Horizon Project

ANR call for proposals number ANR-08-VERS-010
FINEP settlement number 1655/08

Horizon - A New Horizon for Internet

WP3 - TASK 3.1: Service and Resource Overlay Algorithms

Institutions

Brazil	**France**
GTA-COPPE/UFRJ	LIP6 Université Pierre et Marie Curie
PUC-Rio	Telecom SudParis
UNICAMP	Devoteam
Netcenter Informática LTDA.	Ginkgo Networks
	VirtuOR

Contents

List of Figures

Chapter 1

Introduction

The Work Package 3 defines the piloting system needed to handle network emerging problems, such as congestion, failures, QoS requirements, etc. Hence, this work package introduces a set of algorithms and intelligent agents to manage virtual networks taking into account the fulfillment of their individual service level agreements. To reach this goal, this work package must face obstacles such as dealing with the changing requirements of the different virtual networks and also it must deal with the conflicting objectives between infrastructure providers and virtual networks, as well as between co-existent virtual networks sharing the same physical substrate.

The first task of Work Package 3, described in this report, is to investigate the algorithms to control the virtual networks. Data networks represent a dynamic and complex area, in which managers face new problems and challenges every day. The increase of the network complexity and of the amount of information collected make resources and network control more and more difficult. We argue that in such an unpredictable, changing and open environment, a dynamic control gives the opportunity to obtain an optimized network management and monitoring. In this project, we are interested in the adaptive network monitoring approach, which monitors the network state and controls its different components. We aim at including intelligent and dynamic control to guarantee QoS and to provide better network management and global performance.

The algorithms defined in this task consider resources availability and Service Level Agreements (SLAs). These algorithms are designed within the piloting system to guarantee that the agreements of all virtual networks are respected, even in the presence of misbehaved virtual networks that violate their SLAs.

1.1 Related Research Efforts

The control and management of virtual networks can be divided into local and global network control. The global control includes operations such as instantiating virtual nodes and virtual links, as well as virtual network migration. The local control monitors the resources of each physical node that are assigned to each virtual network, dealing, for instance, with isolation between networks.

Schaffrath et al. proposed an architecture for global control of virtualized networks [1]. The proposal was implemented with Xen and assumes a centralized control that creates slices in the network, through the instantiation of virtual machines and virtual links. Thus, upon receiving a request to allocate a new network, the system contacts each of the selected physical nodes and instantiates the virtual machines needed to build the network as well as the virtual links between them through IP tunneling. Other similar approaches to global control are found in the virtualization-based testbeds, such as GENI [2]. The access of the researcher to the testbed is controlled through a central entity called the Clearing House. The Clearing House monitors which physical nodes and services are available in each of the federated testbeds, who is authorized to use them, and which slices are scheduled for each researcher.

Entities with global control also perform other functions, such as migration. Houid et al. proposed a global control system based on multi-agents for dynamic resource allocation to virtual networks through the use of migration [3]. The system monitors the available resources on each physical machine as well as the changing demands of virtual networks. Noting that resources are scarce, the agent on the physical node searches a similar physical node to receive one or more of its virtual machines.

The global network control systems do not deal with the sharing of resources within the physical machine, assuming that the slices are isolated by a locally controlled mechanism. Egi et al. investigate the construction of a platform of virtual routers using Xen and Click [4] as local control, evaluating the provision of isolation and fairness between the networks [5]. The authors investigate the use of different data planes, assuming routing through a privileged domain and through a virtual machine, and evaluate the ability to share resources among virtual networks. The authors extended the CPU scheduler of Click to evaluate the CPU costs of packet forwarding. This work, however, has no mechanisms for defining management capabilities to specify the amount of resources to each virtual network. Also, the authors' proposal does not differentiate or prioritize traffic to ensure QoS in the virtual networks.

Another important aspect of local control is the guarantee of isolation between virtualized environments. Xen has problems not only with respect to the Input/Output (I/O) operations, but also with other aspects such as fairness [6, 7]. Jin et al. proposed a mechanism to ensure fairness in the use of L2 and L3 caches on Xen, whose use is not contemplated by the isolation mechanisms of the Xen hypervisor [7]. The proposed algorithm modifies the allocation of memory pages by the hypervisor using the technique of page coloring.

McIlroy and Sventek proposed a local control based on Xen for the current Internet, in which each flow that requires QoS is allocated to a virtual machine, called QoS routelet [8]. Each virtual machine applies, then, their QoS policies across the incoming traffic. The prototype is implemented in Xen and the traffic without QoS requirements is routed by the privileged domain, while other traffics are routed by virtual machines. The authors note that it is not possible to guarantee QoS in the strictest sense with this model, because the Xen scheduler is not suitable for this task. Other problems related to this proposal are the scalability, because it takes a virtual machine per QoS flow, and low performance in packet forwarding.

Mechanisms for the isolation of virtual environments have also been proposed for other virtualization platforms. Trellis [9] is a system to provide isolation on the VIrtual Network Infrastructure (VINI) [10]. VINI is a testbed similar to PlanetLab, which acts as a private testbed within this project. Because VINI is based on virtualization in the operating system level, i.e. all virtual environments share the same kernel, the performance of packet forwarding using Trellis is lower than that of Xen with plane separation [11]. The major problems of approaches based on operating system virtualization [9, 12, 13] are that all control planes are executed on the same operating system and, in general, these approaches do not allow the creation of differentiated data planes for each virtual network.

Genesis is a kernel to create virtual networks with different architectures [14]. Based on the concepts of hierarchy and inheritance, Genesis proposes that different "child" virtual networks should be created based on a "root" network, from which the children inherit common characteristics. Just as Trellis, Genesis is based on the premise that all control planes work on the same operating system. It allows, however, the usage of different policies and QoS mechanisms for each virtual network. Because Genesis is implemented at the user level and inserts a virtualization layer, it presents low performance on routing.

Another virtualization platform is OpenFlow [15], which is based on a network of simple forwarding elements and a centralized control plane. To share the physical resources of the forwarding elements among the virtual

networks, the OpenFlow platform provides the FlowVisor tool [16], which is a transparent proxy between the forwarding elements and the control planes. FlowVisor controls the use of CPU and memory in the forwarding elements as well as the division of the network space, i.e. which characteristics define each virtual network.

Another approach based on Linux and Click to create a shared data plane is proposed by Keller and Green [17]. In this proposal, each virtual network can create its own data plane, based on generic assumptions for packet forwarding in Click. However, the authors do not address a fair division of resources among the control planes.

The proposals described in this report for the Horizon project are focused on the local and on the global control. The local control is performed by the proposals that guarantee the isolation and the SLAs inside each physical node in Xen networks. The global control autonomously solves physical node failures using a distributed fault-tolerant algorithm and also migrates virtual networks hosted on overcharged physical nodes. The proposals for the global control approach presented can be applied in any virtualization platform, such as Xen or OpenFlow, which are the basis for the project.

1.2 Challenges in Network virtualization using Xen

The virtual network model using Xen considers that virtual machines behave as routers. A virtual network is defined as a set of virtual routers and links, created over the physical infrastructure, as illustrated by Fig. 1.1. The Xen architecture is composed of the hypervisor, the virtual machines, called unprivileged domains (DomU), and a privileged virtual machine called Domain 0 (Dom0). The Xen hypervisor controls the physical resource accesses and handles the I/O operations performed by the domains. Dom0 is a privileged domain that directly accesses the hardware. Since Dom0 is a driver domain, it stores all physical device drivers and creates an interface between the virtual drivers placed in the unprivileged domains and the physical devices. In addition, Dom0 is also the management interface between the administrator and the hypervisor to create virtual machines, modify Xen parameters, and manage Xen operation.

Sending and receiving packets are I/O operations, which require the use of the device drivers located at Dom0. Thus, all network operations of DomUs generate an overhead in terms of both memory and CPU of Dom0. The Xen hypervisor, however, does not efficiently isolate Dom0 resource usage,

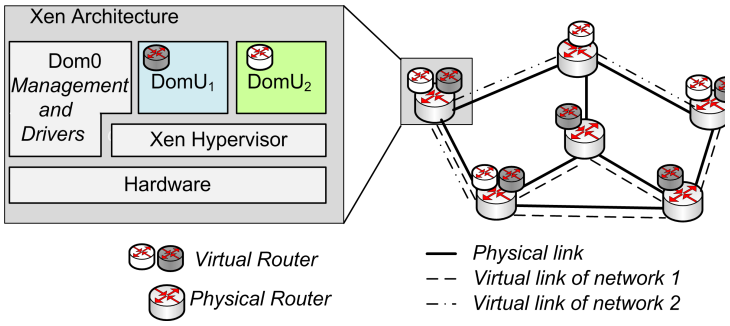


Figure 1.1: Xen architecture with two virtual networks: DomUs behave as virtual routers.

which is a major vulnerability of Xen. Table 1.1 shows that a DomU can easily increase Dom0 CPU consumption by performing network operations[1]. Since data transfer between two DomUs and data transfer between DomU and Dom0 are Dom0 CPU-demanding operations, a malicious or fault action in a DomU can easily exhaust the Dom0 resources and thus compromise the performance of all other domains. One of the goals of Horizon project proposals is to prevent that any operation performed on a virtual network breaks the isolation between networks.

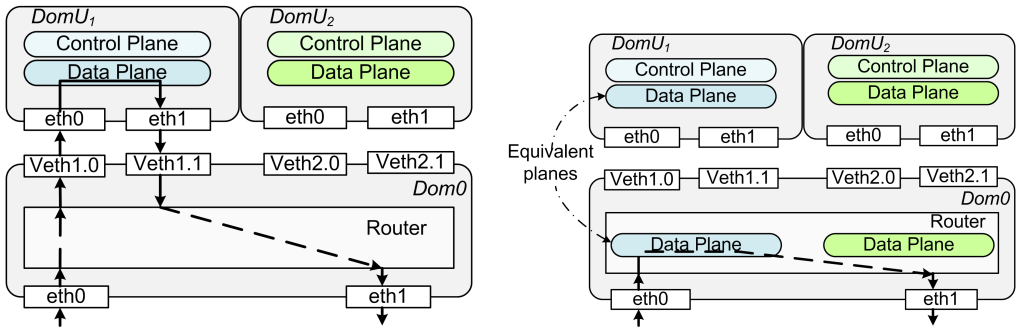
Table 1.1: CPU consumption on Dom0.

CPU (%)	**Description**
$0,71 \pm 0,60$	Basic CPU consumption on $Dom0$
$66,43 \pm 8,93$	TCP traffic from $DomU$ to $Dom0$
$85,49 \pm 5,91$	TCP traffic from $DomU_1$ to $DomU_2$
$1,79 \pm 1,01$	TCP traffic from an external machine to $DomU$

The Xen conventional architecture is not efficient for network operations because DomU packet forwarding takes a long and slow path. As depicted in Fig. 1.2(a), the packet arrives at Dom0, follows to DomU, and returns back to Dom0 to be forwarded to the next router. The plane separation paradigm is an alternative to improve the forwarding performance because packets are directly forwarded by a shared data plane in Dom0, as shown in Fig. 1.2(b).

[1]Tests were performed with the Top tool in a machine with Intel Core 2 Quad processor with 4GB of RAM and Xen 3.4-amd64. Each DomU is configured with one virtual CPU and 128 MB of memory, and Dom0 is configured with one virtual CPU and no memory constraints. Each virtual CPU is associated with an exclusive physical CPU. TCP traffic was generated with Iperf. The basic CPU consumption indicates the CPU usage in Dom0 when there are no operations in the DomUs. We assume a confidence interval of 95%.

The plane separation is accomplished by maintaining a copy of the current forwarding table of DomU in Dom0, which has direct access to the hardware. It is important to note that data packets are directly forwarded by Dom0, but control packets are forwarded to DomU to update the control plane. Also, plane separation does not avoid flexible packet forwarding. If a virtual router needs to do specialized operations not supported by the shared data plane, such as monitoring or modifying a specific header field, it can ignore plane separation by inserting a default route to the virtual machine in the forwarding table in Dom0, as done in the conventional packet forwarding.



(a) Conventional packet forwarding.

(b) Packet forwarding with plane separation: data packets pass only through Dom0.

Figure 1.2: Packet forwarding modes in Xen-based networks.

Finally, Xen also does not provide any scheme for QoS provision in the virtual networks. Thus, a scheme for controlling traffic policies must be built to guarantee the QoS within each virtual network and also between virtual networks.

1.3 Report Outline

The rest of this report is organized as follows. First, we describe the proposals developed for the local control inside physical network nodes. The two first proposals are controllers for the Domain 0 shared resources, which control the resource usage by each virtual network according to the service level agreements (SLAs) of each virtual network. These proposals are described in Chapter 2 and consist of two kinds of controllers: the first controller, which allows the maximum physical resource usage as soon as all the minimum required levels of all virtual networks are respected; the second controller, which restricts to the agreed SLAs the resource consumption of each virtual network, even if there are idle resources. One approach tries to achieve the

highest throughput to all virtual networks, while the other approach tries to achieve the agreed resource utilization, so that there are fewer expenses with physical equipment and management primitives, such as virtual network instantiation and migration, are easily performed. The last proposal for local control, described in Chapter 3, guarantees the minimal agreed bandwidth to each virtual router, assuming the virtual machine as a complete router, which means that both control and data plane are placed inside the virtual machine. This proposal modifies Xen parameters on the fly to guarantee that the required resources will be available to each virtual network when they are demanded. Finally, we describe our proposal for managing the physical resource allocation considering a global view of the network. This proposal, described in Chapter 4 deals with the challenges of allocating virtual networks with respect to all SLAs.

Chapter 2

Controllers for the Domain 0 shared resources

The Domain 0 resources, such as CPU, memory and bandwidth, are shared by the virtual routers in the I/O operations. Since there is no control in this resource sharing among virtual routers and these resources may be depleted during packet forwarding, isolation among virtual routers may be broken. The isolation failure may imply in security problems, as well as in SLA violations. Hence, it is necessary to develop mechanisms for local control that will guarantee the SLAs of each virtual network in each physical node. The main objective of the two proposed controllers is to provide isolation in the Xen virtualization platform by controlling the use of Dom0 resources. Then, the proposed controllers allocate and monitor physical resources used by all DomUs according to the parameters set by the administrator of the physical machine as SLAs of each virtual network.

We propose two kinds of controllers that differ in the physical resource usage policies. The first controller, called Maximum Usage Controller, defines to each virtual network a fixed resource reservation and a parameter called weight. The fixed resource reservation guarantees the minimal amount of resources available to each virtual network, while the weight specifies the how the idle resources are distributed among virtual networks. As long as there are idle resources, they will be provided to the networks that have a demand for that resource. The second controller, called Efficient Usage Controller, provides to each virtual network a more detailed set of parameters to define the SLA. In this controller, only the agreed resources are provided to each virtual network. Hence, even if there are idle resources, a virtual network cannot exceed the amount of resources specified in the SLA.

Both controllers are based on resource monitoring and punishments according to the SLA definitions specified to each virtual network. Also, both

controllers support the plane separation paradigm, so that the resources will be correctly monitored, independent of the data plane location.

We assume in our analysis that all DomUs may be not trustworthy, because each DomU may have a different administrator. Hence a DomU may, intentionally or not, damage other domains. Hence, in our model, a DomU can be malicious or present an unacceptable behavior. The malicious behavior occurs when a DomU intentionally executes an action to break the isolation among virtual networks and then damage other DomUs performance. By unacceptable behavior, we classify any attempt of a DomU to exceed its resource reservation that uses all Dom0 idle resources, which could interfere in the other domains operation. Since both behaviors are harmful, we classify the virtual machines with these behaviors as opponent domains. Other domains are called common domains.

2.1 Maximum usage controller

The Maximum Usage Controller (MUC) [18] allocates Dom0 resources and also monitors their total usage, $U(t)$, and their usage by each virtual router i, $U_i(t)$, in every T seconds. The allocation of Dom0 resources takes place in two ways: by fixed reservation and on demand. In the allocation based on fixed reservation, the administrator reserves a fixed amount of Dom0 resources for each DomU, ensuring a minimum quality for each virtual network. The on-demand allocation guarantees high efficiency in resource usage, because MUC redistributes the idle resources among the DomUs that have a demand greater than their fixed reservation. We classify as idle resources all the non-reserved resources as well as the reserved resources that are not in use by the virtual networks. Thus, a premise of the controller is to provide the fixed resources of a virtual router i, represented as a percentage α_i of the total resources of the Dom0, $R(t)$, whenever there is a demand. Another premise is to allocate all the idle resources on demand to the virtual routers according to the priority preset by the administrator. This priority is a parameter called weight that belongs to $\{W_i \in \mathbb{Z} \mid 1 \leq W_i \leq 1,000\}$. The higher the weight of a virtual router, the more idle resources on Dom0 it has access to. Thus, the on-demand allocation provides an additional differentiated quality for each virtual network.

MUC monitors bandwidth by observing the volume of bits being transmitted by each output physical link. If a router exceeds the allocated bandwidth in an output link, it is punished by having its packets (destined to that link) dropped.

The CPU usage in Dom0 is monitored based on the volume of packets

passing through Dom0. The monitored data is then weighted on the cost of each network operation. The packet processing cost is assigned according to the source and the destination of the packet because, as shown in Table 1.1, the packet impact on the Dom0 CPU depends on whether the packet comes from/goes to a DomU or an external machine. If a router exceeds the allocated CPU, it is punished by having its packets dropped in the input interface. To avoid attacks that generate unfair CPU punishments, it is important to define the responsible domain for each measured operation. In transfers between DomUs, the DomU that sends the packet is responsible for all the costs of CPU usage, because we want to prevent an opponent domain from starting unsolicited traffic to exhaust CPU resources of a common domain. Besides, in transfers between DomU and Dom0, the CPU usage cost is always accounted for the DomU.

MUC controls memory usage by observing the size of the forwarding table of each virtual router. If the Dom0 memory reaches critical limits, the virtual routers whose tables/filters occupy more memory than the fixed reservation are punished, through the disposal of a percentage of routes. To avoid packet losses, a default route to the virtual router is added. Hence, the packets that correspond to the discarded routes are forwarded by the virtual router instead of being discarded by Dom0. Therefore, reducing the size of the routing table does not imply dropping packets, but only in a reduced forwarding performance because the packet is then forwarded by DomU instead of by Dom0.

2.1.1 Punishment computing

Opponent domains are punished by having their packets or routes dropped. MUC searches and converges for a dropping probability that balances the use of Dom0 resources among virtual routers according to the fixed reservation and weight values. To avoid drops when there are idle resources on the physical machine, a virtual router is punished only if its usage overcomes its fixed reservation value and if the total resource usage reaches a critical level, given by a percentage β of total resources $R(t)$ in Dom0. With no idle resources, all nodes that use more than the fixed reservation are punished to avoid that other virtual routers cannot use their fixed reservation. Given that the total non-reserved resources is given by $D(t) = R(t) - \sum_{\forall i} \alpha_i R(t)$, then the dropping probability in $t + T$, given by $\Phi_i(t + T)$, is updated according to Algorithm 1.

It is important to note that even if a DomU consumes fewer resources than its fixed reservation value, the punishment is not immediately reset to

Algoritmo 1: Heuristics for punishment computing.

input : $\Phi_i(t)$, W_i, α_i, $R(t)$, $U(t)$, $U_i(t)$, $D(t)$, β
output: $\Phi_i(t+T)$

if *($\alpha_i \cdot R(t) < U_i(t)$) or ($\Phi_i(t) > 0$)* **then**
  if *($\alpha_i \cdot R(t) < U_i(t)$)* **then**
    % Calculate an idle resource usage indicator
    $\Upsilon_i(t) = (U_i(t) - \alpha_i \cdot R(t))/D(t)$
    if *($\beta \cdot R(t) \leq U(t)$)* **then**
      % Since there are no idle resources, some network can be damaged. Thus, we increase punishment.
      if *($\Phi_i(t) > 0$)* **then**
        $\Phi_i(t+T) = min(\Phi_i(t) + (1 + \Upsilon_i(t)) \cdot (1 + \frac{1}{W_i}) \cdot \frac{\Phi_i(t)}{(3 - \frac{1}{W_i})}, 1)$
      else
        $\Phi_i(t+T) = \Phi_{initial}$ % Set initial punishment
      end
    else
      % Reduce punishment, because there are idle resources
      $\Phi_i(t+T) = max(\Phi_i(t) - (1 + (1 - \Upsilon_i(t))) \cdot (1 - \frac{1}{W_i}) \cdot \frac{\Phi_i(t)}{(3 + \frac{1}{W_i})}, 0)$
    end
  else
    % Reduce punishment, because the router used only its fixed resources.
    $\Phi_i(t+T) = max(\Phi_i(t) - 3 \cdot (1 - \frac{1}{W_i}) \cdot \frac{\Phi_i(t)}{3}, 0)$
  end
else
  $\Phi_i(t+T) = 0$
end

avoid instabilities. Also, to prevent that traffic generated by virtual routers interrupt other Dom0 services due to CPU overload, a residual punishment is constantly applied in the output interfaces of virtual machines. Such punishment should be small enough to not impact the low-volume transmissions, but should prevent that a DomU consumes all the resources of Dom0.

2.1.2 MUC prototype description and analysis

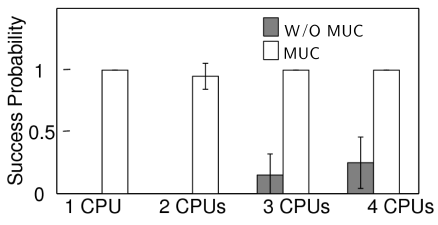
We developed a prototype to analyze the effectiveness of MUC in the presence of opponent domains and to verify the efficiency of the controller to share resources. The prototype was implemented in C and Python and the controller is able to monitor both bandwidth and CPU of Dom0. Monitoring and punishment were implemented with Iptables. To dynamically estimate the capacity of physical links, we used Mii-tool. We compute the CPU usage with MUC by estimating the cost in Dom0 of each network operation, including communication between DomUs, from DomU to Dom0, from Dom0 to DomU, from DomU to external machine, from external machine to DomU, and between external machines. The residual punishment in the output virtual interfaces was estimated as 0.0009 based on the packet rate that caused damage to Dom0 response time.

We performed the tests on a machine, hereafter called router, equipped with an Intel core2 quad processor with 4GB of RAM, using Xen 3.4-amd64 in router mode. The router has five physical Ethernet interfaces of 1 Gb/s each. We instantiated four virtual machines running Debian operating system with Linux kernel 2.6-26-2, each with one virtual CPU, 128 MB of memory, and five network interfaces. The number of virtual CPUs in Dom0 varies according to the test and there are no memory constraints for this domain. The physical CPUs are shared by all virtual CPUs and the hypervisor dynamically maps virtual CPUs to the real CPUs. The tests use two external machines that generate or receive packets, each with a network interface of 1 Gb/s. All traffic is generated with Iperf and the results present a confidence interval of 95%.

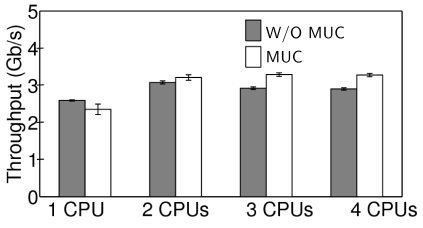
The first test evaluates the availability of the secure data plane update. The test is considered successful if the DomU securely updates its data plane and no operation of other domains that passes through Dom0 prevents the data plane update. We analyzed the impact of the usage of MUC during the data plane update. The test consists of a maximum of three attempts from $DomU_1$ to update the data plane, while $DomU_2$ sends TCP traffic to $DomU_1$. The scenario simulates an opponent virtual router, $DomU_2$, trying to prevent a common router, $DomU_1$, from normally operating. In MUC, $DomU_1$, which is trying to update the data plane, has $\alpha_1 = 0.5$ and the

opponent, $DomU_2$, has $\alpha_2 = 0.3$. All DomUs have weight $W = 500$.

Fig. 2.1(a) shows the success probability of data plane update and Fig. 2.1(b) shows the volume of data transmitted between the virtual machines. The $DomU_2$ attack is effective when using the conventional plane separation, even if there is a great number of CPUs in Dom0. MUC, however, increases by up to 100% the probability of a successful data plane update. Indeed, MUC limits the attack traffic from the $DomU_2$ avoiding the overload of Dom0 resources. Also, MUC reserves the CPU resources required by $DomU_1$ to send the update messages as well as to perform cryptographic operations required by the secure plane separation. Fig. 2.1(b) shows that MUC punishes traffic from $DomU_2$ to $DomU_1$ to ensure that the Dom0 CPU resources are not exhausted. Therefore, the throughput achieved when using MUC is smaller than when using the secure data plane update with only one CPU in Dom0. When we increase the number of CPUs in Dom0, the CPU restriction is relaxed and the throughput using MUC increases. Indeed, MUC throughput is even greater than the throughput of the secure data plane separation. MUC ensures the fixed resources of $DomU_1$, and then $DomU_1$ can handle the data plane update as well as the ACK messages of the TCP connection started by $DomU_2$. Losing ACK messages is worse to throughput than the limitation imposed by MUC to traffic due to the CPU consumption. Thus, MUC ensures a secure data plane update with high availability and also ensures a high performance connection between virtual machines because of the proposed architecture with the controller module.



(a) Probability of a successful data plane update with traffic between $DomU_1$ and $DomU_2$.

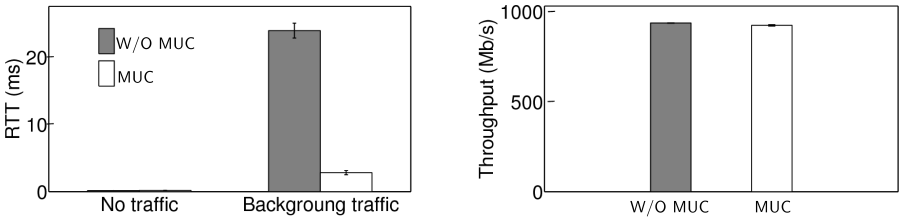


(b) Throughput between $DomU_1$ and $DomU_2$ during data plane update.

Figure 2.1: Availability of the secure data plane update when using MUC.

The second test evaluates the transmission delay when using MUC with the plane separation paradigm. We evaluate the impact of MUC compared to the usage of the plane separation paradigm without any kind of control. This test measures the delay caused by MUC overhead according to Dom0 workload. Because we are not evaluating fairness in resource sharing, we created a virtual network with a fixed reservation of 100%. The test consists

of two experiments that measure the Round Trip Time (RTT) between two external machines using Ping. In the first experiment, there is no background traffic, whereas in the second experiment background TCP traffic was generated between the two external machines. The results of both experiments are in Fig. 2.2(a). Without background traffic, data transmission presents a low RTT for both configurations. However, when there is background traffic, the Dom0 CPU is overloaded, increasing the response time of the system, and, consequently, increasing the RTT. The results show that the CPU and bandwidth control provided by MUC prevents that the Dom0 CPU is overloaded and, thus, MUC presented an RTT up to eight times lower than the conventional plane separation configuration. It is important to note that even though MUC control implies in dropping packets, these drops do not cause a major impact on traffic, as shown in Fig. 2.2(b).



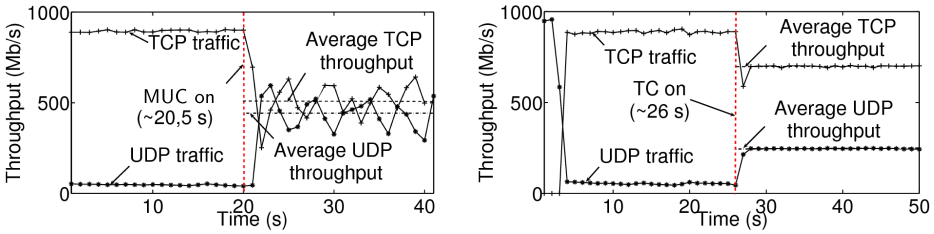
(a) MUC impact over RTT between two external machines with one CPU on Dom0.

(b) Background traffic throughput between two external machines with one CPU on Dom0.

Figure 2.2: Impact of MUC over the RTT.

The third experiment concerns sharing output links. In this experiment, a DomU and an external machine on different virtual networks initiate a communication with another external machine. Thus, both networks share the output link to the destination machine. Both networks have equal access to physical resources, with $\alpha = 0.5$ and $W = 500$ in both virtual routers. To assess MUC, we also test the bandwidth control using Traffic Control (TC), a widely used tool for traffic control on Linux machines. In TC experiments, we use the Hierarchy Token Bucket (HTB) to create two output queues, each one with a minimum bandwidth of 512Mb/s and a maximum bandwidth of up to 1Gb/s to simulate the same resource usage policy than MUC. Figs. 2.3(a) and 2.3(b) present the results when the DomU sends UDP traffic with a maximum rate of 1.5 Gb/s and packets of 1500 B while the external machine sends TCP traffic. In the beginning, there is no control in the network and from the specified moment MUC or TC controls traffic.

The results show that external machine traffic has priority over DomU traffic when there is no control on the resource sharing. This is an isolation



(a) Resource sharing in MUC with UDP traffic from DomU and TCP traffic from external machine

(b) Resource sharing in TC with UDP traffic from DomU and TCP traffic from external machine

Figure 2.3: Resource usage control in MUC.

failure when using plane separation, because external traffic influences the maximum volume of traffic generated by a DomU. Thus, an external machine that belongs to an adversary network could generate attack traffic to damage the performance of a virtual router of another virtual network. In this test, we equally share the resources between the two virtual networks and then both network should have an equal slice of the link, which means 512 Mb/s for each virtual network. Although MUC presents a larger variation in traffic than TC, MUC average throughput has a lower error with respect to the ideal rate of 512 Mb/s for each virtual network than the TC average throughput. In fact, if there is no control of the external machine inflow, UDP traffic from the virtual machine is underprivileged and is unable to achieve high rates. Therefore, MUC control presented a maximum throughput error with respect to the ideal rate of 512 Mb/s of -14.2 % for UDP traffic and of -0.62 % for TCP traffic and TC presented a maximum throughput error with respect to the ideal rate of 512 Mb/s of -52.18 % for UDP traffic and of $+35.68$ % for TCP traffic. Thus, the MUC presented a higher fairness in the link resource sharing because it is adapted to the Xen architecture particularities.

2.2 Efficient usage controller

The Efficient Usage Controller (EUC) is a controller for Xen-based virtual networks with a different policy than MUC. EUC guarantees a robust and accurate isolation between virtual machines and allows a differentiation of the physical resources allocated to each virtual network according to the Service Level Agreements (SLAs).

EUC is responsible for monitoring the use of physical resources by each virtual network and for the punishment of virtual networks that exceed the use of resources employed in the SLA. The control is based on observations

in the short and long term use of the CPU, memory and bandwidth in Dom0.

The controller assumes the existence of a specification of the physical resources allocated to each virtual network according to the SLA. This specification includes: the resources reserved for short term (R_{c_i}), which are the resources which must be met for the network i whenever there is demand in a short time interval I_c; the resources reserved for long-term (R_{l_i}), which are resources that should be guaranteed only in a long time interval I_l, if there is demand; the short-term exclusive reserve (R_{e_i}), which is a type of short-term reserve that should not be made available to other networks, even if network i does not demand this resource; and the maximum reserve (R_{max_i}), which is the maximum resource usage that the network i can use in I_l. The network can also choose a restricted reserve,R_{r_i}, which means that network i must never exceed the resource usage specified by R_{r_i}. Based on this set of features, a virtual network can define different profiles of physical resource usage. The choice of parameters for each virtual network varies according to the requirements of the virtual network operator and the price it is willing to pay for its slice of the network.

The proposed controller monitors if the demand of network i, D_i, is in or out of the SLAs of that network. Algorithm 2 is a simplified version of how the EUC calculates the punishments for each network[1].

Algorithm 2 is run in every I_c for each of the monitored resources of Dom0, i.e., the outgoing bandwidth of each physical link, the CPU, and the memory. This algorithm assumes that the demand D_i of network i in the last I_c is known and this value is used as an estimation of network demand in the next I_c. Based on the estimated demand of all the networks, the resources are divided between the virtual networks. Algorithm 2 further assumes that the number of virtual networks hosted in the physical node (N), the total physical resources available to the virtual networks (R_t), the total resources used by each network up to the current time during the long interval ($used_i$), and current weight of each virtual network i ($weight_i$) are known. The weight is a parameter generated by the controller in each I_c to control the usage of the long-term reserve by each virtual network, so that the greater the weight, the greater the access of the virtual network to the non-reserved resources in the short term.

First, the algorithm resets all punishments, calculates the amount of resources that are not reserved, and the sum of the weights of all networks. These values are used to calculate the amount of resources that must be released for each network in the next I_c, which is represented by the variable $next\_available$. Thus, if the virtual network has not spent its entire long-

[1]The extended version considers the parameters R_{e_i}, R_{max_i}, and R_{r_i}.

Algoritmo 2: Punishment calculation for each virtual network.

input : R_t, N, I_c, I_l, $R_c[\,]$, $R_l[\,]$, $weight[\,]$, $used[\,]$, $D[\,]$
output: $punishment[\,]$

$Zero(punishment[\,])$;
$total\_available = R_t - \sum_{i=1}^{N}(R_c[i])$;
$total\_weight = \sum_{i=1}^{N} weight[i]$;
for $net = 1$ *to* N **do**
  if $(used[net] < R_l[net] \cdot I_l)$ **then**
    $next\_available[net] = R_c[net] + total\_available \cdot weight[net]/total\_weight$;
  else
    $next\_available = R_c[net]$;
  end
  if $(D[net] > next\_available[net])$ **then**
    $punishment[net] = 1 - next\_available[net]/D[net]$;
  end
end
while $(verify(next\_available[\,], D[\,], N) == 1)$ **do**
  $weight\_total = 0$;
  for $net = 1$ *to* N **do**
    if $(next\_available > D[net])$ **then**
      $next\_available = D[net]$;
    end
    if $((D[net] > next\_available[net]) \& (used[net] < R_l[net] \cdot I_l))$ **then**
      $total\_weight += weight[net]$;
    end
  end
  $leftover = R_t - \sum_{i=1}^{N}(next\_available[i])$;
  for $net = 1$ *to* N **do**
    if $((D[net] > next\_available[net]) \& (used[net] < R_l[net] \cdot I_l))$ **then**
      $next\_available[net] += leftover \cdot weight[net]/weight\_total$;
    end
    if $(next\_available[net] < D[net])$ **then**
      $punishment[net] = 1 - next\_available[net]/D[net]$;
    else
      $punishment[net] = 0$;
    end
  end
end

term reserve, the $next\_available$ is given by the short-term reserve plus a slice proportional to weight of the virtual network. Otherwise, the EUC policy defines that the resources are limited by the short-term reserve of the virtual network. All networks that have a demand higher than $next\_available$ receive a punishment commensurated with the excess use.

To improve the resource distribution among virtual networks, EUC checks if

$$D_i < next\_available[i] \tag{2.1}$$

is true for each network i, $\{i \in \mathbb{Z} \mid 1 \leq i \leq N\}$, using the function $verify(\,)$. Then, EUC sets $next\_available[i] = D_i$ for all the networks that checks true for Condition 2.1. Besides, the difference given by

$$Diff = next\_available[i] - D_i \tag{2.2}$$

is distributed among the networks that are being punished because the condition

$$next\_available[i] < D_i \tag{2.3}$$

is satisfied. After that, the $total\_weight$ is recomputed, considering only networks that still satify the Condition 2.3. After, the $next\_available[i]$ of these networks is recalculated based on the released resources in proportion to the weight of each network. Finally, the punishments of all networks are upgraded. This process is repeated until Condition 2.1 is false for all networks. Thus, the physical resources are distributed in proportion to the weight and the demand of each network.

Another objective of the controller module is to calculate the weight of each network in each resource in each short interval. The weight of a network i is an adaptive variable, computed based on the resources that the network has already used ($used$) and long-term reserve, R_{l_i}. The weight adjustment basis is to give to each network a weight proportional unused long-term reserve, i.e. $weight[i] = R_l[i] - used[i] / \sum_{i=1}^{N} R_l[i]$. Thus, the networks that have greater long-term reserve available gain priority in the allocation of available resources, receiving a larger slice of the resources if there is demand. Therefore, EUC increases the likelihood that the long-term reserve is fully provided to all virtual networks.

2.2.1 Punishing virtual networks

EUC computes the punishment as shown in Algorithm 2 for each Dom0 resources that is being monitored. The application of the punishment, however, depends on the resource.

When a virtual network overloads the processing resources, this network is punished by dropping a percentage of the packets on all incoming interfaces. Thus, the percentage of packets specified by the punishment fails to enter Dom0, reducing the CPU power spent with packet forwarding. When there is bandwidth overload in an outgoing interface, packets destined to that interface are dropped. It should be noted that punishment reduces network bandwidth consumption on the outgoing interface, but the CPU cost remains and it is accounted to the network CPU resources.

Memory consumption is estimated at EUC only by the amount of filtering rules and packet forwarding rules in Dom0 of each virtual network. In fact, filtering and forwarding packets demand memory in Dom0, but this demand is small and can be disregarded. Thus, a virtual network that chooses to do packet forwarding by the virtual machine has zero memory cost for EUC, because these two operations will be done within the virtual machine. The networks that opt for plane separation, however, must observe a maximum of filtering rules and routing table entries. The punishment due to excessive memory usage implies on the disposal of a percentage of routes from the routing table. To prevent packet loss, a default route to the virtual machine is set up. Hence, if a packet has no forwarding rules set up in Dom0, this packet is forwarded by the data plane inside the virtual machine. Therefore, if a network uses the plane separation paradigm and depends on high performance in packet forwarding, it must control the number of routes installed in the Dom0.

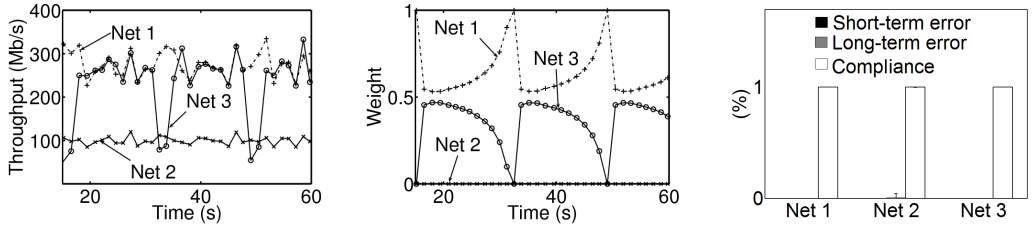
2.2.2 EUC prototype description and results

We developed a prototype of EUC in C++ to evaluate the proposal. We used Xen-4.0 configured in router mode in our test environment. The CPU usage, which is estimated based on the volume of packages forwarded, and the bandwidth usage are monitored using the tool Iptables. The same tool was also used to apply punishments. To control the QoS parameters, we used the tool Traffic Control (TC).

The prototype was implemented on a physical machine with an Intel Core 2 Quad, 4GB RAM, and five gigabit network interfaces, hosting three virtual machines with the operating system Debian 2.6.32-5. The Dom0 is configured with four logical CPUs, while each DomU has a logical CPU. The testbed is composed of four machines, one running Xen and EUC that represents the physical router, and the other, called external machines, connected to this physical router. The traffic is generated by the Linux kernel module 'pktgen', characterized by UDP packets with 1472 B payload.

We analyzed EUC controlling three different virtual networks, but with

equal demands given by $D_i = 300\ Mb/s,\ 0 < i < 3$. The Network 1 has a short-term reserve $R_{c_1} = 50\ Mb/s$ and a long-term reserve $R_{l_1} = 350\ Mb/s$, so that its demand is consistent with SLAs and must be fully provided. Network 2 has $R_{c_2} = R_{l_2} = 100\ Mb/s$, which means that this network demands more than the resources agreed in the SLAs. Network 3 simulates a network with high traffic volume, but without any priority or delay requirement, so that $R_{c_3} = 0$ and $R_{l_3} = 250\ Mb/s$. Therefore, the Network 3 also presents a demand that exceeds its SLA. None of the virtual networks has exclusive reservation and CPU resources and memory were equally divided between the three networks. The traffics of the Network 1 and 2 are transmitted from an external machine to another via Dom0 routing, because these networks use the plane separation paradigm. The traffic of Network 3 goes from the virtual router to the external machine, simulating a traffic generated or forwarded by the virtual router. The output link is shared by all three networks and is the object of this analysis. In this experiment, EUC is configured with a short interval $I_c = 1\ s$ and a long interval $I_l = 15s$, which were obtained by testing the EUC to achieve the best performance. The total test time is $200\ s$.



(a) Dom0 throughput per virtual network.

(b) Weight variation per virtual network.

(c) Error and compliance for each virtual network.

Figure 2.4: EUC control according to different virtual networks patterns, assuming a demand of 300Mb/s for each virtual network.

Figure 2.4(a) shows the throughput V_i of the three networks over time. Because Network 2 has only the short-term reserve, EUC limited its use steadily at 100 Mb/s. The Network 1 had its demand met at all times, while the Network 3 demand was blocked when the total amount of data available in the long-term reservation was used. The use of the long-term reserve is governed by the adaptive weight of the network. Figure 2.4(b) shows the evolution of the weight for each virtual network. The weight of the Network 2 is always zero, because this network has no right to use anything beyond the short-term reserve. The weight of the Network 1 increases while the weight of the Network 3 decreases, because the Network 3 has an average demand that

exceeds its long-term reserve and Network 1 has an average demand lower than its long-term reserve. In fact, the weight fits proportionately between the networks according to the slice of long-term reserve that each network can still use. Thus, the weight of the networks that have a lower demand than its long-term reserve tends to increase while the weight of the networks that have a demand higher than its long-term reserve tends to decrease. Finally, Figure 2.4(c) shows the short-term error, the long-term error, and the compliance, which are defined parameters to check if the SLAs of each of the virtual network were respected. The short-term error of network i, E_{c_i}, is computed at each short interval I_c, and demonstrates to which level the short-term reserve was not met. Similarly, the long-term error of network i, E_{l_i}, is computed at the end of each long interval, I_l, and shows to which level the long-term reserve was disrespected. Thus,

$$E_{c_i} = max(0, 1 - V_i/min(R_{c_i}, D_i)) \quad and \quad E_{l_i} = max(0, 1 - V_i/min(R_{l_i}, D_i)). \tag{2.4}$$

The compliance of network i, C_i, is computed after I_l and shows an average level to which the SLA was respected. Then,

$$C_i = 1 - \frac{mean(E_{c_i})}{2} - \frac{E_{l_i}}{2}. \tag{2.5}$$

According to Figure 2.4(c), all networks had met their compliance over long intervals of every test, because EUC offers an efficient control of shared resources on the physical machine. The Network 2 presents a small error with respect to the short-term error due to the first long interval, in which EUC was still adjusting the control to demand of each network.

Chapter 3

Xen throughput control

In a previous work produced in this project [19], we indentified a significant bottleneck in packet forwarding using virtual routers in Xen due to its default network operation. This occurs because every packet received or sent by virtual router needs to be processed by Domain 0, that manages I/O operations in Xen. When we have flows with high packet rates or when several virtual machines forward packets, the network operations can lead to CPU outage in Domain 0 and, consequently, packets are discarded. Hence, this situation decreases virtual router performance. Furthermore, a virtual router can interfer in the throughput of the other virtual routers due to the bottleneck explained above. We need thus a mechanism that performs resource allocation in Xen in order to guarantee the isolation between the routers and requirements of each one. The current Xen implementation offers tools to perform resource allocation, but do not consider network requirements of the virtual router, such as desired forwarding throughput.

To overcome the Xen network problems, we propose the Xen Throughput Control (XTC) [20] that can limit the throughput allowed to each virtual router, in order to provide isolation between them. This mechanism can be used when a bottleneck is detected to limit the throughput of routers that exceeds their maximum throughput. Nevertheless, we can disable this mechanism when we don't have bottlenecks, allowing the virtual router to forward packets in a throughput greater than specified. The decision to enable or disable our mechanism and also to specify which is the maximum throughput of the virtual router is performed by the Piloting Plane. Hence, our mechanism is a tool that Piloting Plane will use when a bottleneck is detected. The main advantage of our mechanism, compared with simple per interface packet control, such as Linux TC (Traffic Control) [21], relies on the fact that we control the aggregated throughput instead of the throughput of each network interface of the virtual router. Consequently, our mechanism con-

trols the throughput, but give freedom to the virtual router administrator to control the traffic on each of its network interfaces. To control the aggregated throughput, we act on the router capacity to forward packets by limiting the percentage of the physical CPU given to the virtual router. Previous work in this project [22] shows that we can limit the aggregated throughput of the virtual router using CPU allocation. The main problem of this approach is how to map CPU capacity on maximum throughput. Our proposed mechanism thus uses a feedback control loop in order to map CPU and throughput, measuring periodically the router achieved throughput and ajusting the CPU capacity of the virtual router to meet the desired throughput.

3.1 CPU allocation in Xen

In this work we build virtual routers by using virtual machines that have the packet forwarding as their main task. As limiting the CPU time limits also the time that each virtual machine has to execute its tasks, we can use CPU allocation to limit the throughput that a virtual router can achive when forwarding packets.

To control the share of physical CPU time that each virtual machine gets, Xen plataform uses by default the Xen Credit Scheduler [23]. In this scheduler, the share of CPU given to each virtual router depends on two parameters: weight and cap. The first one defines weights to each virtual machine and, in case of CPU contention, the resource sharing is done proportionally to the weights assigned. For example, in case of CPU contention a virtual machine with weight 200 receives twice as CPU time as the virtual machine with weight 100. On another hand, the cap parameter imposes an hard limit on CPU utilization, defining the maximum percentage of CPU time that a virtual machine can receive. For example, a virtual machine with cap 50 can only use 50% of the physical CPU time, even if more CPU resources are available. These two scheduler parameters can be configured in Domain 0. In order to give more control to our mechanism, we use cap instead of weight because the first imposes a hard limit in CPU utilization. The weight, however, can act only in case of CPU contention.

3.2 XTC overview

We develop XTC (Xen Throughput Control) to adjust the cap of each virtual router according to a requested throughput. Once we control the maximum throughput, we can provide isolation among the different virtual routers.

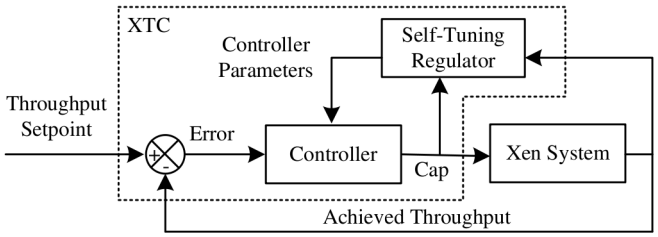


Figure 3.1: XTC Feedback Control Loop.

XTC uses a feedback control loop that acts on the cap entitled to each virtual router to achieve the requested throughput as seen in Figure 3.1. To accomplish this, XTC periodically measures the virtual router throughput and computes the error between this measure and the requested throughput. The error is then used by the Controller block where we implement a Proportional Integral (PI) controller to compute and adjust the virtual router cap according to the requested throughput. The Xen System block represents the behavior of virtual router throughput according to the cap entitled to it. Using an experimental testbed we model this block to design the Controller block, choosing the PI controller parameters. In this report we manually evaluate the parameters. However, XTC uses a Self-Tuning regulator to evaluate the Xen System model autonomously and choose PI controller parameters according to system dynamics. The Self-Tuning regulator block is specified in [20] and it is not used in the equations and experiments below.

Figure 3.2 illustrates XTC utilization. In this system, we have one XTC for each virtual router. The Policing Mechanism (PM) controls all XTCs and is also responsible for activating and deactivating each XTC. Therefore, PM communicates each XTC the requested throughput of the corresponding virtual router. The actions taken by PM are based on its knowledge about the system environment obtained via resource utilization measurements and policies specified by the system administrator. Furthermore, PM can use XTC to provide differentiation between virtual routers. In this report we focus on XTC design and the PM role is manually performed.

3.3 Experimental testbed

Figure 3.3 illustrates our testbed used to model the Xen System block and to further perform experimental analysis of our system. The Traffic Generator machine (TG) produces all data traffic destined to the Traffic Receiver (TR) machine. The Traffic Forwarder (TF) machine hosts the virtual routers used

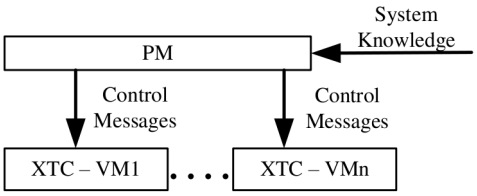


Figure 3.2: Example of XTC utilization.

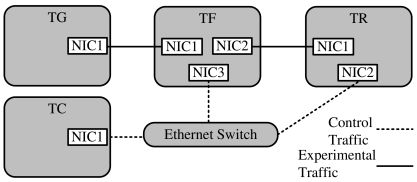


Figure 3.3: Experimental Testbed.

in our experiments. In our Xen configuration Dom0 has two exclusive CPU cores while virtual routers shares another core. TF runs Xen hypervisor version 3.4.2 and has instantiated the virtual routers which forward packets from TG to TR. We also use a Traffic Controller (TC) machine to run our mechanism. It is important to note that the Traffic Generator (TG) and Traffic Receiver (TR) are directly connected to the Traffic Forwarder (TF) whereas TC is connected to TF and TR through different links. We physically separate the traffic to avoid interference between control and data.

TG, TR, and TC are general-purpose PCs equipped with an Intel DP55KG motherboard, an Intel Core I7 860 2.80 GHz processor and 8 GB of RAM. These machines run Debian Linux kernel version 2.6.32. The TF machine is an HP Proliant DL380 G5 server equipped with two Intel Xeon E5440 2.83 GHz processors and 10 GB of RAM. This machine runs Debian Linux paravirtualized kernel version 2.6.26. On the one hand, TG and TR are connected to TF via their on-board Intel PRO/1000 PCI-Express network interface. TF, on the other hand, is connected to TG and to TR via the two interfaces of a PCI-Express x4 Intel Gigabit ET Dual Port Server Adapter.

3.4 Xen System Modeling

The Xen System block models the behavior of virtual router throughput according to the cap entitled to it. To model the system, we use a black-box approach as in [24]. This approach is based on experimental data obtained

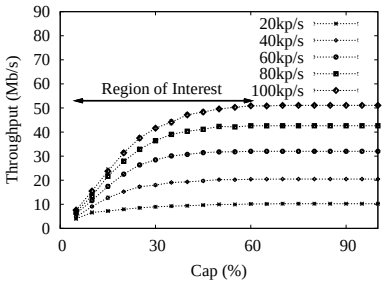


Figure 3.4: Cap Variation with 64-Byte packets.

in our testbed using the following framework.

3.4.1 Training data acquirement

Using the testbed described in Section 3.3 with the TC machine turned off, we model the Xen System by using an experiment to capture the relationship between cap and throughput. We send packets from TG to TR through one virtual router using fixed packet rate and fixed packet length. We then vary the cap given to the virtual router. We send an UDP flow using Iperf [25] traffic generator during 30 seconds. Afterwards, we measure the average throughput obtained in the experiment. Figure 3.4 shows the results using 64-Byte packets with different packet rates. The X axis shows the cap given to the virtual router and the Y axis shows the throughput obtained. Note that the relationship between cap and throughput depends on the packet rate of the flows forwarded by each virtual router. The higher the rates, the more CPU needed. Another characteristic that changes the relationship is packet length. Higher packet lengths results in higher throughputs for the same packet rate.

Figure 3.4 also shows that from a certain cap entitled on, the throughput stops increasing. In this case, the throughput obtained equals to the bit rate produced because the virtual router receives plenty of CPU resources and, therefore, does not need resource management. Nevertheless, below these cap values, the throughput changes according to the cap in a log-scale fashion. Thus, this region is considered on our system modeling. We perform the same experiment with 1470-Byte packets and also observed the same logarithmic behavior seen in the 64-Byte packet experiment. In that experiment, however, the throughput is higher for each cap value, as expected.

3.4.2 Model evaluation

Using the results from Section 3.4.1, we model the Xen System with a discrete transfer function, which will be used in the Controller design. We choose to model the Xen System as a linear first-order system given by Equation 3.1. As we are modeling a non-linear system using a linear model, the $y(k)$ and $u(k)$ are offset values from their operating points given by Equation 3.2, where $\tilde{y}(k)$ and $\tilde{u}(k)$ are the actual values of the Xen System signals and $\bar{y}$ and $\bar{u}$ the operating points. In Equation 3.1, we denote as $y(k)$ the throughput obtained in the Xen System at the k^{th} sample and $u(k)$ as the $log(cap)$ in the system input at the k^{th} sample. We use $log(cap)$ value instead of absolute cap value because the relationship between cap and throughput has a logarithmic behavior over the region of interest. Consequently, the first-order model of Equation 3.1 suits our purposes and simplifies the control system design. To evaluate the Xen System transfer function we apply the Z Transform on both sides of Equation 3.1 and performing algebraic manipulations we obtain $H(z)$ as expressed in Equation 3.3.

$$y(k+1) = ay(k) + bu(k) \tag{3.1}$$

$$\begin{aligned} y(k) &= \tilde{y}(k) - \bar{y} \\ u(k) &= \tilde{u}(k) - \bar{u} \end{aligned} \tag{3.2}$$

$$H(z) = \frac{Y(z)}{U(z)} = \frac{b}{z-a} \tag{3.3}$$

The next step to model the Xen System is to obtain the variables a and b that characterize this system on Equation 3.3. As already explained, the Xen System behavior and thus the variables a and b depend on the packet rate and packet length of the flow being controlled. The parameters a and b can also model aggregated flows considering them as one flow with their average packet rate and packet length.

To show that a first-order system suits our purposes, we model the Xen System forwarding a flow with constant packet rate of 100 kp/s and 64-Byte packet length. This example is used in the remainder of this section as a proof of concept. To estimate a and b for the presented example we use the least squares regression described in [24]. This method uses the training data obtained in Section 3.4.1 for the flow at 100 kp/s with 64-Byte packets and evaluates a and b over an operating point. We assume this operating point as the mean values $\bar{y}$ =40 Mb/s and $\bar{u} = 1.39$ over the region of interest. This

region was chosen because cap still has effect and the throughput does not saturate. In our example, this region corresponds to $cap \leq 60$ as indicated in Figure 3.4. We obtain $a = 0.0915$ and $b = 32259$ from the least squares regression using MATLAB. To evaluate our model accuracy regarding the data collected, we compute the R^2. This metric quantifies the variability explained by the model and varies from 0 (worst model) to 1 (best model). In our model we obtain $R^2 = 0.9899$ which suggests a good fit.

3.5 XTC controller design

In a feedback control system the controller computes the input of the plant being controlled according to the difference between the requested value and the measured value on the plant output. In our case, the controller must decide which value of cap will be given to the virtual machine in order to the Xen System meet the requested throughput. This decision is done periodically according to the measurement of Xen System output and the knowledge about past controller decisions. The Controller block uses a Proportional Integral (PI) controller that has the control law given by Equation 3.4. In this equation, $u(k)$ is the controller decision in the k^{th} sample, denoted as $log(cap)$, and $e(k)$ is the error computed by the difference between requested and achieved throughput in the k^{th} sample. Note that this controller computes the current decision based on the current and previous errors and also on the previous decision of the controller itself. The PI controller is chosen because it has zero steady-state error, which means that $e(k)$ converges to zero for large k values, combined with short settling time.

$$u(k) = u(k-1) + (K_p + K_i)e(k) - K_p e(k-1) \tag{3.4}$$

The design issue of a PI controller is to choose K_p and K_i parameters to meet the system requirements. In this work, we manually choose the parameters. The controller parameters influences the system's poles and zeros placement as shown in the transfer function of system, $Y(z)/R(z)$, given by Equation 3.5. In the equation, $R(z)$ and $Y(z)$ denote the Z transform of the requested throughput and achieved throughput, respectively. The poles and zeros of the system influences the system properties such as stability, settling time, and maximum overshoot. The first indicates that the system converges to a steady-state value, while the second indicates the time that the system would meet this value, and the last indicates the largest difference between the system output and the system steady-state value.

$$\frac{Y(z)}{R(z)} = \frac{z(K_p + K_i)b - K_p b}{z^2 + z[(K_p + K_i)b + a + 1)] + (a - K_p b)} \tag{3.5}$$

The Controller parameters are chosen using the Pole Placement method. This method considers only the influence of the poles in Equation 3.5. However, the zeros placement can also influence system properties increasing, for example, the maximum overshoot. To cope with this influence, we choose a small value of maximum overshoot. Using the example of Section 3.4.2, where $a = 0.0915$ and $b = 32259$, we choose K_p and K_i values to place the system poles to achieve settling time of 5 samples and maximum overshoot of 8%. We found the values $K_p = -3.422 \times 10^{-6}$ and $K_i = 22.158 \times 10^{-6}$. We simulate the control system using Simulink [26] to obtain the real value of the properties explained above. In the simulation, we obtained a settling time of 2 samples and maximum overshoot of 21%, which are still acceptable for our system. With these results we can see the difference of analytical and simulated results as a consequence of the approximation done by the Pole Placement method. Our proposed Controller also uses the concept of dead zone, where it decides to act only when the error exceeds a threshold. As the Controller acts using calls to Dom0, limiting the Controller actions reduce these calls. In systems where an external machine performs these calls this concern is very important, because the TC exchanges messages with TF. Consequently, the dead zone concept also reduces the control traffic. The threshold chosen in our implementation is 10% of the requested throughput.

3.6 Experimental Results

This section provides experimental results showing the XTC operation and some of its features.

3.6.1 Practical implementation

We implement our proposed controller in the experimental testbed of Figure 3.3. We send packets from TG to TR at a fixed packet rate using Iperf. A virtual machine hosted in TF forwards these packets. The Traffic Controller (TC) machine measures the throughput achieved by the Xen System and plays the role of the Controller of the block diagram of Figure 3.1. To measure the achieved throughput, this machine periodically collects the output of the Iperf Server reported by TR. This sensor could be placed at TF, however, we choose to measure the achieved throughput as the Iperf Server output to guarantee that the measurement is independent of the TF machine, which

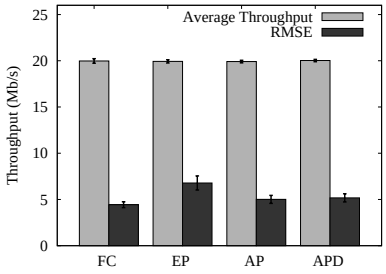


Figure 3.5: Average Throughput and RMSE measurement.

can be overloaded by high packet rates. As a Controller, the TC computes the cap of the virtual router based on the control law of Equation 3.4 and remotely acts on the virtual router cap. Note that the control law computes the $log(cap)$, rather than the absolute cap, and then the actuator must compute the inverse of $log(cap)$. The complexity of this computation is negligible in our testbed. The experiment consists of sending 64-byte packets from TG to TR at 100 kp/s during 100 seconds. The TC machine must adjust the cap of the virtual router to track the requested throughput of 20 Mb/s. Observe that the bit rate sent by TG has the value of 51.2 Mb/s.

Our first evaluation measures the average throughput achieved and the root mean square error (RMSE) with respect to the requested throughput, as seen in Figure 3.5. These measurements are computed using the values obtained during the interval from 20 to 100 seconds of each Iperf run. We use this interval to disregard the system transient behavior before 20 seconds. The average throughput indicates whether the system achieved the throughput of 20 Mb/s as requested. This value of requested output is chosen to show the behavior of the system when it is quite far from the operating point but not so far as to cause undesired system behavior. The RMSE, on the other hand, quantifies the oscillatory behavior of the system showing how the system response deviates from the average throughput. The settling time and maximum overshoot were not evaluated in this experiment due to the oscillation observed in the system output.

We evaluate separately four different configurations. The first one, called FC (Fixed Cap), consists of turning off the XTC and adjusting a fixed cap of 14% to the virtual router. This value is chosen because we expect an Average throughput close to 20 Mb/s. In practice, this implementation is not recommended because it is difficult to know in advance a fixed cap value that leads the system to a specific throughput. The system behavior may vary because of traffic dynamics, which justifies the use of a feedback controller to adjust the cap. We use, however, this result as a reference to analyze

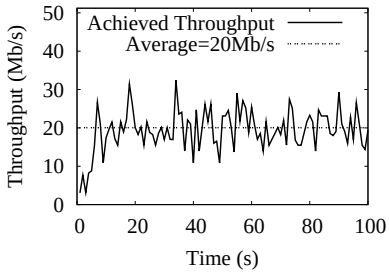


Figure 3.6: Achieved Throughput - APD XTC Experiment.

the XTC performance. Figure 3.5 shows that FC implementation obtains a high RMSE value, which indicates that the system oscillates when the throughput is limited using Xen's cap parameter. Therefore, the Controller will have to cope with this particular behavior of cap adjustment. The EP (Evaluated Parameters) implementation uses the XTC controller parameters $K_p = -3.422 \times 10^{-6}$ and $K_i = 22.158 \times 10^{-6}$ as evaluated already above and the measurement and controller periods of 1 second. Results show that the Average throughput obtained is close to the requested throughput showing the effectiveness of our proposal. The EP implementation, however, inserts more oscillation compared with FC. To cope with this shortcoming, we decrease the K_i parameter to reduce the Integral effect in the Proportional Integral controller, which is partially responsible for the system oscillation. We thus use in the AP (Adjusted Parameters) implementation, the XTC controller parameters as $K_p = -3.422 \times 10^{-6}$ and $K_i = 10.158 \times 10^{-6}$, reducing the RMSE, as seen in Figure 3.5. Finally, we implement APD (Adjusted Parameters with Dead Zone) XTC using the concept of dead zone to reduce the message exchanging between TC and TF. In this experiment, we obtain a reduction of $29 \pm 2.4\%$ of the control messages needed to adjust the cap in comparison to the AP experiment. The comparison between the RMSE values of APD and AP shows that we can reduce the number of control messages without increasing the oscillation. To exemplify the system behavior, we show in Figure 3.6 the system output in a single run of the APD XTC.

In the above experiments, we see that XTC achieves a throughput extremely close to the requested throughput. Nevertheless, the system response oscillates around this value because of cap adjustment, which represents a tradeoff of the Xen platform. Despite this fact, we show that APD XTC introduces negligible oscillation.

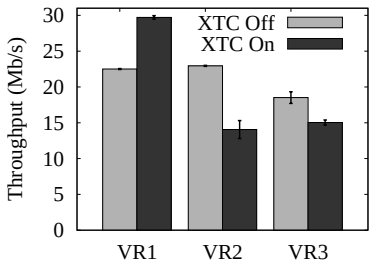


Figure 3.7: Traffic Differentiation Using XTC

3.6.2 XTC features

In this section we discuss two useful features of XTC. The first one is virtual router differentiation. In this feature, XTC dynamically guarantees higher throughput to a virtual router by isolating the amount of CPU resources used by the other routers. In the default network implementation of Xen, all packets sent and received by virtual routers are forwarded by Dom0. Consequently, Dom0 becomes the bottleneck and the packet rate of each virtual network influences each other. We conduct an experiment in our testbed using TF hosting three virtual routers (VR1, VR2, and VR3) forwarding packets from TG to TR. In this experiment, TG sends to TR three 64-byte packet flows at 51.2 Mb/s during 100 seconds. Each virtual router forwards one of the three flows. The virtual routers share the same CPU core, but there is no contention. First, we neither use XTC nor simple cap adjustment and measure the average throughput obtained in the last 80 seconds of each run. This configuration is called XTC Off in Figure 3.7.

The results show that the virtual routers are not able to forward packets at the full rate, 51.2 Mb/s, because of the high contention for resources at Dom0. Consequently, the maximum throughput obtained in a virtual router was 23 Mb/s. To allow VR1 to forward more packets, we can limit the amount of packets the other virtual routers can send to Dom0. As a consequence, VR1 has more opportunity to send packets to Dom0, increasing its throughput. We use XTC on each virtual router and repeat the latter experiment. For VR1, XTC uses the same K_p and K_i used in Section 3.6.1. The only difference is that we now limit the throughput to 30 Mb/s. For VR2 and VR3, XTC is configured to limit the throughput to 15 Mb/s. Because this rate is far from the operating point of the model used in Section 3.6.1, we also evaluate, for VR2 and VR3, a system model for the operating point of 27 Mb/s, resulting in $a = 0.00339$ and $b = 34816$. We then evaluate the controller parameters, as explained in Section 3.5, and find $K_p = -4.825 \times 10^{-6}$

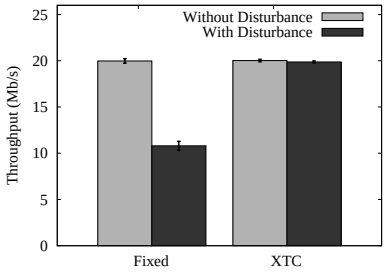


Figure 3.8: Disturbance Tolerance Using XTC

and $K_i = 18.530 \times 10^{-6}$. Results are presented in Figure 3.7 labeled as XTC On. They demonstrate that it is possible to assign priority to a virtual router using XTC. In our experiments, XTC was used in Xen's default configuration, where virtual routers contend for Dom0 resources. Nevertheless, XTC is also used when there is no contention for Dom0 resources (for example using Direct I/0 techniques [27]), but the virtual routers share the same CPU and still contend for CPU resources. In this case, XTC can also reduce the maximum throughput allowed to a virtual router by reducing its CPU utilization. This strategy gives more room to other virtual router forward packets.

The second XTC feature that we show is disturbance tolerance, which is a typical characteristic of feedback control systems. Our next experiment shows another advantage of using a feedback control system instead of static solutions, such as using a table containing the direct correspondence between cap values and requested throughput for a specific packet length and packet rate. With static solutions, wrong decisions may occur as a consequence of variable workload in the virtual router caused by additional packet processing. To demonstrate the problem, we use the same testbed of the experiment for traffic differentiation. In this case, however, TF has only one virtual router forwarding packets. TG generates a 64-byte packet flow at 51.2 Mb/s during 100 seconds and the XTC must limit the throughput in 20 Mb/s. First, we do not induce disturbance in the system. Figure 3.8 shows the results obtained when using a fixed cap of 14% to limit the throughput. In addition, Figure 3.8 plots XTC performance for the same task. These two types of control are represented in X axis respectively as Fixed and XTC. Without disturbance, we show that, despite a slight inaccuracy, fixing a cap value can limit the throughput to 20 Mb/s. We repeat the evaluation inserting disturbance, as seen in Figure 3.8. The disturbance, in our test, is a process running in the virtual router consuming up to 10% of the processing power assigned to it. As shown in Figure 3.8, XTC achieves the requested through-

put even in the presence of disturbance. On the other hand, the disturbance affects the performance of the virtual router in the Fixed case because of the contention for CPU resources between the simultaneous disturbing and packet forwarding processes.

Chapter 4

Adaptive virtual network provisioning

This section presents an overview of our ongoing work on adaptive VN provisioning. The objective is to address the dynamic provisioning of virtual resources acquired from infrastructure providers when the composed virtual networks are activated and running. Indeed, running virtual networks will be subject to dynamic variations due to changes in services demands, in traffic loads, in physical resources and infrastructures and subject to mobility induced variations. Adaptive provisioning frameworks and algorithms are required to maintain virtual network topologies, respect established contracts, expand initial allocations on demand, release resources no longer useful, optimize resource utilization and respond to anomalies, faults and evolving demands.

The purpose of this section is to design and evaluate adaptive embedding algorithms to handle resource re-optimization and fault-tolerant embedding. The proposed adaptive embedding algorithms are carried out in a decentralized manner by autonomous agents integrated in the substrate nodes. The goal is to deal with highly dynamic environments and to react quickly to node failures, performance degradation, etc. Performance results of the embedding algorithms are reported in terms of time delay and message exchange cost.

4.1 Adaptive virtual network provisioning scenario

Virtual Network (VN) provisioning includes matching, embedding and binding of virtual resources. VN embedding consists in assigning the required VN nodes and VN links, specified in the VN request, to a specific set of

substrate nodes and paths extracted from substrate network. The VN embedding relies on a selection process run (or executed) by the infrastructure provider. Among the matched resource candidates, the embedding step consists in choosing and selecting the best (or optimal) resource candidates. Finding the optimal VN embedding satisfying multiple objectives and constraints is a NP-hard problem that has been addressed in several research studies [28, 29, 30, 31, 32]. The general aim is to allow a maximum number of VNs to co-exist in the same substrate while reducing the cost for users and increasing revenue for providers. In our previous work [32], we suggested the use of a decentralized selection and embedding process across the substrate nodes. The distributed embedding relies on the resources non-functional (or dynamic) attributes like actual CPU and bandwidth.

Once the VN is entirely deployed and activated following initial provisioning, the adaptive and dynamic provisioning and maintenance of the VN comes into play. Dynamic changes are induced by variations in the substrates and VNs and are related to resource failures, mobility, migration and maintenance needs. Dynamic and adaptive provisioning deals with the highly dynamic changes expected in VN requests and in the substrate and aims at maintaining topologies and respecting established contracts and service level agreements (SLA).

This section deals with the case where virtual or physical resources supporting VNs fail or suffer from anomalies (e.g. substrate node or virtual node crash or performance degradation). The Infrastructure Provider maintains the VN topologies (affected by the failures) by selecting new substrate or virtual resources to replace or compensate for the affected resources. Infrastructure Providers need also to optimize the use of their resources to accept as many requests as possible thus making room for more users. Several mechanisms can be used to regularly optimize the allocation of resources. Virtual node migration [33, 34] or path splitting/migration [30] are example solutions. This section proposes a Multi-Agent based adaptive embedding framework to handle two algorithms (fault-tolerant embedding and resource re-optimization algorithms) to repair resource failures and dynamically optimize the substrate networks.

4.1.1 Multi-agent based adaptive embedding framework

This subsection provides the design and implementation of an adaptive VN embedding framework to deal with dynamic changes requiring automatic and runtime reparation (scenario 1) and re-optimization (scenario 2). The framework is responsible for:

1. Detecting and identifying local changes through monitoring (e.g. node/link failure, performance degradation)

2. Selecting new substrate resources to maintain VN topologies

3. Migrating virtual nodes from a substrate node to another

4. Running the binding step

The adaptive embedding framework relies on the Multi-Agent based approach to ensure distributed negotiation and synchronization between the substrate nodes. As depicted in Figure 4.1, the Multi-Agent based adaptive embedding framework is composed of autonomous agents integrated in substrate nodes. These autonomous agents communicate, collaborate and interact with others to plan collective reselection of resources for adaptive VN embedding. The agents monitor, supervise and extract the non-functional (i.e. dynamic) attributes from the local repositories and decide locally which reselection actions to undertake. As detailed in next sections, the autonomous agents are responsible for carrying out two proposed algorithms: Distributed fault-tolerant embedding algorithm and Distributed resource re-optimization algorithm.

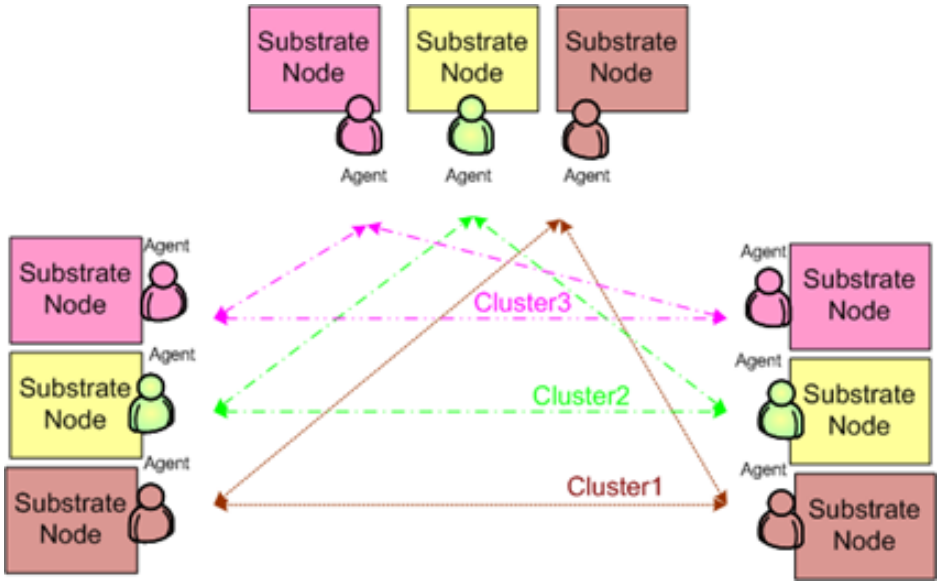


Figure 4.1: Multi-Agent based adaptive embedding framework

The proposed adaptive embedding algorithm relies on situation awareness approaches. In fact, the conceptual clustering technique used during

the matching phase can provide a generic model for a situated multi-agent based infrastructure. The situated view of agents is determined based on similarities between substrate nodes determined by the conceptual clustering algorithms. Only agents within the same cluster can negotiate and cooperate. The situation awareness involves that each agent is aware of what is happening around it. In our case, the agent's situation awareness consists in being aware about its situation and its logic position in the infrastructure and also being able to know the level of perception of other agents (i.e. to which cluster it belongs to and with what agents it should negotiate and synchronize to handle the distributed adaptive embedding decision making).

4.1.2 Distributed fault-tolerant embedding algorithm

This work proposes a distributed fault-tolerant embedding algorithm to maintain the VN topologies by reselecting new resources to replace the no longer available ones. There is no need to consult a central entity upon failures since distributed localized control can react quickly to local changes. The algorithm assumes the existence of supervision (monitoring) and fault diagnosis mechanisms to detect resource failures.

The Multi-Agent based adaptive embedding framework is used to carry out the distributed fault-tolerant embedding algorithm. The substrate node agents exchange messages and cooperate to plan collective reselection decisions. The cooperation between substrate node agents relies on the matching results. When a substrate node ns supporting a virtual node nv fails, the fault-tolerant embedding algorithm can quickly localize the failed node and reselect a new alternative substrate node. This alternative node is selected among the candidate substrate nodes matching the node nv. Thus, only substrate node agents that belong to the same cluster will collaborate to choose an alternative/backup substrate node.

Figure 4.2 depicts a step by step scenario describing the implementation of the distributed fault-tolerant embedding algorithm using the Multi-Agent framework. Each substrate node agent runs the algorithm depicted in Algorithm 3. Upon detecting a substrate node failure, each agent sends a failure notification message (ErrorMSG(ns)) to all agents in the same cluster to notify them that the substrate node supporting the request node nv is no longer available. Next, each agent should check the ability of its associated substrate node to support the request node on behalf of the failed one. To achieve that, each agent extracts the non functional (NF) (or dynamic) attributes of the affected virtual node nv as well as the NF attributes of its own substrate node. The NF attributes are presented in the form of attribute-value pairs: (att, x).

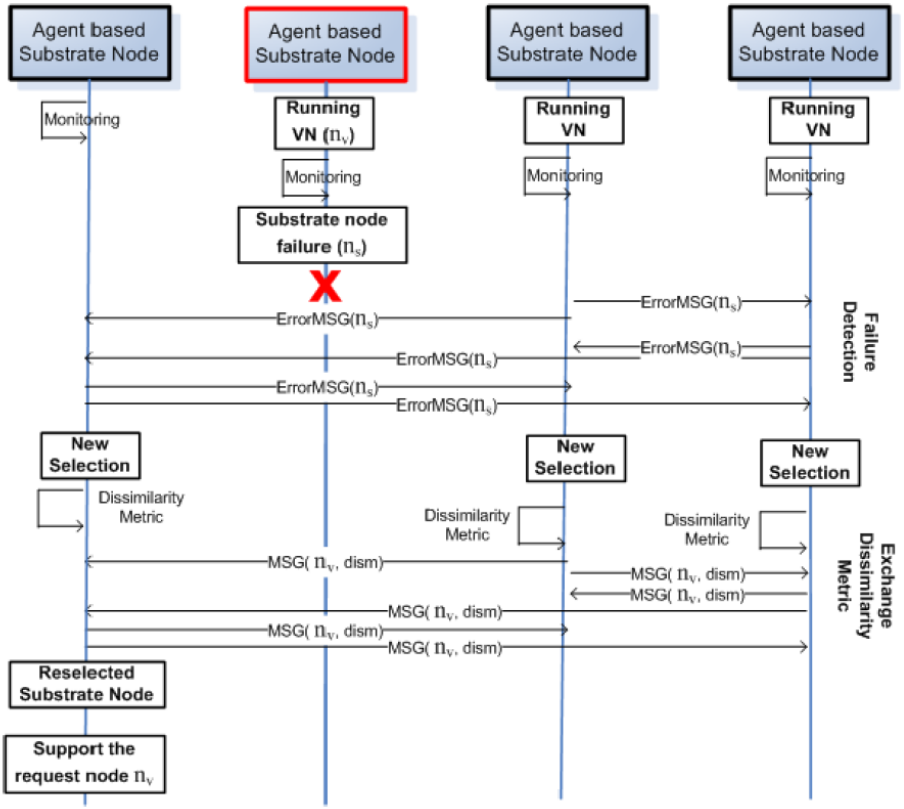


Figure 4.2: Multi-agents based distributed fault-tolerant embedding Scenario

The NF attributes of an arbitrary request node nv are represented as follow:

$$nv = ((att1, xv1), (att2, xv2), ..., attf, xvf), ..., (attp, xvp)). \tag{4.1}$$

The NF attributes of the substrate node (agent host) ns are expressed as:

$$ns = ((att1, xs1), (att2, xs2), ..., (attf, xsf), ..., (attp, xsp)). \tag{4.2}$$

The objective is to check how much the associated substrate node capabilities can respond to the virtual node nv requirements. The substrate node ns that should be reselected must be as similar as possible to the virtual node nv. As stated, each agent computes a dissimilarity metric dism between the NF attributes of request node nv and the NF attributes of its associated substrate node. An example of the dissimilarity function expression is provided below. Once the dissimilarity metrics dism are computed for the affected request node nv, the agent exchanges these metrics, via $MSG(nv, dism)$ message,

with the other agents running in the same cluster. Each agent compares its dissimilarity metrics with those computed by the other agents. For the request node nv, the (substrate node ns, request node nv) pair that has the minimum dissimilarity metric will be selected for the adaptive embedding decision.

Our proposed Distributed fault-tolerant embedding algorithm operates as follows:

Algoritmo 3: Distributed fault-tolerant embedding algorithm running in each substrate node agent ns

input : Network State
output: dissimilarity metric, new allocation map
if *detect a substrate node failure* **then**
    Send a notification message ($ErrorMSG(ns)$) to all agents in the same cluster.
    Compute a dissimilarity metric between the NF attributes of the affected request node nv and the NF attributes of the substrate node ns.
    Exchange, via ($MSG(nv; dism)$) messages, the computed dissimilarity metrics dism within the same cluster.
    The agent compares its dissimilarity metric of the request node nv with all substrate nodes.
    if *the dissimilarity metric of the request node nv is the minimal one compared to the other computed dissimilarity metrics* **then**
        The substrate node hosting the agent will support the request node nv.
    end
end

Once the request node nv is mapped to the reselected substrate node in the substrate, the next step is to map its associated virtual links to the substrate paths. A shortest path algorithm [29, 35] (in the case of unsplittable flows) or a multi-commodity flow algorithm (where the substrate supports splittable flows [30]) can be used for link mapping. These algorithms should be carried out in a distributed manner across the substrate node agents.

4.1.3 Performance results

The experimental facility GRID5000n [36] has been used to generate full mesh substrate topologies with different sizes (from 0 up to 100 nodes) to evaluate the Distributed fault-tolerant embedding algorithm. Autonomous agents [37] are deployed in the GRID5000 machines to emulate the substrate node agents in the multi-agent based embedding framework and to handle the distributed fault-tolerant embedding algorithm.

Our objective is to evaluate the time delay and the number of messages required by the distributed fault-tolerant embedding algorithm to adapt a VN (by reselecting a new node) in the case of substrate node failure. The

aim is to show the benefits of using the clustering approach to reduce the time delay and the number of messages needed to reselect a new substrate node. Clustering refers to set of candidate substrate nodes sharing the same color in the substrate.

Figure 4.3 depicts the time delay required to adapt a VN in full mesh substrate topologies. Four cases are evaluated and compared: substrate topology without clustering (corresponds to curve with one cluster in Figure 4.3) versus substrate topology with 2, 5 or 10 clusters. As shown in Figure 4.3, the time delay required to reassign a VN request to a substrate network without clustering is in order of 2 s. This time delay does not exceed 0.75 s when using clustering in substrate networks. The number of messages exchanged between substrate nodes decreases with clustering. The number of messages is reduced in the distributed localized fault-tolerant embedding algorithm since messages are exchanged per cluster. Figure 4.4 corroborates the delay results shown in the Figure 4.3.

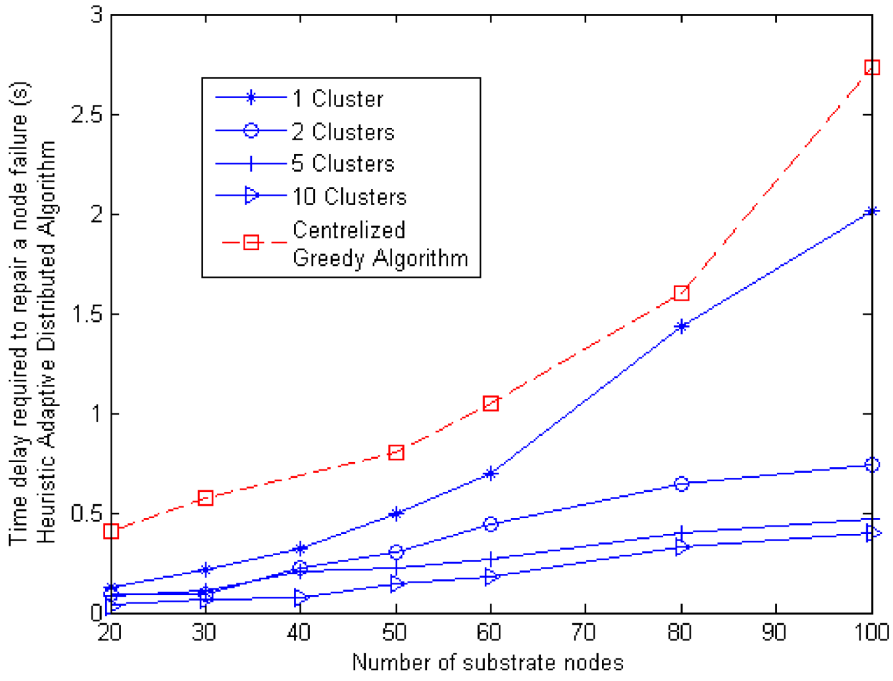


Figure 4.3: Average time delay required to adapt a VN topology in the case of node failure.

As there are currently no known distributed fault-tolerant embedding algorithms available in the literature, at least to our knowledge, this section is not in a position to provide any comparison with existing algorithms.

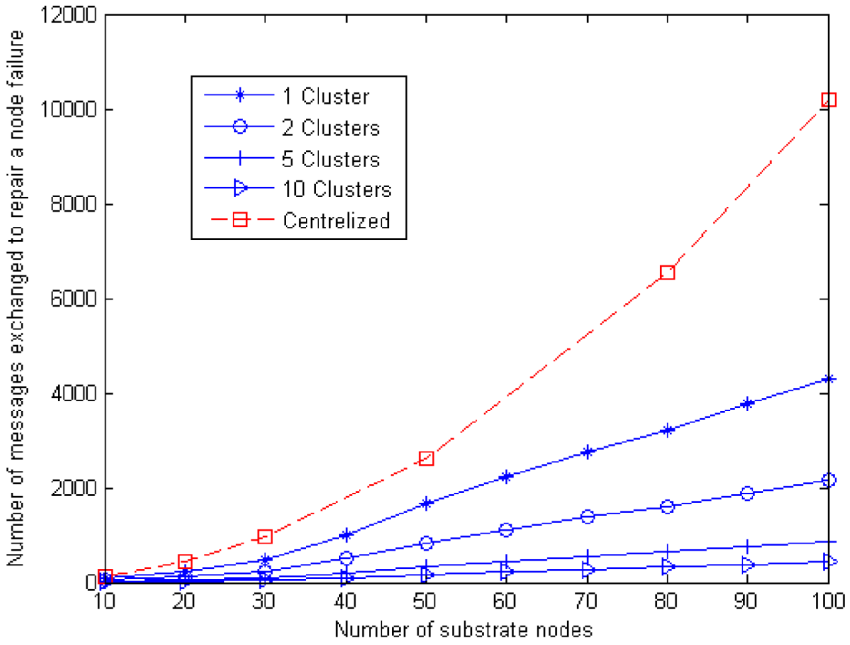


Figure 4.4: Number of messages exchanged to adapt a VN topology in the case of node failure.

We rather compare our algorithm to a centralized fault-tolerant embedding algorithm. As shown in Figure 4.3, the upper curve represents the time delay required to reselect a substrate node in a centralized greedy manner. Results show that the time delay required by our distributed fault-tolerant embedding algorithm to adapt a VN upon failures is much lower than that of the centralized approach. This is due to the important number of messages exchanged between a centralized coordinator and substrate nodes in the case of centralized reselection.

4.1.4 Distributed resource re-optimization algorithm

To make efficient use of the substrate resources and to maximize revenue and acceptance ratio, the Infrastructure Provider needs to re-optimize its resources to make room for other VN requests. Several mechanisms have been proposed in the literature to re-optimize the allocation of substrate resources including virtual node migration [33, 34].

This section provides an overview of a distributed resource re-optimization algorithm responsible for migrating a virtual node from a substrate node to

another. The substrate agents, integrated in the Multi-agent based adaptive embedding algorithm, carry out the distributed resource re-optimization algorithm. The substrate agents collaborate to make a common migration decision. Each agent runs the algorithm depicted in Algorithm 4.

Algoritmo 4: Distributed resource re-optimization algorithm running in each agent based substrate node ns

```
   input : Current topology
   output: Next topology
 1 if the substrate node ns decides to migrate its virtual node nv to other substrate
   node then
 2 |  Send a MIGRATION message to all substrate nodes.
 3 end
 4 if receive a MIGRATION message then
 5 |  if there is a room for the virtual node nv then
 6 |  |  Send an ACCEPT-MIGRATION message along with ns-id identifier.
 7 |  else
 8 |  |  Send REJECT-MIGRATION message.
 9 |  end
10 |  if receive an ACCEPT-MIGRATION message from substrate node ns-id then
11 |  |  Migrate virtual node nv to substrate node ns-id.
12 |  end
13 end
```

Implementation and evaluation of the distributed resource re-optimization algorithm are work in progress which is not achieved yet.

Chapter 5

Conclusions

In this document we explain the developed algorithms to control the virtual networks SLAs. We detail the internal mechanism of our proposals and expose how they provide to the piloting plane an interface for controlling the virtual networks environment according to the desired policies and primitives.

To deal with changing virtual networking environments, adaptive provisioning algorithms are proposed to maintain virtual network topologies. The design, implementation and evaluation of an adaptive embedding framework relying on the Multi-Agent based approach is also proposed. The framework handles distributed fault-tolerant embedding and resource re-optimization algorithms to repair resource failures and dynamically optimize the substrate networks. Future work will consist of implementing the agent based adaptive embedding algorithms using the DIMA [38] - a multi-agents model proposed by one HORIZON partner.

The Piloting Plane plays an important role in the Horizon project because it works as a "brain" in the network. Hence, it must be able to react under different circumstances, such as attacks, wrong configurations, and networks that do not respect the SLAs. The proposed algorithms act pro-actively in the global optimization of the network performance.

Bibliography

[1] G. Schaffrath, C. Werle, P. Papadimitriou, A. Feldmann, R. Bless, A. Greenhalgh, A. Wundsam, M. Kind, O. Maennel, and L. Mathy, "Network virtualization architecture: proposal and initial prototype," in *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, VISA '09, (New York, NY, USA), pp. 63–72, ACM, 2009.

[2] "Geni spiral 1 annual report 2009," tech. rep., National Science Foundation, Dec. 2009.

[3] I. Houidi, W. Louati, D. Zeghlache, P. Papadimitriou, and L. Mathy, "Adaptive virtual network provisioning," in *Proceedings of the 2nd ACM workshop on Virtualized Infrastructure Systems and Architectures*, VISA '10, (New York, NY, USA), pp. 41–48, ACM, Sept. 2010.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *Operating Systems Review*, vol. 5, no. 34, pp. 217–231, Dec. 1999.

[5] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley, "Evaluating Xen for router virtualization," in *International Conference on Computer Communications and Networks*, ICCCN'07, pp. 1256–1261, Aug. 2007.

[6] S.-M. Han, M. M. Hassan, C.-W. Yoon, and E.-N. Huh, "Efficient service recommendation system for cloud computing market," in *ICIS'09: Proceedings of the 2nd International Conference on Interaction Sciences*, pp. 839–845, 2009.

[7] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li, "A simple cache partitioning approach in a virtualized environment," in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 519–524, 2009.

[8] R. McIlory and J. Sventek, "Resource virtualisation of network routers," in *Workshop on High Performance Switching and Routing*, (New Yok, NY, USA), pp. 1–6, IEEE, 2006.

[9] S. Bhatia, M. Motiwala, W. Muhlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford, "Hosting virtual networks on commodity hardware," Tech. Rep. GT-CS-07-10, Princeton University, Georgia Tech, and T-Labs/TU Berlim, Jan. 2008.

[10] *VINI - A Virtual Network Infrastructure*. http://www.vini-veritas.net/, Accessed in August 2010.

[11] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, "Fairness issues in software virtual routers," in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '10, pp. 33–38, Aug. 2008.

[12] Y. Wang, E. Keller, B. Biskeborn, J. V. der Merwe, and J. Rexford, "Virtual routers on the move: Live router migration as a network-management primitive," in *ACM SIGCOMM*, pp. 231–242, Aug. 2008.

[13] M. Zec, "Implementing a clonable network stack in the FreeBSD kernel," in *Proceedings of the 2003 USENIX Annual Technical Conference*, pp. 137–150, 2003.

[14] M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente, and H. Zhuang, "The Genesis kernel: A programming system for spawning network architectures," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 3, pp. 511–26, Mar. 2001.

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S., and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[16] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K.-K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar, "Carving research slices out of your production networks with OpenFlow," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, no. 1, pp. 129–130, 2010.

[17] E. Keller and E. Green, "Virtualizing the data plane through source code merging," in *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pp. 9–14, Aug. 2008.

[18] N. C. Fernandes, M. D. D. Moreira, and O. C. M. B. Duarte, "Xnetmon: A network monitor for securing virtual networks," in *Accepted in ICC 2011 Next Generation Networking and Internet Symposium*, ICC'11 NGNI, 2011.

[19] N. C. Fernandes, M. D. D. Moreira, I. M. Moraes, L. H. G. Ferraz, R. S. Couto, H. E. T. Carvalho, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "Virtual networks: Isolation, performance, and trends," *Annals of Telecommunications*, pp. 1–17, 2010.

[20] R. S. Couto, M. E. M. Campista, and L. H. M. K. Costa, "XTC: Um controlador de vazão para roteadores virtuais baseados em xen," in *To be published in SBRC 2011*, may 2011.

[21] W. Almesberger *et al.*, "Linux network traffic control - Implementation overview." White Paper available in http://diffserv.sourceforge.net/, 2001.

[22] R. S. Couto, H. E. T. Carvalho, L. H. G. Ferraz, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "Cpu resource allocation on xen virtual network environments," in *WNetVirt 2010 First Workshop on Network Virtualization and Intelligence for the Future Internet*, 2010.

[23] D. Ongaro, A. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in *ACM VEE*, pp. 1–10, 2008.

[24] Z. Wang, X. Zhu, and S. Singhal, "Utilization and SLO-based control for dynamic sizing of resource partitions," *Ambient Networks*, vol. 3775, no. 1, no. 1, pp. 133–144, 2005.

[25] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The TCP/UDP bandwidth measurement tool." http://dast.nlanr.net/Projects/Iperf - Accessed in January/2010, 2004.

[26] MathWorkstex, *Simulink*, 2001.

[27] J. Liu, W. Huang, B. Abali, and D. Panda, "High performance VMM-bypass I/O in virtual machines," in *USENIX*, pp. 29–42, 2006.

[28] J. Fan and M. H. Ammar, "Dynamic topology configuration in service overlay networks: A study of reconfiguration policies," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pp. 1 –12, Apr. 2006.

[29] Y. Zhu and M. Ammar, "Algorithms for assigning substrate network resources to virtual network components," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pp. 1 –12, Apr. 2006.

[30] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: substrate support for path splitting and migration," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 17–29, March 2008.

[31] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In vini veritas: realistic and controlled network experimentation," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, (New York, NY, USA), pp. 3–14, ACM, 2006.

[32] R. Ricci, C. Alfeld, and J. Lepreau, "A solver for the network testbed mapping problem," *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 65–81, April 2003.

[33] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford, "Virtual routers on the move: live router migration as a network-management primitive," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, (New York, NY, USA), pp. 231–242, ACM, 2008.

[34] W. Louati, "On demand virtual network service for dynamic networks," Ph.D. dissertation, UPMC / GET-INT, France, 2007.

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

[36] *GRID 5000.* https://www.grid5000.fr/.

[37] *JADE (Java Agent DEvelopment Framework).* http://jade.tilab.com/.

[38] Z. Guessoum, "Dima: Une plate-forme multi-agents en smalltalk," *Revue Objet*, vol. 3, pp. 393–410, dec 1998.