bandwidth $req_l \neq allo_l$, the substrate node hosting src_l ($host(src_l)$) is activated. In more detail, an entry of the form ($host(src_l)$, A, l, req_l) is added to the Controller database, where $host(src_l)$ is the activated node, A is the action that this node should perform, l is the concerned virtual link and req_l is its new bandwidth demand. This active node will trigger a cooperation among the substrate nodes to satisfy the new request.

  Note that there are two ways to define the new bandwidth request: either the VN user *submits* a new bandwidth request to the Controller, or the substrate network provider runs specific workload prediction algorithms to foresee the new resource requirements (Gmach *et al.* (2007); Wei *et al.* (2010); Seddiki *et al.* (2013)), and depending on the prediction results, it schedules an action in the Controller database to a meet the

new demand. In this work we do not investigate this issue and simply suppose that depending on the new bandwidth demand type, a first action is scheduled.

- **Activation when solicited by a neighboring node:** To reach stability, substrate nodes should cooperate. Indeed, a substrate node $\hat{n}$ can request its neighbor $\hat{m} \in Neigh(\hat{n})$ to perform an action A for a specific virtual link l requiring a new amount of bandwidth $req_l \neq allo_l$. To do so, $\hat{n}$ asks the Controller to add ($\hat{m}$, A, l, req_l) in its database (by sending a message).

The different actions that can be executed are described in the following.

5.4.2.2 Algorithms description

Three algorithms are proposed to deal with different types of bandwidth fluctuation. Each algorithm is composed of a list of *actions*. The first algorithm concerns scenarios i) and ii) i.e the case of Decrease in Bandwidth Requirement or Link Removal. The second deals with the Link addition (scenario iii). And the last proposal focuses on the Increase in Bandwidth requirement (scenario iv).

5.4.2.3 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. In such situation, no more required bandwidth should be removed and the virtual link mapping should be updated along its hosting path.

In fact, upon receiving the new request, the origin substrate node, hosting src_l, is activated (as described above). If selected by the Controller, this 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 $host(src_l)$, then activates the next node in $\vec{P}_l$. This process continues until reaching the substrate path last node.

The action **Trigger Allocation** is associated with a priority p_{TA}, and the **De-allocate** action is associated with p_D, such that $p_{TA} > p_D$. The idea behind this choice is to favor managing multiple requests simultaneously. In fact, as meeting a new bandwidth

Algorithm 1: Decrease in Bandwidth Requirement/Link Removal

This algorithm is composed of the following two actions:

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})

Steps:

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

Action 2: De-allocate (Priority=p_D)

Steps:

1: $(\hat{n}, \hat{m})_{av}$ += $(\hat{n}_{allo_l} - req_l)$ (Update available bandwidth in $(\hat{n}, \hat{m})$ to synchronize with the the soliciting node $\hat{m}$, i.e. the previous node in in l hosting path
2: $\hat{n}_{allo_l} = req_l$ (Update the amount of resources allocated to l)
3: **if** $des_l \notin N_{\hat{n}}$ (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 $(\hat{n}, \hat{o})$ where $\hat{o}$ is the next node in l hosting path)
5: Add ($\hat{o}$, $Deallocate$, l, req_l) to the Controller database (Activate $\hat{o}$)
6: **end if**
7: **if** $req_l = 0$ (In case of virtual link removal) **then**
8: $L_{\hat{n}}.remove(l)$ (Remove l from $\hat{n}$ mapping list)
9: **end if**

demand requires *many node moves*, tackling various requests simultaneously will reduce the algorithm convergence time.

The Algorithm 1 and the following examples give more information about these actions.

Examples:

The next two examples will show a step by step execution of Algorithm 1. In the first example, only one bandwidth request is managed. In the second example, we tackle two bandwidth requests at the same time to show how our algorithm deals with multiple bandwidth demands simultaneously. For both examples, we will use the same VN and SN depicted in the initial VN embedding figure 5.1(b).

For each round, we will show two tables: one describing the most relevant substrate

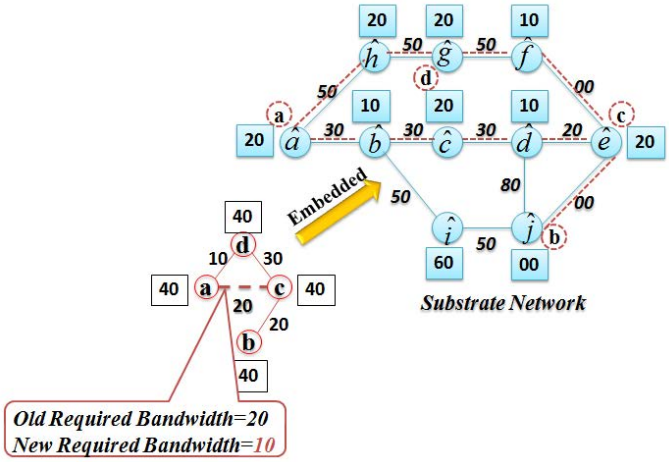


Figure 5.3: Decrease in BW Requirement or Link Removal : Example1

node parameters at the start/ end of the round, and the second illustrating the Controller database (at the start/ end of the round). The latter is represented by a 4 column table, where the first column lists the active nodes, and the others define the i) scheduled actions, ii) the concerned virtual links and iii) the new bandwidth requests. In each round, the *executing* substrate nodes and the main changes will be colored in red.

- **Example 1:**

Suppose that the virtual link $l = (a, c)$, initially demanding 20 bandwidth units and hosted by the substrate path $\vec{P}_l = (\hat{a}, \hat{b}, \hat{c}, \hat{d}, \hat{e})$, is now requiring a new amount of bandwidth = 10 units, and let us run through the algorithm.

- ***Initial situation***

Initially, the substrate network is stable: all the required bandwidth is met for all the embedded virtual links, in particular, $req_l = \hat{a}_{allo_l} = 20$, where $\hat{a}$ is the substrate node hosting src_l. Moreover, $(\hat{a}, \hat{b})_{av} = (\hat{b}, \hat{a})_{av} = 30$ and the Controller database is empty.

- ***Round 1***

After receiving l's new bandwidth request, the substrate node $\hat{a}$ is activated to run the action *Trigger de-allocation*. As there is only one node in the Controller database, $\hat{a}$ is selected to run. It executes the following steps: i) it updates l mapping to have $\hat{a}_{req_l} = \hat{a}_{allo_l} = 10$, ii) it frees the no longer required bandwidth in the substrate link $(\hat{a}, \hat{b})$ $((\hat{a}, \hat{b})_{av} = 40)$, and finally iii) it schedules the next node in $\vec{P}_l$ for execution: $\hat{b}$ is added to the Controller database.

- ***Round 2***

The substrate node $\hat{b}$ is selected by the Controller to execute the action $De-allocate$. It i) updates l mapping and the available bandwidth in $(\hat{a}, \hat{b})$ to synchronize with $\hat{a}$, then ii) releases bandwidth from $(\hat{b}, \hat{c})$ and finally activates $\hat{c}$.

- ***Rounds 3, 4 and 5***

In rounds 3 and 4, substrate nodes $\hat{c}$ and $\hat{d}$ execute the same steps as $\hat{b}$. In the last round, node $\hat{e}$ runs the same steps as $\hat{b}$ excepting the activation of the next node in $\vec{P}_l$, because $\hat{e}$ is the last node in the path.

- ***Comments***

Note that the system has reached stability in 5 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.*

Example 2: This example shows the performance of our algorithms in case of bandwidth fluctuation in multiple virtual links. We imagine that both virtual links $l = (a, c)$ and $l' = (a, d)$ have new demands. More precisely, (a, d) *needs to be removed* and (a, c) *requires only 10 bandwidth units* (like in the previous example).

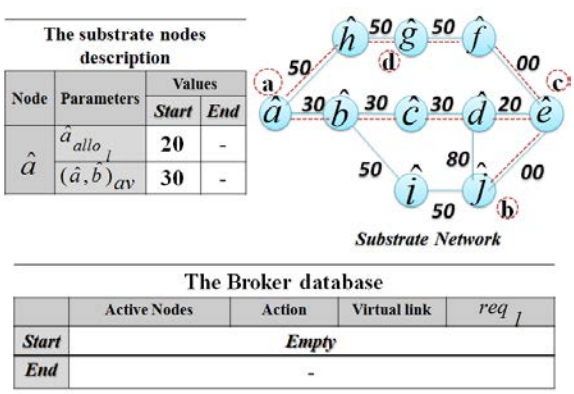
- ***Initial situation***

Initially, $\hat{a}_{allo_l} = req_l = 20$, $\hat{a}.allo_{l'} = req_{l'} = 10$, and $L_{\hat{a}} = \{l, l'\}$. Moreover, $(\hat{a}, \hat{b})_{av} = (\hat{b}, \hat{a})_{av} = 30$, $(\hat{a}, \hat{h})_{av} = (\hat{h}, \hat{a})_{av} = 50$, and the Controller database is empty.

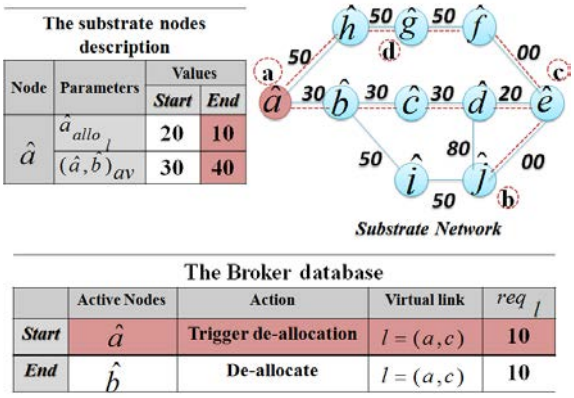
- ***Round 1***

In the first round, $\hat{a}$ is activated to execute *the same action* : *Trigger de-allocation* for *two* demands fluctuation. As there is only one active node, $\hat{a}$ is selected to execute both actions: it updates l and l' mapping and both $(\hat{a}, \hat{b})_{av}$ and $(\hat{a}, \hat{h})_{av}$, then activates $\hat{b}$ and $\hat{h}$.

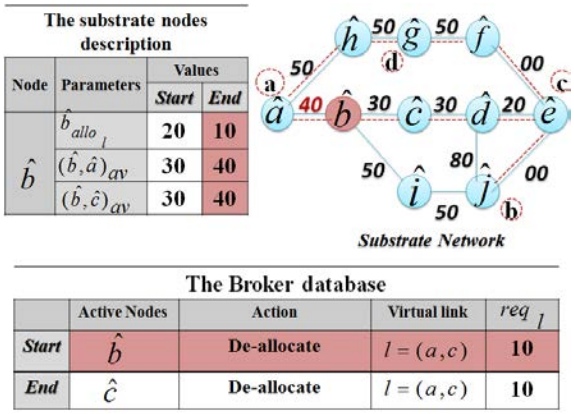
- ***Round 2***



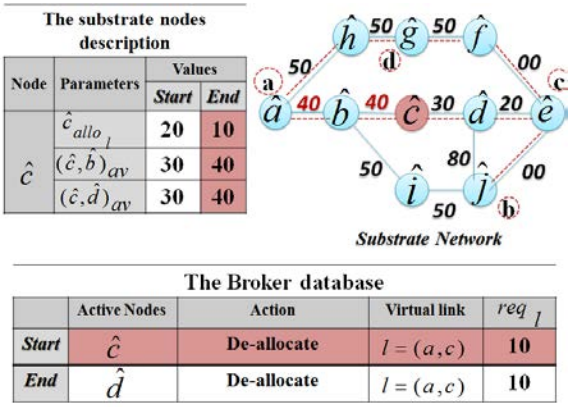
Initial situation



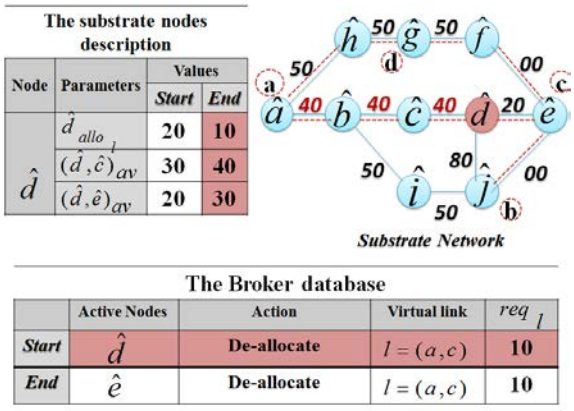
Round 1



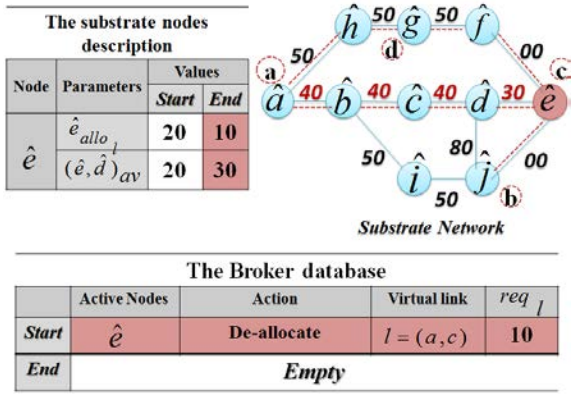
Round 2



Round 3



Round 4



Round 5

Figure 5.4: Decrease in BW requirement OR Link Removal : Example1

In the second round, both $\hat{b}$ and $\hat{h}$ are active to run the same action, hence the Controller selects both of them to execute *simultaneously* the *De-allocate* action. They both *synchronize* with $\hat{a}$, then activate the next nodes in each path: $\hat{b}$ activates $\hat{c}$, and $\hat{h}$ activates $\hat{g}$.

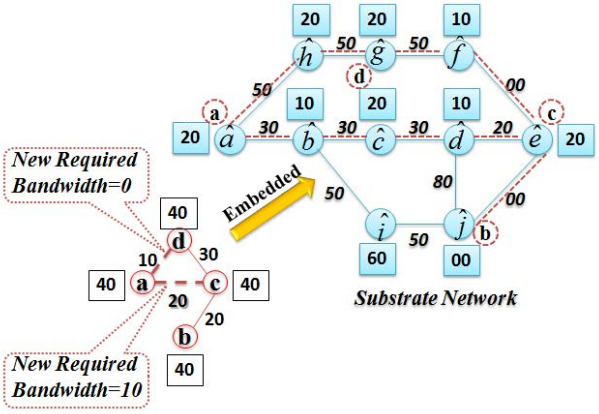


Figure 5.5: Decrease in BW Requirement or Link Removal : Example2

- ***Round 3, 4, 5***

Like in the round 3, the active nodes $\hat{c}$ and $\hat{g}$ will execute simultaneously the *De-allocate* action. They update with $\hat{b}$ and $\hat{h}$, then $\hat{c}$ activates $\hat{d}$. In the rounds 4 and 5, $\hat{d}$ and $\hat{e}$ will run like in the previous example to meet l new request.

- ***Comments***

Note that the algorithm converges in 5 rounds like the case of only one fluctuation. Moreover, remark that in case of multiple fluctuation requests, the number of rounds is at least equal to the number of nodes composing the longest hosting path of evolving links (as these nodes will run in separate rounds). In this example, l has the longest hosting path.

5.4.2.4 Algorithm2: 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 des_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 described in section 2.3.1.1). The aim of this algorithm is to find *the most cost effective* substrate path, in a distributed, and self stabilizing manner. To achieve this, our proposal runs in two steps: first step consists of searching and reserving the substrate path, and second is the bandwidth allocation step.

- **The first step** consists of i) searching all available paths (having enough bandwidth) to connect the two substrate nodes hosting src_l and des_l, and ii) saving the required amount of bandwidth in each of them.

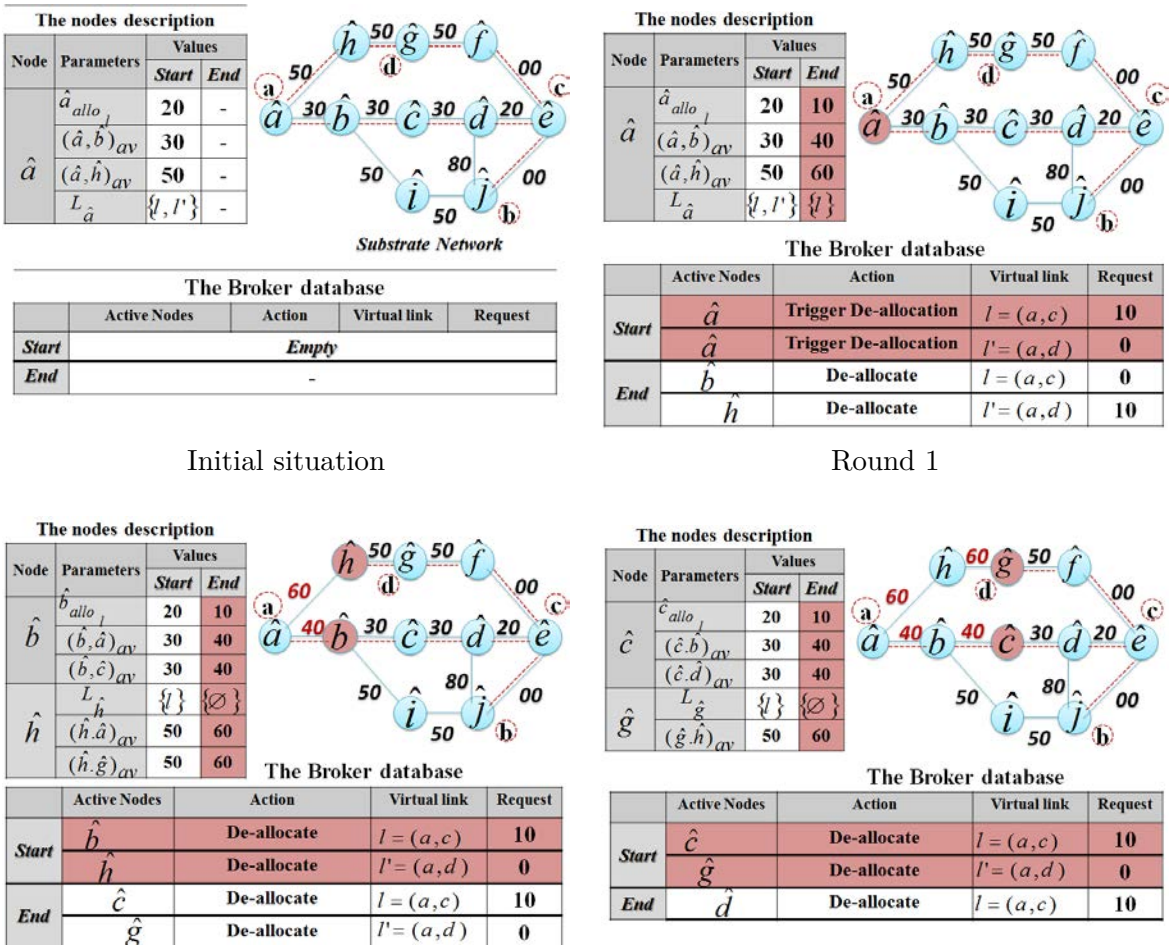


Figure 5.6: Decrease in BW Requirement or Link Removal : Example2

- **In the second step**, the most cost effective substrate path is selected among the K first arriving path proposals (where K is a tuning variable), then, the virtual link l is mapped to the best path, and the bandwidth previously reserved on other paths is released.

Four actions manage this algorithm. *Trigger searching and reserving bandwidth*, *Search and reserve bandwidth*, *Release bandwidth* and *Allocate bandwidth*. Hereafter we give a short description of each action, details can be found in the corresponding algorithms:

- ***Action 1: Trigger searching and reserving bandwidth***:

This action concerns the source substrate node (hosting src_l), called $\hat{n}$. It aims at triggering the first step (path research and reservation). To do so, the node $\hat{n}$ first checks if all its attached substrate paths are saturated, in this situation, the new bandwidth request is immediately rejected, as no path connecting $host(src_l)$ and $host(des_l)$ can be found. Else,

$\hat{n}$ starts building the new hosting path by adding $\hat{n}$ to $\vec{P_l}$, initially empty. Then, for all available attached substrate links, the node first reserves bandwidth on the substrate link and then activates the corresponding neighboring node to continue searching and reserving the path.

Note that the existence of an available substrate path to connect $host(src_l)$ and $host(des_l)$ *is not guarantied*. Hence, we risk to search infinitely for a nonexistent substrate path and *never reach stability*. To avoid this situation, we define $Timer_l$, a "timer" that limits the duration allowed for searching an available path, this duration is defined in terms of rounds, and depends on the SN dimension. $Timer_l$ is launched when the path research starts. If $Timer_l$ expires and no path is found, the new bandwidth request is rejected.

Algorithm 2: Link Addition

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

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

Steps:
1: Launch $Timer_l$ (start l's Timer)
2: **if** $(\forall \hat{o} \in Neigh(\hat{n}), (\hat{n}, \hat{o})_{av} < req_l)$ (if all connected substrate links are saturated) **then**
3: Reject the New bandwidth request
4: **else**
5: $\vec{P_l}.add(\hat{n})$ (start building the new path)
6: **for** $\hat{o} \in Neigh(\hat{n}) \mid (\hat{n}, \hat{o})_{av} > req_l$ (for all attached substrate links having enough bandwidth) **do**
7: $(\hat{n}, \hat{o})_{av}$ -= req_l (reserve bandwidth in $(\hat{n}, \hat{o})$)
8: Add ($\hat{o}$, Search and reserve BW, l, req_l) to the Controller database (activate $\hat{o}$)
9: **end for**
10: **end if**

- ***Action 2: Search and reserve bandwidth***:

This action concerns the other substrate nodes searching for an available path for the evolving virtual link. If the virtual link timer has not expired yet, and no hosting path is found yet, depending on the situation, a node $\hat{n}$ executing this action can run different operations:

- **IF** $\hat{n}$ is not the end node (does not host des_l), then there are two cases:
  - **First case:** If for all adjacent substrate links, *either* there is no more available bandwidth *or* the corresponding neighbor node is already in l path (i.e. there is

a risk to produce a loop!), then we conclude that n is *the end of a no-through path!*. In this situation, previously reserved bandwidth should be released from the saved path. To do so, $\hat{n}$ re-activates $\hat{m}$ (where $\hat{m}$ is the soliciting node) to perform a *Release bandwidth* action, described later, that will be spread *back* through the reserved path.

- **Second case:** else, $\hat{n}$ *synchronizes* with its soliciting node (updates the available bandwidth in the substrate link), then goes on reserving bandwidth on available substrate links, and solicits corresponding neighbors to do so.

- **ELSE** $\hat{n}$ hosts des_l and the path end is reached, hence $\hat{n}$ *synchronizes* with its soliciting node (updates the available bandwidth in the substrate link). As the end node must wait K path proposals to select the most cost effective one, depending on the rank R of the arriving proposal, there are three cases:

  - **First case:** $R < K$: then $\hat{n}$ simply saves the path proposal.
  - **Second case:** $R = K$: then, $\hat{n}$ selects the most cost effective path, thereafter, first it embeds l in this path and adds new l mapping to its list. Next it re-activates the previous node in the selected path to spread *back* the *Allocate bandwidth* update (described below) through the selected path. Third, for all non selected paths, the reserved bandwidth is released, and the previous node in each path is activated to execute a *Release bandwidth* action. Finally, $\hat{n}$ informs the Controller that a path proposal was found for l to stop reserving other paths.
  - **Third case:** $R > K$: In this case, the proposal is arriving late, and it is rejected: a *Release bandwidth* action is triggered among the reserved path.

- **ENDIF**

Note that if a path proposal is already found or the timer has expired, all the nodes activated to continue searching and reserving BW for the concerned link will no longer execute the previously described steps, but will only spread back a *Release BW* action through the previously reserved paths to cancel previous reservation.

- ***Action 3: Allocate bandwidth***:

To allocate *already reserved bandwidth*, to a virtual link l, the substrate node $\hat{n}$ first updates l mapping (add l to $L_{\hat{n}}$), note that no update for the available bandwidth on substrate links is required as the bandwidth was already reserved. Then,**if** $\hat{n}$ is not the origin node (not

hosting src_l), it spreads *back* a the Allocate bandwidth request by activating the *previous* node in $\vec{P_l}$ (note that this action is spread *back* through the previously reserved path, i.e. it starts from $host(src_l)$ to $host(des_l)$).

- ***Action 4: Release bandwidth***:

$\hat{n}$ releases previously reserved bandwidth in $(\hat{n}, \hat{o})$ where $\hat{o}$ is the soliciting node (i.e. the next node in the path, as this action is spread *back* through the previously reserved path). If $\hat{n}$ is not the origin node (does not host src_l), it releases reserved bandwidth in $(\hat{n},\hat{m})$ ($\hat{m}$ is the previous node in $\vec{P_l}$) and activates $\hat{m}$ to make a Release bandwidth update for l.

Note that a reserved substrate link can be common to many paths, as shown in the next example, hence, before updating available bandwidth on substrate links ($(\hat{n}, \hat{o})$ and $(\hat{n},\hat{m})$), $\hat{n}$ checks if the bandwidth was not already *released* or *used* in a previous move, in this case, no other updates are required.

Action priority: We choose the following priorities to schedule the previous actions:

- **Trigger searching and reserving BW**: p_{TSR}
- **Search and reserve bandwidth** : p_{SR}
- **Allocate bandwidth** : p_{All}
- **Release bandwidth** : p_{Rel}

Such that $p_{SR}<p_{Rel}<p_{All}<p_{TSR}$. 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 an available path to increase the number of path solutions.

Note that when two *neighboring* substrate nodes execute the actions *Trigger searching and reserving BW* and *Search and reserve bandwidth*, they risk to reserve bandwidth on the same substrate link connecting them *simultaneously*, thinking that there is enough bandwidth, which is not always the case as both nodes are still executing and their parameters values are not determined yet. To avoid such situation, the Controller selects a set of *non adjacent* active nodes to execute **Trigger searching and reserving BW** or **Search and reserve bandwidth** actions. The other nodes should wait for the next round. More details of theses actions can be found in Algorithm 2 and the following example.

Algorithm 2: Link Addition

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

Inputs:

1: $\hat{n}$, l, req_l

Steps:

1: **if** $Timer_l$ has not expired yet and no path is found for l yet **then**
2: **if** $des_l \notin N_{\hat{n}}$ (if $\hat{n}$ does not host the destination node) **then**
3: **if** $(\forall \hat{o} \in Neigh(\hat{n})$, $(\hat{n}, \hat{o})_{av} < req_l$ **or** $\hat{o} \in \vec{P_l})$ (if for all neighboring nodes, either the substrate link is saturated, or the node is already in $\vec{P_l}$, i.e. we risk to form a loop) **then**
4: Add ($\hat{m}$, Release BW, l, req_l) to the Controller database, where $\hat{m}= \vec{P_l}^{end}$ (activate $\hat{m}$ to Release BW, $\hat{m}$ is the soliciting node, i.e. the last node added to $\vec{P_l}$)
5: **else**
6: $(\hat{n}, \hat{m})_{av}$ -= req_l, $\vec{P_l}.add(\hat{n})$ (reserve bw to synchronize with $\hat{m}$, and add $\hat{n}$ to the path)
7: **for** $\hat{o} \in Neigh(\hat{n}) \mid (\hat{n}, \hat{o})_{av} > req_l$ **and** $\hat{o} \notin \vec{P_l}$ (for all neighbors connected to $\hat{n}$ with enough bw and not in $\vec{P_l}$) **do**
8: $(\hat{n}, \hat{o})_{av}$ -= req_l (reserve bw in $(\hat{n}, \hat{o})$)
9: Add ($\hat{o}$, Search and reserve BW, l, req_l) to the Controller database (activate $\hat{o}$ to Reserve BW)
10: **end for**
11: **end if**
12: **else**
13: The path end is reached and this is the Rth path proposal for l, $\hat{n}$ should select the best path among K proposals
14: $(\hat{n}, \hat{m})_{av}$ -= req_l, $\vec{P_l}.add(\hat{n})$ (reserve bw to synchronize with $\hat{m}$, and add $\hat{n}$ to the path)
15: **if** $R < K$ **then**
16: Save the path solution
17: **end if**
18: **if** $R = K$ **then**
19: $BestPath$= The most cost effective path among the K proposals
20: $L_{\hat{n}}.add(l)$ ($\hat{n}_{\vec{P_l}} = BestPath$, and $\hat{n}_{allo_l} = req_l$) (save the new mapping)
21: Add ($\hat{j}$, Allocate BW, l, req_l) to the Controller database, where $j = BestPath^{previous}$ (activate $\hat{j}$ to Allocate BW, where j is the previous node in $BestPath$)
22: **for** all non selected paths **do**
23: $(\hat{n}, \hat{i})_{av}$ -= req_l (release previously reserved BW, where $\hat{i}$ is the last node in a non selected path)
24: Add ($\hat{i}$, Release BW, l, req_l) to the Controller database (activate $\hat{i}$ to continue releasing Release BW through the path)
25: **end for**
26: **end if**
27: **if** $R > K$ (this proposal is arriving late, the virtual path is already embedded) **then**
28: $(\hat{n}, \hat{m})_{av}$ += req_l (release previously reserved bw on $(\hat{n}, \hat{m})$)
29: Add ($\hat{m}$, Release BW, l, req_l) to the Controller database (activate $\hat{m}$ to Release BW)
30: **end if**
31: **end if**
32: **else**
33: Add ($\hat{m}$, Release BW, l, req_l) to the Controller database (activate $\hat{m}$ to Release BW)
34: **end if**

- **Example:**

Algorithm 2: Link Addition

Action 3: Allocate BW (Priority=p_{All})

Inputs:

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

Steps:

1: $L_{\hat{n}}.add(l)$ save l new mapping
2: **if** $src_l \notin N_{\hat{n}}$ (if $\hat{n}$ does not host l source, i.e. this is not the origin node) **then**
3: Add ($\hat{m}$, Allocate BW, l, req_l) to the Controller database, where $\hat{m}= \vec{P}_l^{previous}$ (activate $\hat{m}$ to continue propagating back the Allocate BW action through the path)
4: **end if**

Algorithm 2: Link Addition

Action 4: Release BW (Priority=p_{Rel})

Inputs:

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

Steps:

1: **if** the reserved BW on $(\hat{n}, \hat{o})$ was not already released or allocated, where $\hat{o}= \vec{P}_l.next$ ($\hat{o}$ it is the next node in $\vec{P}_l$ because this action travels **back** through the path) **then**
2: $(\hat{n}, \hat{o})_{av}$ += req_l (release previously reserved bandwidth in $(\hat{n}, \hat{o})$)
3: **end if**
4: **if** $src_l \notin N_{\hat{n}}$ (if this is not the origin node) **then**
5: **if** the reserved BW on $(\hat{n}, \hat{m})$ was not already released or allocated, where $\hat{m} = \vec{P}_l^{previous}$ **then**
6: $(\hat{n}, \hat{m})_{av}$ += req_l (release previously reserved bandwidth in $(\hat{n}, \hat{m})$)
7: **end if**
8: Add ($\hat{m}$, Release bandwidth, l, req_l) to the Controller database (activate $\hat{m}$ to Release bandwidth), i.e. propagate back the Release bandwidth action
9: **end if**

This example gives a round per round explanation of the Algorithm 2 execution. To do so, imagine that a new virtual link l should be added to the VN topology to connect the two virtual nodes a and b, with $req_l = 30$. To simplify the example, we set K, the number of proposals that the end node should wait to 1 (the first path proposal is embedded) and suppose that all substrate links have the same cost 1, besides, we imagine that $Timer_l$ is long enough to find a path solution for l.

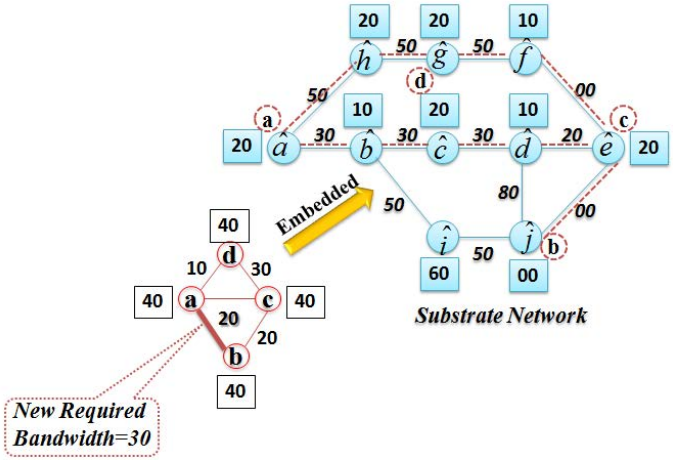


Figure 5.7: Link addition: Example

- ***Round 1***

To handle the new request, the substrate node $\hat{a}$ is scheduled to execute a T*rigger BW research and allocation action*, to initiate the research for a substrate path for l. When selected by the Controller, $\hat{a}$ updates $(\hat{a},\hat{b})_{av}$ and $(\hat{a},\hat{h})_{av}$ to reserve bandwidth on the available substrate links, then $\vec{P_l}$ is updated to save the reserved path ($\vec{P_l} = \{\hat{a}\}$). Finally, $\hat{a}$ activates $\hat{b}$ and $\hat{h}$ to continue searching for available paths.

- ***Round 2 and 3***

In the second round, $\hat{b}$ and $\hat{h}$ execute simultaneously: they reserve bandwidth in all available attached substrate links ($(\hat{h},\hat{g})$ for $\hat{h}$, $(\hat{b},\hat{c})$ and $(\hat{b},\hat{j})$ for $\hat{b}$), update the reserved path and then activate the next nodes ($\hat{g}$, $\hat{c}$ and $\hat{i}$), that will execute the same steps in the following round.

- ***Round 4***

In the round 4, $\hat{f}$, $\hat{d}$ and $\hat{j}$ execute simultaneously. $\hat{f}$ remarks that all its attached substrate links are unavailable ($(\hat{f}.\hat{e}).av = 0$ and $\hat{g} \in \vec{P_l}$), hence, it activates $\hat{g}$ to perform a *Release BW* action to cancel the bandwidth reservation among the saved path.

As for $\hat{d}$, only the substrate link $(\hat{d}.\hat{j})$ is available ($(\hat{d}.\hat{e}) = 20 < req_l$) , so, it reserves bandwidth on this link and activates $\hat{j}$ to do so.

$\hat{j}$ who was activated by $\hat{i}$ in round 3, is the end node: it hosts $des_l = b$, it concludes that it has just received the first path proposal for l. As $K = 1$, $\hat{j}$ immediately embeds l onto this path, it updates its mapping list L_j (add l), it synchronizes with $\hat{i}$ (update $(\hat{j}.\hat{i}).av$) and then it activates back $\hat{i}$ to *Allocate BW* and update l mapping among the chosen entire

substrate path: $\vec{P}_l = \left\{\hat{a}, \hat{b}, \hat{i}, \hat{j}\right\}$. Finally, $\hat{j}$ informs the Controller that a path was found for l.

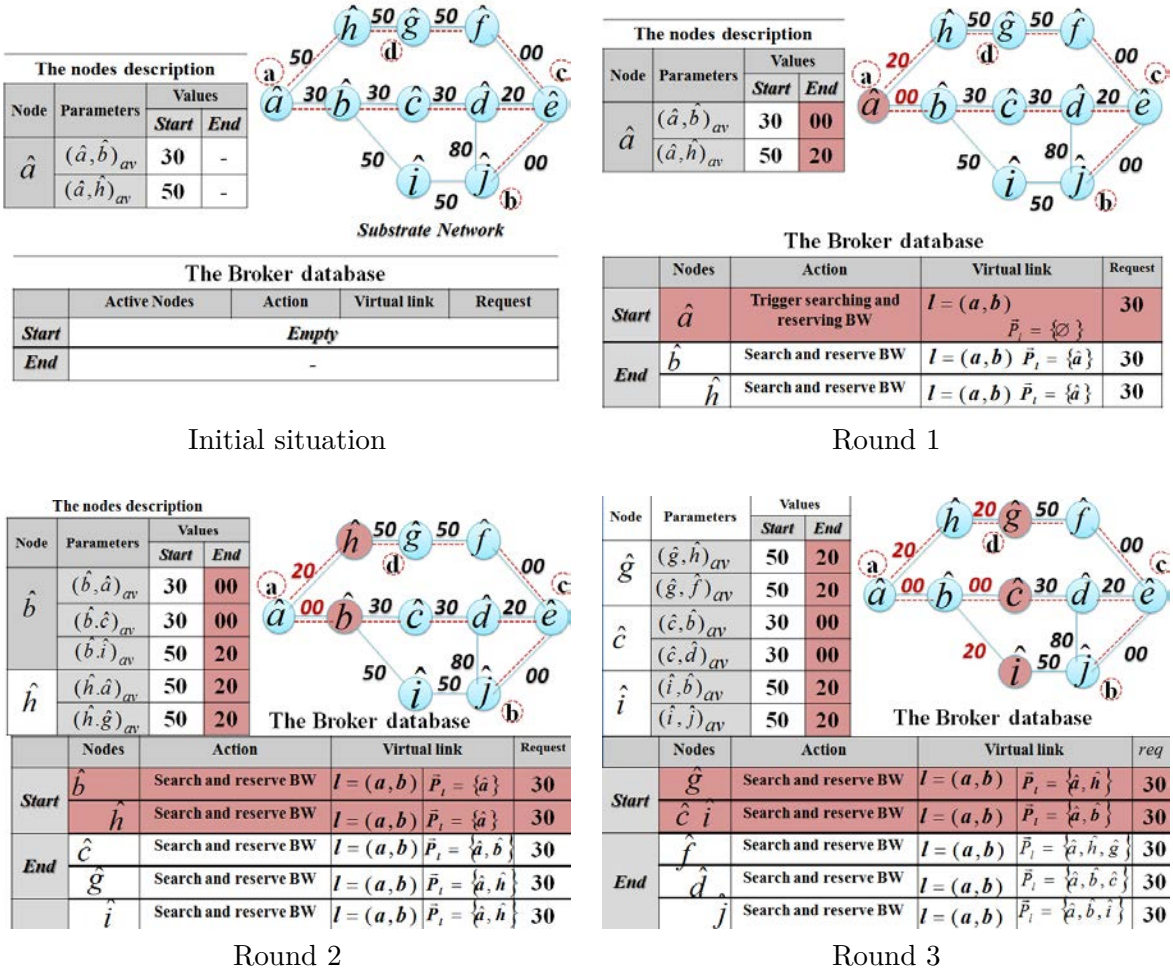


Figure 5.8: Link addition, Rounds 1-3

- ***Rounds 5, 6, 7***

In the round 5, three substrate nodes are active to execute three different actions: $\hat{g}$ is scheduled to *Release BW* ($priority = p_{Rel}$), $\hat{j}$ is scheduled to *Search and reserve BW* ($priority = p_{SR}$) and $\hat{i}$ is scheduled to Allocate BW ($priority = p_{All}$). Hence the Controller selects the one with highest priority: $\hat{All}$, because $p_{All} > p_{Rel} > p_{SR}$.

Thus, $\hat{i}$ updates its mapping to add l, and activates b. b and a also update their mapping in rounds 6 and 7, to embed l on $\vec{P}_l = \left\{\hat{a}, \hat{b}, \hat{i}, \hat{j}\right\}$

- ***Rounds 8, 9, 10***

In the round 8, only the nodes $\hat{g}$ and $\hat{j}$ are active (they where activated in round 4), as $\hat{a}$, did not activate any node in the previous round. Node $\hat{g}$, having the highest priority is selected to execute a *Release bandwidth* action, hence it updates the available resources on substrate links $(\hat{g}, \hat{f})$ and $(\hat{g}, \hat{h})$ and activates $\hat{h}$ to do the same thing in order to release the reserved bandwidth on the substrate link $\vec{P_l} = \left\{\hat{a}, \hat{h}, \hat{g}, \hat{f}\right\}$. This is accomplished in rounds 9 and 10.

- ***Rounds 11, 12, 13, 14, 15***

In round 11, only $\hat{j}$ is active, it is selected by the Controller to execute *a Search and reserve BW* action. As a path was already found, $\hat{j}$ triggers a Release BW update among $\vec{P_l} = \left\{\hat{a}, \hat{b}, \hat{c}, \hat{d}, \hat{j}\right\}$ to release previously reserved bandwidth. This will be completed in rounds 12, 13, 14 and 15.

- ***Comments:*** Note that although this algorithm converges in 15 rounds (for this example), the substrate path hosting l was found since the fourth round.

5.4.2.5 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**, we check 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, we will use a slightly modified version of already defined actions in previous Algorithms.

- ***Action 1: Trigger reserving bandwidth***:

This action concerns the substrate node hosting src_l and aims at triggering bandwidth reservation *among the path supporting* l. Note that the main differences between this action and the *Trigger searching and reserving bandwidth* action defined in Algorithm 2 are: i) no path research is required as the bandwidth is reserved only among l's hosting path, already known, and ii) the amount of reserved bandwidth is only $req_l - allo_l$, (compared to req_l in *Trigger searching and reserving bandwidth* action) as $allo_l$ is already allocated to l.

The node $\hat{n}$ executing this action first checks if there is enough bandwidth on the substrate link $(\hat{n}, \hat{o})$, where $\hat{o}$ is the next node in $\vec{P_l}$. If it is the case, $\hat{n}$ reserves bandwidth on

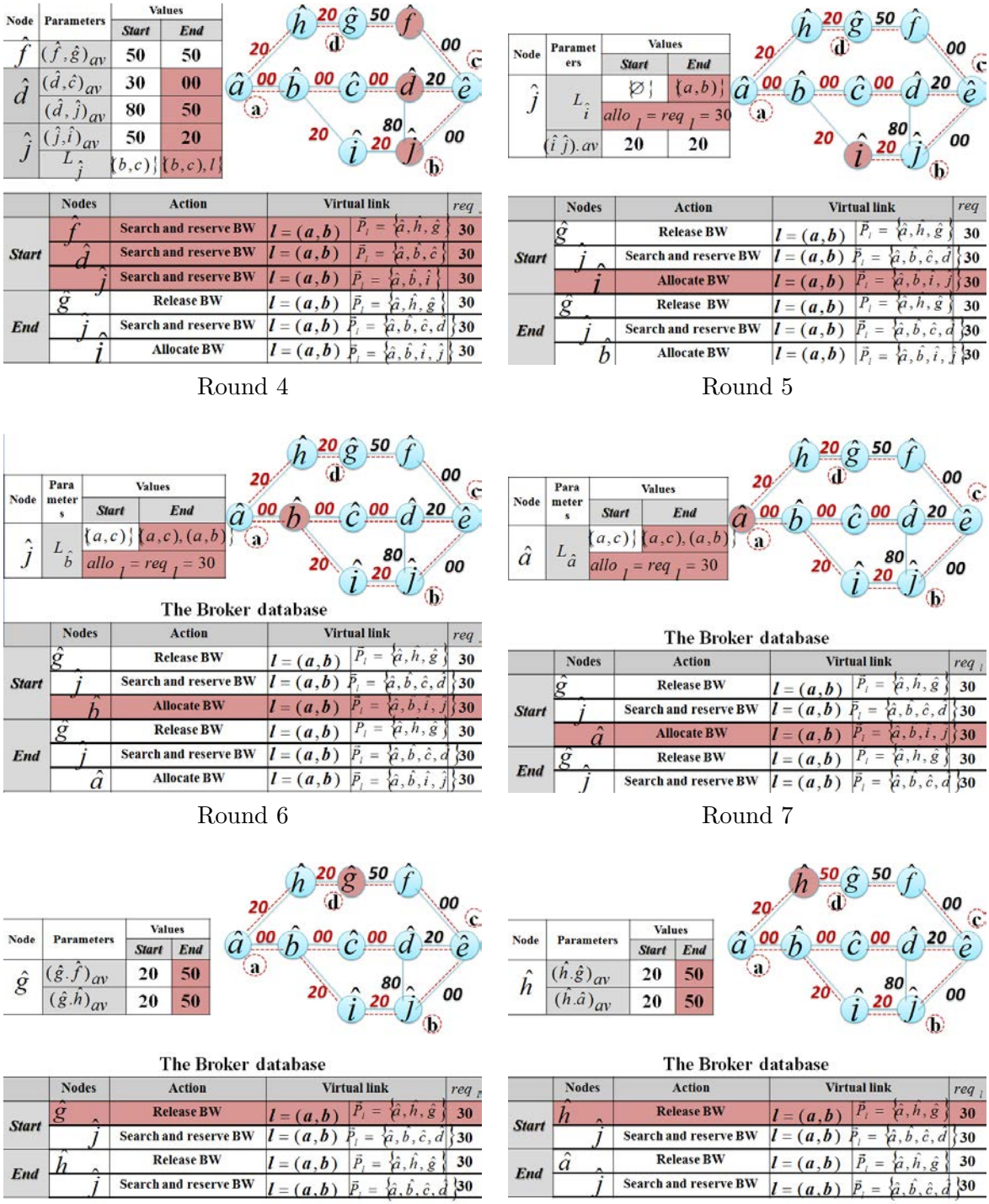
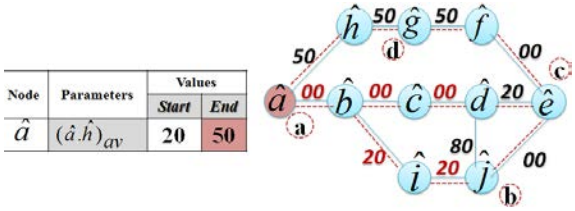


Figure 5.9: Link addition, Rounds 4-9

$(\hat{n}, \hat{o})$ and activates $\hat{m}$ to do so (using action ***Reserve bandwidth***, defined below). Else, the new bandwidth demand can not be supplied over $\vec{P}_l$, and a *virtual link migration* is required. To do so, $\hat{n}$ performs the steps of *Trigger searching and reserving bandwidth* action defined in Algorithm 2 in order to *start searching and reserving a new path* to support

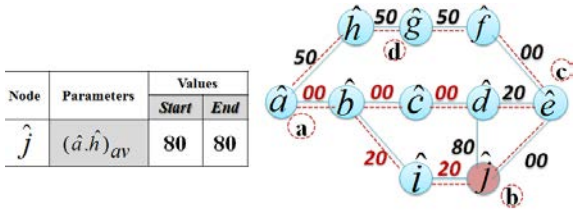


Node	Parameters	Values	
		Start	End
$\hat{a}$	$(\hat{a}.\hat{h})_{av}$	20	50

The Broker database

	Nodes	Action	Virtual link	req_l
Start	$\hat{a}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{h},\hat{g}\}$	30
	$\hat{j}$	Search and reserve BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d}\}$	30
End	$\hat{j}$	Search and reserve BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d}\}$	30

Round 10

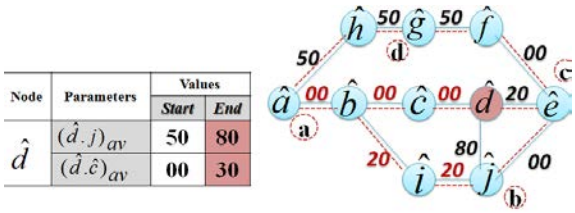


Node	Parameters	Values	
		Start	End
$\hat{j}$	$(\hat{a}.\hat{h})_{av}$	80	80

The Broker database

	Nodes	Action	Virtual link	req_l
Start	$\hat{j}$	Search and allocate BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30
End	$\hat{d}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30

Round 11

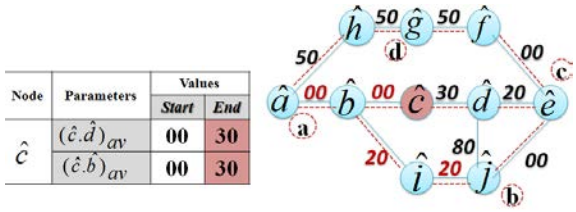


Node	Parameters	Values	
		Start	End
$\hat{d}$	$(\hat{d}.\hat{j})_{av}$	50	80
	$(\hat{d}.\hat{c})_{av}$	00	30

The Broker database

	Nodes	Action	Virtual link	req_l
Start	$\hat{d}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30
End	$\hat{c}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30

Round 12

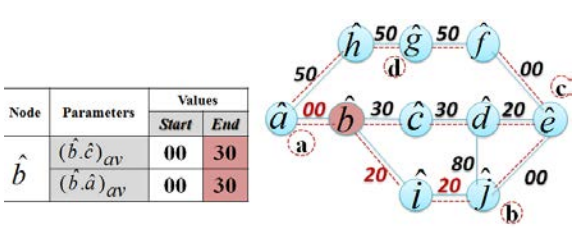


Node	Parameters	Values	
		Start	End
$\hat{c}$	$(\hat{c}.\hat{d})_{av}$	00	30
	$(\hat{c}.\hat{b})_{av}$	00	30

The Broker database

	Nodes	Action	Virtual link	req_l
Start	$\hat{c}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30
End	$\hat{b}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30

Round 13

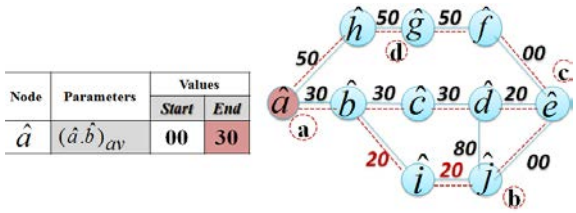


Node	Parameters	Values	
		Start	End
$\hat{b}$	$(\hat{b}.\hat{c})_{av}$	00	30
	$(\hat{b}.\hat{a})_{av}$	00	30

The Broker database

	Nodes	Action	Virtual link	req_l
Start	$\hat{b}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30
End	$\hat{a}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30

Round 14



Node	Parameters	Values	
		Start	End
$\hat{a}$	$(\hat{a}.\hat{b})_{av}$	00	30

The Broker database

	Nodes	Action	Virtual link	req_l
Start	$\hat{a}$	Release BW	$l=(a,b)$ $\vec{P}_l=\{\hat{a},\hat{b},\hat{c},\hat{d},\hat{j}\}$	30
End	*Empty*			

Round 15

Figure 5.10: Link addition, Rounds 10-15

l, then executes the stages of action *Trigger de-allocation* of Algorithm 1, to delete l old mapping and release bandwidth on the old path. Details about this action can be found in algorithm 3.

Algorithm 3: Increase in BW requirement

Action 1: Trigger reserving bandwidth (Priority=p_{TR})

Inputs:

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

Steps:

1: **if** $(\hat{n}, \hat{o})_{av} > (req_l - allo_l)$, where $\hat{o} = \vec{P}_l^{next}$, if there is enough BW on the substrate link $(\hat{n}, \hat{o})$, where $\hat{m}$ is the next node in $\vec{P}_l$ **then**
2: $(\hat{n}, \hat{o})_{av} -= (req_l - allo_l)$ (reserve bandwidth in $(\hat{n}, \hat{m})$)
3: Add ($\hat{o}$, Reserve bandwidth, l, req_l) to the Controller database (activate $\hat{o}$ to spread the BW reservation along the path)
4: **else**
5: A virtual link migration is required:
6: **First :** Perform the steps of the action *Trigger searching and reserving bandwidth* of Algorithm 2:
7: Start $\hat{n}.Timer_l$ (start l Timer)
8: **if** $(\forall \hat{m} \in Neigh(\hat{n}), (\hat{n}, \hat{m})_{av} < req_l)$ (if all connected substrate links are saturated) **then**
9: Reject the New bandwidth request
10: **else**
11: $\vec{P}_l.add(\hat{n})$ (save the new path)
12: **for** $\hat{m} \in Neigh(\hat{n}) \mid (\hat{n}, \hat{m})_{av} > req_l$ (for all attached substrate links having enough bandwidth) **do**
13: $(\hat{n}, \hat{m})_{av}$ -= req_l (reserve bandwidth in $(\hat{n}, \hat{m})$)
14: Add ($\hat{m}$, Search and reserve bandwidth, l, req_l) to the Controller database (activate $\hat{m}$)
15: **end for**
16: **end if**
17: **Second :** Perform the steps of the action *Trigger de-allocation* of Algorithm 1, with $req_l = 0$:
18: $(\hat{n}, \hat{m})_{av}$ += $\hat{n}_{allo_l}$ where $\hat{m} = \hat{n}_{\vec{P}_l}^{next}$ (release previously allocated bandwidth in the substrate link $(\hat{n}, \hat{m})$ where $\hat{m}$ is the next node in l's hosting path)
19: Add ($\hat{m}$, *Deallocate*, l, req_l) to the Controller database (Activate the next node in $\vec{P}_l$)
20: $L_{\hat{n}}.remove(l)$ (Remove l from $\hat{n}$ mapping list)
21: **end if**

- ***Action 2: Reserve bandwidth***:

This action aims at spreading the bandwidth reservation through l's hosting path. To do so, $\hat{n}$ performs the following steps depending on the situation:

- **IF** $\hat{n}$ is not the last node in $\vec{P}_l$, it checks if there is enough bandwidth on $(\hat{n}, \hat{o})$, where $\hat{o}$ is the next node in $\vec{P}_l$.
  - **IF** it is the case, it synchronizes with its soliciting node (the previous node in $\vec{P}_l$), then reserves bandwidth on $(\hat{n}, \hat{o})$ and activates $\hat{o}$ to continue bandwidth

reservation.

- **ELSE**, we conclude that l's supporting substrate path can not provide more bandwidth to meet req_l. In this case, previously reserved bandwidth among $\vec{P}_l$ should be released: node $\hat{n}$ activates $\hat{m}$ to do so (using action ***Release bandwidth v2***, described below, a slightly modified version of action *Release bandwidth* of Algorithm 2), where $\hat{m}$ is the previous node in $\vec{P}_l$.

- **ELSE** the end of the path is reached and required bandwidth was successfully reserved, and we only need to assign it to l. Hence, $\hat{n}$ synchronizes with its soliciting node, then updates l mapping ($\hat{n}_{allo_l} = req_l$), and spreads *back* through $\vec{P}_l$ an ***Update bandwidth allocation*** action, described below.

Algorithm 3: Increase in BW requirement

Action 2: Reserve BW (Priority=p_{Res})
Inputs:

1: **if** $\hat{n} \neq \vec{P}_l.end$ (if $\hat{n}$ is not the last node in the path) **then**
2: **if** $(\hat{n}, \hat{o})_{av} \geqslant (allo_l - req_l)$, where $\hat{o}$= $\vec{P}_l^{next}$, (if there is enough BW on $(\hat{n}.\hat{o})$, with $\hat{o}$ the next node in $\vec{P}_l$) **then**
3: $(\hat{n}, \hat{m})_{av}$ -= $(allo_l - req_l)$, where $\hat{m}$= $\vec{P}_l^{previous}$ (reserve bandwidth to synchronize with the soliciting node)
4: $(\hat{n}, \hat{o})_{av}$ -= $(allo_l - req_l)$ (reserve bandwidth on $(\hat{n}, \hat{o})$)
5: Add ($\hat{o}$, Reserve bandwidth, l, req_l) to the Controller database (activate $\hat{o}$ to Reserve bandwidth)
6: **else**
7: We conclude that l hosting substrate path can not support the new required bandwidth, hence
8: Add ($\hat{m}$, Release bandwidth v2, l, $(allo_l - req_l)$) to the Controller database (activate $\hat{m}$ to Release previously reserved bandwidth)
9: **end if**
10: **else**
11: ($\hat{n}$ is the last node in the path, hence)
12: $\hat{n}_{allo_l} = req_l$ (update l mapping, i.e. allocate the reserved bandwidth to l)
13: Add ($\hat{m}$, Update bandwidth allocation, l, req_l) to the Controller database (activate $\hat{m}$ to update l mapping through its hosting path)
14: **end if**

- ***Action 3: Release bandwidth v2***:

This action aims at releasing reserved bandwidth among l path. The same steps of action *Release bandwidth* of Algorithm 2 are used, except when $\hat{n}$ is the source node (hosting src_l),

Algorithm 3: Increase in BW requirement

Action 4: Update bandwidth allocation (Priority=p_U)

Inputs:

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

Steps:

1: $n_{allo_l} = req_l$ (update l mapping
2: **if** $src_l \notin N_{\hat{n}}$ (if $\hat{n}$ does not host l source, i.e. this is not the origin node) **then**
3: Add ($\hat{m}$, Allocate bandwidth, l, req_l), where to the Controller database, where $\hat{m}= \vec{P}_l^{previous}$ activate the previous node in the path to continue the update (because this action travels **back** through the substrate path)
4: **end if**

in fact, in this case, $\hat{n}$ concludes that l path can not support the new bandwidth requirements, and that a *virtual link migration* is required. Hence, it starts searching a new path for l and de-allocating l from its old path (like in the *Trigger reserving bandwidth* action). More detail can be found in the algorithm.

- ***Action 4: Update bandwidth allocation***:

This action simply serves to update l's mapping ($allo_l$) through the hosting path.

Action priority

Similarly to Algorithm 2, we choose the following priorities to schedule the previous actions:

- **Trigger reserving bandwidth**: p_{TR}
- **Reserve bandwidth** : p_{Res}
- **Release bandwidth** : p_{Relv2}
- **Update bandwidth allocation** : p_U

Such that $p_U < p_{Res} < p_{TR} < p_{Relv2}$. In fact, the order $p_U < p_{Relv2} < p_{TR}$ follows the chronological order of these actions execution: we first check if there are enough bandwidth in old path, then allocate more bandwidth if possible. Moreover, we always give releasing bandwidth the highest priority to avoid SN saturation when the reserved bandwidth is not used (p_{Relv2} is the highest).

As actions of all defined algorithms can be performed simultaneously, we will set the following order of priority to orchestrate their execution (see next table):

Algorithm 3: Increase in bandwidth requirement

Action 3: Release BW v2 (Priority=p_{Relv2})

Inputs:

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

Steps:

1: $(\hat{n}, \hat{o})_{av}$ += req_l, where $\hat{o}$= $\vec{P}_l.next$ (release previously reserved bandwidth in $(\hat{n}, \hat{o})$), where $\hat{o}$ is the soliciting node, it is the next node in $\vec{P}_l$ because this action travels **back** through the path
2: **if** $src_l \notin N_{\hat{n}}$ (if this is not the source node) **then**
3: $(\hat{n}, \hat{m})_{av}$ += req_l, where $\hat{m} = \hat{n}_{\vec{P}_l}^{previous}$ (release previously reserved bandwidth in $(\hat{n}, \hat{m})$), where $\hat{m}$ is the previous node in $\vec{P}_l$
4: Add ($\hat{m}$, Release bandwidth, l, req_l) to the Controller database (activate $\hat{m}$ to execute a release bandwidth action), i.e. propagate back the release bandwidth action
5: **else**
6: This is the source node, and a path migration is required to satisfy the new request:
7: **First :** Perform the steps of the action *Trigger searching and reserving bandwidth* of Algorithm 2:
8: Start $\hat{n}.Timer_l$ (start l Timer)
9: **if** $(\forall \hat{m} \in Neigh(\hat{n}), (\hat{n}, \hat{m})_{av} < req_l)$ (if all connected substrate links are saturated) **then**
10: Reject the New bandwidth request
11: **else**
12: $\vec{P}_l.add(\hat{n})$ (save the new path)
13: **for** $\hat{m} \in Neigh(\hat{n}) \mid (\hat{n}, \hat{m})_{av} > req_l$ (for all attached substrate links having enough bandwidth) **do**
14: $(\hat{n}, \hat{m})_{av}$ -= req_l (reserve bandwidth in $(\hat{n}, \hat{m})$)
15: Add ($\hat{m}$, Search and reserve bandwidth, l, req_l) to the Controller database (activate $\hat{m}$)
16: **end for**
17: **end if**
18: **Second :** Perform the steps of the action *Trigger deallocation* of Algorithm 1, with $req_l = 0$:
19: $(\hat{n}, \hat{m})_{av}$ += $\hat{n}_{allo_l}$ where $\hat{m} = \hat{n}_{\vec{P}_l}^{next}$ (release previously allocated bandwidth in the substrate link $(\hat{n}, \hat{m})$ where $\hat{m}$ is the next node in l's hosting path)
20: Add ($\hat{m}$, *Deallocate*, l, req_l) to the Controller database (Activate the next node in $\vec{P}_l$)
21: $L_{\hat{n}}.remove(l)$ (Remove l from $\hat{n}$ mapping list)
22: **end if**

Th selected order is as follows: we give the first actions of each algorithm the three highest priorities ($p_TD = 10$,$p_{TR} = 9$, $p_TSR = 8$) in order to favor dealing with different fluctuations simultaneously. Then, we enhance releasing no used reserved resources ($p_D = 7$, $p_Rel = 6$, $p_{Relv2} = 5$). Finally, we respect the previously selected order for Algorithms 2 and 3.

Algorithm 1: Decrease in bandwidth requirements/ Link Removal		
Action1: *Trigger de-allocation*	$p_{TD} = 10$	Starts the de-allocation of an embedded virtual link
Action2: *De-allocation*	$p_D = 7$	Continuous the de-allocation of an embedded virtual link
Algorithm 2: Link Addition		
Action1: *Trigger searching and reserving bandwidth*	$p_{TSR} = 8$	Starts searching an available path to host a non embedded virtual link
Action2: *Search and reserve bandwidth*	$p_{SR} = 1$	Continuous searching an available path to host a non embedded virtual link
Action3: *Allocate bandwidth*	$p_{All} = 4$	Continuous allocating a virtual link in an already reserved path (this action travels **back** among the substrate nodes hosting the reserved path)
Action4: *Release bandwidth*	$p_{Rel} = 6$	Continuous releasing bandwidth from a reserved path (this action travels **back** among the substrate nodes hosting the reserved path)
Algorithm 3: Increase in Bandwidth Requirement		
Action1: *Trigger reserving bandwidth*	$p_{TR} = 9$	Starts checking if enough bandwidth exist in the old hosting path of an evolving link and reserves this bandwidth
Action2: *Reserve bandwidth*	$p_{Res} = 3$	Continuous reserving bandwidth through the old hosting path of an evolving link
Action3: *Release bandwidth v2*	$p_{Relv2} = 5$	Continuous releasing reserved bandwidth in the old hosting of an evolving link, then triggers searching for a new hosting path (this action travels **back** among the substrate nodes hosting the reserved path)
Action4: *Update bandwidth allocation*	$p_U = 2$	Continuous updating the evolving link mapping through its hosting path (this action travels **back** among the substrate nodes of the hosting path)

Table 5.4: Summary of all devised actions

5.5 Simulation results and evaluation

In this section, we will evaluate and validate the effectiveness of our proposed framework and algorithms by conducting extensive simulations. To achieve this, we will first describe the simulation environment and present the used performance parameters. Then, we will present our main simulation results.

5.5.1 Simulation environment

We adjusted the C++ simulator used in previous chapters to fit our scenario: substrate nodes with local view, and a round per round execution of the algorithms.

As in previous chapters, the GT-ITM tool (Zegura *et al.* (1996)) is used to generate random topologies of the substrate and VN networks. The SN (Substrate Network) 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 50%. The node resource capacity is randomly selected between 0 and 20 and the edge resource capacity is uniformly distributed between 0 and 50.

As in chapter 3, 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 $Convergence\_Time$ later. Other metrics will be presented later.

5.5.2 Simulation results

This section presents preliminary results of simulations conducted to evaluate our proposal.

5.5.2.1 Algorithm1: Decrease in Bandwidth requirement (DBR) or Link Removal (LD)

We simulate two scenarios to figure out the effectiveness of this algorithm:

- **Case 1: only one bandwidth fluctuation is considered**

In the first case, only one bandwidth fluctuation is considered, and results showed that, in accordance with the example 5.10, $Convergence\_Time$ only depends on the number of substrate nodes supporting the substrate path hosting the evolving virtual link. In more detail: $Convergence\_Time = NbNodes_{Path}$, (see fig 5.11), where $NbNodes_{Path}$ is the number of nodes in the hosting path.

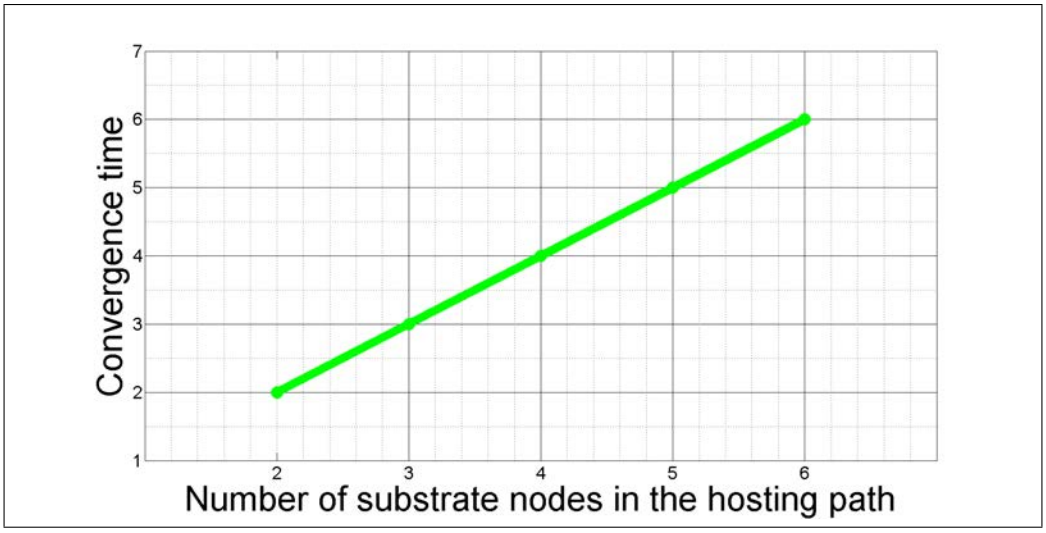


Figure 5.11: DBR or LR: case of only one bandwidth fluctuation

- **Case 2: Multiple bandwidth fluctuations occur simultaneously**

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

Note that, as explained in the example , if only one bandwidth request is considered, the $Convergence\_Time$ depends on the length (in term of number of nodes) of the substrate path hosting the evolving link. Hence, in case of multiple bandwidth demands, the $Convergence\_Time$ will be *at least equal to* the number of substrate nodes supporting *the longest path* among those hosting the evolving links.

With this idea in mind, we try to evaluate the performance of our algorithm in handling *multiple* bandwidth fluctuations at the same time. To do so, we consider N bandwidth requests, and 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.

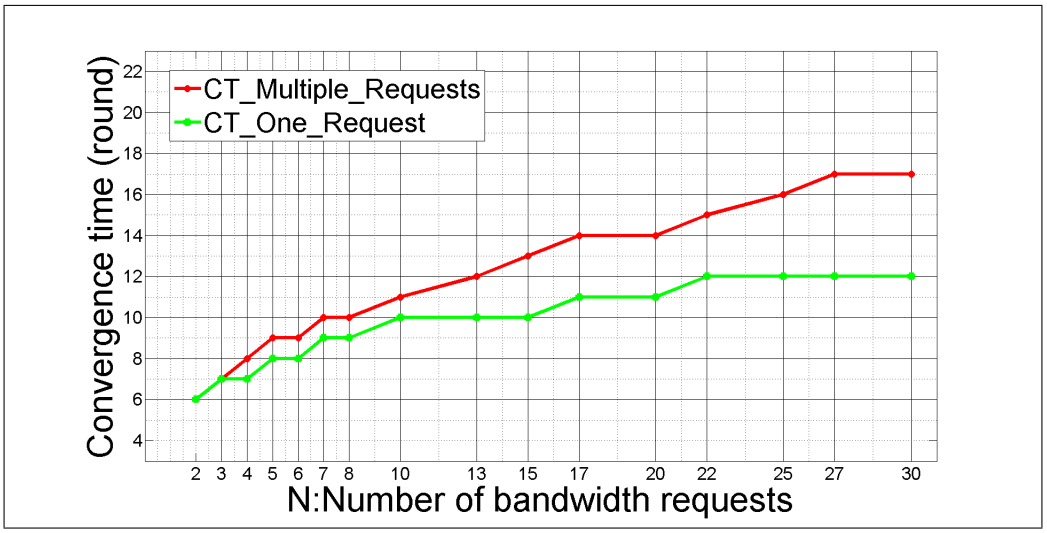


Figure 5.12: DBR or LR: Case of multiple bandwidth fluctuations

Figure 5.12 depicts the results of 200 averaged runs and shows that the convergence time increases with the number of bandwidth requests for both *CT_Multiple requests* and *CT_One request*. Moreover, note that the gap between the convergence time of the two scenarios is small ($CT\_Multiple\_Requests/CT\_One\_Request < 1.4$), compared to the number N of requests handled in the second case: 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). Note also that $CT\_One\_Request$ stabilizes at 12, which is the number of substrate nodes supporting the *longest* hosting path in the network.

In order to understand better the behavior of the algorithm, we plot the number of substrate nodes executing simultaneously in each round, called $Nb\_Executing\_Nodes$ for $N = 10$, N=15, $N = 20$ and $N = 25$.

Figure 5.13 depicts the results of 200 averaged runs and shows that all the curves have the same shape: the number of executing nodes is initially high, then drops sharply in the second round, thereafter it increases again in the third and fourth rounds and finally drops off linearly until reaching stability.

This can be explained as follows: initially, all the active nodes in the system are scheduled to execute the same action (*Trigger de-allocation*), hence have the same priority. As there is generally at least a node wishing to execute *only one action*, the Controller selects all the active nodes to run *one* action (note that $Nb\_Executing\_Nodes$ in the first round is not equal to the N, the total number of handled requests as some nodes are active for more then an action). In the second round, there will be substrate nodes active to execute the *Trigger*

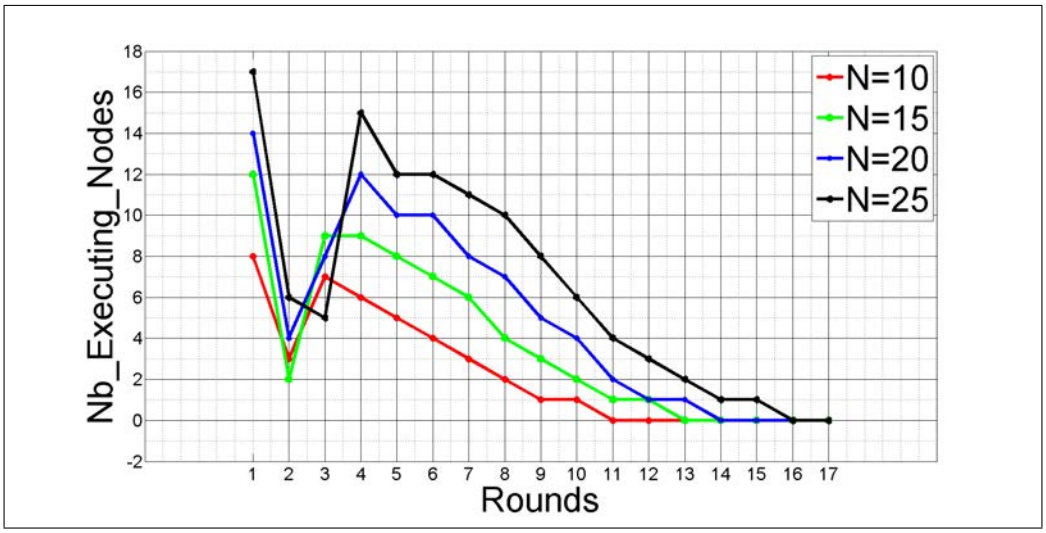


Figure 5.13: DBR or LD: Number of executing nodes per round

de-allocation action (those remaining from the previous round, i.e. nodes supporting more then a virtual link with new bandwidth requirements), and nodes active to execute the action *De-allocate*, which were activated by the executing nodes of the first round.

Hence the nodes scheduled for the *Trigger de-allocation* are selected to execute first, because they have the highest priority, which explains the sharp decrease of $Nb\_Executing\_Nodes$ in round 2, as the majority of active nodes for *Trigger de-allocation* already performed in the first round. Since the fourth round, all the active nodes are scheduled to execute the same action and thus are always selected to run simultaneously. $Nb\_Executing\_Nodes$ will then decrease progressively as the request of the virtual links with shortest hosting paths will be met rapidly.

5.5.2.2 Algorithm 2: Link addition (LA)

Remind that this is the case where a new virtual link is added to the VN topology. To meet the request, our proposal searches available paths to connect the source and destinations nodes, then select the most cost effective one for embedding. In this simulation, we will concentrate on the case where *only one new bandwidth request* is submitted, and we will examine three metrics to evaluate the performance of our algorithm: *the evolving requests acceptance ratio*, *the virtual link embedding cost* and *the convergence time*.

In order 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 both nodes composing each pair belong to the same VN and are non adjacent (not connected by a virtual link, in order to avoid multi-graphs).

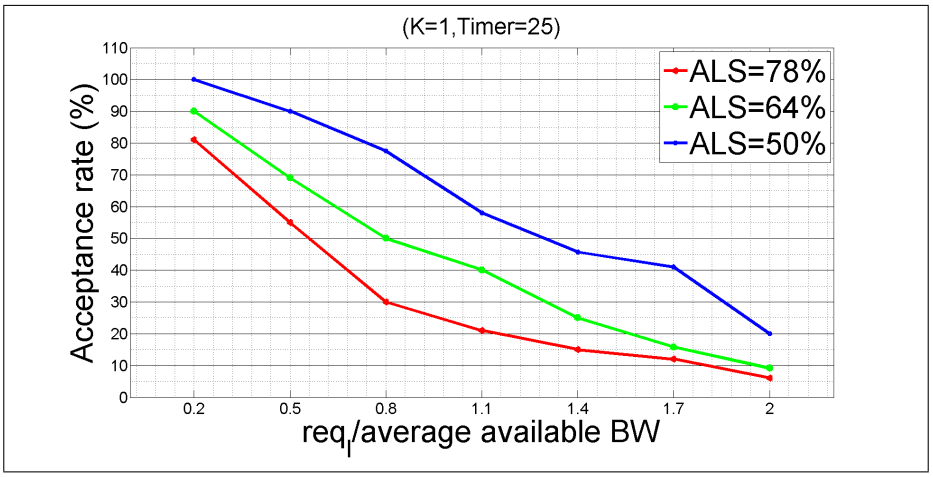


Figure 5.14: LA:Acceptance ratio depending on req_l

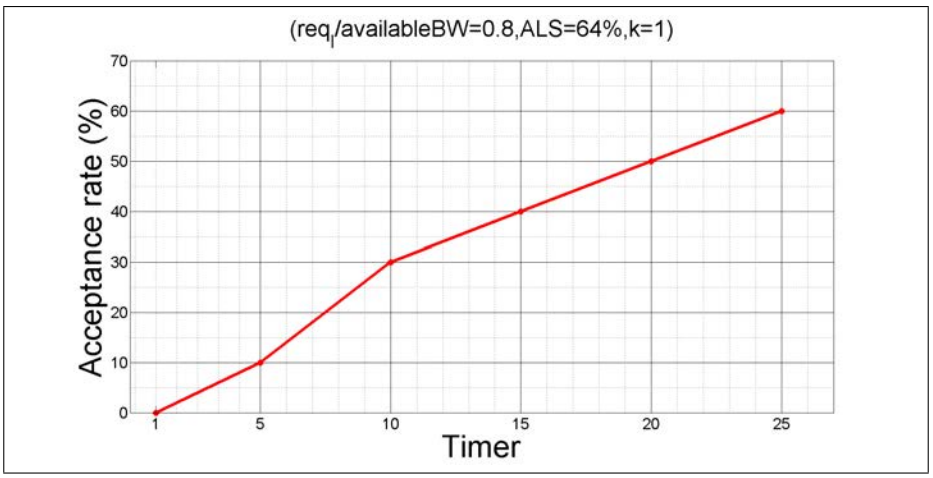


Figure 5.15: LA: Acceptance ratio depending on Timer

- **Evolving requests acceptance ratio**

We will evaluate the acceptance ratio of evolving requests depending on three parameters: i) the substrate network saturation, ii) the amount of new required bandwidth, and iii) the evolving virtual link Timer (the maximum allowed time to search for a path solution). All the figures will depict the results of 100 averaged runs.

First, we measure the acceptance ratio of evolving demands in three substrate networks *with different substrate links saturation*. We keep the same definition of *the average links saturation* as in section 4 4.2, and consider three values: $ALS = 78\%$, 64% and 50%. In each scenario, we calculate the average available bandwidth on substrate links $Average\_available\_BW$, and consider different values of new bandwidth demand req_l, such that the ratio $req_l/Average\_available\_BW$ is equal to 0.2, 0.5, 0.8 ... 2. and measure the acceptance ratio for different values of this ratio:

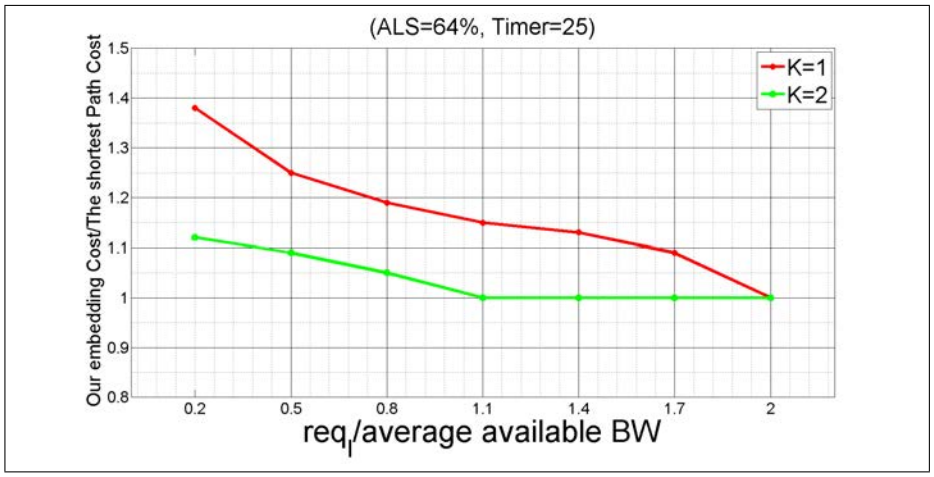


Figure 5.16: LA: Embedding cost

Figure 5.14 shows that the acceptance ratio decreases when req_l increases for all scenarios, in fact, when the bandwidth demand is small, it is easier to find a hosting path. Moreover, notice that the best acceptance ratios are obtained when the substrate links are less congested, this is predictable as there is more available bandwidth in such network.

Second, we fix the ALS to 64%, and the $req_l/Average\_available\_BW$ to 0.8, and measure the acceptance ratio for different values of $Timer$ (in terms of rounds). Figure 5.14 shows that the acceptance ratio increases with the Timer, in fact, the more time we have to search for an available path, the more chance we find a solution.

- **Virtual link embedding cost**

Now we will evaluate the cost-efficiency of our proposal. To do so, recall that end node of an evolving virtual link should wait for K path proposals to select the most cost effective one. We will measure the virtual link embedding cost for $K = 1$ and $K = 2$, for different values of new bandwidth demand req_l, and compare it to the shortest path cost, found with a global view of the system. Figure 5.15 shows the ratio of *the embedding cost found with our algorithm*, and that of *the shortest path algorithm*,

$Our\_embedding\_Cost/The\_shortest\_Path\_Cost$ for different values of

$req_l/Average\_available\_BW$, for 100 accepted requests.

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 (for $K = 1$ and $K = 2$) 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. For instance, for $K = 2$,

we see that since $req_l/Average\_available\_BW = 1.1$, the two algorithms have the same embedding cost, that means that there are at most two available paths, so if we find both of them, we necessarily find the shortest path.

- **Convergence time**

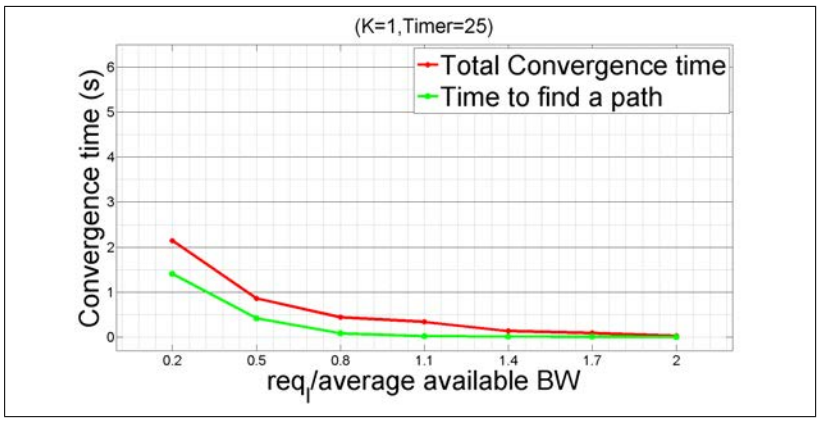


Figure 5.17: LA: Convergence Time, ALS=78%

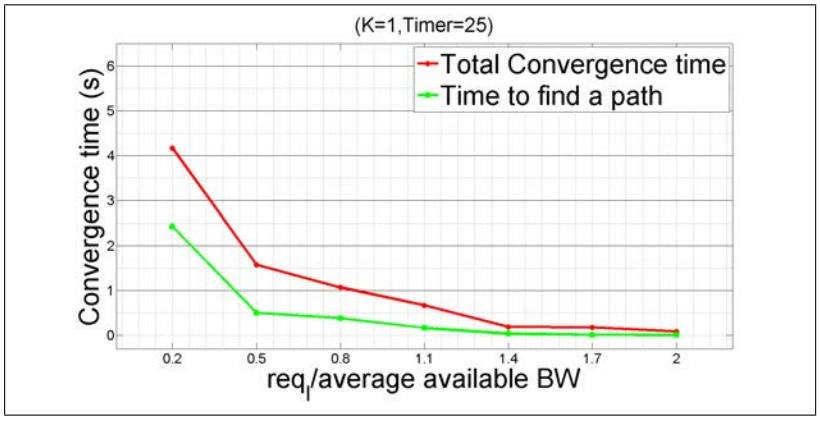


Figure 5.18: LA: Convergence Time, ALS=64%

Remember that this algorithm is composed of two steps: a first step to search for available paths, and a second step to embedd the selected path and release bandwidth from other paths. In this simulation, we will measure two times: the duration of the first phase: the time required to find a path solution, and the total convergence time, i.e. the total time required to reach stability (after completing the first and second step). We set K to one, and $Timer$ to 25 rounds, and make evaluation for three SN configurations: $ALS = 78\%$, 64% and 50%. We will measure the time in seconds for more precision as all rounds do not necessarily have the same duration (depending on the number and type of executed actions in each round).

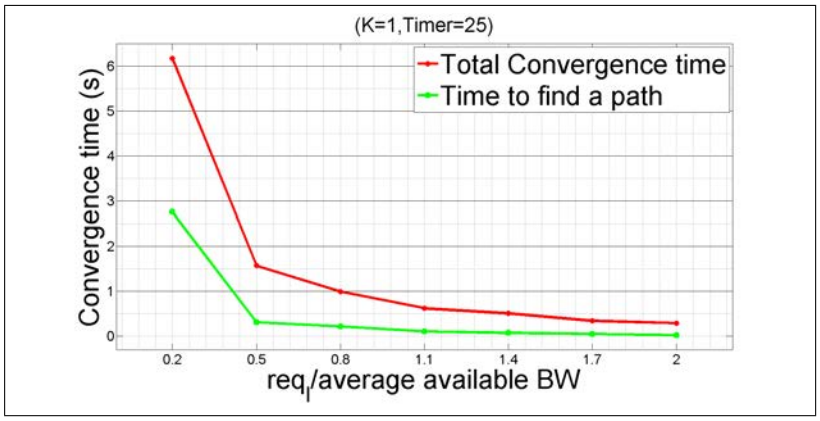


Figure 5.19: LA: Convergence Time, ALS=50%

Figures 5.16, 5.17, 5.18 depict the results of 100 averaged runs and show that the *Total convergence time* and the *Time to find a path* decrease when req_l increases. This is explained by the fact that, when the required bandwidth is high, there are few available links, hence few substrate nodes will be activated to search for a path, and thus 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. For instance, for $ALS = 50\%$ and $req_l/Average\_available\_BW = 0.2$, the convergence time exceeds 6 seconds.

Moreover, notice that both the *Total convergence time* and the *Time to find a path* decrease with the average link saturation, for the same reasons explained above. Finally, note that the gap between *Total convergence time* and the *Time to find a path* increases with ALS, we can explain this as follows: 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.

5.6 Conclusion

In this chapter, 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 in a time effective way. Moreover, our distributed algorithms find solutions (cost effective paths to embedd new virtual links added to the VN topology) that are very close to the global solutions (using a global view) .

Chapter 6

Conclusion and Future Research Directions

Contents

This chapter outlines the contributions of this thesis and discusses the work to be carried out in the future. In section 6.1, we will summarize the proposals described in this thesis, then we will formulate possible research for the future in section 6.2.

6.1 Conclusion and discussion

In this thesis, we addressed the virtual network resource provisioning problem, with a focus on the virtual and substrate resource management. In fact, we seperate the virtual network resource provisioning issue into two sub-problems: the initial virtual network embedding (VNE) that aims at finding an optimal mapping between virtual nodes and links and substrate nodes and links, and the *dynamic management of virtual and substrate resources* that deals with resource demand fluctuation of embedded virtual networks, and the re-optimization of the substrate network usage. The key contributions of the thesis are listed below:

- A heuristic algorithm that deals with virtual nodes demand fluctuations. It manages the case where an embedded virtual node requires more resources, whereas the hosting substrate node does not have enough available resources. The main idea of the algorithm is to re-allocate one or more co-located virtual nodes from the substrate node, hosting the evolving node, to free resources (or make room) for the evolving node.

In addition to minimizing the re-allocation cost, our proposal takes into account the service interruption during migration and reduces it.

- The previous algorithm was extended to design a preventive re-configuration scheme to enhance substrate network profitability. In fact, our proposal "takes advantage" of the resource demand perturbation to tidy up the SN at minimum cost and disruptions. When re-allocating virtual nodes to make room for the extending node, we shift the most congested virtual links to less saturated substrate resources to balance the load among the Substrate network. Our proposal offers the best trade off between re-allocation cost and load balancing performance.

- A distributed, local-view and parallel framework was devised to handle all forms of bandwidth demand fluctuations of the embedded virtual links. It is composed of a Controller and three algorithms running in each substrate node in a distributed and parallel manner. The framework is based on the self-stabilization approach, and can manage many and different forms of bandwidth demand variations simultaneously.

6.2 Future research directions

Suggested future research work resulting from this thesis can be summarized as follows:

- **Enhance the previous contributions by:**
  - Extending the self-stabilizing framework to manage the node demand fluctuations: in fact, when re-allocating a virtual node, its attached links should be re-embedded too. To do so, the proposed algorithms for allocating virtual links and deleting others can be used.
  - Boosting the Controller performance to allow the execution of different types of actions simultaneously, to reduce the convergence time.
  - Making more exhaustive simulations on the self-stabilizing algorithms to better evaluate their performance and understand their limits in different conditions, for instance in case of *multiple* bandwidth requests of *different types*.
  - Improving the substrate network profitability at the same time as managing the bandwidth demand fluctuation, by balancing the load among substrate links. To do so, the cos unit of substrate links can be defined according to their stress.

  - Expanding the contribution of chapter 3 to manage bandwidth demand fluctuation.
  - Managing the case where a whole sub-graph is added to the embedded virtual network

- **Explore other Virtual Network resource provisioning problems**, namely the Substrate Network Survivability problem Rahman *et al.* (2010)

Bibliography

Amshavalli, R.S., & Kavitha, G. 2014 (May). Increasing the availability of cloud resources using broker with semantic technology. *Pages 1578–1582 of: Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on.*

Andersen, David G. 2002. *Theoretical Approaches To Node Assignment.*

Armbrust, Michael, Fox, Armando, Griffith, Rean, Joseph, Anthony D., Katz, Randy, Konwinski, Andy, Lee, Gunho, Patterson, David, Rabkin, Ariel, Stoica, Ion, & Zaharia, Matei. 2009 (February). *Above the Clouds: A Berkeley View of Cloud Computing.* Tech. rept. University of California at Berkeley.

Baveja, Alok, & Srinivasan, Aravind. 2000. Approximation Algorithms for Disjoint Paths and Related Routing and Packing Problems. *Math. Oper. Res.*, **25**(2), 255–280.

Blenk, Andreas, & Kellerer, Wolfgang. 2013. Traffic Pattern Based Virtual Network Embedding. *In: Proceedings of the 2013 Workshop on Student Workhop.*

Botero, J.F., & Hesselbach, X. 2009 (Sept). The bottlenecked virtual network problem in bandwidth allocation for network virtualization. *Pages 1–5 of: Communications, 2009. LATINCOM '09. IEEE Latin-American Conference on.*

Boyd, S., Xiao, L., Mutapcic, A., & Mattingley, J. 2006. Notes on Decomposition Methods. *Stanford University.*

Buyya, Rajkumar, Yeo, Chee Shin, Venugopal, Srikumar, Broberg, James, & Brandic, Ivona. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Syst.*, **25**(6), 599–616.

Carter, R.L., St.Louis, D., & Andert, E.P., Jr. 1998 (Oct). Resource allocation in a distributed computing environment. *Pages C32/1–C32/8 vol.1 of: Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, vol. 1.

Cheng, Xiang, Su, Sen, Zhang, Zhongbao, Wang, Hanchi, Yang, Fangchun, Luo, Yan, & Wang, Jie. 2011. Virtual network embedding through topology-aware node ranking. *Computer Communication Review*, **41**(2), 38–47.

Chowdhury, M., Rahman, M.R., & Boutaba, R. 2012. ViNEYard: Virtual Network Embedding Algorithms With Coordinated Node and Link Mapping. *Networking, IEEE/ACM Transactions on*, **20**, 206–219.

Chowdhury, N.M.M.K., & Boutaba, R. 2009. Network virtualization: state of the art and research challenges. *Communications Magazine, IEEE*, **47**(7), 20–26.

Costa, Paolo, Migliavacca, Matteo, Pietzuch, Peter, & Wolf, Alexander L. 2012. NaaS: Network-as-a-Service in the Cloud. *In: Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*.

Dab, B., Fajjari, I., Aitsaadi, N., & Pujolle, G. 2013 (Dec). VNR-GA: Elastic virtual network reconfiguration algorithm based on Genetic metaheuristic. *Pages 2300–2306 of: Global Communications Conference (GLOBECOM), 2013 IEEE*.

Di, Hao, Yu, Hongfang, Anand, Vishal, Li, Lemin, Sun, Gang, & Dong, Binhong. 2012. Efficient Online Virtual Network Mapping Using Resource Evaluation. *J. Network Syst. Manage.*, **20**(4), 468–488.

Dijkstra, Edsger W. 1974. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, **17**(11), 643–644.

Dubois, Swan, & Tixeuil, Sebastien. 2011. A Taxonomy of Daemons in Self-stabilization. *CoRR*, **abs/1110.0334**.

Eppstein, David. 1999. Finding the K Shortest Paths. *SIAM J. Comput.*, **28**(2), 652–673.

Esposito, F., Di Paola, D., & Matta, I. 2014. On Distributed Virtual Network Embedding With Guarantees. *Networking, IEEE/ACM Transactions on*, **PP**(99), 1–1.

Fajjari, I, Aitsaadi, N., Pujolle, G., & Zimmermann, H. 2011a. VNE-AC: Virtual Network Embedding Algorithm Based on Ant Colony Metaheuristic. *In: (ICC) 2011.*

Fajjari, I, Aitsaadi, N., Pujolle, G., & Zimmermann, H. 2012 (Dec). Adaptive-VNE: A flexible resource allocation for virtual network embedding algorithm. *Pages 2640–2646 of: Global Communications Conference (GLOBECOM), 2012 IEEE.*

Fajjari, Ilhem, Aitsaadi, Nadjib, Pujolle, Guy, & Zimmermann, Hubert. 2011b. Vnr algorithm: A greedy approach for virtual networks reconfigurations. *Pages 1–6 of: Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE.* IEEE.

Farooq Butt, Nabeel, Chowdhury, Mosharaf, & Boutaba, Raouf. 2010. Topology-Awareness and Reoptimization Mechanism for Virtual Network Embedding. *In: NETWORKING 2010.* Springer Berlin Heidelberg.

Fischer, A, Botero, J.F., Till Beck, M., de Meer, H., & Hesselbach, X. 2013. Virtual Network Embedding: A Survey. *Communications Surveys Tutorials, IEEE*, **15**(4), 1888–1906.

Gambette, J. Beauquierand P. 2006. Introduction à l'algorithmique répartie et à l'auto-stabilisation.

Gmach, Daniel, Rolia, Jerry, Cherkasova, Ludmila, & Kemper, Alfons. 2007. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. *Pages 171–180 of: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization.* IISWC '07. Washington, DC, USA: IEEE Computer Society.

He, Jiayue, Zhang-Shen, Rui, Li, Ying, Lee, Cheng-Yen, Rexford, Jennifer, & Chiang, Mung. 2008. Davinci: Dynamically adaptive virtual networks for a customized internet. *In: Proceedings of the 2008 ACM CONEXT Conference.*

Hochbaum, Dorit S., & Shmoys, David B. 1986. A unified approach to approximation algorithms for bottleneck problems. *J. ACM*, **33**(3), 533–550.

Houidi, I, Louati, W., & Zeghlache, D. 2008 (May). A Distributed Virtual Network Mapping Algorithm. *Pages 5634–5640 of: Communications, 2008. ICC '08. IEEE International Conference on.*

Houidi, Ines, Louati, Wajdi, Ben Ameur, Walid, & Zeghlache, Djamal. 2011. Virtual Network Provisioning Across Multiple Substrate Networks. *Comput. Netw.*, **55**, 1011–1023.

Jmila, Houda, & Zeghlache, Djamal. 2015 (Jan.). An Adaptive Load Balancing Scheme for Evolving Virtual Networks. *Pages 494–500 of: 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC) (CCNC 2015).*

Jmila, Houda, Houidi, Ines, & Zeghlache, Djamal. 2014 (June). RSforVNE: Node Reallocation Algorithm for Virtual Networks Adaptation. *In: 19th IEEE Symposium on Computers and Communications (IEEE ISCC 2014).*

Jmila, Houda, Drira, Kaouther, & Zeghlache, Djamal. 2016. A Self-Stabilizing framework for dynamic bandwidth allocation in Virtual Networks. *IEEE/IFIP Network Operations and Management Symposium (NOMS).*

Kapil, D., Pilli, E.S., & Joshi, R.C. 2013. Live virtual machine migration techniques: Survey and research challenges. *In: Advance Computing Conference (IACC), 2013 IEEE 3rd International.*

Kolliopoulos, Stavros G., & Stein, Clifford. 1997. Improved Approximation Algorithms for Unsplittable Flow Problems. *Pages 426–435 of: FOCS.* IEEE Computer Society.

Krauter, Klaus, Buyya, Rajkumar, & Maheswaran, Muthucumaru. 2002. A taxonomy and survey of grid resource management systems for distributed computing. *Softw., Pract. Exper.*, **32**(2), 135–164.

Lu, Jing, & Turner, Jonathan. 2006. *Efficient Mapping of Virtual Networks onto a Shared Substrate.* Tech. rept. Washington University in St. Louis.

Mahmood, Zaigham, & Hill, Richard. 2011. *Cloud Computing for Enterprise Architectures.* London: Springer.

Marquezan, C.C., Nobre, J.C., Granville, L.Z., Nunzi, G., Dudkowski, D., & Brunner, M. 2009. Distributed Reallocation Scheme for Virtual Network Resources. *In: Communications, 2009. ICC '09. IEEE International Conference on.*

Marquezan, C.C., Granville, L.Z., Nunzi, G., & Brunner, M. 2010. Distributed autonomic resource management for network virtualization. *In: Network Operations and Management Symposium (NOMS), 2010 IEEE.*

Mell, Peter, & Grance, Timothy. 2011 (September). *The NIST Definition of Cloud Computing.* Tech. rept. 800-145. National Institute of Standards and Technology (NIST), Gaithersburg, MD.

Mijumbi, R., Gorricho, J.-L., Serrat, J., Claeys, M., Famaey, J., & De Turck, F. 2014a (June). Neural network-based autonomous allocation of resources in virtual networks. *Pages 1–6 of: Networks and Communications (EuCNC), 2014 European Conference on*.

Mijumbi, Rashid, Gorricho, Juan-Luis, Serrat, Joan, Claeysy, Maxim, Turcky, Filip De, & Latré, Steven. 2014b. Design and Evaluation of Learning Algorithms for Dynamic Resource Management in Virtual Networks. *Network Operations and Management Symposium (NOMS 2014)*.

Mijumbi, Rashid, Gorricho, Juan-Luis, Serrat, Joan, Shen, Meng, Xu, Ke, & Yang, Kun. 2015. A neuro-fuzzy approach to self-management of virtual network resources. *Expert Syst. Appl.*, **42**(3), 1376–1390.

Nürnberger, Andreas. 2001. *A Hierarchical Recurrent Neuro-Fuzzy System*.

Parashar, Manish, & Hariri, Salim. 2005. Autonomic computing: An overview. *Pages 247–259 of: Unconventional Programming Paradigms*. Springer Verlag.

Rahman, MuntasirRaihan, Aib, Issam, & Boutaba, Raouf. 2010. Survivable Virtual Network Embedding. *Pages 40–52 of:* Crovella, Mark, Feeney, LauraMarie, Rubenstein, Dan, & Raghavan, S.V. (eds), *NETWORKING 2010*. Lecture Notes in Computer Science, vol. 6091. Springer Berlin Heidelberg.

Razzaq, A., & Rathore, M.S. 2010 (Sept). An Approach towards Resource Efficient Virtual Network Embedding. *Pages 68–73 of: Evolving Internet (INTERNET), 2010 Second International Conference on*.

Schneider, Marco. 1993. Self-stabilization. *ACM Comput. Surv.*, **25**(1), 45–67.

Seddiki, M.S., Nefzi, B., Song, Ye-Qiong, & Frikha, M. 2013 (Dec). Automated controllers for bandwidth allocation in network virtualization. *Pages 1–7 of: Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*.

Sun, Gang, Anand, V., Yu, Hong-Fang, Liao, Dan, Cai, Yanyang, & Li, Le Min. 2012 (Dec). Adaptive provisioning for evolving virtual network request in cloud-based datacenters. *Pages 1617–1622 of: Global Communications Conference (GLOBECOM), 2012 IEEE*.

Sun, Gang, Yu, Hongfang, Anand, Vishal, & Li, Lemin. 2013. A cost efficient framework and algorithm for embedding dynamic virtual network requests. *Future Generation Comp. Syst.*, **29**, 1265–1277.

Sutton, Richard S., & Barto, Andrew G. 1998. *Reinforcement Learning I: Introduction.*

Till Beck, M., Fischer, A., de Meer, H., Botero, J.F., & Hesselbach, X. 2013 (June). A distributed, parallel, and generic virtual network embedding framework. *Pages 3471–3475 of: Communications (ICC), 2013 IEEE International Conference on.*

Tran, Phuong Nga, Casucci, Leonardo, & Timm-Giel, Andreas. 2012. Optimal mapping of virtual networks considering reactive reconfiguration. *In: Cloud Networking (CLOUDNET), 2012.*

Tran, P.N., & Timm-Giel, A. 2013. Reconfiguration of virtual network mapping considering service disruption. *In: (ICC), 2013.*

Wang, Anjing, Iyer, M., Dutta, R., Rouskas, G.N., & Baldine, I. 2013. Network Virtualization: Technologies, Perspectives, and Frontiers. *Lightwave Technology, Journal of*, **31**(4), 523–537.

Wei, Yongtao, Wang, Jinkuan, Wang, Cuirong, & Hu, Xi. 2010 (Sept). Bandwidth Allocation in Virtual Network Based on Traffic Prediction. *Pages 1–4 of: Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference on.*

Xu, Li, Tan, Guozhen, & Zhang, Xia. 2014a (Oct). A cost sensitive approach for Virtual Network reconfiguration. *Pages 191–196 of: Computing, Communications and IT Applications Conference (ComComAp), 2014 IEEE.*

Xu, Zichuan, Liang, Weifa, & Xia, Qiufen. 2014b. Efficient virtual network embedding via exploring periodic resource demands. *Pages 90–98 of: LCN.* IEEE Computer Society.

Yu, Minlan, Yi, Yung, Rexford, Jennifer, & Chiang, Mung. 2008. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. *SIGCOMM Comput. Commun. Rev.*, **38**, 17–29.

Zegura, E.W., Calvert, K.L., & Bhattacharjee, S. 1996. How to model an internetwork. *In: INFOCOM.*

Zhang, Min, Wu, Chunming, Yang, Qiang, & Jiang, Ming. 2012. Robust dynamic bandwidth allocation method for virtual networks. *Pages 2706–2710 of: Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012.*

Zhang, Qi, Cheng, Lu, & Boutaba, Raouf. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, **1**(1), 7–18.

Zhang, Sheng, Qian, Zhuzhong, Wu, Jie, & Lu, Sanglu. 2014a (Aug). Leveraging tenant flexibility in resource allocation for virtual networks. *Pages 1–8 of: Computer Communication and Networks (ICCCN), 2014 23rd International Conference on.*

Zhang, Sheng, Qian, Zhuzhong, Wu, Jie, Lu, Sanglu, & Epstein, L. 2014b. Virtual Network Embedding with Opportunistic Resource Sharing. *Parallel and Distributed Systems, IEEE Transactions on*, **25**(3), 816–827.

Zhani, M.F., Zhang, Qi, Simon, G., & Boutaba, R. 2013. VDC Planner: Dynamic migration-aware Virtual Data Center embedding for clouds. *In: Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on.*

Zhou, Ye, Li, Yong, Sun, Guang, Jin, Depeng, Su, Li, & Zeng, Lieguang. 2010a (Dec). Game Theory Based Bandwidth Allocation Scheme for Network Virtualization. *Pages 1–5 of: Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE.*

Zhou, Ye, Li, Yong, Jin, Depeng, Su, Li, & Zeng, Lieguang. 2010b. A virtual network embedding scheme with two-stage node mapping based on physical resource migration. *In: Communication Systems (ICCS), 2010 IEEE International Conference on.*

Zhou, Ye, Yang, Xu, Li, Yong, Jin, Depeng, Su, Li, & Zeng, Lieguang. 2013. Incremental Re-Embedding Scheme for Evolving Virtual Network Requests. *Communications Letters, IEEE*, **17**, 1016–1019.

Zhu, Yong, & Ammar, Mostafa H. 2006. Algorithms for Assigning Substrate Network Resources to Virtual Network Components. *In: INFOCOM.*