D4.2
Specification of Self-Management of Network Security and Resilience

Project number:	643964
Project acronym:	**SUPERCLOUD**
Project title:	User-centric management of security and dependability in clouds of clouds
Project Start Date:	1st February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Report
Reference Number:	ICT-643964-D4.2/ 1.0
Work Package:	WP 4
Due Date:	Oct 2016 - M21
Actual Submission Date:	2nd November, 2016
Responsible Organisation:	FFCUL
Editor:	Fernando M. V. Ramos, Nuno Neves
Dissemination Level:	PU
Revision:	1.0
Abstract:	In this deliverable we describe the SUPERCLOUD resilient network virtualization platform. We present its main components and the techniques used to improve the dependability, scalability, and security of the platform.
Keywords:	network virtualization, multi-cloud, software-defined networking, self-management, security

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643964.

This work was supported (in part) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

Editor

Fernando M. V. Ramos, Nuno Neves (FFCUL)

Contributors (ordered according to beneficiary numbers)

Marc Lacoste, Nizar Kheir (ORANGE)
Max Alaluna, André Mantas, Luis Ferrolho, José Soares (FFCUL)
Gregory Blanc, Fabien Charmet, Khalifa Toumi (IMT)

Disclaimer

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The users thereof use the information at their sole risk and liability.

This document has gone through the consortium's internal review process and is still subject to the review of the European Commission. Updates to the content may be made at a later stage.

Executive Summary

In this deliverable we describe the SUPERCLOUD resilient network virtualization platform. We present its main components and the techniques used to improve the dependability, scalability, and security of the platform. The architecture follows a Software-Defined Networking (SDN) approach. The main components are divided into two modules: address and topology virtualization, and virtual network embedding. Our network hypervisor provides address and topology virtualization, allowing each tenant to use the full header space and to define arbitrary topologies. The virtual network embedding module maps virtual network requests to physical resources, taking into account the security and dependability demands by users. To avoid the network hypervisor being a single point of failure, we present the design and implementation of a fault-tolerant SDN controller. Our design guarantees consistent event processing under faults, without requiring changes to current SDN protocols or switches. To allow the system to scale, we propose the design of a distributed SDN controller centered around a consistent data store that allows coordination amongst the different controllers instances. Finally, we describe the autonomic security management framework of the network virtualization platform, a component that offers data plane monitoring and proactive attack detection across the different providers of the SUPERCLOUD.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The limitations of traditional networking infrastructure allow only primitive forms of virtualization (e.g., VLANs) that lack the flexibility and scalability required in cloud environments. As such, the offer of virtualization to cloud users has been restricted to compute and storage resources until very recently.
A recent advance in computer networking – Software Defined Networking (SDN) [32] – has proved to be a key enabling technology for network virtualization, as it can support logical communication endpoints coupled with on-the-fly data forwarding reconfiguration. Newly proposed platforms [29, 3, 17] rely on SDN to offer full virtualization of the network topology and addressing schemes, while guaranteeing the required isolation among tenants.
These state-of-the-art platforms show the feasibility of network virtualization but they have been confined to a datacenter controlled by a single cloud operator. This restriction can become an important barrier as more critical applications are moved to the cloud. For instance, compliance with privacy legislation may demand certain customer data to remain local (either in an on-premise cluster or in a cloud facility located in a specific country). This sort of requirement is particularly severe in the health and public administration sectors, which normally need to resort to ad hoc approaches if they want to offload part of their infrastructure to the cloud.
Being able to leverage from several cloud providers can potentiate important benefits. First, a tenant can be made immune to any single datacenter or cloud availability zone outage by spreading its services across providers. Despite the highly dependable infrastructures employed in cloud facilities, several recent incidents give evidence that they can still generate internet-scale single points of failures [38]. Second, user costs can potentially be decreased by taking advantage of dynamic pricing plans from multiple cloud providers. Amazon's EC2 spot pricing is an example, which was recently explored to significantly reduce the costs on certain workloads when compared to traditional on-demand pricing [60]. As providers increase the support of dynamic prices, the opportunity for further savings increases with the user ability to move VMs to less costly locations. Third, increased performance can also be attained by bringing services closer to clients or by migrating VMs that at a certain point in time need to closely cooperate.
A second problem of existing SDN-based network virtualization solutions is that they do not consider security and dependability in their design. The main benefits that make SDN a suitable architecture for network virtualization – network programmability and control logic centralization – open the doors for security threats and dependability concerns that did not exist before [31]. Traditional networks have "natural protections" against network attacks, including the closed (proprietary) nature of network devices, the heterogeneity of the software, and the distributed nature of the control plane that represent defences against common vulnerabilities. This level of protection is comparatively smaller in SDNs, increasing the surface for attack.

1.1 SUPERCLOUD resilient network virtualization

In SUPERCLOUD we propose a new architecture that allows network virtualization to extend across multiple cloud providers, including a tenant's own private facilities, therefore increasing the versatility of the network infrastructure. This will allow distribution and replication of resources, enabling the platform to scale out to multiple clouds while tolerating cloud faults. In this setting, the tenant can

specify the required network resources as usual but now they can be spread over the datacenters of several cloud operators. This is achieved by creating a new network layer above the existing cloud hypervisor to hide the heterogeneity of the resources from the different providers while providing the level of control to setup the required (virtual) links among the VMs. We follow an SDN approach, where the new network layer contains an Open vSwitch (OVS) [45] that is configured by an SDN controller, in order to perform the necessary virtual-to-physical (and virtual-to-virtual) mappings and the set up of tunnels to allow the network to be virtualized.
The design of our platform includes security and dependability principles and techniques from the outset. Resilience is included by design into the platform, ensuring that correct operation can be maintained under relevant failure scenarios, encompassing both problems of accidental and malicious natures. This includes the use of replication and distribution techniques to increase the dependability and scalability of SDN control.

1.2 Outline of the document

In this deliverable we describe the SUPERCLOUD resilient network virtualization platform. As the different components of the architecture present distinct levels of maturity, an important objective of this document is to give an outlook of the current status of the design, implementation, and evaluation of each component.
Chapter 2 presents an overview of the architecture [4]. Then, Chapter 3 describes the fundamental components for virtualization: virtual network embedding; address and topology virtualization. The virtual network embedding module maps virtual network requests to physical resources, taking into account the security and dependability demands by users. Our network hypervisor provides address and topology virtualization, allowing each tenant to use the full header space and to define arbitrary topologies, while guaranteeing isolation of traffic.
Afterwards, Chapter 4 describes the dependability and scalability of the platform. We present the design and implementation of a fault-tolerant SDN controller that guarantees consistent event processing, even under fault. Importantly, this is achieved without requiring changes to current SDN protocols or switches. In this chapter we also detail the architecture of a distributed SDN controller [9]. The central component of this controller is a consistent data store that allows coordination amongst the different controllers instances.
Chapter 5 addresses security, by describing the autonomic security management framework of the SUPERCLOUD network hypervisor. This component offers data plane monitoring and proactive attack detection across the different providers of the SUPERCLOUD, thus enabling the characterization and classification of security incidents, and implementation of customizable reaction policies. Chapter 6 concludes this document and sets the main objectives for the next stage.

Chapter 2 An overview of the network virtualization architecture

In this chapter we make a description of the general framework with a high-level view of all components and services. We start by describing its main requirements in Section 2.1, as defined in SUPERCLOUD Deliverable D4.1. Then, we present a high-level view of the architecture in Section 2.2. We conclude this chapter with a preliminary evaluation of its basic functionality.

2.1 Design requirements

The network hypervisor platform leverages on network infrastructure from both public cloud providers and private infrastructures (or private clouds) of the tenants [1]. This heterogeneity impacts the level of network visibility and control that may be achieved, affecting the type of configurations that can be pushed to the network, with obvious consequences on the kind of services and guarantees that can be assured by the solution.
On one extreme case, the public cloud provider gives very limited visibility and no (or extremely limited) network control, which is often the case with commercial cloud service providers (e.g., Amazon Web Services, AWS). Even in this case, these clouds offer a full logical mesh among local VM instances (i.e., they provide a "big switch" abstraction), which we can use to implement logical software-defined datapaths and thus present a virtual network to the tenant. In this setting, one of the fundamental challenges that has to be addressed is how to interconnect the various cloud providers, as this kind of support is normally unavailable, while ensuring levels of service compatible with the tenant requirements. In these clouds only software switching can be employed, as our platform does not have access to the hardware. On the other extreme, full access may be attainable if the cloud is private (i.e., the datacenter belongs to the tenant). This results in a flexible topology that may be (partially) SDN-enabled, where both software and hardware switching may be employed. This property is particularly interesting to build hybrid solutions. Hardware switches offer high-speed packet processing (hundreds of Gbps) but have small rule tables (switch memories, TCAMs, are expensive). On the other hand, software switches do not forward packets at high speeds (10/20 Gbps per core maximum), but have a large rule space.
Considering this setting, in this deliverable we aim to fulfill four requirements in the design of our multi-cloud network hypervisor. The first requirement is to have *remote, flexible control over the network elements*. Traditional networks' lack of such control has been identified as the main reason for the limitations of current forms of network virtualization [29]. Cloud tenants define their virtual networks, which are then deployed on the clouds of various providers (and eventually on their own datacenters). In order to connect the actual elements the platform needs to be able to control the operation (namely, the packet forwarding rules) of the network devices to establish the channels that carry the traffic exchanged among the VMs. In addition, it is also necessary to devise mechanisms that allow the creation of links that cross facilities, as typical cloud offerings do not provide such support.
The second requirement is to offer *full network virtualization*, including topology and addressing abstraction, and isolation between tenants. For topology abstraction, different mappings should be created when the network is setup. For instance, a virtual link can correspond to multiple network

[1] In this document we use the terms *user* and *tenant* interchangeably.

paths connecting the two endpoints. Virtual switches may hide large portions of the infrastructure, where several (physical) switches have to be interconnected in order to provide the required abstraction. In addition, tenants should have complete autonomy to manage their own address space of the virtual network. Lastly, isolation between users should be enforced at different levels. A first level is attained by separating the virtual networks (VN) of the users and then hiding VN instances deployed by different customers from one-another. A second level is to prevent the actions of one user to influence the network behaviour observed by the others. For example, if one of the users attempts to clog a particular link, this should not cause a significant decrease of the bandwidth available to the other users.
The third requirement is *security and dependability* of the infrastructure. Tenants should observe an available network, where communications can occur with high probability. Users should also experience a secure network, where appropriate measures are applied to protect the communications from attacks. In SDN technologies, the controller is logically centralized, and therefore it can become a bottleneck in data processing or a single point of failure.
The final requirement is *scalability and performance*. Tenants should be able to deploy arbitrarily sized virtual networks without significant performance degradation. The network virtualization solution should not introduce constraints on the number of VMs, network devices and amount of traffic in the virtual infrastructure, as long as enough (physical) resources are made available to support them. The platform should not introduce unnecessary overheads in the physical network.

2.2 Architecture

Recently proposed platforms for network virtualization [29, 3, 17] share several common characteristics. First, they target datacenter environments where there is a high level of control over the resources. Second, they rely on logically centralized control to achieve full network virtualization. An important differentiating factor of our solution arises from tackling the challenges of using multiple clouds, including public clouds on which we have very limited control. In addition, its particular focus on resilience (security and dependability), and the integration of mechanisms in its design such as network snapshot and migration. The high-level view of the network virtualization architecture we propose is shown in Figure 2.1.

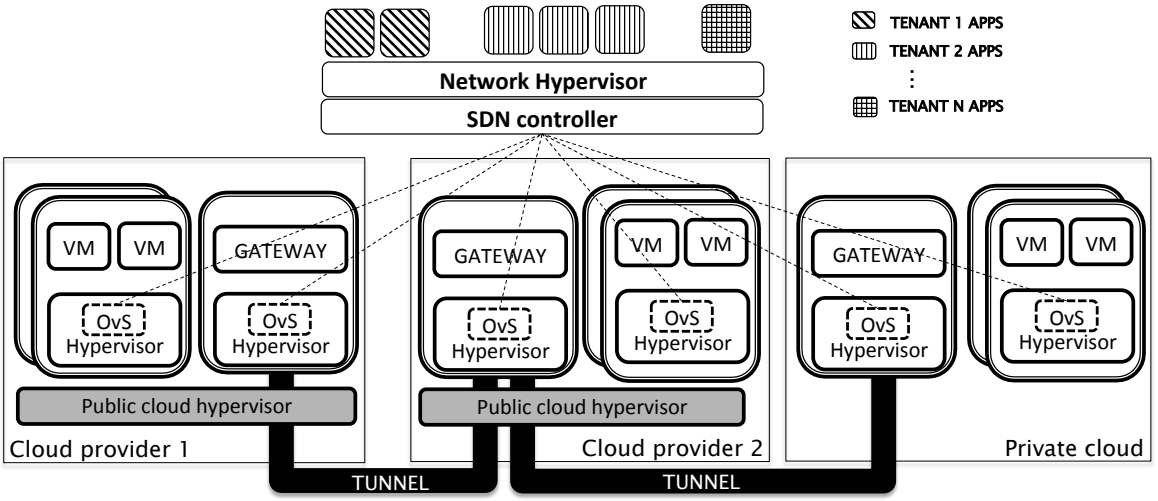


Figure 2.1: Network virtualization architecture

The network hypervisor controls and configures the OVS switches that are installed in all VMs (along with the OpenFlow hardware switches that may be present in the private cloud). This hypervisor is built as an application that runs in the SDN controller, similar to NVP [29] and FlowN [17], and in contrast with the proxy-based approach followed by OpenVirteX [3]. Each cloud has a specific VM,

the gateway, that establishes tunnels with another clouds. As such, only one public IP address per cloud is needed in our solution. We build a minimum spanning tree to minimize the number of tunnels needed. In a distributed configuration the gateway also hosts an instance of the SDN controller. For each tenant, a specific set of network applications that control the tenants' virtual network runs on top of the network hypervisor.
The design of the architecture aims to fulfil the requirements defined before. To fulfil the first requirement – network control – the platform resorts to the SDN paradigm [32]. The data plane element of our solution is Open vSwitch (OVS) [45], a software switch for virtualized environments that resides within the hypervisor or management domain. OVS exports an interface for fine-grained control of packet forwarding (via OpenFlow [40]) and of switch configuration (via OVSDB [44]). This allows SDN-based logically centralized control.
In public clouds, our platform does not have access to the cloud hypervisor. For this reason, we have an additional virtualization layer on top of the cloud hypervisor to provide virtualization between multiple tenants. OVS is part of this hypervisor that runs inside each VM. Private clouds include the proposed network hypervisor, with OVS, running on bare metal.
For the second requirement – full network virtualization – the network hypervisor has to guarantee isolation between tenants, while enabling them to use their desired addressing schemes and topologies. Our network hypervisor runs on top of the SDN controller to map the physical to virtual events by intercepting the flow of messages between the physical network and the users' applications. This, along with flow rule redefinition at the edge of the network, allows isolation between tenants' networks. For addressing virtualization, the traffic that originates from tenant VMs is all tagged with a tenant ID. As a consequence, each tenant is able to use the full IP address range and the full MAC address (that is, complete L2 and L3 virtualization). We discuss these techniques in more detail in the next chapter.
Having a resilient network virtualization platform – our third requirement – requires resilience of the SDN infrastructure. Our controller is thus fault-tolerant, giving also strong guarantees of consistency and correctness. The consistency of both data and control planes (and their interaction) is necessary to avoid network anomalies (loops or/and security breaches).
Security is considered along three axes. First, secure channels are used in the control plane connection (for communication between the SDN controller and all switches) and for the inter-cloud links. Second, virtual network embedding – the process that maps virtual network requests to the physical resources – includes security and dependability constraints set by the user. Third, our solution entails autonomic security management, including data plane monitoring and proactive attack detection across the different providers of the SUPERCLOUD.
Finally, to guarantee the scalability and performance of the network virtualization platform, the controller used in our solution is distributed. As distribution brings with it network state consistency challenges, we consider techniques to guarantee consistency and correctness between the different controller instances.
As a summary, Figure 2.2 illustrates all components of our modular network hypervisor: the module for addressing virtualization, topology abstraction, and isolation; virtual network embedding; and autonomic security management. The hypervisor is replicated for fault-tolerance. This is represented in the figure by having different boxes of the same colour managing the same part of the infrastructure. The hypervisor is also distributed for scalability. Different instances of the hypervisor (represented with different colours) manage different parts of the infrastructure.

2.3 Preliminary implementation and evaluation of basic functionality

The first prototype of our network hypervisor consists of nearly 4000 lines of Java code and is implemented as a module of the Floodlight controller. GRE tunnels are used between the gateways, and a proactive SDN approach is used. The flows rules are installed in the switches when the virtual networks are set up.

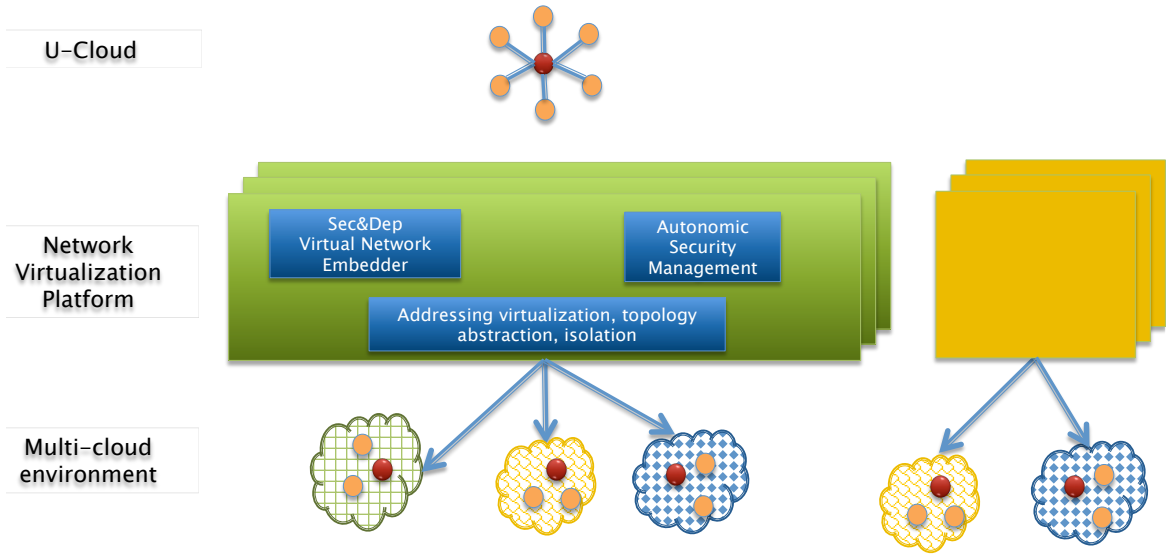


Figure 2.2: Modular architecture of the network hypervisor

The evaluation of our solution answers two main questions. First, it evaluates the cost of deploying the environment by analysing the different components that make up the setup time. In particular, we study how the creation of tunnels and the tunnel topology itself influence the setup time, and how this variable scales with network size. Second, we evaluate the overhead introduced by our virtualization layer. We analyse the overhead of the control plane by measuring the controller's request processing time. For the data planes we evaluate the overhead in terms of throughput and communications latency.

The experiments were run on a testbed composed of two servers equipped with 2 Intel Xeon E5520 quad-core, HT, 2.27 GHz, and 32 GB RAM. The hypervisor used is XenServer 6.5, running OVS 2.1.3. There is a router between the servers to simulate a multi-cloud environment. One of the servers hosts one VM dedicated to the Floodlight controller, and another to host Mininet 2.2.0.

Setup time. The setup time is the time between the moment the tenant submits a virtual network request until the instant when the whole network components are initialized and instantiated. This time is composed of two components: time to populate network state in the resilient network hypervisor, and time to configure and initialize all tunnels.

We compare two different tunnel topologies. The first is a setup with a full mesh of tunnels between all VMs, creating a one-hop tunnel between each pair of VMs, to serve as baseline. The second is our solution: we set up a minimum spanning tree (MST) between those same VMs. The objective was to compare between the two extreme scenarios. The results are shown in Figure 2.3.

As expected, for the MST case the setup time grows linearly with network size. By contrast, a full mesh has an $O(n^2)$ cost, and hence the setup time grows quadratically. As can be seen in the full mesh case, tunnel creation has a visible effect on setup time as the network grows, making it a fundamental component for large scale scenarios. This motivates the need to minimize the number of tunnels for the system to scale. In any case, these setup times are still two to three orders of magnitude below the time to provision and boot a VM in the cloud [35]. For scalability sake it is therefore important to use a minimum spanning tree as the base for tunnel setup, adding redundant tunnels as needed to increase network resilience.

Control plane overhead. We measure the cost of network virtualization in the control plane using cbench [55], a control plane benchmarking tool that generates packet-in events for new flows. In this test, cbench is configured to spawn a number of switches equal to the number of virtual networks, each switch having 5 hosts with unique MAC addresses. The tests are run with cbench in latency mode. In this mode cbench sends a packet-in request and waits for a response before sending the next request. This allows measuring the controller's request processing time. We consider two scenarios:

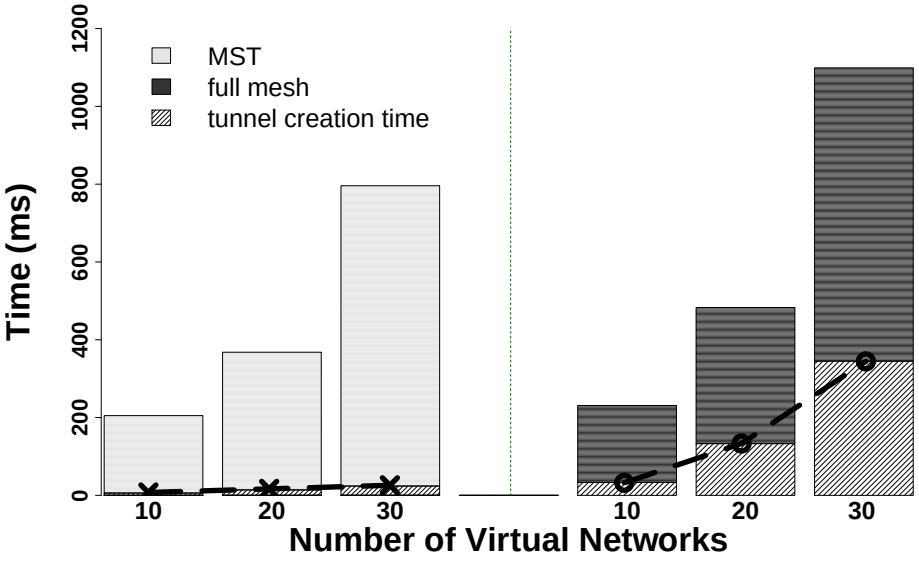


Figure 2.3: Setup time (left: MST; right: full mesh)

one with network virtualization, and another without network virtualization. We present the results in Figure 2.4.

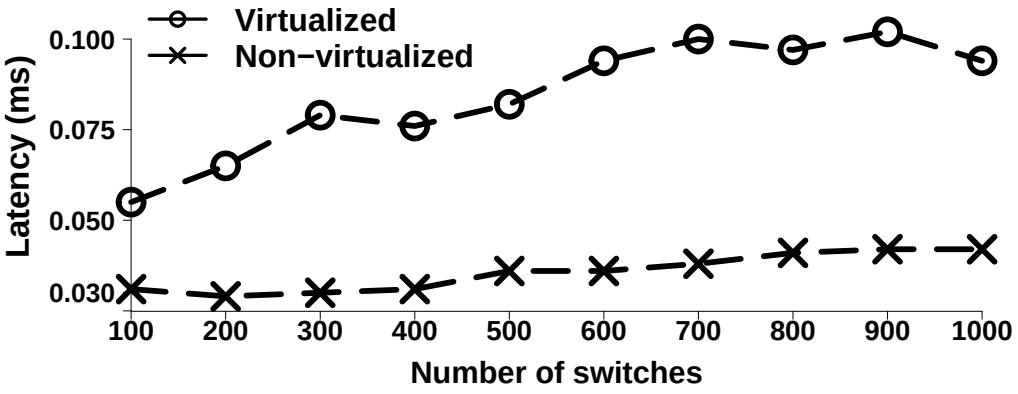


Figure 2.4: Control plane overhead

As can be seen, the virtualization layer adds a very small overhead of less than 0.1 ms compared to the baseline. Importantly, the latency overhead is mainly independent of network size (i.e., as the network grows the latency overhead remains relatively stable). Further, for multi-cloud scenarios the inter-cloud latency is in the order of the tens of hundreds of milliseconds [35], and hence this overhead is negligible.

Data plane overhead. To evaluate data plane overhead, we make two experiments. We measure network latency by running several pings between two virtual machines executing in different servers (emulating different clouds). To measure network throughput we run netperf's TCP_STREAM test between those same virtual machines. Again, we consider two scenarios: one virtualized and one non-virtualized.

The results are shown in Figure 2.5. The virtualization layer introduces an overhead, in particular at very high bit rates. The cost is particularly clear in the 10G experiment, with the maximum throughput reduced to half the original value when virtualization is introduced. The latency also increases more than 50% in this worst case.

The virtualization overhead is mainly due to the use of tunnels. Our context precludes the use of hardware offload techniques, such as those used in NVP [29], to increase performance, as in public

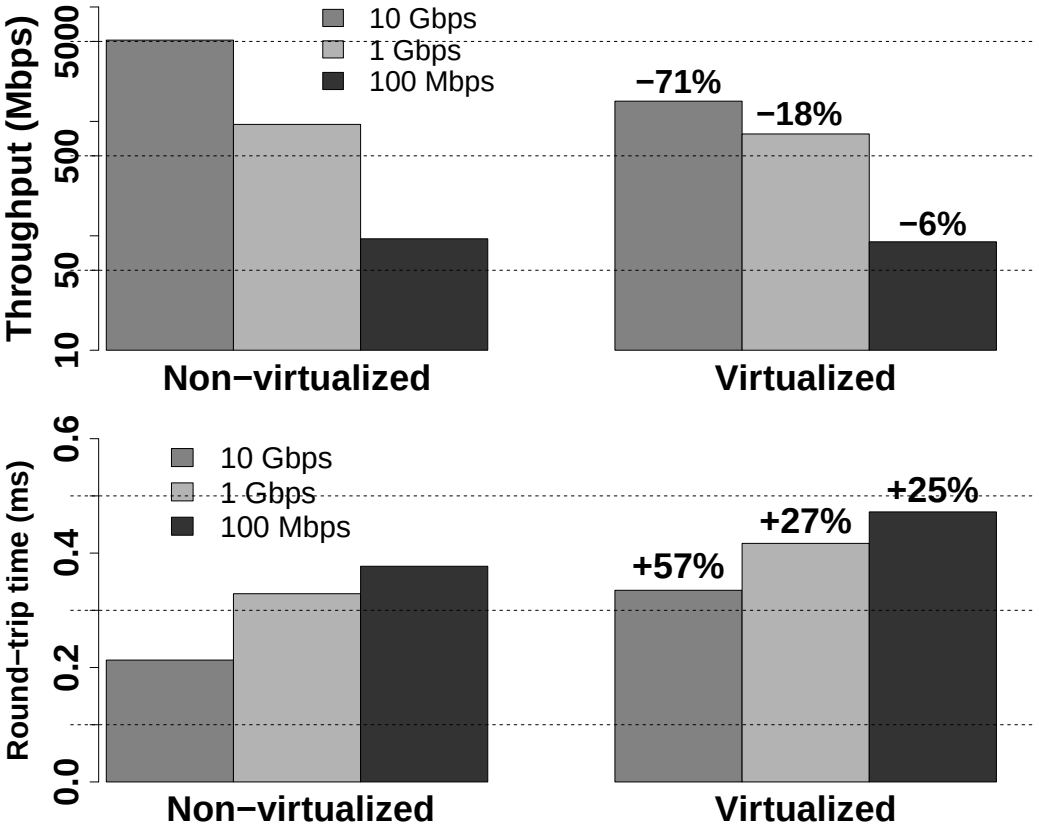


Figure 2.5: Data plane overhead

clouds we do not have access to the cloud virtualization layer, nor to the hardware. This motivates the need to minimize the use of tunnels by increasing traffic locality as much as possible. This can be done by maintaining VMs that communicate frequently closer to each other. For instance, VM migration could be triggered when this type of communication pattern is detected. Anyway, for the multi-cloud scenarios we target the inter-cloud throughput is in the order of the hundreds of Mbps [35]. At these rates, the overhead is relatively low (a reduction of 6% in throughput). The additional latency is also negligible when compared with typical inter-cloud latencies.

Chapter 3 Network virtualization components

In this chapter we present the components of the SUPERCLOUD platform that enable network virtualization: virtual network embedding, and address & topology virtualization. Then, in Chapter 4 we focus on the dependability and scalability of the solution, and in Chapter 5 we describe its security autonomic management module.
In Section 3.1 we address the virtual network embedding module, responsible for mapping the virtual network requests to the SUPERCLOUD physical resources, taking into account the security and dependability demands by users. Then, in Section 3.2 we detail the techniques we propose for address and topology virtualization. These allow SUPERCLOUD users to choose any L2 (Ethernet MAC) or L3 (IP) address – our network hypervisor offers full address virtualization – and any topology.

3.1 Virtual network embedding

In this section we tackle a fundamental issue in network virtualization – the *Virtual Network Embedding (VNE) problem* – from the SUPERCLOUD perspective. VNE addresses the problem of providing the virtual networks specified by a tenant while making efficient use of the shared resources. The goal is to find an effective mapping of the virtual nodes and links onto the substrate network. The VNE problem has been widely studied in the literature [2]. This problem is traditionally formulated with the objective of maximizing network provider revenue by efficiently embedding incoming virtual network (VN) requests. This objective is subject to constraints, such as processing capacity on the nodes and bandwidth resource on the links.
A mostly unexplored perspective on this problem is providing security and dependability, an important set of requirements of SUPERCLOUD. Indeed, to shift their workloads to the cloud, tenants trust their cloud providers to guarantee that their workloads are secure and available. Unfortunately, there is increasing evidence that problems do occur, of both benign nature (e.g., a cloud outage), and malicious (e.g., caused by a corrupt cloud insider) [38]. We thus argue that *security and dependability are becoming critical factors that should be considered by virtual network embedding algorithms.*
In this section we propose a VNE solution that considers security and dependability as first class citizens. We formulate the problem as a Mixed Integer Linear Program (MILP)[1]. In order to fulfil SUPERCLOUD requirements, we introduce specific *security constraints* including, for instance, the possibility of a virtual machine attacking another virtual machine (e.g., a side-channel attack) or replay attacks on physical links. As substrate resources may fail, we also take into account *dependability constraints*, including the ability to tolerate failures, by ensuring that additional computing and communication resources are allocated during the process of embedding. The resiliency properties of our solution are further extended by assuming the multiple cloud provider model of SUPERCLOUD. We consider the coexistence of multiple clouds, both public and private, and assume that each individual cloud may offer potentially different levels of trust for a user. A private facility is in principle more trustworthy, while different public clouds may offer a different level of security and trust (due to its location and/or its particular infrastructure). By not relying on a single cloud provider we avoid internet-scale single points of failures, avoiding cloud outages by replicating workloads across clouds.

[1] A Mixed Integer Linear Programming problem is a mathematical optimization program in which some of the variables are constrained to be integers, while other variables are allowed to be non-integers.

In addition, security can be enhanced by leaving sensitive workloads in the tenant's private clouds or on trustworthy public alternatives.

3.1.1 Background and Related Work

The main resource allocation challenge in network virtualization is the problem of embedding virtual networks in a substrate network. This is usually referred to as the Virtual Network Embedding (VNE) problem. The goal is to maximize the benefit gained from the efficient mapping of virtual resources onto the physical hardware.
There is already a rich literature on this problem [2]. Yu et al. [58] where the first to solve it efficiently, by assuming the capability of path splitting (multi-path) in the substrate network, which enable the computationally harder part of the problem to be solved as a multicommodity flow (MCF), for which efficient algorithms exist. The authors solve the problem considering two independent phases – an approach commonly used by most algorithms. In the first phase, a greedy algorithm is used for virtual node embedding. Then, to map the virtual links, either efficient MCF solutions or k-shortest path algorithms can be used. In [14], Chowdhury et al. proposed two algorithms for VNE that introduce coordination between the node and link mapping phases. The main technique proposed in this work is to augment the substrate graph with meta-nodes and meta-links that allow the two phases to be well correlated, achieving more efficient solutions.
As failures in networks are inevitable, the issue of failure recovery and survivability in VNE has gained attention recently. H. Yu et al. [57] have focused on the failure recovery of nodes. They proposed to extend the basic VNE mapping with the inclusion of redundant nodes. Rahman et al. [48] formulated the survivable virtual network embedding (SVNE) problem to incorporate single substrate link failures. Other works, such as [56], aim to achieve efficient dependable embedding, for example by pooling backup resources.
A mostly unexplored perspective on VNE problem is providing security guarantees. Fischer et al. [19] have introduced this problem and considered including constraints in the VNE formulation that take security in consideration. The authors proposed the assignment of security levels to every physical resource, allowing virtual network requests to include security demands. Liu et al. [37] have proposed VNE algorithms based on this idea.
More recently a few works started considering cloud environments. Nonde et al. [41] have proposed energy-efficient VNE, where power savings are introduced by consolidating resources in datacenters. The setting of this work is, however, a single datacenter.

3.1.2 Problem description

The multi-cloud environment of SUPERCLOUD increases the flexibility, dependability, and security of the network virtualization solution. This increase in the options offered to users makes the problem different from the VNE problems considered to this date. In our environment, when a user wants to instantiate a virtual network (VN), besides the processing (CPU) capacity for its nodes and the bandwidth resources for its links, it may also include security and dependability demands. To fulfill these requirements, we give users the possibility to choose specific security and dependability levels for its VN (namely, to its virtual nodes and virtual links). Figure 3.1 gives an example of typical security and dependability levels that can be chosen by the user for its virtual resources.
In the example, virtual nodes can be embedded onto machines with three different types of security: normal containers, normal VMs, or secure VMs (e.g., VMs that include trusted components, such as a Trusted Platform Module, TPM). Users also can choose the security requirements for their virtual links: default security, an intermediate security level (e.g., where authenticity and integrity are guaranteed), or a higher security level (e.g., where authenticity, integrity, and confidentiality are guaranteed). Finally, users can choose the type of cloud where they want their virtual nodes to be located. In the example, we define three types of clouds: public clouds (belonging to cloud providers), trusted public clouds (belonging to cloud providers that are considered more trustworthy), and private

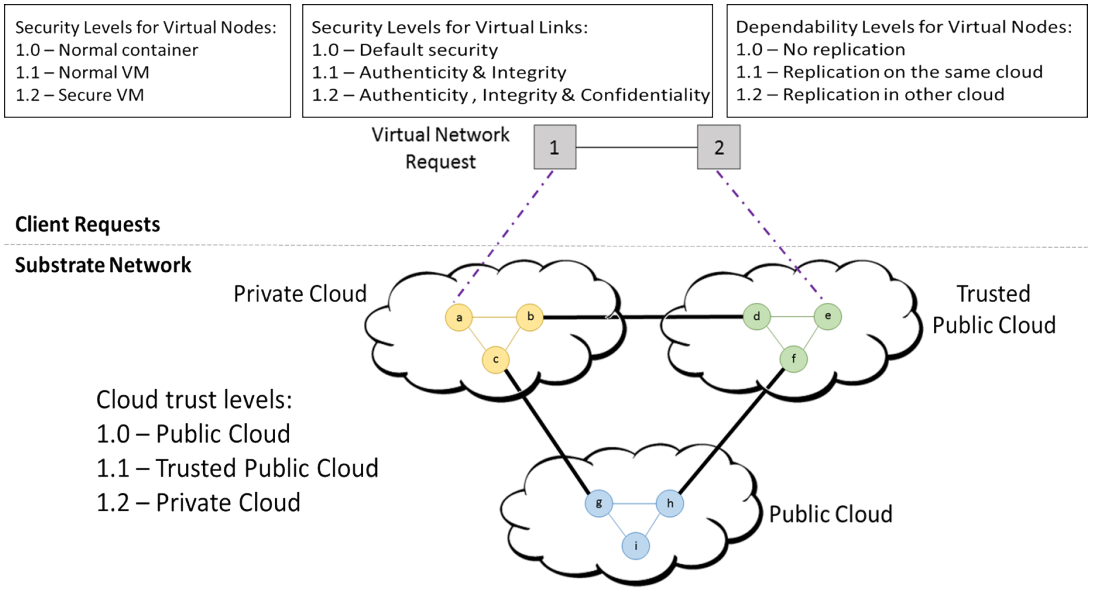


Figure 3.1: Possible levels of security and dependability the user can choose for its virtual network

clouds (belonging to the tenant, assumed to be the most secure option). With these options, the user may choose private clouds or trusted public clouds for more sensitive workloads, while leaving the others in public clouds, to scale out.
Besides security, tenants also can require backups for their virtual resources, for fault-tolerance. To ensure dependability additional physical resources need to be allocated in the substrate network (SN) to the user's VN. In addition, when replication is required tenants can choose the location of each backup node, further improving dependability. To avoid cloud outages (caused by a natural disaster or a malicious attack), they may choose replication for virtual nodes in different clouds. Note that when dependability is required substrate paths between the backup nodes of a VN need to be set up. With the features of our solution described, we now define the Secure and Dependable Virtual Network Embedding (SecDep VNE) problem:

> Given the virtual network G^V with resources requested and corresponding security and dependability requirements, and substrate network G^S with resources to serve incoming Virtual Network Requests (VNRs), can G^V be mapped to the substrate network with the minimum resources while satisfying the following? (i) Each virtual node and link is mapped to the substrate network meeting the CPU capacity and bandwidth constraints, respectively, and also the security and dependability constraints, namely of node security type, node location, node backup, and link security type; (ii) Each virtual node is mapped to a substrate node that is located in a cloud that covers its cloud type requirements; (iii) The virtual network is protected against faults in the substrate network or cloud outages, when backups are required by the user.

Our model handles the SecDep VNE problem, trying to map a VN onto a SN while respecting all the requirements and constraints. When a VNR arrives, our solution tries to find the best mapping for the VN while reducing the costs of embedding it (i.e., reduce the total quantity of substrate resources allocated to it). If there is no possible solution to embed the incoming VN, then the VN is rejected. When a solution is found, the quantity of resources demanded by the VN is allocated.
In our formulation, after the embedding of a VN the SN is augmented with the virtual nodes that were embedded. These virtual nodes have meta-links with the substrate nodes on which they are mapped.

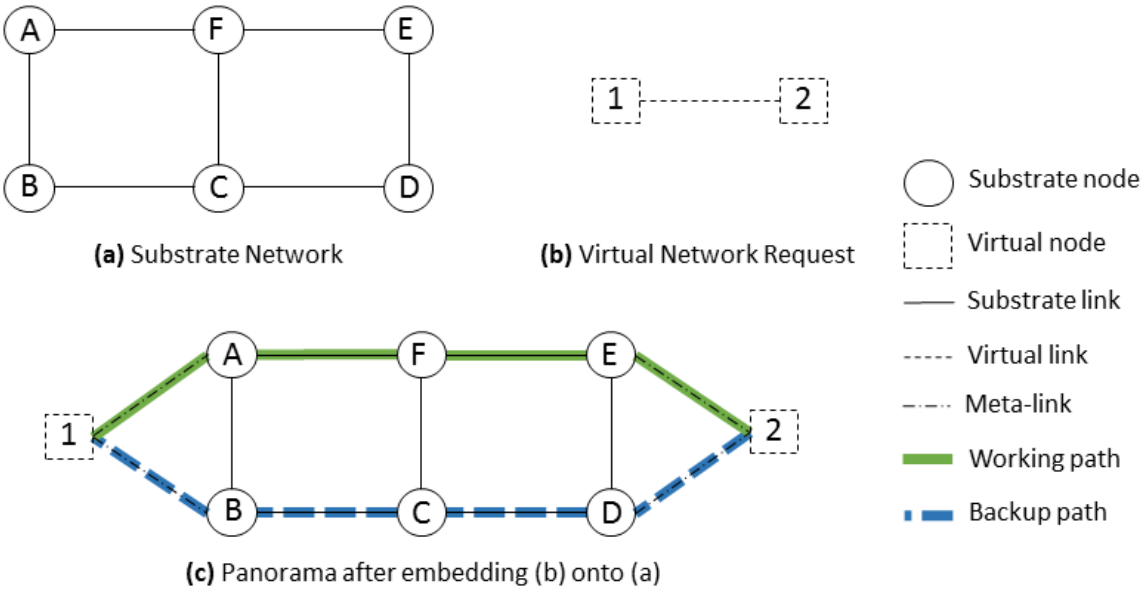


Figure 3.2: (a) Simple substrate network. (b) Simple virtual network request that arrives to be embedded. (c) Example of an embedding result of (b) onto (a) after the execution of our MILP formulation (supposing that all constraints are achieved - resource capacity, security, cloud type and dependability constraints).

The meta-links allow a certain virtual node to be mapped onto a certain substrate node.
In Figure 3.2(c) we illustrate the result of embedding the VN request presented in 3.2(b) onto the SN presented in 3.2(a) (assuming that all constraints are guaranteed). Virtual node 1 has a meta-link with substrate node A, and virtual node 2 has one with E, which are their primary nodes. They also have meta-links with substrate nodes B and D, the backup nodes that are used when a failure occurs. In this figure it is also possible to observe that the substrate paths (both working and backup) correspond to more than one substrate link (e.g., the substrate links (A,F) and (F,E) are assigned to the working path).

3.1.3 Network Model

In this section we describe the characteristics and attributes that define a physical and a virtual network. More precisely, we describe each attribute of the network resources (its nodes and links).

3.1.3.1 Substrate Network

We model the substrate network as a weighted undirected graph. It is denoted by $G^S = (N^S, E^S, A_N^S, A_E^S)$, where N^S is the set of substrate nodes, E^S is the set of substrate links, A_N^S is the set of attributes of substrate nodes, and A_E^S is the set of attributes of substrate links.
A_N^S contains the following attributes for substrate nodes:

$$A_N^S = \{\{cpu^S(n), sec^S(n), cloud^S(n)\} | n \epsilon N^S\}$$

- $cpu^S(n)$ - Total CPU capacity of the substrate node n. This attribute can take any value between 0 and 1.
- $sec^S(n)$ - Security provided by the substrate node n. This attribute can take any positive integer (including zero) as its value. In the example given above in Figure 3.1, if node n is a normal container, then $sec^S(n) = 0$. If it is a normal VM, $sec^S(n) = 1$. Lastly, if n is a secure VM, then $sec^S(n) = 2$.

- $cloud^S(n)$ - Defines in which type of cloud the substrate node n is located. This attribute can take any positive integer (including zero) as its value. In the example, if n is located in a public cloud, then $cloud^S(n) = 0$. If it is located in a trusted public cloud, then $cloud^S(n) = 1$. Lastly, if n is located at a private cloud, then $cloud^S(n) = 2$.

A_E^S contains the following attributes for substrate links:

$$A_E^S = \{\{bw^S(l), sec^S(l)\}|l \epsilon E^S\}$$

- $bw^S(l)$ - Total bandwidth capacity of the substrate link l. This attribute can take any values greater or equal to 0.
- $sec^S(l)$ - Security provided by the substrate link l. This attribute can take any positive integer (including zero) as its value. In the example, $sec^S(l) = 0$ if link l only provides simple security mechanisms defined as default. If link l supports protocols that provide authenticity and integrity guarantees, then $sec^S(l) = 1$. Lastly, if l supports the use of protocols that provide authenticity, integrity and confidentiality guarantees, then $sec^S(l) = 2$.

Figure 3.3 shows an example of a SN. Substrate nodes A and D are in the same cloud (a trusted public cloud) and B and C are in a public cloud. Links (D,A) and (B,C) are intra-domain links; links (A,B) and (C,D) are inter-domain links. It is also possible to observe the diversity of the resource attributes. Some nodes are more secure (e.g., node A, a secure VM), some have weaker computing or bandwidth capacities, such as node C or link (A,B), etc.

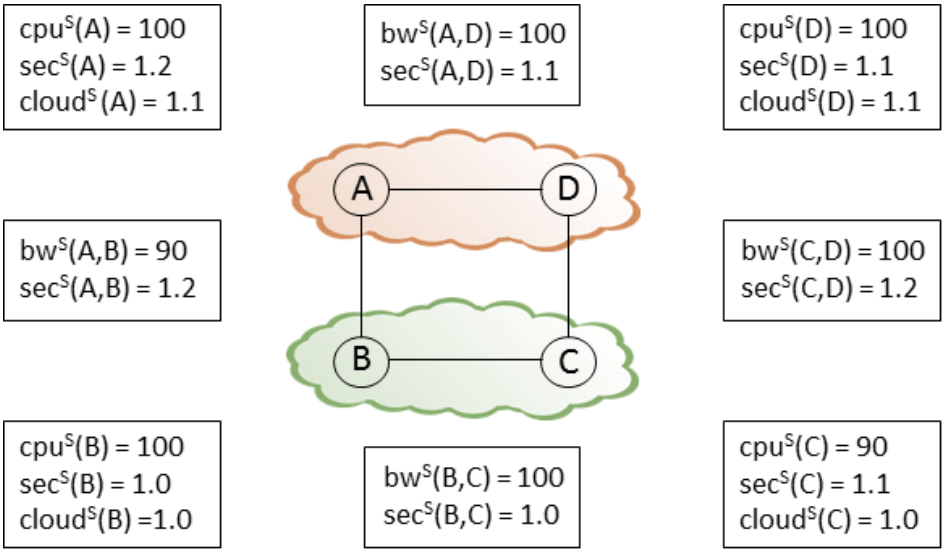


Figure 3.3: Example of a SN G^S where $N^S = \{A, B, C, D\}$ and $E^S = \{(A, B), (B, C), (C, D), (D, A)\}$. The sets A_N^S and A_E^S (i.e., the attributes of the substrate resources) are presented in the figure.

3.1.3.2 Virtual Network Requests

VNRs are defined by the users of the system. Similar to the substrate network, the VNRs are also modeled as weighted undirected graphs. Each virtual network request is denoted by $G^V = (N^V, E^V, Time^V, Dur^V, A_N^V, A_E^V)$, where N^V is the set of virtual nodes, E^V is the set of virtual links, $Time^V$ is the arrival time of the VNR, Dur^V is the time period for which the VN is valid, A_N^V is the set of attributes of substrate nodes, and A_E^V is the set of attributes of substrate links.
A_N^V contains the following attributes demanded by virtual nodes:

$$A_N^V = \{\{cpu^V(n), sec^V(n), cloud^V(n), dep^V(n)\} | n \epsilon N^V\}$$

- $cpu^V(n)$ - CPU capacity demanded by the virtual node n. This attribute can take any value between 0 and 1.
- $sec^V(n)$ - Security demanded by the virtual node n. This attribute can take any positive integer (including zero) as its value, similar to the SN case above.
- $cloud^V(n)$ - Defines the type of cloud on which the virtual node n should be mapped. This attribute can take any positive integer (including zero) as its value, similar to the SN case above.
- $dep^V(n)$ - Defines where the backup of virtual node n should be mapped. This attribute can take any positive integer (including zero) as its value. Considering the same example, if replication is not needed for the VNR , then $dep^V(n) = 0$. If virtual node n should have a backup in the same cloud, then $dep^V(n) = 1$. Finally, if n should have a backup located in another cloud (e.g., in order to survive to a cloud outage), then $dep^V(n) = 2$.

A_E^V contains the following attributes demanded by virtual links:

$$A_E^V = \{\{bw^V(l), sec^V(l)\} | l \epsilon E^V\}$$

- $bw^V(l)$ - Bandwidth capacity demanded by the virtual link l. This attribute can take any value greater than 0.
- $sec^V(l)$ - Security demanded by the virtual link l. This attribute can take any positive integer (including zero), again similar to the SN case.

Figure 3.4 shows an example of a VNR. Similar to Figure 3.3, here it is also possible to observe the diversity of the virtual resources. The user requires virtual node 1 to be mapped onto a substrate node that is located in a trusted public cloud. In addition, it requires its backup to be located in another cloud with the same level of security. Virtual node 2 can be mapped onto a substrate node that is located in a public cloud, and its backup can be located in the same cloud. Virtual link (1,2) can be mapped onto one or more substrate links, but requires all constituent links to have an equal or greater security level than the one required. In this case, the physical links where virtual link (1,2) will be mapped should have $sec^S = 1$ at least.

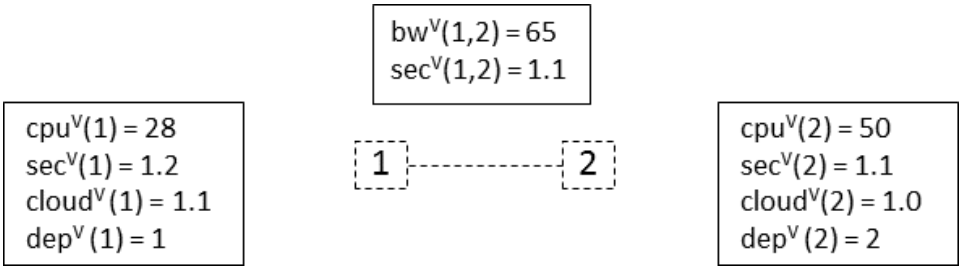


Figure 3.4: Example of a VN G^V where $N^V = \{1, 2\}$ and $E^V = \{(1, 2)\}$. The sets A_N^V and A_E^V, i.e., the demands of virtual nodes and links, are presented in the figure.

It is important to note that substrate resources with higher security (or dependability) levels can map virtual resources that have lower or equal security level demands. For instance, a substrate node with $sec^S = 2$ and $cloud^S = 2$ (located in a private cloud) can map either one of the virtual nodes presented in Figure 3.4, since it provides the highest level of security demanded by virtual nodes.

3.1.3.3 Measurement of Substrate Network Resources

The residual capacity (or available capacity) of a substrate node, $R_N(n^S)$ is defined as the available CPU capacity of the substrate node $n^S \epsilon N^S$.

$$R_N(n^S) = cpu^S(n^S) - \sum_{\forall n^V \uparrow n^S} cpu^V(n^V),$$

where $n^V \epsilon N^V$ and $x \uparrow y$ denotes that the virtual node x is hosted on the substrate node y.
Similarly, the residual capacity of a substrate link, $R_E(e^S)$ is defined as the total amount of bandwidth available on the substrate link $e^S \epsilon E^S$.

$$R_E(e^S) = bw^S(e^S) - \sum_{\forall e^V \uparrow e^S} bw^V(e^V),$$

where $e^V \epsilon E^V$ and $x \uparrow y$ denotes that the flow of the virtual link x traverses the substrate link y.
Since a virtual link can be mapped onto multiple substrate links, i.e., mapped onto a substrate path, it is also important to define the available bandwidth capacity of a substrate path P. This can be defined as the minimum available bandwidth among all the substrate links belonging to P.

$$R_E(P) = \min_{e^S \epsilon P} R_E(e^S)$$

3.1.3.4 Objectives

The main goal of VNE is to maximize the profit of the SUPERCLOUD provider. For this purpose, and similar to [58, 15], we define the revenue of accepting a VNR as:

$$\mathbb{R}(G^V) = \lambda_1 \sum_{e^V \epsilon E^V} bw^V(e^V)\ sec^V(e^V) + \lambda_2 \sum_{n^V \epsilon N^V} cpu^V(n^V)\ sec^V(n^V)\ cloud^V(n^V),$$

where λ_1 and λ_2 are weights that denote the relative proportion of each revenue component to the total revenue. We will address these weights later in Section 3.1.4.2. Although the revenue gives us an idea of how much the SUPERCLOUD provider will gain by accepting a certain VNR, it is not useful if we do not know the cost the provider will incur for embedding that request. Therefore, it is also necessary to define the cost. The cost of embedding a VNR can be defined as the sum of total substrate resources allocated to that VN. Normally, the numerical cost of embedding a VNR is equal or higher than the revenue generated by that request. This is due to the fact that virtual links may be embedded to one or more physical links. In our work, the cost may also increase if the VNR requires backups or higher security or dependibility for its virtual nodes and links. We define the cost of embedding a VNR as:

$$\mathbb{C}(G^V) = \lambda_1 \sum_{e^V \epsilon E^V} \sum_{e^S \epsilon E^S} f_{e^S}^{e^V}\ sec^S(e^S) + \lambda_2 \sum_{n^V \epsilon N^V} \sum_{n^S \epsilon N^S} cpu_{n^S}^{n^V}\ sec^S(n^S)\ cloud^S(n^S),$$

where $f_{e^S}^{e^V}$ denotes the total amount of bandwidth allocated on the substrate link e^S for virtual link e^V and $cpu_{n^S}^{n^V}$ denotes the total amount of CPU allocated on the substrate node n^S for virtual node n^V (either working or backup parts). λ_1 and λ_2 are weights that denote the relative proportion of each cost component to the total cost.

3.1.4 MILP Formulation for Sec&Dep VNE

We have developed a MILP formulation to solve the Secure and Dependable VNE problem. In this section we start by explaining the variables used in our formulation, the objective function, and finally the constraints that were defined to solve the problem.

3.1.4.1 Variables

In Table 3.1 the variables that are used in our MILP formulation are showed (and explained). Briefly, $wf_{u,v}^{i,j}$, $bf_{u,v}^{i,j}$, $wl_{u,v}^{i,j}$, $bl_{u,v}^{i,j}$ and $rl_{u,v}$ are related to working and backup links; $wn_{i,v}$, $bn_{i,v}$ and rn_v are related to working and backup nodes; $wc_{i,c}$ and $bc_{i,c}$ are related to the embedding location of virtual nodes.

Symbol	Meaning
$wf_{u,v}^{i,j}$	The amount of working flow, i.e., bandwidth, on the physical link (u,v) for the virtual link (i,j)
$bf_{u,v}^{i,j}$	The amount of backup flow, i.e., backup bandwidth, on the physical link (u,v) for the virtual link (i,j)
$wl_{u,v}^{i,j}$	Denotes whether the virtual link (i,j) is mapped on to the physical link (u,v). (1 if (i,j) is mapped on (u,v), 0 otherwise)
$bl_{u,v}^{i,j}$	Denotes whether the backup of virtual link (i,j) is mapped onto the physical link (u,v). (1 if backup of (i,j) is mapped on (u,v), 0 otherwise)
$rl_{u,v}$	The reserved backup resources on a physical link (u,v), i.e., the total quantity of bandwidth that is allocated for backup flows.
$wn_{i,v}$	Denotes whether virtual node i is mapped onto the physical node v. (1 if i is mapped on v, 0 otherwise)
$bn_{i,v}$	Denotes whether virtual node i's backup is mapped onto the physical node v. (1 if i's backup is mapped on v, 0 otherwise)
rn_v	The reserved backup resource on a physical node v, i.e., the total quantity of CPU that is allocated to backups.
$wc_{i,c}$	Denotes whether virtual node i is mapped on cloud c. (1 if i is mapped on c, 0 otherwise)
$bc_{i,c}$	Denotes whether virtual node i's backup is mapped on cloud c. (1 if i's backup is mapped on c, 0 otherwise)

Table 3.1: List of all the variables used in our MILP formulation

3.1.4.2 Objective Function

The objective function of our formulation has three goals: to minimize 1) the sum of all computing costs, 2) the overall number of hops of the substrate paths for the virtual links, and 3) the overall number of hops of the substrate paths for the virtual links.
Since we have different objectives, and these objectives are measured in different units, we have to unify them. Thus, in our formulation we consider a weighted sum function with three different weights, λ_1, λ_2, and λ_3, which should be reasonably parameterized for each objective, in order to mitigate the differences between the units.
The first and the second parts of Eq. 3.1 are the sum of all working and backup link bandwidth costs, respectively. The third and the fourth parts are the sum of all working and backup computing node costs. In this function, the level of security provided by the physical resources is considered. The parameter α is a weight for each physical link that assumes some value defined previously and that depends if (u,v) is an inter-cloud connection (link between two clouds) or an intra-domain link (link inside a cloud). This is due to the expectation that virtual links that use links connecting two clouds (inter-domain links) to have higher cost (monetary, delay, or other). Also, mapping a VN onto

substrate resources that provide higher security or dependability requirements are expected to increase costs. The fifth and last parts of the objective function aim to address this cost.

$$
\begin{aligned}
min \quad & \lambda_1 \sum_{(i,j)\epsilon E^V} \sum_{u,v\epsilon N^S} \alpha_{u,v} \quad wf_{u,v}^{i,j} \quad sec^S(u,v) \\
& + \lambda_1 \sum_{u,v\epsilon N^S} rl_{u,v} \quad sec^S(u,v) \\
& + \lambda_2 \sum_{i\epsilon N^V} \sum_{v\epsilon N^S} cpu^V(v) \quad wn_{i,v} \quad sec^S(v) \quad cloud^S(v) \\
& + \lambda_2 \sum_{v\epsilon N^S} rn_v \quad sec^S(v) \quad cloud^S(v) \\
& + \lambda_3 \sum_{(i,j)\epsilon E^V} \sum_{u,v\epsilon N^S} wl_{u,v}^{i,j} \\
& + \lambda_3 \sum_{(i,j)\epsilon E^V} \sum_{u,v\epsilon N^S} bl_{u,v}^{i,j}
\end{aligned} \tag{3.1}
$$

Intuitively, with our formulation when a VNR arrives to our system the embedder will try to match the resources to requests in such a way that it saves the more "powerful" resources (e.g., those with higher security levels) to the virtual resources that require them explicitely. For instance, virtual nodes with $sec^V = 1$ will be mapped onto substrate nodes with $sec^S = 2$ if and only if there are no substrate nodes with $sec^S = 1$ available.

3.1.4.3 Typical Constraints

In this section we define the constraints typically defined in most VNE MILP formulations.
Domain Constraints. The following constraints define the values space for each variable defined in our MILP formulation.

$$wf_{u,v}^{i,j} \geq 0, \forall u,v\epsilon N^S, (i,j)\epsilon E^V \tag{3.2}$$
$$bf_{u,v}^{i,j} \geq 0, \forall u,v\epsilon N^S, (i,j)\epsilon E^V \tag{3.3}$$

$$wl_{u,v}^{i,j}\epsilon \{0,1\}, \forall u,v\epsilon N^S, (i,j)\epsilon E^V \tag{3.4}$$
$$bl_{u,v}^{i,j}\epsilon \{0,1\}, \forall u,v\epsilon N^S, (i,j)\epsilon E^V \tag{3.5}$$

$$wn_{i,v}\epsilon \{0,1\}, \forall i\epsilon N^V, v\epsilon N^S \tag{3.6}$$
$$bn_{i,v}\epsilon \{0,1\}, \forall i\epsilon N^V, v\epsilon N^S \tag{3.7}$$

$$rl_{u,v} \geq 0, \forall u,v\epsilon N^S \tag{3.8}$$
$$rn_v \geq 0, \forall v\epsilon N^S \tag{3.9}$$

$$wc_{i,c}\epsilon \{0,1\}, \forall i\epsilon N^V, c\epsilon C \tag{3.10}$$
$$bc_{i,c}\epsilon \{0,1\}, \forall i\epsilon N^V, c\epsilon C \tag{3.11}$$

Eq. 3.2 and 3.3 ensure that the bandwidth allocated for a virtual link (i,j) in a substrate link (u,v), either for the working or backup parts, never takes negative values.

Eq. 3.4, 3.5, 3.6 and 3.7 ensure, respectively, that the variables $wl_{u,v}^{i,j}$, $bl_{u,v}^{i,j}$, $wn_{u,v}$ and $bn_{u,v}$ take either the value 0 or 1.
Eq. 3.8 and 3.9 ensure that the bandwidth reserved for backup on substrate links and the capacity (e.g., CPU) reserved on substrate nodes, also for backup, never take negative values.
Eq. 3.10 and 3.11 ensure variables $wc_{i,c}$ and $bc_{i,c}$ only take the value 0 or 1.
Note: The set C present in these restrictions (and later) is the set of existent clouds.
Link Mapping for Working Traffic
Constraints 3.12, 3.13 and 3.14 refer to the working flow conservation conditions, which denote that the network flow to a node is zero, except for the source node and the sink node, respectively (i.e., no flow appears or disappears in any node, unless it is a source or a sink node).

$$\sum_{u \epsilon N^S \cup N^V} wf_{u,v}^{i,j} - \sum_{u \epsilon N^S \cup N^V} wf_{v,u}^{i,j} = 0, \forall (i,j) \epsilon E^V, v \epsilon N^S \tag{3.12}$$

$$\sum_{v \epsilon N^S} wf_{i,v}^{i,j} - \sum_{v \epsilon N^S} wf_{v,i}^{i,j} = bw^V(i,j), \forall (i,j) \epsilon E^V \tag{3.13}$$

$$\sum_{v \epsilon N^S} wf_{j,v}^{i,j} - \sum_{v \epsilon N^S} wf_{v,j}^{i,j} = -bw^V(i,j), \forall (i,j) \epsilon E^V \tag{3.14}$$

Eq. 3.15 and 3.16 guarantee that the working flow of a virtual link (i,j) always departs from the correspondent working node of i and arrives to the correspondent working node of j.

$$wn_{i,v} \quad bw^V(i,j) = wf_{i,v}^{i,j}, \forall v \epsilon N^S, (i,j) \epsilon E^V \tag{3.15}$$

$$wn_{j,v} \quad bw^V(i,j) = wf_{v,j}^{i,j}, \forall v \epsilon N^S, (i,j) \epsilon E^V \tag{3.16}$$

Figure 3.5 shows an example of how these constraints contribute to our formulation. In short, they define the values of variables wf. Constraints 3.12, 3.13 and 3.14 are responsible for $wf_{D,A}^{1,2} = 4$, i.e., they are responsible for the working flows in the substrate network. Eq. 3.15 and 3.16 are responsible for the working flows on meta-links. Supposing that virtual nodes 1 and 2 of a VNR are mapped onto substrate nodes A and D, respectively, we have $wn_{1,A} = 1$ and $wn_{2,D} = 1$. Therefore, $wf_{1,A}^{1,2} = 4$ and $wf_{2,D}^{1,2} = 4$. All other wf and wn variables, such as $wf_{1,B}^{1,2}$ or $wn_{2,C}$ are equal to 0.

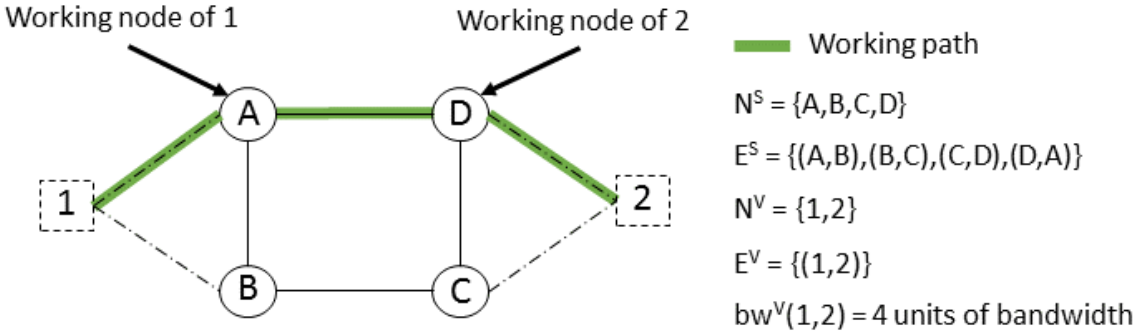


Figure 3.5: Example of how variables wf contribute to our formulation.

Node Capacity Constraints. Substrate nodes can map nodes from different VNRs simultaneously. They can be the correspondent working node of a virtual node i from a VNR x and simultaneously be the correspondent backup node of a virtual node j from a VNR y. Considering this, for a substrate node, the total allocated capacity depends on the total capacity that is allocated for working nodes, plus the total capacity that is allocated for backup nodes, which should be less than the current capacity of the substrate node. This is represented by Eq. 3.17 and 3.18.

$$\sum_{u \epsilon N^V} bn_{u,v} \quad cpu^V(u) \leq rn_v, \forall v \epsilon N^S \tag{3.17}$$

$$\sum_{u \epsilon N^V} wn_{u,v} \quad cpu^V(u) + rn_v \leq R_N(v), \forall v \epsilon N^S \tag{3.18}$$

Link Capacity Constraints. Like substrate nodes, substrate links also can map virtual links from different VNRs simultaneously. Eq. 3.19 and 3.20 define the allocated link capacity of a substrate link as the sum of the capacity allocated for the active flows and the reserved resources for backup. The allocated capacity of a substrate link should be less than the residual capacity of that physical link.

$$\sum_{(i,j) \epsilon E^V} (bf^{i,j}_{u,v} + bf^{i,j}_{v,u}) \leq rl_{u,v}, \forall u, v \epsilon N^S \tag{3.19}$$

$$\sum_{(i,j) \epsilon E^V} (wf^{i,j}_{u,v} + wf^{i,j}_{v,u}) + rl_{u,v} \leq R_E(u,v), \forall u, v \epsilon N^S \tag{3.20}$$

Figure 3.6 shows an example of how this set of restrictions works. In Figures 3.6(a) and 3.6(b) we show two separate VNRs with different parameters. Taking into account only CPU and bandwidth resources as attributes, we can observe in 3.6(c) that nodes A, B, C and D, and links (A,D) and (B,C), are sharing their capacity to two different VNs at the same time. Node A is the working node of 100 and the backup node of 1, node B is the working node of 1 and the backup of 100, node C is the working node of 2 and the backup of 99, and node D is the working node of 99 and the backup of 2. Link (A,D) carries the data that goes from 99 to 100 and is the backup link for (1,2), i.e., if a failure occurs in B, C or (B,C) the data that goes from 1 to 2 will pass through (A,D). Link (B,C) is the working link for the virtual link (1,2) and the backup for the virtual link (99,100). This is possible because the substrate resources have sufficient capacity to map multiple virtual resources from different VNRs.

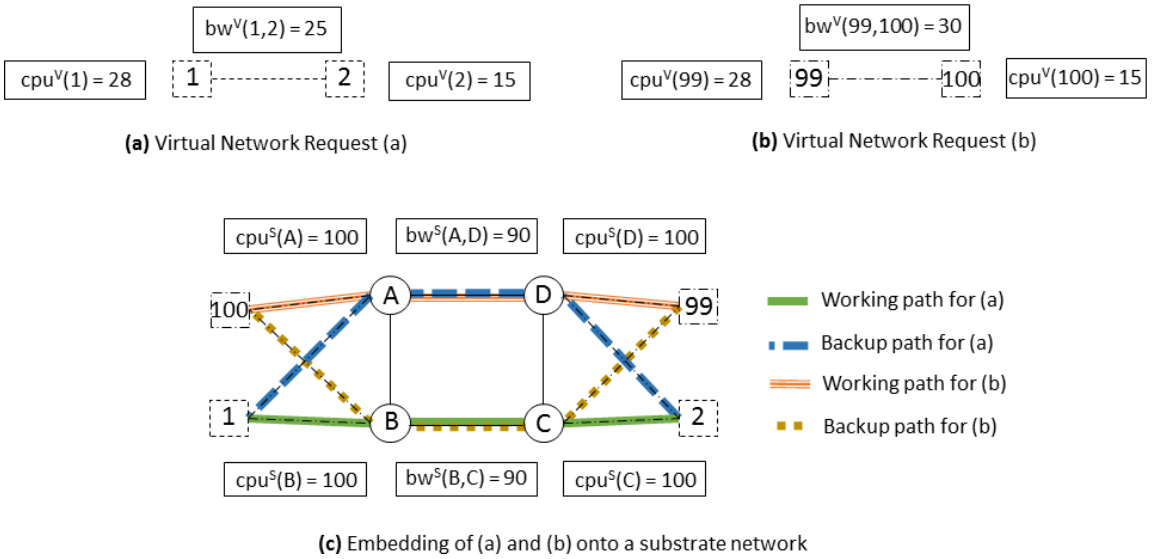


Figure 3.6: Allocation of substrate resources from a SN in (c) to multiple VNRs ((a) and (b)).

3.1.4.4 Security Constraints

Our VNE formulation includes security constraints, for both the nodes, links, and clouds.
Node Security Constraints. The security restrictions for nodes are defined as:

$$wn_{u,v} \quad sec^V(u) \leq sec^S(v), \forall u \epsilon N^V, v \epsilon N^S \tag{3.21}$$
$$bn_{u,v} \quad sec^V(u) \leq sec^S(v), \forall u \epsilon N^V, v \epsilon N^S \tag{3.22}$$

Eq. 3.21 guarantees that a virtual node u is only mapped to a physical node that has an equal or higher security level than u's security demand.
Eq. 3.22 ensures the same as the previous one, but for backup nodes.
This ensures, returning to our initial example, that a secure VM from the physical infrastructure can map virtual nodes request for normal containers, VMs, or secure VMs, whereas a physical container can only map virtual nodes that are looking for normal containers.
Link Security Constraints. The following equations are related with the working and the backup link security constraints:

$$wl_{u,v}^{i,j} \quad sec^V(i,j) \leq sec^S(u,v), \forall (i,j) \epsilon E^V, u, v \epsilon N^S \tag{3.23}$$
$$bl_{u,v}^{i,j} \quad sec^V(i,j) \leq sec^S(u,v), \forall (i,j) \epsilon E^V, u, v \epsilon N^S \tag{3.24}$$

Here, it is necessary to ensure that each virtual link is mapped to one or more physical links that provide a security level equal or higher than the security demand of the virtual link. Similarly to the previous case, a physical link that provides default security can only map virtual links that demand for that low level of security, while a physical link that provides authenticity, integrity and confidentiality guarantees can map virtual links with any security demand.
Cloud Security Constraints. Eq. 3.25 ensures that a virtual node u is mapped to a certain physical node v only if the cloud where v is located is of a type of equal or higher security than the type of cloud demanded by node u. For instance, a virtual node that requires to be mapped on a public cloud may be mapped to either a public, a trusted public or a private cloud, considering again our example. On the other side, a virtual node that requires the highest level of security can only be mapped to nodes located in a private cloud. Eq. 3.26 guarantees the same restriction for the backup nodes.

$$wn_{u,v} \quad cloud^V(u) \leq cloud^S(v), \forall u \epsilon N^V, v \epsilon N^S \tag{3.25}$$
$$bn_{u,v} \quad cloud^V(u) \leq cloud^S(v), \forall u \epsilon N^V, v \epsilon N^S \tag{3.26}$$

3.1.4.5 Dependability Constraints

Finally, we define the constraints related to fault-tolerance and dependability.
Link Mapping for Backup Traffic. For the backup traffic, it is necessary to define the same set of flow constraints defined for working traffic, but using the variables $bf_{u,v}^{i,j}$ and $bn_{u,v}$. Constraints 3.27, 3.28 and 3.29 refer to the backup flow conservation conditions, which denote that the network flow to a node is zero, except for the source node and the sink node, respectively. *wantBackup* is a parameter defined by the tenant and it assumes the value 1 if backups are needed or the value 0 otherwise.

$$\sum_{u \epsilon N^S \cup N^V} bf_{u,v}^{i,j} - \sum_{u \epsilon N^S \cup N^V} bf_{v,u}^{i,j} = 0, \forall (i,j) \epsilon E^V, v \epsilon N^S \tag{3.27}$$
$$\sum_{v \epsilon N^S} bf_{i,v}^{i,j} - \sum_{v \epsilon N^S} bf_{v,i}^{i,j} = bw^V(i,j) * wantBackup, \forall (i,j) \epsilon E^V \tag{3.28}$$
$$\sum_{v \epsilon N^S} bf_{j,v}^{i,j} - \sum_{v \epsilon N^S} bf_{v,j}^{i,j} = -bw^V(i,j) * wantBackup, \forall (i,j) \epsilon E^V \tag{3.29}$$

Eq. 3.30 and 3.31 guarantee that the backup flow of a virtual link (i,j) always departs from the correspondent backup node of i and arrives to the correspondent backup node of j. Normally, the

backup path only carries information of a virtual link if a failure in the working substrate path as occurred.

$$bn_{i,v} \quad bw^V(i,j) = bf_{i,v}^{i,j} * wantBackup, \forall v \epsilon N^S, (i,j) \epsilon E^V \tag{3.30}$$

$$bn_{j,v} \quad bw^V(i,j) = bf_{v,j}^{i,j} * wantBackup, \forall v \epsilon N^S, (i,j) \epsilon E^V \tag{3.31}$$

Figure 3.7 shows an example of how these constraints contribute to our formulation. In short, they define the values of variables bf. Constraints 3.27, 3.28 and 3.29 are responsible for $bf_{B,C}^{1,2} = 4$, i.e. for the backup flows in the substrate network. Eq. 3.30 and 3.31 are responsible for the backup flows on meta-links. If we assume that virtual nodes 1 and 2 of a VNR are mapped onto substrate nodes B and C, respectively, we have $bn_{1,B} = 1$ and $bn_{2,C} = 1$. Therefore, $bf_{1,B}^{1,2} = 4$ and $bf_{2,C}^{1,2} = 4$. All other bf and bn variables, such as $bf_{1,A}^{1,2}$ or $bn_{2,D}$ are equal to 0.

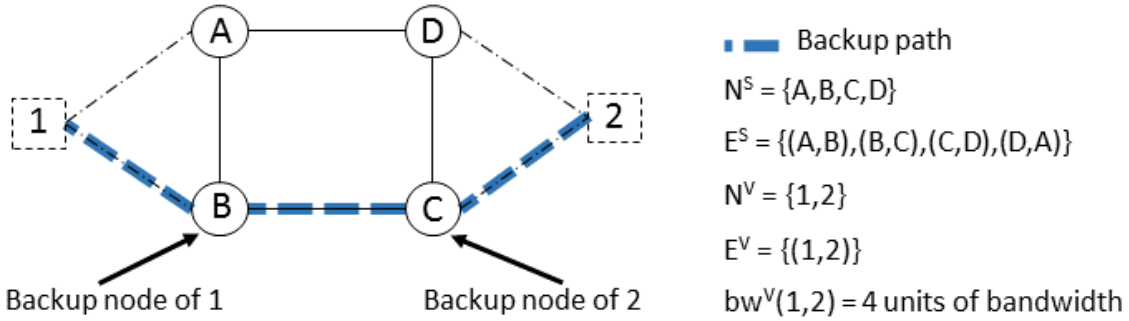


Figure 3.7: Example of how variables bf contribute to our formulation.

The equation below guarantees that the meta-links only carry working or backup traffic to their correspondent virtual nodes. This means that, if a virtual node 1 needs to send information to virtual node 2, the data does not need to pass through the meta-links of a virtual node 3.

$$\sum_{j,k!=i \;\; j,k \epsilon N^V} wf_{i,v}^{j,k} + wf_{v,i}^{j,k} + bf_{i,v}^{j,k} + bf_{v,i}^{j,k} = 0, \forall v \epsilon N^S, i \epsilon N^V \tag{3.32}$$

Virtual Node Mapping. Eq. 3.33 and 3.34 state that each virtual node has to be mapped to exactly one working node and $wantBackup$ backup nodes in the substrate node, i.e., if $wantBackup = 0$, virtual nodes will not have backup, if $wantBackup = 1$, virtual nodes will have backup. For the same VN, Eq. 3.35 guarantees that *(i)* two different virtual working nodes are not mapped to the same substrate node; and *(ii)* a substrate node that is the backup for a virtual node does not have any virtual working nodes on it. Eq. 3.36 guarantees that a substrate node can be the backup of a single virtual node, for the same VN.

$$\sum_{v \epsilon N^S} wn_{u,v} = 1, \forall u \epsilon N^V \tag{3.33}$$

$$\sum_{v \epsilon N^S} bn_{u,v} = wantBackup, \forall u \epsilon N^V \tag{3.34}$$

$$\sum_{u \epsilon N^V} wn_{u,v} + bn_{z,v} \leq 1, \forall v \epsilon N^S, z \epsilon N^V \tag{3.35}$$

$$\sum_{u \epsilon N^V \setminus \{z\}} bn_{u,v} + bn_{z,v} \leq 1, \forall v \epsilon N^S, z \epsilon N^V \tag{3.36}$$

Note that we define Eq. 3.35 because we want to minimize the number of virtual resources of a VN affected if a failure occurs in a certain substrate node. Figures 3.8 and 3.9 try to clarify this idea. In Figure 3.8, if a failure occurs in the physical node A, and nodes 1 and 2 are mapped onto it, all the virtual links will be affected. Instead, if all the nodes of the VN are mapped onto different physical nodes and the failure occurs in node A, only the virtual link (1,3) will be affected.

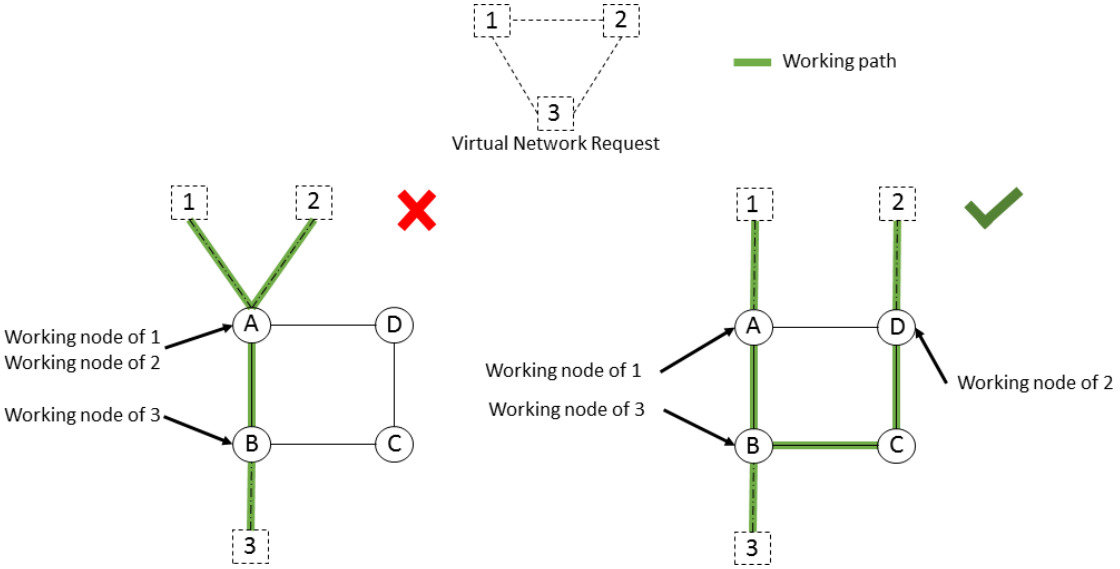


Figure 3.8: Example of an embedding that respects the first part of eq. 3.35.

In Figure 3.9, if a failure also occurs in the physical node A, and A is the working node for 1 and the backup node for 2, both working and backup paths will be affected. If physical nodes do not assume the working and backup function simultaneously for the same VN, this problem is solved, as we can observe in the right part of the figure. Here, if a failure occurs in node A, there will always exist a backup path through which nodes 1 and 2 can continue to communicate while A is down.

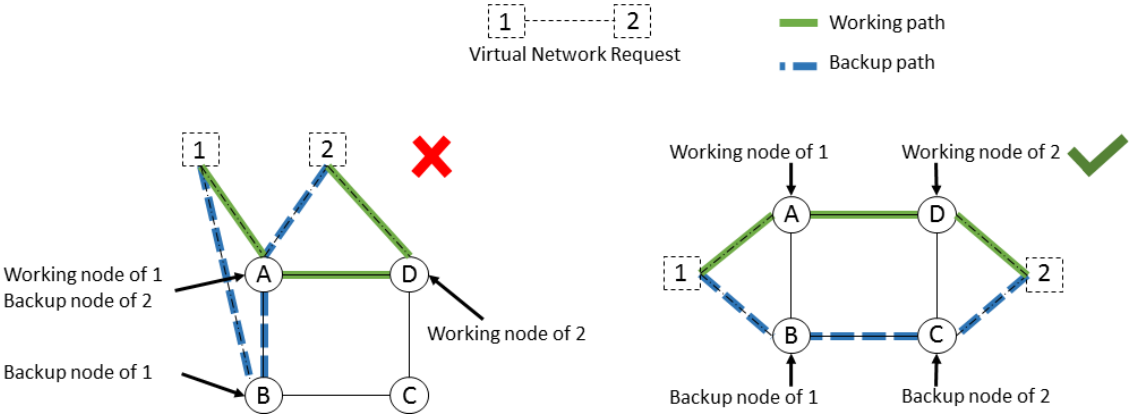


Figure 3.9: Example of an embedding that respects the second part of eq. 3.35.

Since in our work we allow the user to choose between having no replication, replication in the same cloud or in different clouds, it is necessary to specify these restrictions.

$$\sum_{c \epsilon C} wc_{u,c} = 1, \forall u \epsilon N^V \tag{3.37}$$

$$\sum_{c \epsilon C} bc_{u,c} = wantBackup, \forall u \epsilon N^V \tag{3.38}$$

$$wantBackup * wc_{u,c} + bc_{u,c} \leq dep^V(u), \forall u \epsilon N^V, c \epsilon C \tag{3.39}$$

$$wc_{u,c} \geq bc_{u,c} * dep^V(u) - 1, \forall u \epsilon N^V, c \epsilon C \tag{3.40}$$

Eq. 3.37 states that each virtual node is mapped on exactly one cloud. Eq. 3.38 ensures that, when a VN needs backup ($wantBackup = 1$), the backup of each virtual node is mapped to exactly one cloud. Eq. 3.39 and 3.40 are restrictions that address if a virtual node u and its correspondent backup will be on the same cloud or in different clouds, depending on the dependability level required by u ($dep^V(u)$).
Figure 3.10 represents the embedding of a VNR accordingly to the dep^V of its nodes. Node 1 requires a backup in the same cloud ($dep^V = 1$), while node 2 requires to have a backup in another cloud ($dep^V = 2$).

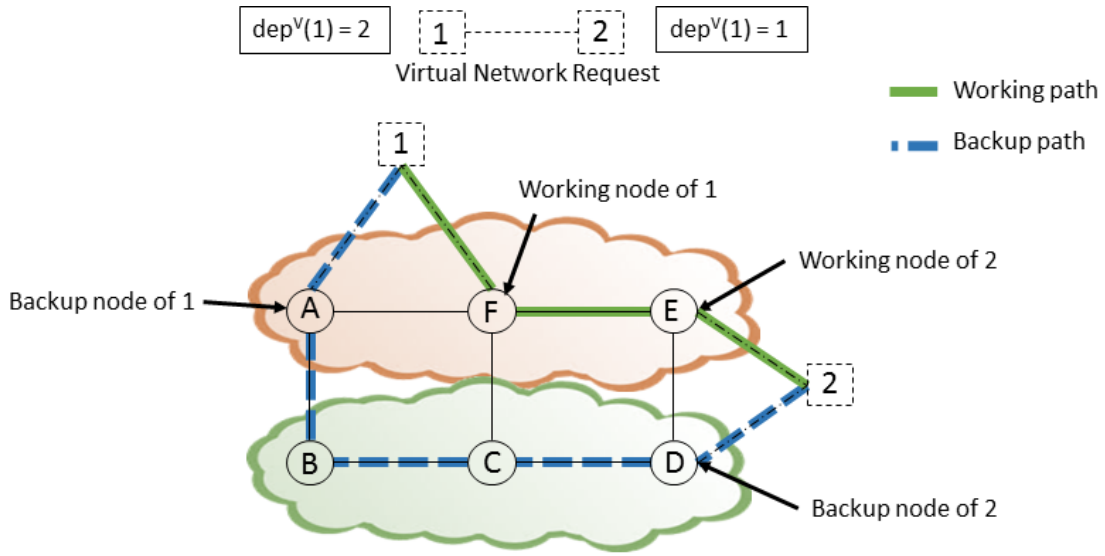


Figure 3.10: Example of an embedding respecting the dep^V required by the virtual nodes.

Eq. 3.41 and 3.42 establish a relation between the virtual and physical nodes and the clouds (the first one related to working nodes, and the second related with the backup nodes).

$$\sum_{v \epsilon N^S} (wn_{u,v} * doesItBelong_{c,v}) \geq wc_{u,c}, \forall u \epsilon N^V, c \epsilon C \tag{3.41}$$

$$\sum_{v \epsilon N_S} (bn_{u,v} * doesItBelong_{c,v}) \geq bc_{u,c}, \forall u \epsilon N_V, c \epsilon C \tag{3.42}$$

The aim of these equations is the following. If a virtual node u is mapped onto a physical node v and v belongs to cloud c ($doesItBelong_{c,v} = 1$ if substrate node v belongs to cloud c, 0 otherwise), then u is mapped on cloud c.
In a similar fashion, we also need restrictions to create relationships between the variables wf, wl and wn, and bf, bl and bn:

$$wn_{i,v} * bw^V(i,j) \geq wl_{i,v}^{i,j}, \forall (i,j) \epsilon E^V, v \epsilon N^S \tag{3.43}$$

$$wn_{j,v} * bw^V(i,j) \geq wl_{v,j}^{i,j}, \forall (i,j) \epsilon E^V, v \epsilon N^S \tag{3.44}$$

$$bw^V(i,j) * wl_{u,v}^{i,j} \geq wf_{u,v}^{i,j}, \forall (i,j) \epsilon E^V, u, v \epsilon N^S \cup N^V \tag{3.45}$$

Eq. 3.43 and 3.44 are constraints that ensure that, if a meta-link is established between a virtual node i and a physical node v, then this means that i is mapped onto v. For instance, if $wl_{i,v}^{i,j} = 1$, then $wn_{i,v} = 1$. Eq. 3.45 ensures that, if there is a flow between nodes u and v for a virtual link (i,j), then this means that (i,j) is mapped to a meta-link or a physical link whose end-points are u and v. For example, if $wf_{u,v}^{i,j} = 1$, then $wl_{u,v}^{i,j} = 1$.
Eq. 3.46, 3.47 and 3.48 achieve the same goals as before, but for the backup:

$$bn_{i,v} * bw^V(i,j) \geq bl_{i,v}^{i,j}, \forall (i,j) \epsilon E^V, v \epsilon N^S \tag{3.46}$$

$$bn_{j,v} * bw^V(i,j) \geq bl_{v,j}^{i,j}, \forall (i,j) \epsilon E^V, v \epsilon N^S \tag{3.47}$$

$$bw^V(i,j) * bl_{u,v}^{i,j} \geq bf_{u,v}^{i,j}, \forall (i,j) \epsilon E^V, u, v \epsilon N^S \cup N^V \tag{3.48}$$

Finally, we include two binary constraints to guarantee the symmetric property of the binary variables related with links.

$$wl_{u,v}^{i,j} = wl_{v,u}^{i,j}, \forall (i,j) \epsilon E^V, u, v \epsilon N^S \cup N^V \tag{3.49}$$

$$bl_{u,v}^{i,j} = bl_{v,u}^{i,j}, \forall (i,j) \epsilon E^V, u, v \epsilon N^S \cup N^V \tag{3.50}$$

Nodes and Links Disjointness. Since any substrate nodes and links of a working path can fail, we have to ensure that backup paths connecting the backups of the virtual nodes are disjoint from the substrate resources that are being used for the working part (otherwise a backup path can be compromised if a physical resource belonging to the working and backup part fails).

$$BigConstant * working_u >= \sum_{v \epsilon N^S} wl_{u,v}^{i,j}, \forall (i,j) \epsilon E^V, u \epsilon N^S \tag{3.51}$$

$$BigConstant * backup_u >= \sum_{v \epsilon N^S} bl_{u,v}^{i,j}, \forall (i,j) \epsilon E^V, u \epsilon N^S \tag{3.52}$$

$$backup_u = 1 - working_u, \forall u \epsilon N^S \tag{3.53}$$

Eq. 3.51 - 3.53 together with Eq. 3.35 ensure path disjointness between the working and the backup parts. As we already observed, Eq. 3.35 ensures that a substrate node mapping the working virtual node can not be a backup of any other node, and vice-versa. Relatively to Eq. 3.51 - 3.53 we ensure that if a substrate node u is an end point of a certain link that is being used as a working resource, u can not be an end point of a link that is being used as a backup resource, and vice-versa. Variables *working* and *backup* are auxiliary variables that define if a certain physical node u belongs to the working or backup part. *BigConstant* is a constant big enough to ensure that the restriction is valid when its rightmost part is greater than 0.
To close the explanation of our algorithm, in Figure 3.11 we present a full embedding of a VN onto a SN. An embedding is only successful if all of the previous restrictions explained are fulfilled. In this figure, in addition to the CPU and bandwidth capacities, the VN requests for security for its nodes and links, and also for backups in another cloud. All the requirements of the VN and the characteristics of the substrate resources are presented in the figure, according with the attributes defined in Sections 3.1.3.1 and 3.1.3.2.

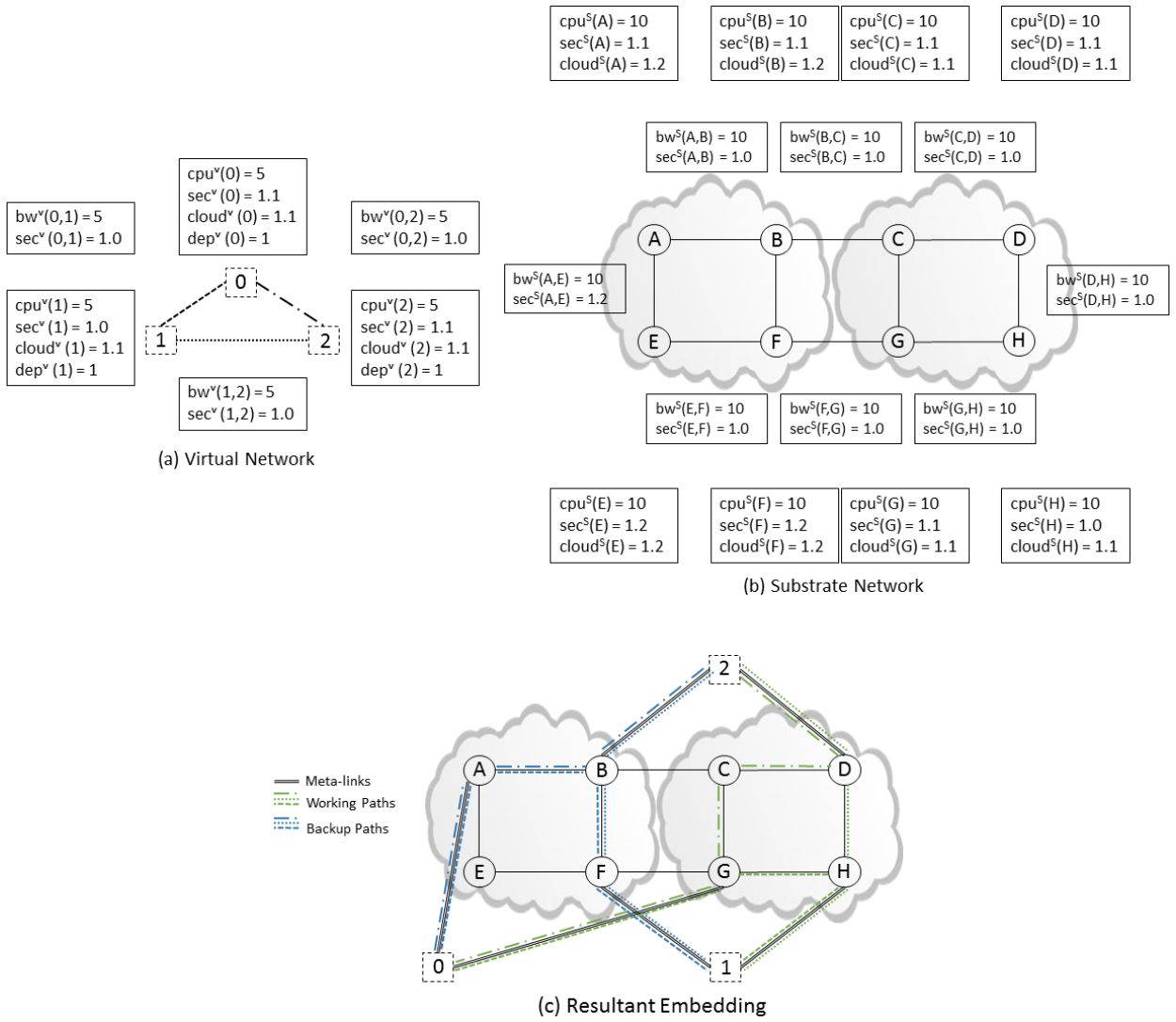


Figure 3.11: Example of a full embedding of a VN onto a SN with our solution.

3.1.5 Evaluation

In this section we present some performance results of our solution. We have implemented a simulator to reproduce an environment where VNRs with different requirements arrive over time. Section 3.1.5.1 presents the simulation setup. In Section 3.1.5.2 we present the algorithms with which we compared our own. Finally, in Section 3.1.5.3 we show the results of our evaluation, followed by a discussion.

3.1.5.1 Simulation Setup

We have implemented an event simulator to evaluate the performance of our algorithm against the performance of D-ViNE [15, 14]. We have chosen D-ViNE due to its availability as open-source software and the fact that it has been considered as the baseline for most VNE work. This simulator was based on the simulator presented in [1] and, in short, it simulates the dynamical arrival of VNRs to the system. For our evaluation, the SN topology was randomly generated with 25 nodes using the GT-ITM tool [59] in (10x10) grids. Each pair of substrate nodes was randomly connected with probability between 0.1 and 0.3. The CPU and bandwidth resources of the substrate nodes and links were real numbers uniformly distributed between 50 and 100. We assumed that VNRs arrivals ($Time^V$) are modelled as a Poisson process with an average rate of 4 VNRs per 100 time units, each one having an exponentially distributed lifetime (Dur^V) with an average of $\mu = 1000$ time units. In each VNR, the number of virtual nodes was randomly determined by a uniform distribution between 2 and 4. Each

pair of virtual nodes was randomly connected with probability between 0.1 and 0.3. The capacity requirements of the virtual nodes and links were real numbers uniformly distributed between 10 and 20.
We chose to only address a small scale environment (25 nodes to the SN and 2-4 nodes to the VNs) because optimal solutions, such as the MILP we used in SecDepVNE, do not scale for large networks and, therefore, it would not be possible to have results in a reasonable time. Future work includes looking for efficient heuristics for this problem.
For SecDepVNE we also need to include security and dependability attributes. The probability of substrate nodes and links having $sec^S = 0$ was 0.05, for $sec^S = 1$ was 0.4, and for $sec^S = 2$ was 0.55. The substrate nodes belonged to three clouds, each one with a different security level (public, trusted public and private). The weight (α) of all substrate links was 1. The sec^V of the virtual nodes and links was distributed uniformly by the three different security types. The requests always asked for backups to their virtual nodes, and $cloud^V$ and dep^V for the virtual nodes of a request were also generated randomly. In this evaluation, we considered that $\lambda_1 = 1$, $\lambda_2 = 1$ and $\lambda_3 = 1$.
To solve the MILP solutions we used the open source MILP library GLPK [21]. The simulation ran for 50000 time units, and during this period, the MILPs tried to embed 1000 VNRs. The order of arrival of VNRs and the capacity requirements of each VNR were the same for both algorithms, in order to ensure that both dealt with similar problem instances.

3.1.5.2 Comparison Method

In our evaluation, we compared two algorithms that solve the VNE problem with different requirements, D-ViNE [15] and our SecDep. D-ViNE requirements include only CPU and bandwidth capacities, while our algorithm adds to these requirements also security demands, cloud preferences and dependability requirements. This means the algorithms are not in an equal footing (our has more requirements and consequently restrictions), we chose D-ViNE to comparison because it has been considered in the VNE literature as the baseline. Furthermore, the comparison of SecDep against D-ViNE is interesting in order to understand what are the implications when we introduce security and dependability aspects to a typical embedding problem. The objective function of D-ViNE is presented in 3.54. Besides the minimization of CPU and bandwidth resources allocated to a VN (i.e., minimization of costs) the objective function of this algorithm also tries to balance the load introducing weights. In our evaluation, we considered those weights as 1, as its authors [15, 14] (we also follow their notation).

$$min \sum_{uv \epsilon E^S} \frac{\alpha_{uv}}{R_E(u,v) + \delta} \sum_i f_{uv}^i + \sum_{w \epsilon N^S} \frac{\beta_w}{R_N(w) + \delta} \sum_{m \epsilon N^{S'} \setminus N^S} x_{mw} c(m) \tag{3.54}$$

To evaluate our solution, we executed six different types of experiments:

1. Solve the embedding problem where there are no security and dependability requirements for VNs and the SN;

2. Solve the embedding problem where the SN and all VNs have random security requirements for their resources, and only 10% of them requests dependability;

3. Solve the embedding problem where the SN and all VNs have random security requirements for their resources, and 30% of them requests dependability;

4. Solve the embedding problem where the SN and all VNs have random security requirements for their resources, and 50% of them requests dependability;

5. Solve the embedding problem where the SN and all VNs have random security requirements for their resources, and 100% of them requests dependability. This case is considered the worst-case in our evaluation, since all the VNRs seek for security and dependability;

6. Solve the embedding problem where there are four different types of requests: 25% of the VNRs do request neither security nor dependability, 25% of the VNRs request only dependability, 25% of the VNRs request only security, and the last 25% of VNRs have both security and dependability requirements.

The notations that we used to refer the different experiments are enumerated in Table 3.2.

Notation	Algorithm Description
D-ViNE	VNE MILP model presented in [15, 14]
SecDep0	Secure and Dependable VNE MILP model with the environment explained in Point 1).
SecDep10	Secure and Dependable VNE MILP model with the environment explained in Point 2).
SecDep30	Secure and Dependable VNE MILP model with the environment explained in Point 3).
SecDep50	Secure and Dependable VNE MILP model with the environment explained in Point 4).
SecDep100	Secure and Dependable VNE MILP model with the environment explained in Point 5).
MixedSecDep	Secure and Dependable VNE MILP model with the environment explained in Point 6).

Table 3.2: Compared algorithms

3.1.5.3 Evaluation Results

We used several performance metrics for evaluation in our experiments. We have considered the same metrics as in [15]:

- VNR acceptance ratio: the percentage of requests accepted over time;
- Average revenue: the revenue that the SUPERCLOUD provider gets over time;
- Average cost of accepting a VNR: the cost the SUPERCLOUD provider will incur by embedding a request;
- Average node utilization: the load of the SN nodes over time;
- Average link utilization: the load of the SN links over time.

To calculate the revenue $\mathbb{R}$ and costs $\mathbb{C}$ we used the equations presented in Section 3.1.3.4. We now present all results from the simulations and discuss each outcome.

1) **The embedding performance of SecDep without considering security and dependability requirements is similar to the embedding performance of D-ViNE.** When security and dependability are not taken into account, our algorithm performs similar to D-ViNE, as can be seen in all figures we present in this section. This is important as it shows our baseline to be identical to the most commonly used VNE algorithm.

2) **A richer set of features (namely, security and dependability) decreases the acceptance ratio.** Figure 3.12 shows that D-ViNE leads to a higher acceptance ratio over time when compared with our SecDep VNE. This was expected, as SecDep VNE is richer in terms of the features provided (security and dependability, in addition to CPU and bandwidth). Consequently, in SecDep VNE to accept a VNR the number of constraints is higher and more condition need

to be satisfied. For instance, a VNR with some virtual nodes demanding the maximum security level may be rejected by SecDep if the SN does not have enough substrate nodes available to cover that demand at that moment. Another important factor that contributes to a smaller acceptance ratio is the higher use of substrate resources due to VNRs that require dependability features, as more resources need to be allocated to these VNRs. When compared to D-ViNE, the worst-case of SecDep VNE (SecDep100) has an acceptance ratio around 35% smaller. The scenarios SecDep50 and MixedSecDep have similar acceptance ratios, both accepting around 20% less requests when compared to D-ViNE. In general, and as expected, a secure and dependable VNE algorithm will present a lower acceptance ratio as providing security and dependability is more demanding.

3) **A richer set of features (namely, security and dependability) increases the revenue until the point when the acceptance ratio becomes too low.** Figure 3.13 illustrates the time average of generated revenue. When our algorithm is set without security and dependability requirements (SecDep0), it generates the same revenue as D-ViNE. In this figure it is also possible to observe that SecDep10 and SedDep30 generate more revenue than D-ViNE. This is due to security having a weight that is considered in the revenue function (as we expect richer resources to be more more expensive). Since the acceptance ratio in these is not far from D-ViNE, the revenue will increase. All the other cases (SecDep100, SecDep50, and MixedSecDep) generate less revenue than D-ViNE as a consequence of the smaller acceptance ratio they present.

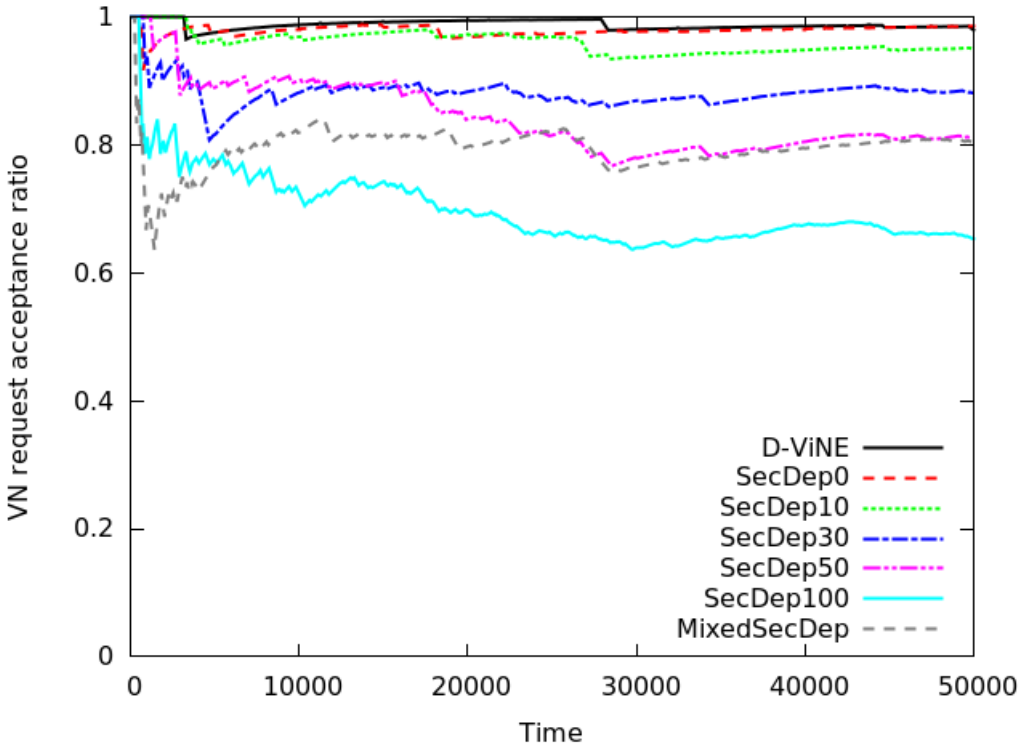


Figure 3.12: VNR acceptance ratio over time.

4) **A relatively small increase in the price of security resources leads to revenues that are inline with traditional VNE algorithms.** As per the previous point, SecDep0, SecDep10, and SecDep30 generate similar or slightly better revenues than D-ViNE, so we address this point only to the other case.

As stated earlier, a way to increase the revenue is to increase the price of each security resource. To have a better notion of the scale of the necessary increase, in Table 3.3 we show the target

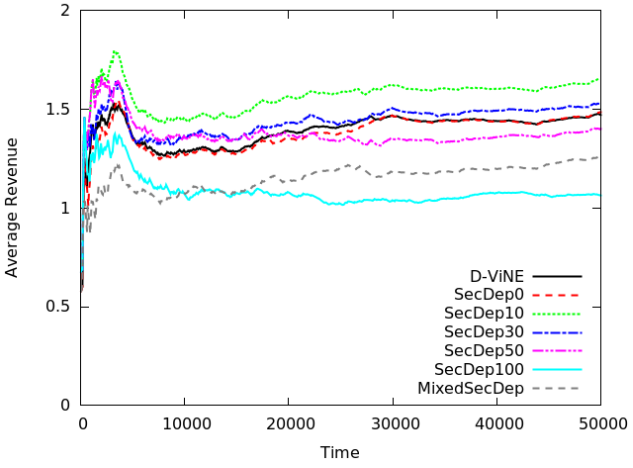


Figure 3.13: Time average of generated revenue.

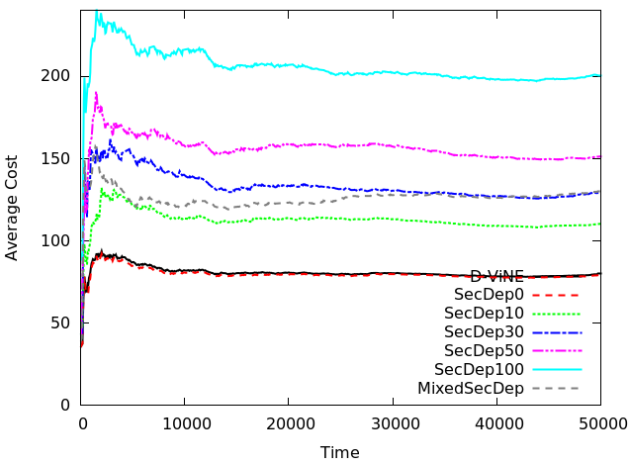


Figure 3.14: Average cost of accepting VNRs over time.

price security resources should have in cases SecDep50, SecDep100 and MixedSecDep, in order to achieve a total revenue similar to D-ViNE. For this analysis we change the prices of node, link and cloud securities proportionally. The column of security prices shows the price changes we made. The first line of each cell in this column refers to the price increase of the intermediary security levels, while the second one refers to the price increase of the highest security levels. The important take away from this table is the fact that it is possible to achieve a revenue inline with D-ViNE with a very small increase in the price of each security resource.

5) **Dependability and security requirements increase the costs of embedding a VNR.** Figure 3.14 shows that the costs of embedding a VNR with the SecDep algorithm are higher when compared to D-ViNE, as expected. The cost is higher for one main reason. When backups are required by a VNR, the CPU resources demanded by a virtual node are always allocated twice, one allocation for the working node and another to the backup node. There will also be additional allocations of bandwidth resources for the virtual links, since it is necessary to have substrate paths connecting the working nodes (working paths composed by at least one working link) and substrate paths connecting the backup nodes (backup paths composed by at least one backup link).

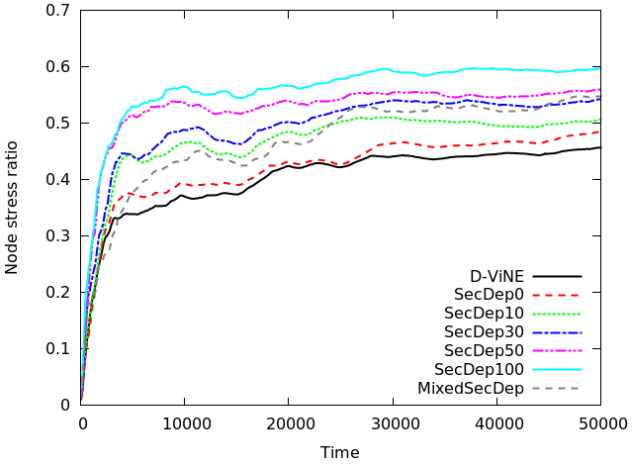


Figure 3.15: Average node utilization.

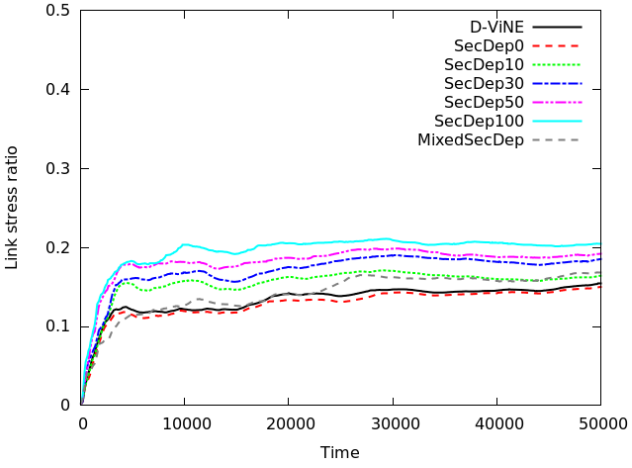


Figure 3.16: Average link utilization.

Algorithm	Total $\mathbb{R}$	Security Prices (Original price → New price)	Average % Price Increasing
D-ViNE	≈ 76796		
SecDep50	≈ 76773	Medium sec. level: 1.1 → 1.1 Highest sec. level: 1.2 →1.3	4.165
SecDep50	≈ 80320	1.1 → 1.2 1.2 →1.3	8.71
SecDep50	≈ 80372	1.1 → 1.1 1.2 →1.4	8.335
SecDep100	≈ 77800	1.1 → 1.3 1.2 →1.8	34.09
SecDep100	≈ 77931	1.1 → 1.4 1.2 →1.7	34.47
SecDep100	≈ 78061	1.1 → 1.5 1.2 →1.6	34.845
MixedSecDep	≈ 75473	1.1 → 1.3 1.2 →1.5	20.19
MixedSecDep	≈ 77522	1.1 → 1.4 1.2 →1.5	26.135
MixedSecDep	≈ 77544	1.1 → 1.3 1.2 →1.6	24.355
MixedSecDep	≈ 79635	1.1 → 1.4 1.2 →1.6	30.3
MixedSecDep	≈ 81764	1.1 → 1.5 1.2 →1.6	34.845

Table 3.3: Prices increasing to achieve total revenues near to D-ViNE.

6) **Dependability requirements increase substrate resources utilization.** Figure 3.15 and Figure 3.16 show the average substrate node and link utilization, respectively. In both figures we observe that there more resources are allocated in the SN with SecDep VNE than with D-ViNE. The reasons are the same as in Point **5**) above.

3.1.5.4 Discussion

Security and dependability have a cost that may affect the profit of the SUPERCLOUD provider. However, it is possible to reduce the costs by increasing the embedding efficiency. For instance, our solution adopts a backup scheme where each virtual node has a dedicated backup, and additional links are allocated to ensure that VNs survive in case of a failure. This approach is the most conservative, and consequently the most expensive.
Some well-known techniques can be used to reduce the costs of embedding. For example, the use of a backup pooling mechanism, as in [56], can could reduce the cost of embedding as only a set of substrate nodes (instead of all) would be allocated to be backup of any VNR when a failure occurs. Although this solution may reduce costs and resource utilization, it brings some disadvantages: the substrate nodes pooled may not be sufficient to all VNRs if a cloud outage occurs. In addition, when a failure occurs making a VNR operational again takes more time, since it is necessary to search which pooled nodes can allocate resources for the failed nodes.
In a similar way, a backup bandwidth sharing technique, where bandwidth resources of backup links (or backup paths) can be allocated on-demand by any VNRm could also reduce the costs and resources utilization. The downside is that this technique does not ensure that there are sufficient backup bandwidth resources available when a failure occurs.

All these techniques target efficiency, aiming to reduce costs. Other possibilities include increasing revenue. For instance, the acceptance ratio of our algorithm could be improved by, instead of refusing a VNR when there are no resources available at the moment of its arrival, postponing the request to be embedded later, as proposed in [14].

3.1.6 Conclusion

A major challenge in network virtualization is how to make efficient use of the shared resources. Virtual Network Embedding (VNE) addresses this problem by finding an effective mapping of the virtual nodes & links onto the substrate network. For some scenarios, VNE has been studied in some detail in the network virtualization literature.
The VNE problem is traditionally formulated with the objective of maximizing network provider revenue by efficiently embedding incoming virtual network (VN) requests. This objective is subject to constraints, such as processing capacity on the nodes and bandwidth resource on the links. A mostly unexplored perspective on this problem is providing some security assurances, a gap increasingly more acute. With the advent of network virtualization platforms, cloud operators now have the ability to extend their cloud computing offerings with virtual networks. To shift their workloads to the cloud, tenants trust their cloud providers to guarantee that their workloads are secure and available. Unfortunately, there is an increasing number of evidence that problems do occur, of both the malicious kind (e.g., caused by a corrupt cloud insider) or benign (e.g., a cloud outage). Security and dependability is thus becoming a critical factor that should be considered by virtual network embedding algorithms.
In this section we proposed a VN embedding solution that considers security and dependability as first class citizens. For this purpose, we introduced specific security and dependability constraints into the formulation. The SUPERCLOUD allows us to extend the resiliency properties further, by assuming a multiple cloud provider model, considering the coexistence of multiple clouds: both private, belonging to the tenant, and public, belonging to cloud providers. By not relying on a single cloud provider we avoid internet-scale single points of failures, avoiding cloud outages by replicating workloads across clouds. In addition, we can enhance security by leaving sensitive workloads in the tenant's private clouds or in public trusted facilities.
The results from our experiments show that there is an (already expected) cost in providing security and availability that may reduce the SUPERCLOUD provider profit. However, a relatively small increase in the price of the richer features of our solution (security resources, for instance), coupled with efficient techniques to reduce the embedding cost (e.g., pooling of backup resources) enables the provider to offer secure and dependable network services at a profit.

3.2 Addressing & topology virtualization

The SUPERCLOUD network virtualization platform has to fulfil three requirements:

- The first requirement is to enable the specification of *arbitrary virtual network topologies*, which are independent of the existing physical infrastructure. This means, for instance, that a virtual link can correspond to multiple network paths connecting the two endpoints. Virtual switches may also hide large portions of the infrastructure, with several physical switches interconnected to provide the required abstraction.

- In addition, *address virtualization* is required, as users should have complete autonomy to manage the address space of their virtual networks.

- Lastly, as resources may be shared to enable an efficient use of the substrate resources, *isolation* between virtual networks should also be enforced.

Our network hypervisor runs on top of the SDN controller, mapping the physical to virtual events by intercepting the flow of messages between the physical network and the users' applications. This logical centralization of control is the main enabler for addressing and topology virtualization, and isolation.

3.2.1 Topology Virtualization

Our network hypervisor allows users to specify their own arbitrary topologies. These topologies do not have to correspond to the actual physical network – they can exactly match what the user desires. As explained in the previous section, virtual network requests are delivered to a network embedder, which generates a virtual-to-physical mapping. This mapping is then instantiated on the physical topology. The mapping itself does not specify how the virtualization is actually implemented. This is achieved by the hypervisor pushing the necessary flow rules to the switches. For instance, if a virtual link connecting two virtual hosts is composed of multiple physical links, then the hypervisor will install the necessary flows in every switch on the path to enable connectivity between the two end-points.

3.2.2 Address Virtualization

The SUPERCLOUD network hypervisor provides address virtualization. Users are able to use the full header space, including both Layer 3 (IP) and Layer 2 (Ethernet MAC) addresses. By offering the full header space users are free to use *any* MAC and IP address. As such, the virtual addresses of different users can overlap. This is fundamentally different from techniques such as FlowVisor [53], which slice the address space (hence users are restricted to use an address sub-space). Address virtualization is an important facilitator for network migration. The virtual addresses of a user do not need to be reconfigured when the substrate resources are migrated.
Multi-tenant virtualization platforms such as NVP [29] allow address virtualization by implementing logical datapaths in the software virtual switches that run on each host, leveraging a set of tunnels between every pair of host-hypervisors. This option has the advantage of leaving the network infrastructure untouched (the network only sees IP packets), but the use of tunnels implies a non-negligible overhead.
As we have SDN control over the network elements, we have opted for a less expensive translation approach. In our solution the edge switches (those that connect to user's VMs) re-write the virtually assigned MAC addresses into disjoint addresses to be used within the physical network. This allows different users to use potentially overlapping IP and MAC address blocks. To differentiate between virtual networks, the hypervisor generates a globally unique label: the tenant ID. The first 16 bits of the virtual MAC address are replaced by the tenant ID, forming the physical address that is sent to the network. By such, the user can use any MAC address, but is limited to 2^{32} hosts (the same limitation imposed by the use of IPv4 addresses). Collisions of addresses are avoided by installing flow rules to rewrite addresses at the edge switches of the network, from the user-assigned (virtual) addresses to the physical MAC address with tenant ID encoded at the ingress edge, and vice-versa at the egress edge. This rewriting at the edge imposes a (negligible) overhead on the dataplane. Encapsulation (tunneling) techniques (e.g., GRE) are then only used to allow the virtual network to span across clouds.
The current version of the hypervisor is proactive. This means that when a virtual network is instantiated the required translation rules are set up at the edge switches and the necessary flow rules are installed in the substrate switches.

3.2.3 Isolation

The SUPERCLOUD network virtualization platform has to guarantee isolation between users. Several dimensions have to be considered: isolation of traffic, bandwidth, device CPU, and forwarding tables [53].

Two techniques are required to achieve isolation of traffic. First, as explained in Section 3.2.2, flow rule redefinition at the edge of the network. The traffic is tagged with the tenant IDs and the flows installed guarantee connectivity between hosts from the same tenant only. Second, we avoid ARP flooding in the network. When ARP traffic is detected at the edge, the hypervisor inserts the necessary flow rules for the traffic to be confined to hosts of the same tenant. This technique guarantees isolation and does not require modification in the hosts (they process the ARP protocol as usual).
For bandwidth isolation it is necessary each virtual link to have its own fraction of bandwidth on each physical link. Failure to isolate bandwidth would allow one user to affect other user's throughput. To achieve this goal our platform uses the bandwidth slicing capabilities of OpenFlow, in the form of per-port queues. The hypervisor creates a per-user queue on each port on the switch, configured for a fraction of link bandwidth. To enforce bandwidth isolation each user traffic is forwarded via its specific queue (and not directly to the output port). As such, all traffic is mapped a specific class, and a minimum bandwidth queuing discipline is used (although any other, such as Weighted Fair Queueing, can be used).
The CPU of each device (hosts and switches) also needs to be limited. Switches, in particular, use low-power embedded processors that typically have very limited computational resources and so need proper CPU slicing. Rate-limiters are used to enforce CPU isolation.
Finally, as forwarding tables have limited capacity (in terms of TCAM entries) each user has a finite quota of forwarding rules in each switch. This is also important because the failure to isolate forwarding entries between users might allow one user to prevent another from forwarding packets. Our hypervisor maintains a counter of the number of flow entries used per user per switch, and ensures that it does not exceed a preset limit.

3.2.4 Status of implementation

The current prototype of the SUPERCLOUD network hypervisor provides virtualization of L2 and L3 addresses. Users are thus able to define the IP and Ethernet MAC addresses of their VMs. In addition, users can specify arbitrary virtual topologies. At this stage isolation between virtual networks is not yet guaranteed.
The next version of the prototype will include the isolation mechanisms described in Section 3.2.3 to enforce isolation between tenant's networks. Future work also includes a thorough evaluation of the functional and non-functional requirements of the platform.

Chapter 4 Resilient control plane

The SUPERCLOUD network hypervisor has resilience built-in by design into the control plane (the SDN controller), ensuring that correct operation can be maintained under relevant failure scenarios. Scalable and fault-tolerant SDN controllers usually give up strong consistency for the network state, adopting instead the more efficient eventually consistent storage model. This decision is mostly due to the performance overhead of the strongly consistent replication protocols (e.g., Paxos, RAFT), which limits the responsiveness and scalability of network applications. Unfortunately, this lack of consistency leads to a complex programming model for network applications and, more importantly in the context of SUPERCLOUD, can lead to network anomalies that may result in security breaches. In this chapter we show how the lack of control plane consistency can lead to network problems and propose fault-tolerant and distributed SDN control plane architectures to address this issue. In Section 4.1 we present the design and implementation of the SUPERCLOUD fault-tolerant SDN controller. Our solution guarantees consistent processing of control plane events and commands even in the presence of faults, without requiring changes to SDN protocols or switches. To address scalability, in Section 4.2 we detail the architecture of the SUPERCLOUD distributed SDN controller. The solution we propose is supported by a fault-tolerant data store that provides the strong consistency properties necessary for transparent distribution of the control plane. In order to deal with the fundamental concern of such design, we apply a number of techniques tailored to SDN for optimizing the data store performance.

4.1 Fault-tolerant SDN controller

The SDN controller is the crucial enabler of the Software-Defined Networking paradigm. Its fundamental function is to maintain the logically centralized network state, which is acted upon by network applications to define the network policy. The controller then instantiates the policy dictated by applications by installing flow rules in the data plane switches, which set how traffic should be handled. Figure 4.1 illustrates the normal execution in an SDN environment. A switch receives a packet that it does not know how to forward. Upon receiving this packet, the controller installs flow rules on the switch that allows it to forward subsequent packets directly without contacting the controller. As such, typically only the first packet of a specific flow needs controller intervention.

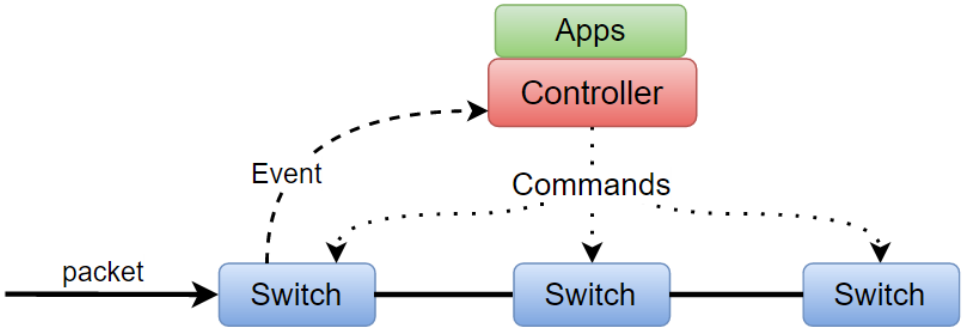


Figure 4.1: SDN flow execution. Switches send events to the controller as needed and the controller replies with one or more commands that modify the switch's table.

The first SDN controllers (including NOX [22], the original) were centralized: a single controller instance would control all network switches. This design has inherent scalability and dependability problems. As the size of the network increases, the controller becomes a bottleneck. In addition, such a centralized design makes the controller a single point of failure.
To increase the dependability of the system, the typical solution is to replicate the controller. With $f + 1$ controller replicas an SDN can tolerate up to f controller crash faults. In case of contoller failure, however, it is important that the network view remains consistent. Otherwise, applications will operate in a stale state, which can lead to degraded performance [34] and other network anomalies [27], including loops and security breaches.
To build a consistent global network view across replicated, fault-tolerant controllers, events (packets) need to be processed in a consistent way that guarantees three properties: (i) events are processed in the same (total) order in all controllers, (ii) no events are lost (processed at least once) and (iii) no events are processed repeatedly (at most once). These properties ensure that all controllers will reach the same internal state and thus build a consistent network view.
However, building a consistent network view in the controllers is not enough to logical centralization. In SDN, it is necessary to include switch state into the system model and handle it consistently [27]. Since switches are programmed by controllers (and controllers can fail), there must be mechanisms to ensure that controllers actually send commands to the switches (at least once), but never send repeated commands (at most once).
In a fault-tolerant scenario, if a controller fails while it is processing an event, it may have sent, or not, some commands. A naive approach to avoid no commands being missed would be for the new controller to simply repeat all commands for that event. However, this would lead the switch to receive duplicate commands which could result in its state becoming inconsistent, since some commands are not idempotent.
Summing up, to achieve a consistent, fault-tolerant SDN environment, one needs to ensure that the following properties are met:

- **Total Event Ordering:** Controller replicas should process events in the same order and subsequently all controller application instances should reach the same internal state.
- **Exactly-Once Event Processing:** All the events are processed, and are neither lost nor processed repeatedly.
- **Exactly-Once Execution of Commands:** Any given series of commands are executed once and only once on the switches.

Recently, Kata *et al.* [27] have proposed a fault-tolerant controller, Ravana, that achieves this level of consistency for SDN control. To achieve these properties, however, Ravana requires modifications to the OpenFlow protocol and to existing switches. Specifically, switches need to maintain two buffers, one for events and one for commands, and four new messages need to be added to the OpenFlow protocol. These modifications preclude the adoption of Ravana on existing systems and hinder the possibility of it being used in (at least) the near future. Indeed, there are no guarantees the necessary messages will be included in the OpenFlow specification anytime soon or that switches will include the necessary mechanisms. Motivated by this fact, in this section we propose a fault-tolerant controller that offers the same consistency guarantees as Ravana *without* requiring changes to OpenFlow or modifications to switches.

4.1.1 Background & Related Work

We start by introducing some background on SDN fault-tolerance. Kim *et al.* [28] where among the first to identify the three fault domains in SDN: (i) the switches and links between them (*data plane*), (ii) inter-controller links and switch-controller links (*control plane*) and (iii) the controllers' machines.

As a general approach to handle faults in fault domain (i), most OpenFlow controllers take a reactive approach: port-down events sent by switches are fed to applications that program switches accordingly (i.e., reroute traffic around failures). Nonetheless, this approach incurs high restoration time and additional load on the controllers. A possible solution is to have pre-installed backup paths in the switches to avoid the overhead of communicating with the control plane. An example is CORONET [28], which provides fast recovery and is able to recover from multiple link failures. It incurs into minimal control traffic, by changing VLAN configuration after detecting faults in switches or links, using the OpenFlow API. Alternatively, FatTire [49] proposes a new programming language that eases the development of fault tolerant applications by compiling regular expressions into OpenFlow rules. This provides an easy way for developers to specify the set of paths that packets can take in the network and the degree of fault-tolerance required.
Covering fault domain (ii), Ros et al. [51] address the problem of dynamically adapting the number and locations of controllers, as introduced by Bari *et al.* [5], but for fault-tolerance. The goal is to determine the optimal number, placement and switch-connections of controllers in order to achieve at least five nines of reliability in the southbound interface (connectivity between controllers and switches).
In previous work [10] we have addressed controller fault-tolerance while achieving strong consistency in the state shared between replicas. Our proposal, SMaRtLight, is a fault-tolerant controller architecture that uses an hybrid replication approach: passive replication in the controllers (one primary and multiple backups) and active replication in a distributed data store to achieve durability and strong consistency. The controllers are coordinated through the data store and achieve acceptable performance by using caching mechanisms. However, SMaRtLight requires that applications are modified to use the data store directly and, more importantly, do not achieve the level of consistency required in an SDN. Specifically, it does not include switch state into its system model to handle it consistently.
In this respect, the closest work to ours is Ravana [27]. This system provides a transparent fault-free control platform for applications in the face of both controller and switch crashes. To achieve this, Ravana processes control messages transactionally and exactly once (at both the controllers and the switches) using a replicated state machine approach, but without involving the switches in a complex consensus protocol. The protocol proposed by Ravana is shown in Figure 4.2.
In Ravana switches buffer events (in case they need to re-transmitted; e1 and e2 in the figure) and send them to the master controller that will replicate them in a shared log with the slaves (e1r is a replicated event). The master then replies back to the switch acknowledging reception of the events. Then, events are delivered to applications that will generate and send one or more commands to the switches. Switches reply back to acknowledge the reception of these commands and buffer them (c1 in the figure) to filter possible duplicates.
While Ravana allows unmodified applications to run in a fault-tolerant environment, it requires modifications to the OpenFlow protocol and switches. Namely, Ravana leverages on buffers implemented on switches to retransmit events and filter possible repeated commands received from the controllers. In addition, explicit acknowledge messages must be added to the OpenFlow protocol so that the switch and the controller are able to acknowledge received messages.

4.1.2 Design

The goal of this work is to build a strongly consistent and fault-tolerant control plane for SDN, without requiring modifications to switches or to the OpenFlow protocol. Our system is driven by the following requirements:

- Reliability: the system should maintain a correct and consistent state even in the presence of failures (in both the controllers and switches).
- Transparency: the consistency and fault-tolerance properties should be completely transparent to applications and switches.

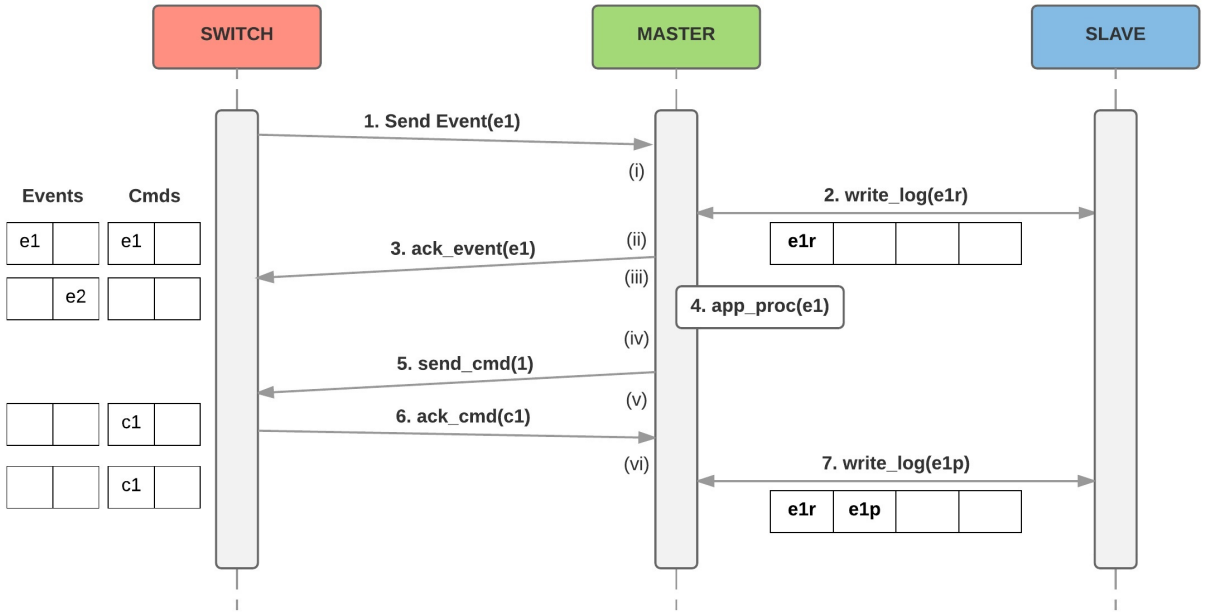


Figure 4.2: Ravana protocol. In Ravana switches maintain two buffers (displayed on the left) to re-transmit events and filter repeated commands in case of master failure. New acknowledge messages (`ack_event` and `ack_cmd`) are exchanged between the switch and the master to guarantee the consistency requirements.

- Performance: the performance of the system should not degrade as the number of network elements (events and switches) grows.

The proposed architecture for our system, Rama[1] is depicted in Figure 4.3. Its main components are: (i) OpenFlow enabled switches (switches that are implemented according the OpenFlow specification), (ii) controllers that manage the switches and (iii) a coordination service that abstracts controllers from complex primitives like fault detection and total order. In our model, we consider only one network domain with one primary controller and f backup controllers, where f is the number of faults to tolerate. Each switch connects to one primary controller and f backup controllers. This primary/backup model is supported by OpenFlow in the form of master/slave and allows the system to tolerate controller faults. When the master controller fails, the remaining controllers will elect a new leader to act as the new master for the switches managed by the crashed master. This election is supported by the coordination service.
The coordination service offers strong consistency and abstracts the controllers from coordination and synchronization primitives, making them simpler and more robust. The strong consistency model assures that updates to the coordination service made by the master will only return when they are persistently stored. This means that slaves will always have the most recent modifications available when the master receives confirmation of the update. This results in a consistent network view among all controllers even in the presence of faults. In addition to the controllers' state, the switches also maintain state that must be handled consistently in the presence of faults. This will be further discussed in section 4.1.4.

4.1.3 Why Consistency Matters

In this section we summarize the importance of maintaining a consistent state in both controllers and switches in a fault-tolerant SDN setting. The first step to have fault-tolerance in the control plane is to have more than one controller available to manage the network switches, which can lead to problems.

[1] In the Hindu epic Ramayana, Rama kills the evil demon Ravana, who abducted his wife Sita.

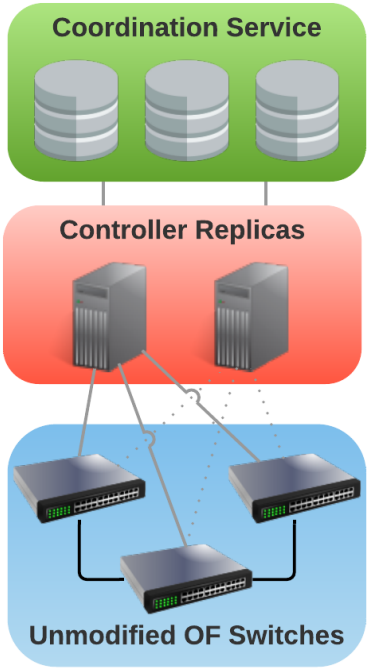


Figure 4.3: High level architecture of the system. One controller manages the OpenFlow (OF) switches (master) and one or more backup controllers are ready in case of failure (slaves, connected in dashed line). All controllers keep an updated and consistent network view using the coordination service. Network applications run inside controller replicas.

4.1.3.1 Inconsistent event ordering

In OpenFlow, each switch maintains a TCP connection with each controller it knows and sends them messages using these channels. If we configure switches to send all their events to all known controller replicas, and let replicas process events as they are received, each one will end up building a different internal state. This is because, although each TCP channel orders events sent by each switch, there is no ordering guarantee between the events sent to controllers by all switches.
Consider a simple scenario with two controllers (c1 and c2) and two switches (s1 and s2) that send two events, respectively: (e1, e2) and (e3, e4). One possible outcome where both controllers receive events in different order (while respecting the TCP FIFO property) is c1 receiving events in the order e1, e3, e2, e4 and c2 receiving in the order e3, e4, e1, e2.
As a result of this consistency problem we derive our first design goal for a fault tolerant and consistent control plane:

> **Total event ordering:** all controllers should process the same (total) order of events and consequently reach the same internal state.

4.1.3.2 Unreliable event delivery

In order to achieve a total ordering of events between controller replicas two approaches can be used:

1. The master (primary) replica can store controller state (including state from network applications) in an external consistent data-store (as in Onix [30], ONOS [6] and SMaRtLight [10]);

2. The controller replicas can maintain consistent state using replicated state machine protocols (as in Ravana [27]).

Although both approaches ensure a consistent ordering of events between controller replicas, they are not fault-tolerant in a standard case where only the master controller receives all events. If we consider

— for the first approach – that the master replica can fail between receiving an event and finishing persisting the controller state in the external data-store (which happens after processing the event through controller applications), that event will be lost and the new master (i.e., one of the other controller replicas) will never receive it. The same can happen in the second approach: the master replica can fail right after receiving the event and before replicating it in the shared log (which in this case happens before processing the event through the controller applications). In these cases, since only the crashed master received the event, the other controller replicas will not have an updated view of the network and, depending on the type of the lost event or on the type of applications running on the controller, this can cause serious performance or security problems.
Additionally to not losing events, it is also important not to process them repeatedly in each controller replica, since that could also lead to controllers having an inconsistent view of the network. From both these problems comes our second design goal:

> **Exactly-once event processing:** all events sent by switches are processed and are never lost nor processed repeatedly.

4.1.3.3 Repetition of commands

In SDN, mechanisms to ensure exactly-once event processing are not enough to achieve a consistent system in the presence of faults. This is due to the state maintained by switches taking an important role on how the whole system works.
Consider the case where the master replica is processing an event that generated three commands to be sent to one switch and the slave replica has knowledge of this event. If the master fails while sending these commands, the new elected master will process the event (to reach an updated state) and may send repeated commands. This happens because the old master failed before informing the slave replica of its progress (i.e., which commands have been sent for that specific event) and therefore it cannot decide which commands to send and which commands to filter.
Additionally, to make things worse, one of the differences between traditional client-server models and SDN is that controllers (servers) may reply (i.e., send commands) to multiple switches (clients) as a result of one event sent by a switch. Therefore it is essential to integrate switch state into a consistent and fault-tolerant protocol and handle it carefully, which leads to our third and final design goal:

> **Exactly-once command execution:** any set of commands for a given event sent by controllers is executed only once on the switches.

4.1.4 Consistent and fault-tolerant protocol

In an SDN setting, switches generate events (e.g., when they receive packets or when the status of a port changes) that are forwarded to controllers. The controllers run (potentially) multiple applications that process the received events and may send (potentially) multiple commands to one or more switches in reply to each event. This cycle repeats itself in multiple switches across the network as needed.
In order to maintain a correct system in the presence of faults, one must handle the state in the controllers (the received events) and the state in the switches (the received commands) consistently. In this work, to ensure this, the entire cycle (figure 4.4) is processed as a transaction and exactly once. This means that (i) the events are processed exactly once at the controllers and in a total order (i.e., all controllers process events in the same order to reach the same state) and (ii) the commands are processed exactly once in the switches. Because the standard operation in OpenFlow switches is to simply process commands as they are received, the controllers must coordinate to send a command exactly once (since we do not want to modify the protocol or the switches). Ravana [27] does not need this coordination because the (modified) switches can simply buffer received commands and discard repeated commands (with the same identifier) sent by the new controller.
By default, in OpenFlow a master controller receives all asynchronous messages (e.g., `OFPT_PACKET_IN`) and the slaves controllers only receive a subset (e.g., port modifications). This means that only the

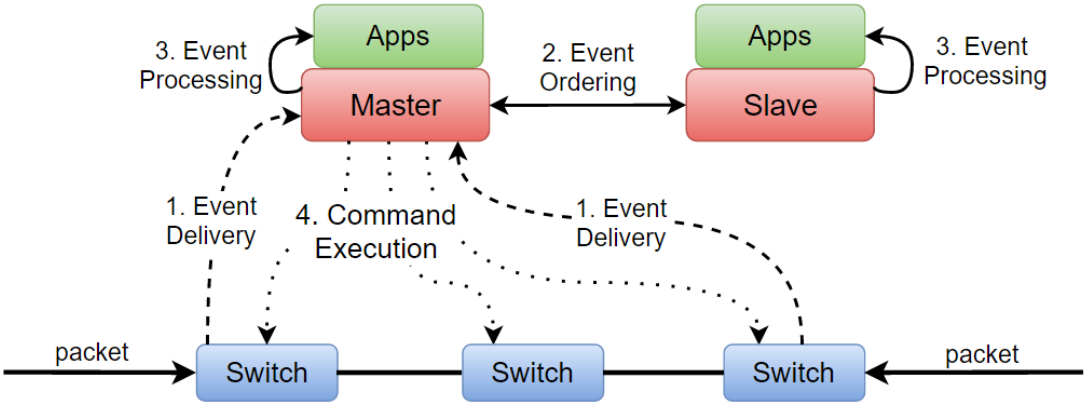


Figure 4.4: Control loop of (1) event delivery, (2) event ordering, (3) event processing, and (4) command execution. Events are delivered to the master controller, which decides a total order on the received events. The events are processed by applications in the same order in all controllers. Applications issue commands to be executed in the switches.

master controller receives all generated events from the switches. To change this behavior, it is possible the slaves to send an `OFPT_SET_ASYNC` message to each switch that modifies the asynchronous configuration in a way that switches will send events also to the slaves. Alternatively, controllers can set their role to `EQUAL`, and the coordination (to decide which one processes and sends which commands) is done amongst controllers. In our case, and conforming to the OpenFlow protocol specification, switches send all events to every controller by having the role `EQUAL`.
The fault-free execution of the protocol we propose is represented in figure 4.5. For simplicity, in this case a switch is connected with one master controller and one slave controller. The main idea is that switches must send messages to *all controllers*, so that they can coordinate themselves even if some fail at any given point. In Ravana, since switches simply buffer events (so that they can be retransmitted to a new master if needed), switches can send events only to the current master, instead of to every controller as in our case.
The master controller replicates the event in a shared log with the other controllers that imposes a total order over the received events. Replicating events in the log can be seen as the master sending an update to the coordination service and the slaves receiving it (to simplify, the coordination service is omitted from the figures).
When the event is replicated across controllers, it is processed by the master controller applications, which will generate zero or more commands. The commands are sent to the switches in bundles (a feature introduced in OpenFlow 1.4, see figure 4.6). A controller can open a bundle, add multiple commands to that bundle and then tell the switch to commit the commands present in the bundle in an atomic and ordered fashion. If for some reason an event does not generate any commands, the master controller will still add one single message as part of the protocol (see below). Note that Ravana does not rely on bundles since switches buffer all received commands so that they can discard possible duplicates from a new master.
When the event is processed by all modules (with each having sent their zero or more commands), the master controller sends a `OFPBCT_COMMIT_REQUEST` message to each switch affected by the event. The switch processes the request and tries to apply all the messages in the bundle by order. It then sends a reply message indicating if the Commit Request was successful or not. Again, we need to make sure that this reply message is sent to all controllers. This is a challenge, as it is not possible directly via the OpenFlow protocol. Bundle Replies are Controller-to-Switch messages and hence are only sent to the controller that made the request (in the same TCP connection). To overcome this challenge we inform other controllers if the bundle was committed or not (so that they can decide wether or not to resend commands), by including one `OFPT_PACKET_OUT` message in the end of every bundle

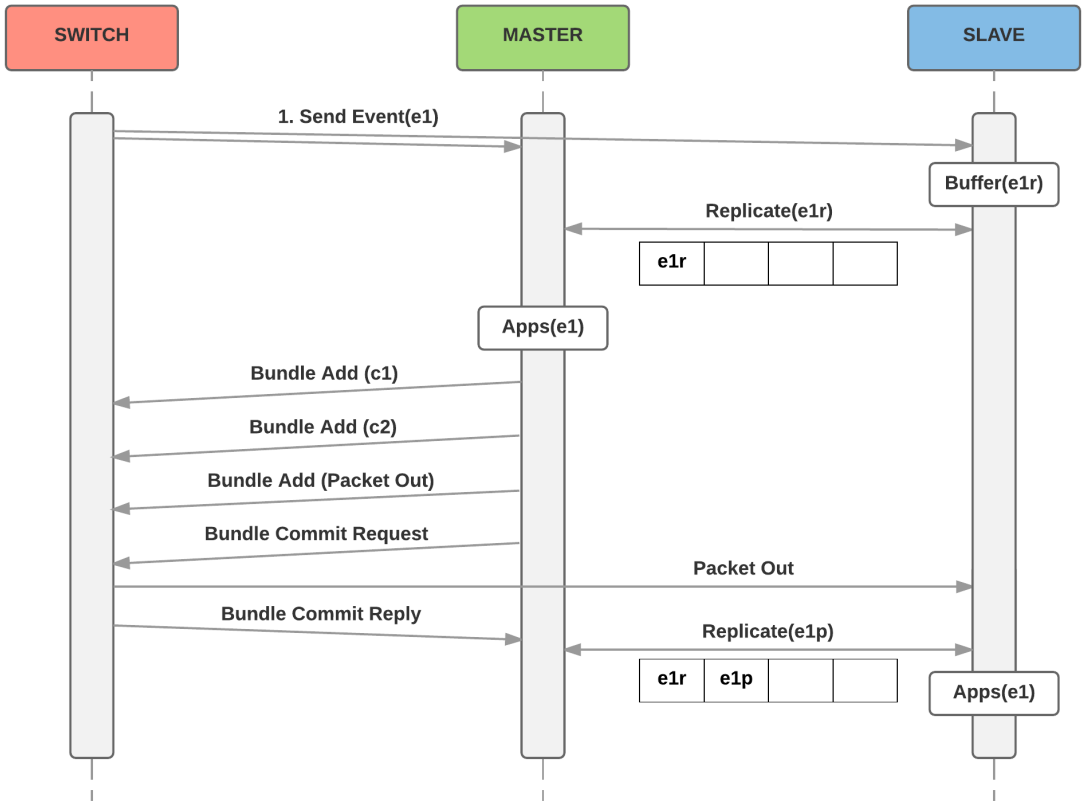


Figure 4.5: Fault-free case of the protocol. Switches send generated events to all controllers so that no event is lost. The master controller replicates the event in the shared log and then feeds its applications with the events in log order. Commands sent are buffered by the switch until the controller sends a Commit Request. The corresponding Commit Reply message is forwarded to all controllers to make sure that a new master never tries to commit repeated commands.

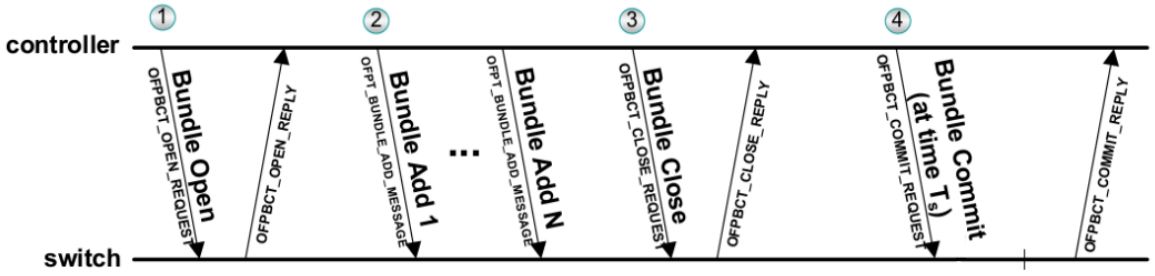


Figure 4.6: OpenFlow Bundles.

with action `output=controller`. The result is that the switch will send the data attached in the `OFPT_PACKET_OUT` message to all connected controllers in a `OFPT_PACKET_IN` message. This data is set by the master controller and assures that slave controllers know which events were fully processed by the switch, so that they do not send repeated commands (thus guaranteeing exactly-once semantics). The master finishes the transaction by replicating an `event processed message` in the log, which tells the slaves controllers that they can safely feed the corresponding event in the log to their applications. This is done to simply bring the slaves to an updated state equal to the master controller (the resulting commands sent by the applications are discarded).

4.1.4.1 Fault cases

When the master controller fails, the other controllers will detect the failure (i.e., by timeout) and run a leader election algorithm to elect a new master for the switches. Upon election, the new master must send a Role Request message to each switch, to register as the new master. There are three main cases where the master controller can fail:

1. Before replicating the received event in the distributed log (figure 4.7),
2. After replicating the event but before sending the Commit Request (figure 4.8)
3. After sending the Commit Request message.

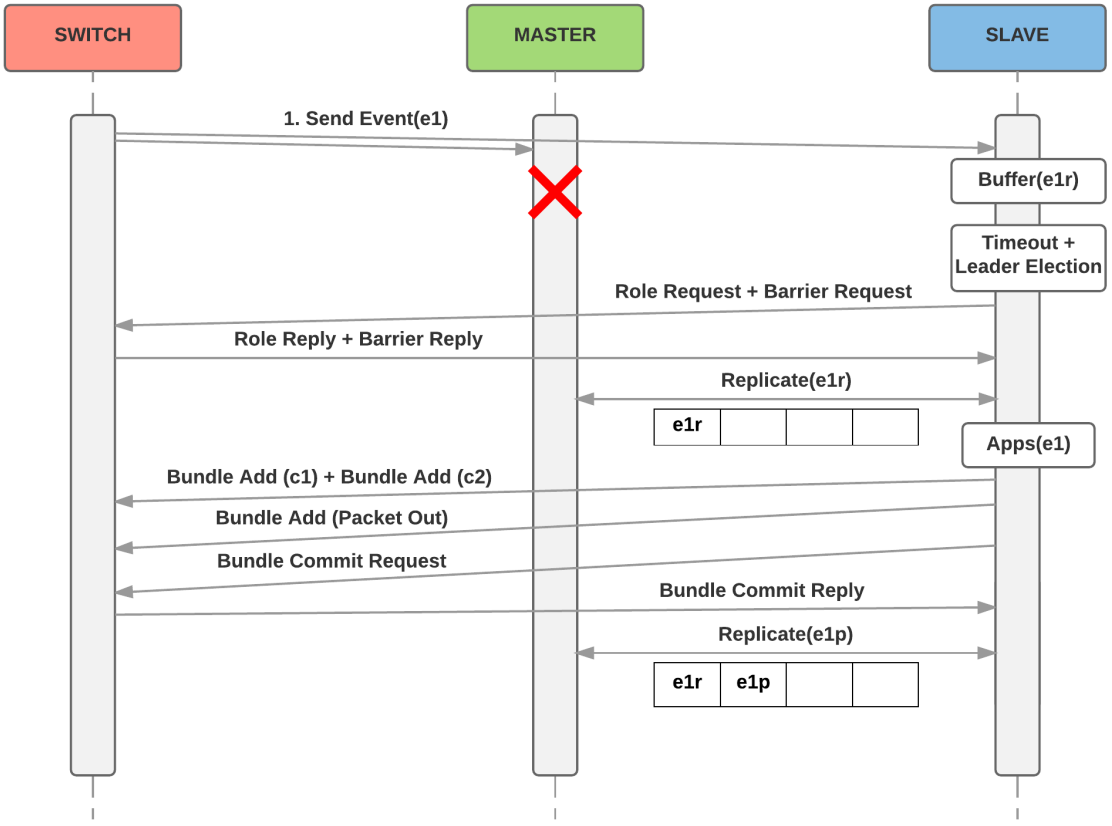


Figure 4.7: Case of the protocol where the master fails before replicating the received event. Because the slaves buffer all events, the event is not lost and the new master can resume the execution of the failed controller. The new master chooses the order of the events to replicate in the log and can only append new events to the log.

In the first case, the master failed to replicate the received events to the shared log but, since slave controllers receive and buffer all events, no events are lost. The new elected master appends the buffered events in order to the shared log and continues operation (feed the new events to applications and send commands). Note that before doing this, the new master must finish processing any events logged by the older master, having or not the corresponding `event processed message` in the log) – events marked as processed have their resulting commands filtered. This makes the new master reach the same internal state as the previous one before choosing the new order of events to append to the log (this is also valid to the other fault cases).
However, if the event was indeed replicated in the log (cases 2 and 3), the crashed master may have already issued the Commit Request message. Therefore, the new master must carefully verify if the

switch has processed everything it has received before re-sending the commands and the Commit Request message. To guarantee ordering, OpenFlow provides a Barrier message, to which a switch can only reply after processing everything received before (including generating and sending possible error messages). If a new master receives a Barrier Reply message without receiving a Commit Reply message (in form of `OFPT_PACKET_OUT`), it can safely assume that the switch did not receive nor execute a Commit Request for that event from the old master (case 2)[2]. Even if the old master sent all commands but did not send the Commit Request message, the bundle will never be committed and will eventually be discarded. Therefore, the new master can safely resend the commands. In case 3, since the old master sent the Commit Request before crashing, the new master will receive the confirmation that the switch processed respective commands for that event and will not resend them (otherwise one would break the exactly-once commands property).

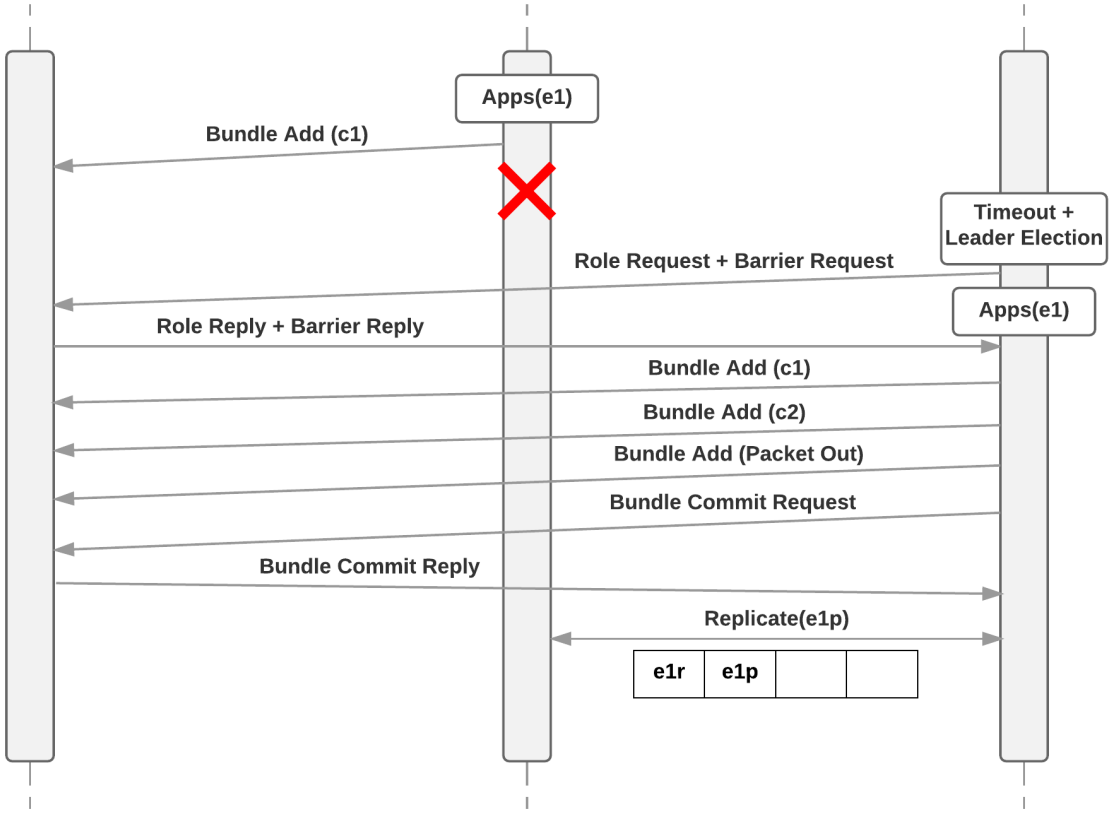


Figure 4.8: Case of the protocol where the master fails after replicating the event. The first part of the protocol is identical to the fault-free case and is omitted from the figure. In this case, the crashed master may have already sent some commands or even the Commit Request to the switch. If the new master does not receive a Commit Reply before the Barrier Reply, he can safely repeat the commands and order a commit. If the new master received the Commit Reply (the crashed master did send the Commit Request), the new master simply continues operation without sending the old commands.

Note that to ensure that the backups will know that the switch completely executed the received commands it is important that the switch sends the Commit Reply message to all controllers (in our case, via the `OFPT_PACKET_OUT` messages). Imagine the other scenario where the switch only sends the Commit Reply message to the master controller. If the master controller fails after issuing the Commit Request to the switch but before receiving or replicating the reply to the other controllers, the new elected master would never know that the switch had committed the commands. As such, the inclusion of the `OFPT_PACKET_OUT` message to the bundles is the key to guarantee the level of

[2]This relies on the FIFO properties of the controller-switch TCP connection.

consistency required.

4.1.4.2 Consistency Properties

The protocol described in the previous section was designed to achieve the same consistency properties as Ravana but without the need to modify the OpenFlow protocol or the switches. In this section we summarize the desired properties and the mechanisms used to achieve them. For a brief comparison, see table 4.1. Rama aims to achieve three main consistency properties:
Total event ordering: To guarantee that all controller replicas reach the same internal state, they must process a sequence of events in the same order. For this, both Rama and Ravana rely on a shared log across the controller replicas (implemented using the external coordination service) which allows the master to dictate the order of events to be followed by all replicas. Even if the master fails, the new elected master always preserves the order of events in the log and can only append new events to it.
Exactly once event processing: Events cannot be lost (processed *at least once*) due to controller faults nor can they be processed repeatedly (they must be processed *at most once*). Contrary to Ravana, Rama does not need switches to buffer events neither that controllers acknowledge each received event to achieve *at-least once event processing* semantics. Instead, Rama relies on switches sending the generated events to *all (f+1)* controllers so that at least one of them will known about the event (even if the other f fail – considering a control plane with $f+1$ replicas that tolerates f faults). Upon receiving these events, the master replicates them in the shared log while the slaves buffer them. If the master fails before replicating the events, the new elected master can append the buffered events to the log. If the master fails after replicating the events, the slaves will filter the buffered events so that they do not append the same events to the log. This ensures *at-most once event processing* since the new master only processes each event in the log one time. Together, sending events to all controllers and filtering buffered events ensures *exactly-once event processing*.
Exactly once command execution: For any given event received from a switch, the resulting series of commands sent by the controller are processed by the affected switches exactly *once*. Here, Ravana relies on switches acknowledging and buffering the received commands (to filter duplicates) from controllers. As this requires changes to the OpenFlow protocol and to switches, Rama relies on OpenFlow Bundles to guarantee transactional processing of commands. Additionally, the Commit Reply message that is triggered after the bundle finishes, is sent to *all* controllers and thus acts as an acknowledgment that is independent of controller faults. If the master fails, the new master needs to know if it should resend the commands for the logged events or not. A Packet Out message at the end of the bundle acts as a Commit Reply message to the slave controllers. This way, upon becoming the new master, the controller replica has the required information to know if the switch processed the commands inside the bundle or not, without relying on the crashed master. Furthermore, the new master sends a Barrier Request message to the switch that, upon receiving the corresponding reply, guarantees that all previous messages were processed. Therefore, the use of existing mechanisms in OpenFlow – Bundles with a Packet Out at the end, in addition to the use of a Barrier message – ensures that commands will be processed by the switches *exactly-once*.

It is important to note that we also consider the case where switches fail. However, this is not a special case of the protocol because it is already treated by the normal operation of the OpenFlow protocol. A switch failure will generate an event in the controller which will be delivered to the applications, for them to act accordingly (e.g., route traffic around the failed switch). As a special case, a switch may fail before sending the Commit Reply to the master and the slave controllers. However, this does not mean the transaction fails. This is a normal scenario in SDN, with controller replicas simply marking pending events for the failed switch as processed and moving on.
While we detail our reasoning as to why our protocol meets the described consistency properties, modeling the Rama protocol and giving a formal proof is left as future work and out of the scope of this deliverable.

Property	Ravana	Rama
At least once events	Buffering and retransmission of switch events	Switches send events to every controller with role EQUAL
At most once events	Event IDs and filtering in the log	
Total event order	Master appends events to a shared log	
At least once commands	RPC acknowledgments from switches	Bundle commit is known by every controller by piggybacking PacketOut in OpenFlow Bundle
At most once commands	Command IDs and filtering at switches	

Table 4.1: How Rama and Ravana achieve the same consistency properties using different mechanisms

4.1.5 Implementation

In this section we gives details on the implementation of Rama. Our proposal, Rama, could be implemented in any existing SDN controller that supports OpenFlow 1.4. We have built our prototype on top of Floodlight [46], an open source controller implemented in Java. As coordination service, we opted for ZooKeeper [25], for its reliability, simplicity and widespread use.

4.1.5.1 ZooKeeper

ZooKeeper [25] is a well-known coordination service that enables highly reliable distributed coordination. It exposes a set of primitives like naming, synchronization, and group services to be used by distributed applications. In the case of Rama, ZooKeeper abstracts controllers from fault detection, leader election and event transmission and storage (for controller recovery).
The most important design goals of Zookeeper are high performance, high availability, strictly ordered access and reliability. The reliability aspect means that ZooKeeper itself should be replicated across a set of hosts (called an ensemble). Each ZooKeeper server has an atomic broadcast component and maintains a replicated in-memory database that contains the entire data tree (see figure 4.9). The atomic broadcast is the core of ZooKeeper: it is an atomic messaging system that keeps all servers in sync. This agreement protocol ensures that every server processes messages in the same (total) order. To split workload across ZooKeeper servers, clients can connect to any server and send requests to it. Servers can process and reply to read requests locally (using the local database) but write requests (that change the state of the service), are processed by the agreement protocol. In this protocol, servers forward write requests from clients to a single server (the leader). These servers (followers) receive message proposals from the leader and agree on the order of messages to be delivered proposed by the master. In conclusion, the agreement protocol implemented by ZooKeeper ensures that local replicas of the database never diverge in each server.
The ZooKeeper data model resembles the traditional directory tree structure of file systems, with each node in the tree (that can be seen as a directory) having data and children nodes associated with it. For Rama, two important concepts from ZooKeeper are *ephemeral nodes* and *watches*. The former allows clients (in our case, controllers) to create nodes that will be deleted when their session ends (allowing for fault detection), while the latter enables triggers for node modification (for event replication). A simplified ZooKeeper API is shown in table 4.2.
In the next sections we will discuss how ZooKeeper and its features helped us building Rama using its simple design and API.

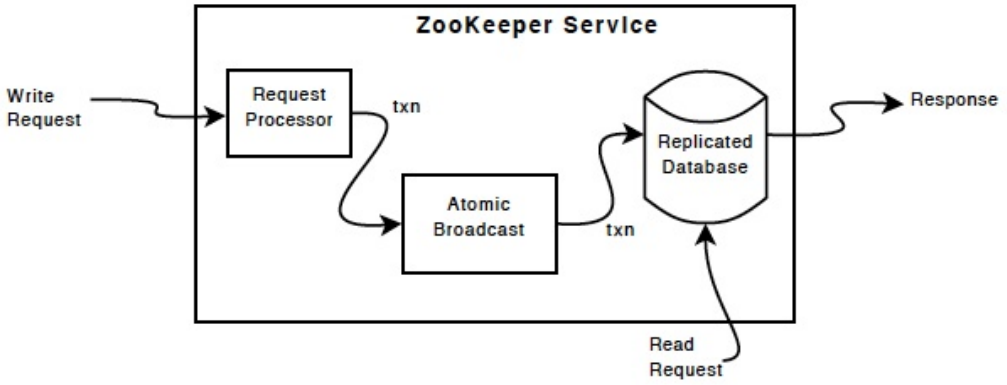


Figure 4.9: ZooKeeper components in each server

Method	Description
create(path, data, mode)	Creates a node with the given path and mode.
delete(path)	Deletes the node with the given path.
exists(path, watch)	Checks if the node with the given path exists or not
getChildren(path, watch)	Returns the list of the children of the node in the given path.
getData(path, watch)	Return the data of the node with the given path
setData(path, data)	Sets the data for the node of the given path if it exists
multi(operations)	Executes multiple ZooKeeper operations or none of them

Table 4.2: Simplified ZooKeeper API

4.1.5.2 Floodlight architecture

To understand some of the design decisions we made in Rama, it helps to understand Floodlight architecture. Floodlight is a modular controller, which means that multiple modules can be plugged into its core functionality. There are two kinds of modules in Floodlight: controller modules and application modules (see figure 4.10). Controller modules implement core network services (e.g., Link Discovery, Device Manager and Topology Manager) to be used by other modules. Application modules, making use of the core modules as needed, implement the network administrator's desired network logic (e.g., learning switch, firewall, traffic enginnering, etc.).
Modules can register to receive different type of events (e.g., Packet-In messages, switch modifications) and act accordingly to program the switches. These events are processed in a pipeline that traverses all modules that registered to receive that type of event.
Floodlight uses Netty, an asynchronous event-driven framework, for its I/O operations with switches. A thread pool with a fixed number of threads (worker threads) collects network events, with each worker thread processing one event at a time through the pipeline. Figure 4.11 describes the life cycle of worker threads. Note that each worker thread is only available to pick up a new network event when it finishes processing the pipeline of modules.

4.1.5.3 Rama architecture

Rama introduces two main modules into Floodlight: the *Event Replication* module (section 4.1.5.4) and the *Bundle Manager* module (section 4.1.5.7). Additionally, the Floodlight architecture was modified for performance reasons (see figure 4.12).

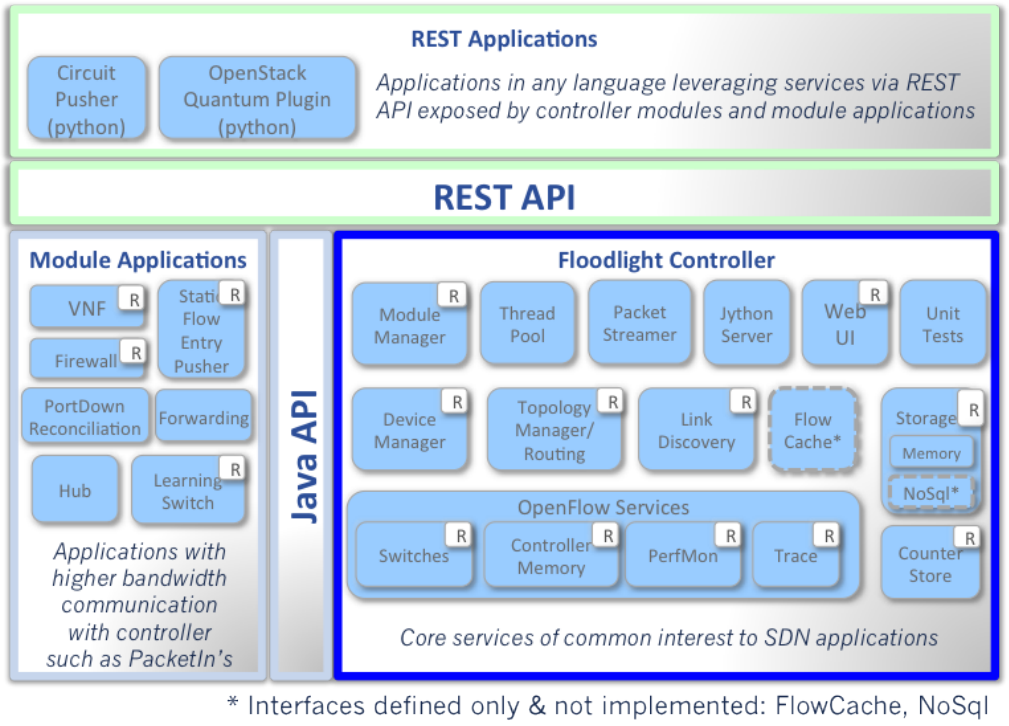


Figure 4.10: Floodlight modules architecture

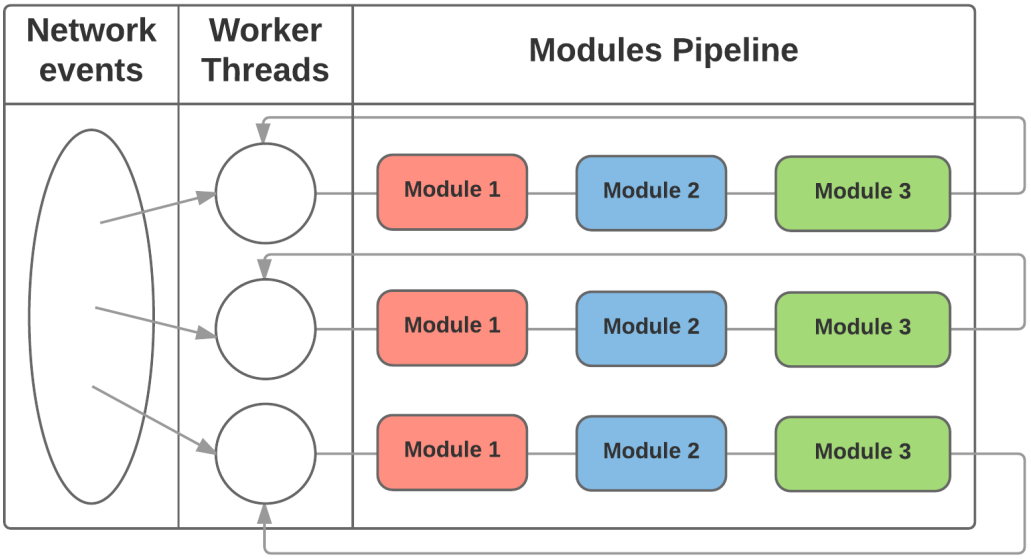


Figure 4.11: Floodlight thread architecture

In the original Floodlight, the worker threads are used to collect network events and to process the modules pipeline. If we kept this design it would not be possible to perform event batching before sending the events to ZooKeeper (this is further explained in section 4.1.5.4). In Rama, we want to free the threads that collect network events (netty worker threads) as soon as possible so that they can keep collecting more events. For this purpose, the worker threads' only job is to put events in a queue (Replication Queue). The Replication threads will take events from this queue and execute the logic in the Event Replication module, which will send the events to ZooKeeper in batches (see section 4.1.5.6). When ZooKeeper replies to the batched request, events will be added to the Pipeline Queue to be processed by the Floodlight modules.
One of the requirements for this work is to make the control plane transparent for applications (in

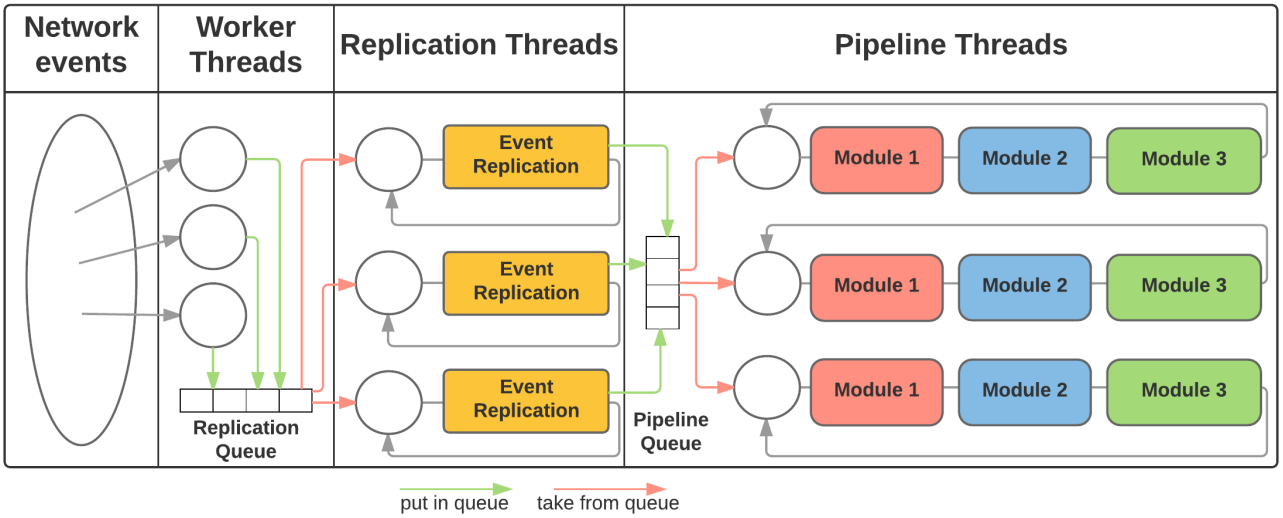


Figure 4.12: Rama thread architecture

Floodlight, to its modules). The Event Replication module is thus made completely transparent to other modules since it acts before the pipeline. The modules will continue to receive events as usual in Floodlight and process them by changing their internal structures and sending commands to switches. Returning to the proposed protocol (see section 4.1.4), we need to send commands to switches inside bundles, in a transparent way for modules. To achieve this, we only modified a core class of Floodlight, `OFSwitch.java` to interact with our Bundle Manager (see section 4.1.5.7). This core class contains the method `write(OFMessage m)` that sends a message to the switch using the established TCP connection and is (already) used by all modules to send commands to switches as part of the Floodlight architecture and design. Therefore, this process of sending messages inside OpenFlow Bundles is completely transparent to the existing and future Floodlight modules.

4.1.5.4 Event Replication and ZK Manager

The Event Replication module is the bridge between receiving events from the netty worker threads and putting them in the pipeline queue to be processed by the Floodlight modules. Events are only added to the pipeline queue after being stored in ZooKeeper. To separate tasks, Event Replication leverages on the ZK Manager, an auxiliary class that acts as ZooKeeper client (establishing connection, making requests and processing replies) and keeps state regarding the events (an event log and an event buffer in case of slaves) and switch leadership. We consider that an event is a pair <switch, message> and it is processed as such through the whole Rama architecture. Floodlight also has the notion of *context* that is passed to the modules pipeline, but we can ignore it for now. Event Replication and the ZK Manager work together to attain exactly-once event delivery and total order.
When an event arrives at the Event Replication module, we check whether the controller is in master or slave mode (for that switch). In master mode the event is replicated in ZooKeeper and added to its in-memory log. This log is a collection of `RamaEvent` objects which, apart from the switch and message, contains an unique event identifier (an incremental long number given by the current master), information related to the switches affected by the event, and of which switches already processed the commands sent. The events are replicated in ZooKeeper in batches (see section 4.1.5.6), so each replication thread simply adds an event to the current batch and becomes free to process a new event. Eventually the batch will be sent to ZooKeeper containing one or more events to be stored. Upon receiving the reply, the events that were stored will be added to the pipeline queue ordered according to identifier given by the master (i.e., event *i* can only be added to the queue after event *i-1*).
In slave mode, the event is simply buffered in memory for the case where the master controller fails. A special case is when the event received is the Packet Out that the master controller added to the

bundle. In this case, the slave marks that this switch already processed all commands for this event. Slaves also keep an event log as the master, but only events that come from the master are added to it. Events from the master arrive via *watches* set in ZooKeeper nodes. An important detail is that event identifiers are set by the master controller, and when slaves deserialize the data obtained from nodes stored in ZooKeeper, they get the same exact `RamaEvent` objects created by the master. Therefore, the events will be queued in the same order as they were in the master controller replica.

4.1.5.5 Fault Detection and Leader Election

The ZK Manager module is also responsible for detecting and reacting to controller faults. At start up, each controller will try to create an *ephemeral node* called *master* under the main *rama* folder, with the data set for its identifier within the group. Note that creating a node that already exists results in error. This means that only one controller (the master) will be able to create the node (and succeed), while the other controllers will get an error saying that the node already exists. In this case, (slave) controllers will leave a watch in the *master* node. When the master controller (the one that was successful in creating the *master* node) fails, the *ephemeral node* will be deleted by ZooKeeper and it will send a notification to every controller that has a watch in the *master* node. Upon receiving this notification, controllers will retry the procedure mentioned above and only one of them will be the new master, while the others reset the watch to be notified again in the future.

4.1.5.6 Event batching

Floodlight thread architecture was modified to allow event batching, which is done for performance reasons. Considering that ZooKeeper is running on a separated machine from the master controller replica, sending one event at a time to ZooKeeper would significantly degrade performance. Therefore, the ZKManager groups events before sending them to ZooKeeper in batches. Batches are sent to ZooKeeper using a special request called `multi`, which contains a list of operations to execute (e.g., create, delete, set data). For event replication, the multi request will have a list with multiple create operations as parameter. This request is sent after reaching the maximum configured amount of events (e.g., 1000) or some time after receiving the first event in the batch (e.g., 50ms). This means that each event has a maximum delay time (regarding event batching). Furthermore, to minimize the number of nodes created in ZooKeeper, each create operation will have data representing a list of events. This list size is bounded by the maximum allowable size of the data in each node, which is 1MB (1,048,576 bytes).
We have two goals when batching events: (i) we want to send as few requests as possible to ZooKeeper and (ii) we want to create as few nodes as possible in ZooKeeper. For (i) we use the multi request which can group multiple operations in one single request and for (ii) we group multiple events in a list and serialize it to use as the data for each create node operation (as opposed to having one *create node* operation for each event inside the multi request).
To give a concrete example, suppose a master controller receives 2050 events in less than 50ms, the batch size is 1000 events and we want each node in ZooKeeper to hold at maximum 100 events. Three multi requests will be made: two consisting of a list with 10 *create node* operations (each node with 100 events) and another with only one create node (with the remaining 50 events). In total, this processing required three requests to ZooKeeper and 21 nodes to be saved. Without batching, one would have 2050 requests to ZooKeeper and 2050 nodes.

4.1.5.7 Bundle Manager

The Bundle Manager and the ZK Manager work jointly to attain exactly-once command execution on switches. The Bundle Manager module keeps state related to all the bundles opened for each switch (as result of an event) and is responsible for adding messages, closing and committing them. We modified the write method in `OFSwitch.java` (class that is used by all modules to send commands to switches) to call the Bundle Manager, which will wrap the message sent by modules in a `OFPT_BUNDLE_ADD_MESSAGE`

and send it to the switch. Upon receiving this message, the switch will add the inner message to the (previously) opened bundle. This process is transparent to all modules: they do not need to be modified and are unaware of the presence of the Bundle Manager module. In the end of the pipeline, the Bundle Manager module is called to close and commit the bundles containing the messages added by the modules for this event. Note that one event may cause modules to send commands to multiple switches, so in this step the Bundle Manager may send `OFPBCT_COMMIT_REQUEST` to one or more switches. Before committing the bundle, the Bundle Manager also adds a `OFPT_PACKET_OUT` message to it, so that slave controllers will know if the commands for an event were committed or not in the switch (as explained in Section 4.1.4). This message will be received by the slave controllers as a `OFPT_PACKET_IN` message with data set by the master controller. This data contains the identifiers of the event, switch, and bundle.

4.1.6 Evaluation

In this chapter we evaluate Rama to understand its viability, the costs associated with the mechanisms used to achieve the desired consistency properties (without modifying the OpenFlow protocol or switches), and how it compares with the alternatives (Ravana [27]).
For our tests we used 3 machines connected to the same switch via 1Gbps links as depicted in figure 4.13. Each machine has an Intel Xeon E5-2407 2.2GHz CPU and 32 GB (4x8GB) of memory. Machine 1 runs one or more Rama instances, machine 2 runs ZooKeeper 3.4.8[3] and machine 3 runs Cbench to evaluate the controller performance. This setup tries to emulate a scenario similar to a real one with ZooKeeper on a different machine for fault-tolerance purposes and Cbench on a different machine to add some network latency.

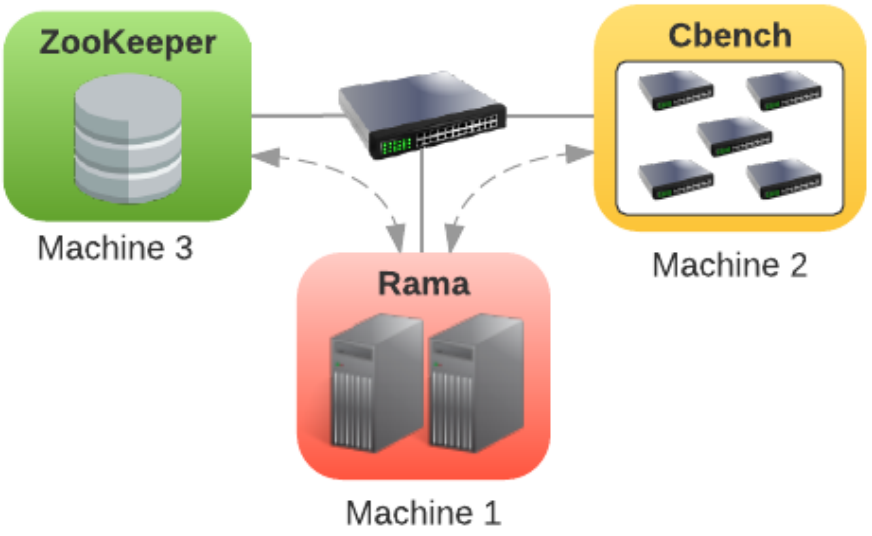


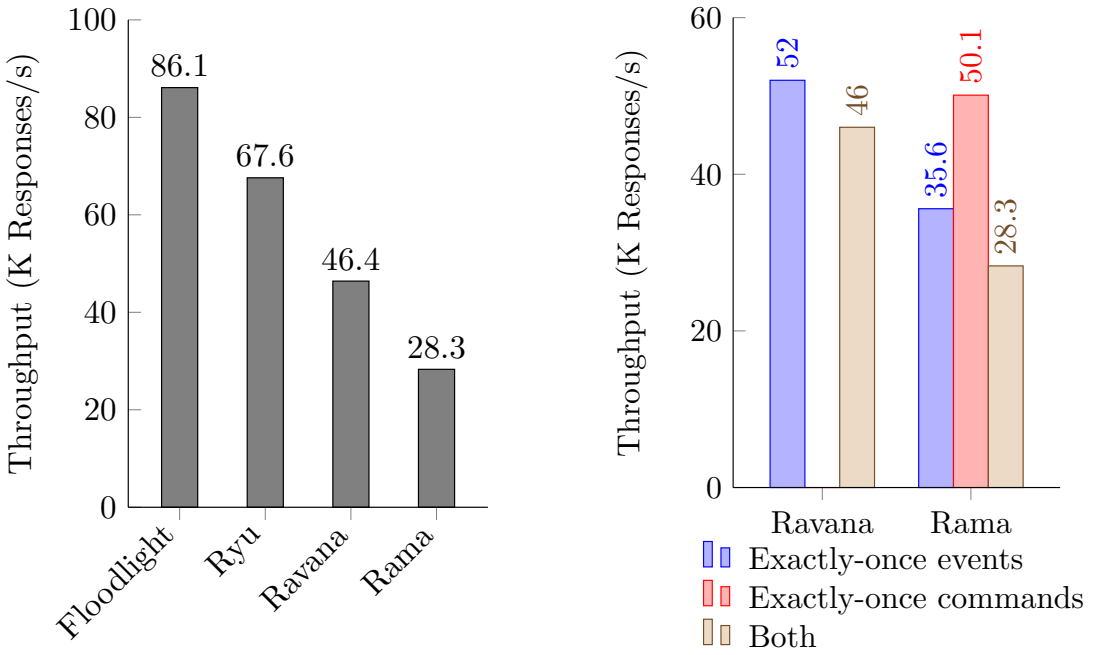
Figure 4.13: Experiment setup

Cbench is a tool that simulates a configurable number of OpenFlow switches that connect to a SDN controller and tests its performance by sending Packet In messages and measuring reported times. Cbench can run in two distinct modes: throughput and latency. In throughput mode Cbench always keeps its network buffer full of Packet In messages for each switch-controller connection and counts replies (Flow Mod or Packet Out messages) as they arrive. In latency mode Cbench sends one Packet In message and waits for a reply before sending the next one. Note that we modified Cbench to handle OF Bundle messages

[3] http://zookeeper.apache.org/doc/r3.4.8/

4.1.6.1 Rama Performance

We have compared the performance of Rama against Floodlight, Ravana [27] and Ryu (the base controller for Ravana). Figure 4.14a shows the throughput for each controller (for Rama and Ryu we use the results reported in [27], as they considered a similar setup). For Floodlight and Rama measurements we run Cbench emulating 16 switches.



(a) Controllers throughput

(b) Throughput with different consistency guarantees

Figure 4.14: Throughput comparison

Floodlight is optimized for performance achieving around 85K responses per second, with a configuration where only some core modules and a Hub application module is running. Note that we run Floodlight with a single worker thread so it comes down to Ryu's level (which has around 67K responses per second). Rama achieves close to 30K responses per second. We can see that this throughput is not quite at Ravana's level, as our solution incurs in higher costs compared to Ravana for the consistency guarantees provided.

In figure 4.14b we show, separately, throughput results considering the different levels of consistency provided by both Rama and Ravana. The exactly-once events consistency level ( ) ensures that no events are lost and that controllers do not process repeated events. Additionally, controllers must agree on a total order of events to be delivered to applications. For the latter, both Rama and Ravana rely on ZooKeeper to build a shared log across controllers. In our case, the master controller batches events in multiple requests to ZooKeeper, waits for replies, and orders the events before adding them to the Pipeline Queue. In Ravana the processing is equivalent.

The Exactly-once commands semantics ( ) ensures that commands sent by controllers are not lost and that switches do not receive duplicate commands. Ravana relies on switches to explicitly acknowledge each command and filter repeated ones. For Rama, this includes maintaining state of all opened bundles for switches, and sending additional messages to the switches. Instead of replying only with a Packet Out as in Floodlight, Rama must send messages to open the bundle, add the Packet Out to it, close the bundle and commit it. To evaluate this case, we had to modify Cbench. In our modified version of Cbench, a switch only counts a reply when it receives a Commit Request message from the controller (not when it receives a Bundle Add message with the Packet Out). This allows a faithful emulation of the performance of Rama in a real system – indeed, in Rama a packet will only be

forwarded after committing the bundle on the switch to guarantee consistent process.
Note that neither Rama nor Ravana wait for ZooKeeper to persistently store requests on disk (they both use ZooKeeper in-memory). In our case, the multi request is sent asynchronously (i.e., threads are freed to continue operation) and a callback function is registered. This function will be activated when ZooKeeper replies to our multi request and enqueues the logged events (in order) in the Pipeline Queue to be processed by the modules.
As show in Figure 4.14b, some guarantees are costlier to ensure than others. For instance, the cost of providing Exactly-once events semantics is higher than Exactly-once commands semantics. Note that we do not include the results from Exactly-once commands in Ravana as these are not available in [27].
Figure 4.15 shows how maintaining multiple switch connections affects Rama throughput. As switches send events at the highest possible rate, the throughput of the system saturates with around 16 switches. Importantly, the throughput does not decrease with a higher number of switches, which is similar to Ravana.
Rama batches events to reduce the communication overhead of contacting ZooKeeper. In practice, events are sent to ZooKeeper after reaching a configurable number of events in the batch (batching size) or after a configurable timeout (batching time).
To evaluate batching we conducted a series of tests with different configurations to understand how the batching size and time affects Rama performance (figure 4.16). Note that throughput is only affected by batching size and never by batching time. This happens because in throughput mode Cbench sends events at the highest rate possible and therefore the batching time is never reached.

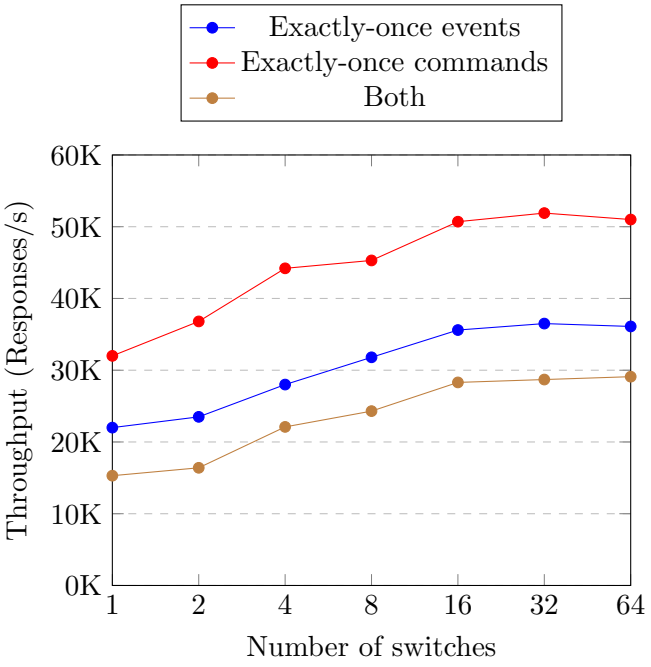


Figure 4.15: Rama throughput with different number of switches

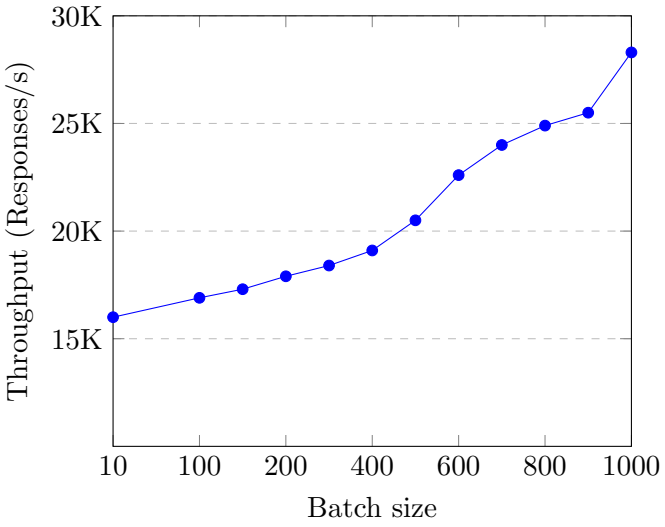


Figure 4.16: Variation of Rama throughput with batch size

4.1.6.2 Latency

For latency measurements, we want to know the average time that one request takes to be processed by Rama.
If we consider that the controller will be batching events until the batch is full or until a timeout is reached, the maximum (worse) possible latency is around the batching time (e.g., 100ms). This corresponds to the case where switches do not send enough events to fill the batch and so Rama waits until the timeout is reached.
On the other hand, if the number of events received fill the batch, Rama will process the events as soon as possible without waiting, resulting in a better latency. To test this, we use the results from running cbench in throughput mode (to trigger the batching limit) with 1 switch (to test the worst case and give a more realistic value) from figure 4.15, which is 15.3K responses per second. To get the seconds it takes for one request to be processed (the latency), we need to calculate the inverse (1/15300) which gives around 65 microseconds.

4.1.6.3 Failover Time

To measure the time for Rama to react to failures we use mininet, our modified version of OpenvSwitch, and iperf.
Mininet creates a virtual network with multiple switches and hosts that runs in one machine. This allows us to run command line programs in one host to communicate with other hosts (e.g., ping) using the virtual switches. The virtual switch used by mininet is a version of OpenvSwitch that we modified to handle bundle related messages and only forward packets after a Commit Request message. Iperf is a network bandwidth measurement tool that runs in both client and server mode. The client generates IP or UDP packets and sends them to the server (usually at a constant rate) and the server reports the results in terms of bandwidth, packet loss, and other parameters.
We setup a simple topology in Mininet with one switch and two hosts, one to act as iperf server and another as client. We start the client and sever in UDP mode, where the client generates 1 Mbit/sec during 10 seconds. The switch connects to two Rama instances and sends all events to both. Each Rama instance is connected to ZooKeeper server running on another machine (as before) with a negotiated session timeout of 500ms (the minimun we could set in ZooKeeper). To make sure that no rules are installed on the switch – so that events are sent to controllers each time a packet arrives – we run Rama with a module that only forwards packets (using Packet Out messages) without modifying the switch's tables.
Figure 4.17 shows the reported bandwith from the iperf server and indicates the time taken by Rama to react to failures.
Namely, the slave replica takes around 550ms to react to faults. This includes the time for:

(a) ZooKeeper to detect the failure and notify the slave replicas (500ms).

(b) Electing a new leader for the switches.

(c) The new leader to transition to master (finish processing logged events from the old master to reach the same internal state).

(d) Append buffered events to the log and start delivering unprocessed events in the log to applications so they start sending commands to the switches.

The major factor is the time ZooKeeper needs to detect the failure of the master controller. As it may be possible to reduce the session timeout, we were not able to achieve it without incurring into problems (either ZooKeeper did not accept client requests when the timeout was too small, or clients were constantly losing their session even when running). For comparison, Ravana reports a failover time of 75ms with 40ms to detect the failure (as opposed to our 500ms to detect the failure).

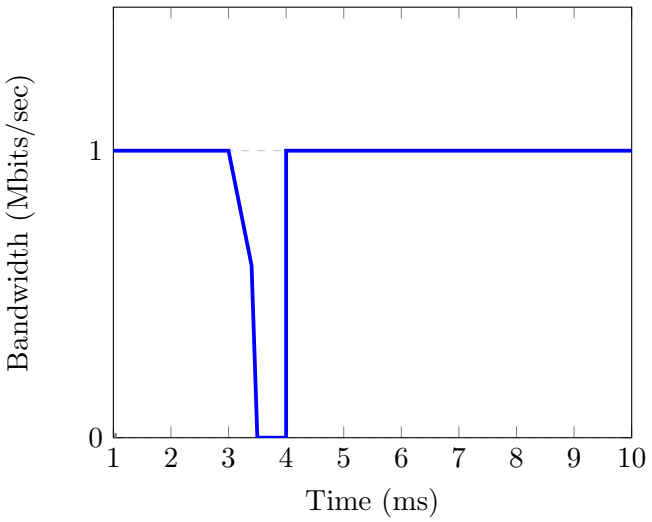


Figure 4.17: Rama failover time

Summing up, Rama comes close, but does not achieve the performance of Ravana. This is due to the fact that Rama incurs into higher costs (requires more messages to be sent over the network) in order to achieve the same properties as Ravana. Despite the small loss in performance, the value proposition of Rama of guaranteeing consistent command and event processing without requiring modifications to switches or to the openflow protocol makes it an effective enabler for immediate adoption of fault-tolerant SDN solutions.

4.1.7 Conclusion

In a fault-tolerant SDN, maintaining consistent controller state is not enough to achieve a correct system. Unlike traditional distributed systems, in SDN it is necessary to consistently handle not only controller state but also network state, to avoid loss or repetition of commands under controller failures.
A recently proposed fault-tolerant controller, Ravana [27], addresses this challenge and guarantees the required transactional semantics in the SDN context. Unfortunately, for this purpose it requires the OpenFlow protocol to be modified. In addition, switches need to be extend with techniques hitherto not available.
To address the above two challenges, we proposed Rama, a consistent and fault-tolerant SDN controller that also handles the entire SDN event processing cycle (event delivery by switches, event processing by controllers, command delivery by controllers, and execution by switches) *exactly-one*. Crucially, Rama differs from Ravana by not requiring modifications to the OpenFlow protocol nor to existing OpenFlow switches, allowing its immediate adoption.
Our extensive evaluation of Rama demonstrates that its advantage is achieved at a relatively small cost in performance when compared to Ravana.

4.2 Distributed SDN controller

The SUPERCLOUD network hypervisor follows an SDN approach. As such, the network control plane is physically separate from the data plane and the network view is logically centralized. The ability to write the network hypervisor based on a global, centralized network view is a fundamental concept of this new paradigm that we leverage in SUPERCLOUD. Reasoning based on a global view simplifies the design and development of our solution. The materialization of this concept can be made by means of a centralized SDN controller [22] that manages the network by configuring, solo, the underlying switches. However, the requirements of performance, scalability, and dependability of SUPERCLOUD

make a centralized solution unfeasible and demand a distributed and dependable control plane (such as ONIX [30]). Production-level SDNs such as Google's worldwide inter-datacenter network [26] and VMware's Network Virtualization Platform [29] indeed resort to such distributed solutions.
Unfortunately, distributed systems are difficult to understand, design, build, and operate [13]. In such systems, partial failures are inevitable, testing is challenging, and the choice of the "right" consistency model is hard. Ideally, the development of control applications should not be exposed to such a complex environment.
In an SDN, as the network is programmed based on a global network view, we argue it is fundamental this view to be consistent. This implies that upon a change in the network state all controllers should maintain the same consistent view of the network. Indeed, in this section we show, by example, that maintaining an eventually consistent view is not enough and may lead to network anomalies that range from transient outages to unexpected security breaches (Section 4.2.1).
Motivated by the need of strong consistency in the control plane to assure correct network policy enforcement, and well informed of the complexity of building a distributed system, we propose a novel, modular architecture for the SDN control plane of SUPERCLOUD (Section 4.2.2). Contrary to alternative designs (such as ONOS [6] and Onix [30]), the central element of this architecture is a consistent, fault-tolerant data store that keeps relevant network and applications state, guaranteeing that SDN applications operate on a consistent network view. This property ensures coordinated, correct behavior, and consequently simplifies application design. In our architecture each controller is responsible for managing a specific subset of the network switches, and is a client of this replicated, fault-tolerant data store. The architecture is modular in the sense that it separates the network configuration problem in two. On the one hand, the applications running in the controllers read state from the data store to program the network switches under their control, and write/update network state when informed of changes by the switches. On the other hand, the data store is responsible for coordination, and consequently for guaranteeing a strongly consistent view of the network and applications state across all controllers. This *separation of concerns* allows the division of the problem into more tractable pieces and is therefore a relevant aspect of the proposed design.
The main concern of this approach is the overhead required to guarantee consistency on a fault-tolerant replicated data store, which may limit its responsiveness and scalability. To mitigate this problem, we apply several optimization techniques for improving the performance of the data store operation (Section 4.2.3). In order to evaluate the impact of using these techniques, we analyzed the workloads generated by real SDN applications (learning switch, load balancer and device manager) as they interact with the data store (Section 4.2.5). Our results show that the proposed optimizations can substantially improve the latency and throughput of these applications (Section 4.2.7).
An important contribution of this work is to show that an architecture as the one we propose here results in a distributed control infrastructure that can efficiently handle representative workloads, while guaranteeing a network view that is always consistent. This property thus precludes network anomalies resultant from an inconsistent network view.

4.2.1 (Strong) consistency matters

Feamster et al. [18] have argued for the need to have a consistent view of routing state as a fundamental architectural principle for reducing routing complexity Indeed, network state consistency has been a recurring topic in the networking literature, and it gained significant momentum with SDN. For instance, Levin *et al.* [34] have analyzed the impact an eventually consistent global network view would have on network control applications and concluded that state inconsistency may significantly degrade their performance. In this section we intend to show, by means of an example, how inconsistencies in the control plane can lead to (potentially) severe network anomalies.
Consider Fig. 4.18. Controller C1 controls the left subset of the network, whereas C2 controls the right subset of switches. The solid arrows represent the path the packets that flow from host H1 to H2 take initially. Assume that at a certain time t controller C2 realizes the link between switches S3 and S4 is becoming congested and diverts H1-H2 traffic to S2 (as it is aware of the alternative S2-S4

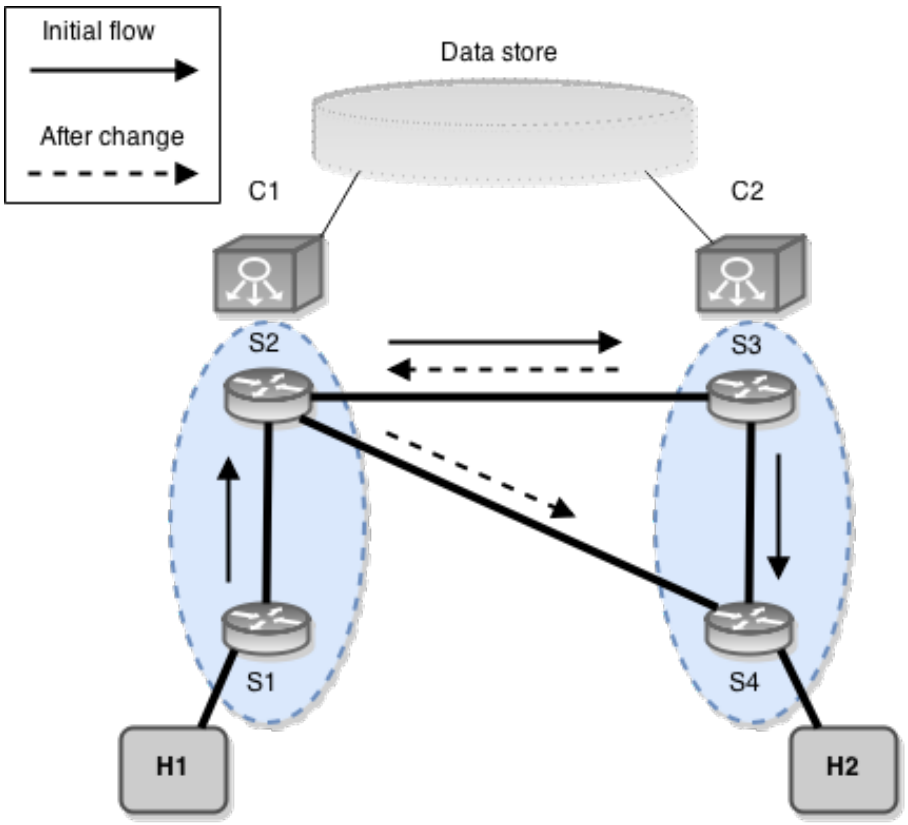


Figure 4.18: Consistency loop scenario

being less congested). Controller C2 writes the change to the data store to update the network view. An eventually consistent data store will reply to the controller C2 as soon as it starts processing its request. The controller will immediately install the new rules on the switches it controls. This will create a transient loop in the traffic from H1 to H2 (dashed arrow in link S3-S2) even if, as we assume in this example, the controllers use data plane consistency mechanisms. This network anomaly will *eventually* be corrected when the network state in the data store converges and controller C1 installs the new rules in its switches (the flow then starts using link S2-S4). A transient problem such as the one presented here can have consequences that range from an inconvenient hiccup in a VoIP call or a lost server connection to more alarming problems such as security breaches.
Importantly, this problem would not occur if one considers a strongly consistent data store.[4] When controller C2 sends the update to the data store, the data store replicas will have to reach an agreement before replying to this client. As a consequence, controller C2 has the guarantee that, when it receives the reply from the data store, *all* controllers are already informed of the fact and will act in unison. When coupled with data plane mechanisms (as the one proposed in the previous section) this guarantee will make it possible to give strong consistency guarantees in a fully distributed scenario.

4.2.2 Controller Architecture

We propose a novel distributed controller architecture that is fault-tolerant and strongly consistent. The central element of this architecture is a replicated data store that keeps relevant network and application state, guaranteeing that SDN applications operate on a consistent network view.
The architecture is based on a set of controllers acting as clients of the fault-tolerant replicated data store, reading and updating the required state with the application demands. In this sense, the architecture is data-centric – it is through the data store, acting as a virtual shared memory, that

[4]Formally, we say a replicated data store is *strongly consistent* if it satisfies linearizability [24], i.e., it mimics a centralized system.

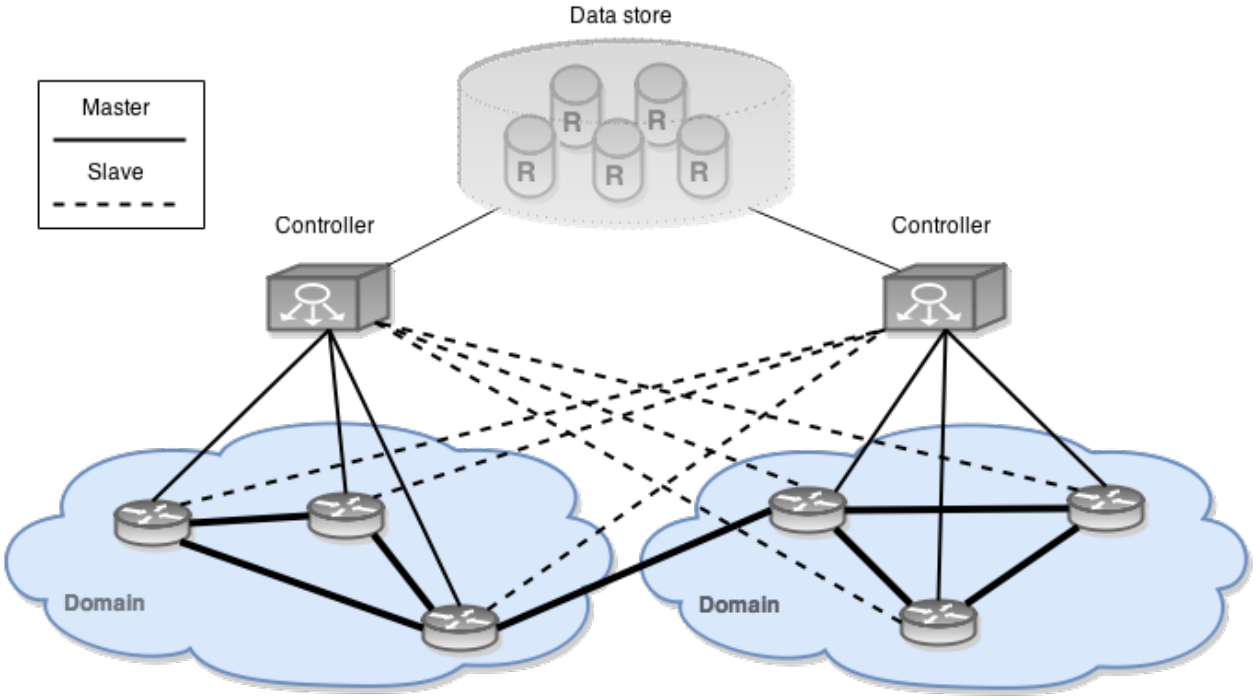


Figure 4.19: The controllers of different network domains coordinate their actions using a logically centralized (consistent and fault-tolerant) data store.

we support distribution. The data store mimics the memory model existent in centralized controllers such as Floodlight [46]. Therefore, other controllers can be easily integrated as a component of our architecture.

Fig. 4.19 illustrates our distributed controller architecture. The figure shows a set of controllers responsible for managing different subsets of the network switches (called a domain). All decisions taken by the control plane applications that run on the controller are based on data plane events triggered by the switches and the consistent network state (the global view) the controllers share using the data store. The fact that we have a consistent data store as the central piece of the architecture simplifies the interaction between controllers to reads and writes on a shared memory: there is no need for code that deals with conflict resolution or the complexities due to possible corner cases arising from weak consistency. This additional modularity of the design (when compared with alternative solutions [6]) allows a clean separation of concerns we deem important for its evolution.

In terms of dependability, our distributed controller architecture covers the two most complex fault domains in an SDN, as introduced in [28]. It has the potential to tolerate faults in the controller (if the controller itself or associated machinery fails) by having the state stored in the data store. It can also deal with faults in the control plane (the connection controller-switch) since each switch is connected to several controllers.

The controllers (and the applications they host) keep only soft state, which can easily be reconstructed after a crash. All the important network and application state is maintained in the data store. This simplifies the implementation of fault tolerance for the system since (1) all complex fault-tolerant protocols (e.g., fault-tolerant distributed consensus) are kept inside of the data store, reusing thus the large body of work existent in this area (e.g., [7, 8, 33, 42, 52]), and (2) the recovery of controllers is made very simple because there is no hard state to synchronize. Regarding the last point, once a controller fails any of the existent controllers can take over its place based on the network state that resides in the data store. The switches can tolerate controller crashes using the master-slave configuration introduced in OpenFlow 1.2 [43], which allows each switch to connect to $f+1$ controllers (f is a parameter of the system representing an upper bound on the number of faults tolerated), with a single one being master for each particular switch. In our design, controller fault tolerance is per

domain, meaning that the primary of one domain is the backup of another domain (as depicted in Fig. 4.19).
The bottleneck of the architecture is the data store. The reason are the complex coordination protocols that run between the replicas, which limit the scalability of the architecture. This is an unavoidable consequence of the property we want to guarantee: a consistent global view across controllers. Anyway, it is possible to significantly improve the data store performance by equipping it with a set of state-of-the-art distributed systems techniques that are particularly useful for the target environment. In the next sections we thus focus on the design and implementation of the data store, considering several optimizations, and using representative workloads generated by real and non-trivial network applications to evaluate it.

4.2.3 Data Store Design

A common way to implement a consistent and fault-tolerant data store is by using replicated state machines [52]. This technique considers a set of replicas implementing a service (e.g., the data store) accessed through a total order multicast protocol that ensures all replicas process the same sequence of requests. The core of a total order multicast protocol is a consensus algorithm such as Paxos [33], Viewstamped Replication [36], BFT-SMaRt [8] or RAFT [42]. In this work we are using BFT-SMaRt for this purpose.
A fundamental concern of these systems is their limited scalability and performance overhead. However, recent work in this field has started showing interesting performance figures of up to tens of thousands of small updates per second [25, 8]. To have an idea of how these figures compare with previous solutions, this performance is three orders of magnitude better than what was reported for the initial consistent database used in the Onix distributed controller [30]. These performance numbers start justifying the use of such systems as a consistent backend for supporting a distributed controller, especially if complemented with specific optimizations.
In the following we present a set of techniques used in the design of an *efficient* data store for network control applications. As a starting point we consider a data store supporting an arbitrary number of tables (uniquely identified by their name). Each table maps unique keys to opaque values of arbitrary sizes (i.e., raw data). The server has no semantic knowledge of the data present in the data store and supports simple operations such as create, read, update, and delete.

4.2.3.1 Cross References

A classical key-value table is restricted to a single key to identify a value despite the number of unique attributes that are associated with the value. However, in some cases it is useful to have an additional table that relates a "secondary" key with the value indexed by some "primary" key. As an example, consider an application tracking hosts accessing a network that assumes a device is uniquely identified either by an IP or MAC address. Therefore, we could use two tables: table `IPS`, relating IPs (key) to MACs (value), and table `MACS`, relating MACs (key) to devices (value). This is a reasonable scheme in a local environment given that the cost to obtain a device with a MAC address or its IP is the same. However, in a distributed environment, this requires two accesses to the data store just to obtain a single device with an IP address (one to fetch the MAC, and another to fetch the device), incurring in a significant latency penalty.
To avoid this penalty for obtaining a single value we implemented a Cross Reference table, which in this example is able to obtain the device with a single access to the data store. Fig. 4.20 illustrates how our Cross Reference table works. In this example, the client (*controller*) configures the `IPS` table as a cross reference to the `MACS` table. In practice, this is understood as: the values of the `IPS` table can be used in the `MACS` table. With this setting, the client can fetch a device from the `IPS` table with a single data store operation (the *getCrossReference* method). Thus, this operation halves the latency penalty required to obtain the device.

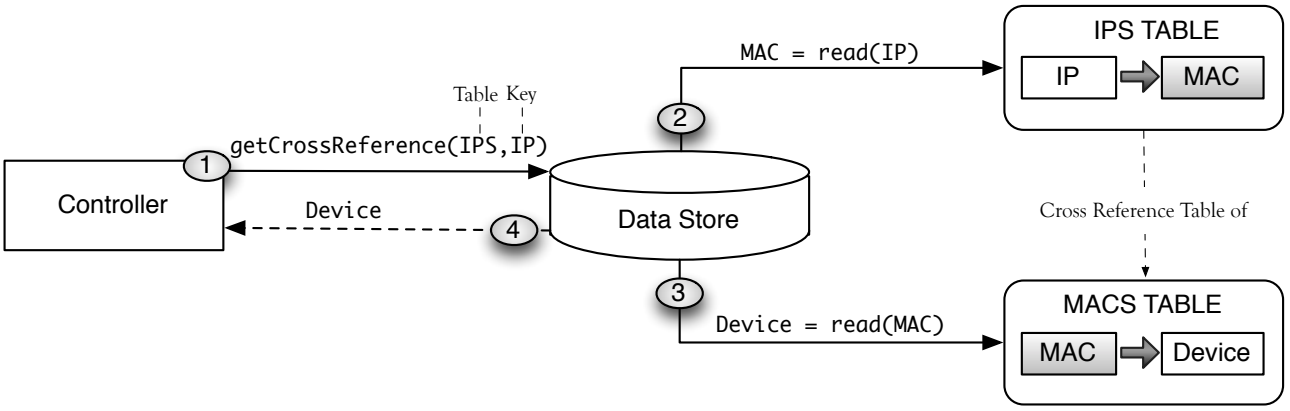


Figure 4.20: Cross Reference table example with Table *IPS* configured as a cross reference to table *MACS*. First, the controller sends a cross reference read request to the data store for table `IPS` and key *IP* (1). Then, the data store performs a read in table *IPS* to obtain the key *MAC* (2), that is used in table `MACS` (3) to finally reply to the client the *Device* (4).

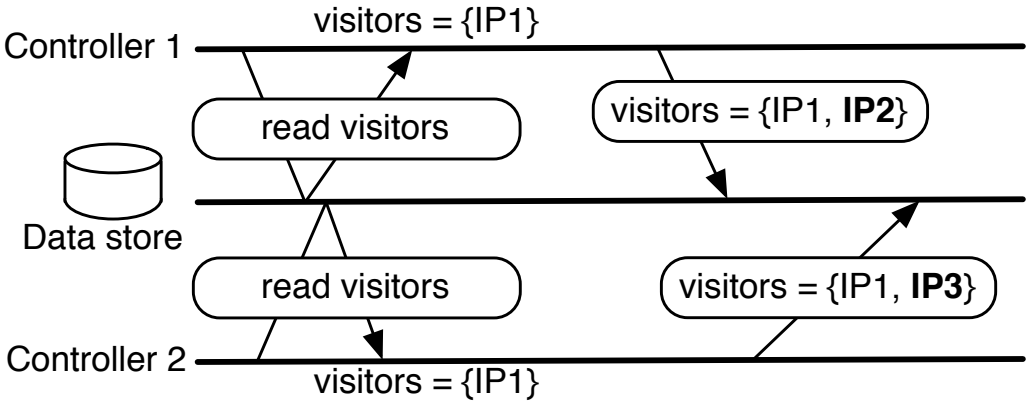


Figure 4.21: Concurrent updates lead to loss of data

4.2.3.2 Versioning

Despite being strongly consistent, our data store is still exposed to the pitfalls of concurrent updates performed by clients. Namely, the loss of data caused by overlapping writes. As an example, consider an HTTP network logger running in a controller that maintains a key-value table in the data store to map each web page accessed to the set of IP addresses that have visited it. For updating the set, the controllers need first to fetch it, add an element locally, and update the data store with the new set. Fig. 4.21 illustrates how, in this setting, concurrent updates can lead to data loss.
Controllers 1 and 2 fetch the same `visitors` set for a particular web site (uniquely identified by the URL), and then they replace it by a new set that includes IP2 and IP3, respectively. The lack of concurrency control results in the loss of the write operation that includes the IP2 visit to the site (visitors={IP1,IP2}), because the last write (visitors = {IP1, IP3}) overwrites the previous.
To solve this problem we make each table entry (i.e., key-value pair) be associated with a monotonically increasing counter (the version number) that is incremented in every update executed for the key. By creating this Versioning mechanism, we empower the data store with the capability to detect and prevent conflicting updates that otherwise could result in data loss.

4.2.3.3 Columns

With a key value data model, clients are able to map a unique key to any arbitrary value with no semantic meaning for the data store (it is just raw data). This is a quite limited data model since

values are often composed of multiple attributes. Consequently, when the client interest lies towards a small portion of the value (e.g., a single attribute), this model can be a bottleneck, since both the update messages (sent to the data store), and reply messages (received from the data store) may contain unnecessary attributes (thus increasing the latency penalty for the client). Therefore, we expanded the key value table to allow clients to access the individual components of a value with an additional key (i.e., the column name). With Columns, we enhance the unidimensional model of a key value table to a bi-dimensional one whereby two keys (as opposed to one) can access an individual attribute of a value inside a table.
Despite the fact that a Column table decomposes a value into columns, the client is still able to manipulate the entire value. Namely, the client is still able to retrieve or update a value "entirely" even if she is not aware of the column names that compose a value. Furthermore, the column names are not static, not even in the context of a table. Each key-value entry may have different columns, and clients can add and delete columns from a value as they see fit (in run-time).

4.2.3.4 Micro Components

So far, we have focused in particular client use cases (i.e., multiple keys to obtain a value, concurrent updates, and manipulation of attributes) to introduce techniques that reduce the number and size of messages during the interaction between clients and the data store. However, for an arbitrary number of operations that have no explicit connection to each other we need a more general abstraction. Imagine a control application needs to execute the following transaction in the data store: *"read two values from different tables: the total number of bytes allowed to be forwarded and the byte counter from the forwarding table (that gives the number of bytes effectively forwarded)"*, *"subtract them"* and *"update the first table with the new number of bytes allowed to be forwarded"*. With the current interface, this set of operations will require multiple controller-data store interactions, thus revealing a significant latency penalty for such a simple task.
To address this limitation, we propose the use of a mechanism for running the whole transaction at the server side. More specifically, we deploy specific data store extensions, called Micro Components. This is similar to the use of extensions in coordination services [16] or stored procedures in transactional databases. The most significant advantage of a micro component is performance since it allows the client to merge multiple operations in a single method reducing the number of accesses to the data store.

4.2.3.5 Cache

With a cache the client can keep frequently accessed values locally, for a particular data store table. For this table, each value that is read or written from and to the data store is added to the local cache. As the cache affects consistency (a point we will return to below), the client has the option to define a bound on the window of inconsistency she is willing to tolerate. For each client request, the cache returns the local value if the request is within the staleness bound. Otherwise, the cache retrieves the value from the data store. This is shown in Fig. 4.22. Of course, if the bound specified by the client is zero, the cache is bypassed and the request is sent to the data store.
A strong point of our architecture is the guarantee of a consistent global view for network and application state. As such, the reader may correctly question why we consider the use of a cache. Our goal in this respect is to offer some level of flexibility, as we anticipate not all state to require the same level of consistency. We thus leave the client with explicit control over the window of inconsistency she is willing to accept. Particular state may lead to inconsistencies, as that concerning cross-domain operations, such as the example in Section 4.2.1. But since in our design a single controller controls all switches in its domain, no concurrency issues will occur for non-cross domain operations and a part of the state may be served from the cache.

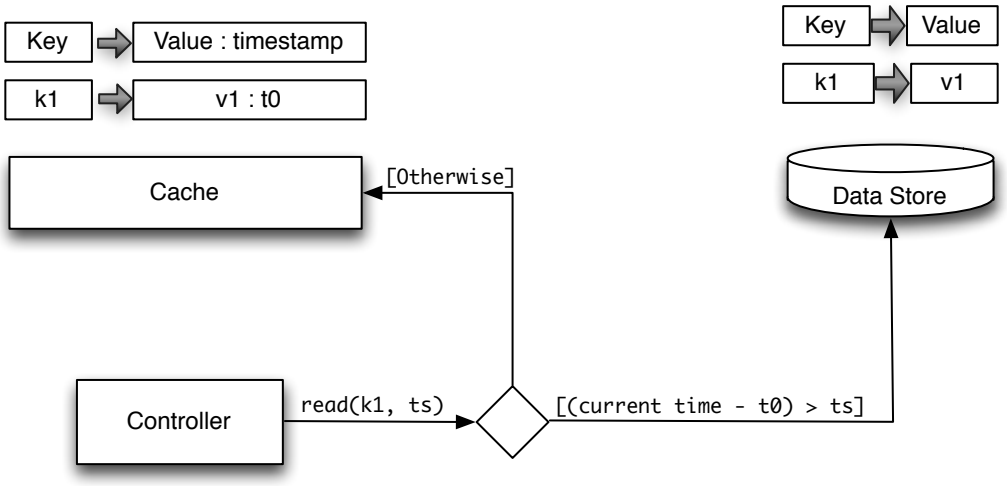


Figure 4.22: Reading Values from the Cache: the client performs a read on the data store for key *k1* and accepted staleness *ts*. The cache returns a local value iff: it was added to the cache for less than *ts* time. Otherwise, it obtains the value from the data store (and updates the cache).

4.2.4 Implementation

We implemented a prototype of the previously described architecture by integrating the Floodlight controller with a data store built on top of a state-of-the-art State Machine Replication library, BFT-SMaRt [8]. Furthermore, we modified three SDN applications provided with Floodlight in order to operate with our data store: Learning Switch (a common layer2 switch), Load Balancer (a round-robin load balancer), and Device Manager (an application that tracks devices as they move around a network).
The BFT-SMaRt library supports a tunable fault model and durability. The fault model can be either Byzantine[5] or crash-recovery. For performance reasons, we consider the crash-recovery model whereby a process (i.e., replica) is considered faulty if either the process crashes and never recovers or the process keeps infinitely crashing and recovering [11]. The library operates under an eventually synchronous model for ensuring liveness. For durability, a state transfer protocol guarantees that state survives the failure of more than f replicas (the number of replicas that can fail simultaneously).
Our data store is, therefore, replicated and fault-tolerant, being up and running as long as a majority of replicas is alive [33]. More formally, $2f + 1$ replicas are needed to tolerate f simultaneous faults.
The implemented data store supports all optimizations described in the previous section. The only noticeable limitation of our proof-of-concept prototype is related with the support for Micro Components. Currently, they are statically included in the data store codebase along with the classes that each micro component requires to operate.

4.2.5 Workload Analysis

To evaluate our data store design, we consider three applications in isolation and analyze how they interact with the data store (with and without the optimizations).
Fig. 4.23 illustrates the scenario. Whenever a switch triggers a message to be sent to an application, the latter executes one or more operations on the data store. Then, as soon as the application finishes, it can reply to the switch with a message (named "controller reaction" in the figure). In the end, a *data store workload* is a trace (or log) of data store requests and replies resulting from the processing of a data plane event by a particular application.

[5]In a Byzantine fault model, processes can deviate from the protocol in any way. Namely, they can lie, omit messages, and crash.

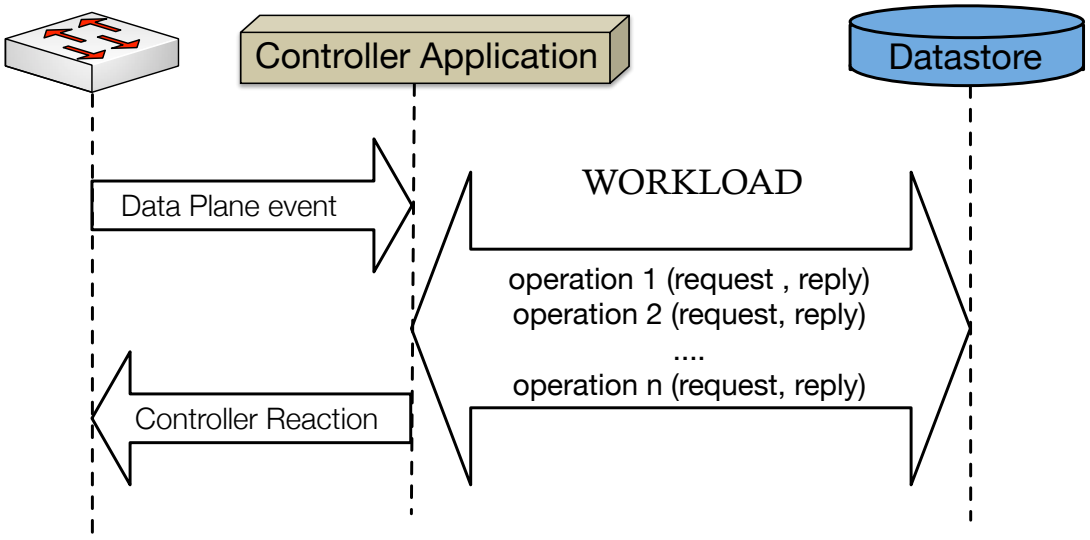


Figure 4.23: Each data plane event triggers a variable number of operations in the data store. The trace of those operations and their characteristics is a workload.

We selected three representative control applications for validating our design, *Learning Switch*, *Load Balancer* and *Device Manager*. First, we analyze the *Learning Switch*, a simple application commonly used to benchmark SDN designs. Our second application, the *Load Balancer*, implements a flow distribution policy and represents a class of applications that benefits from accessing up to date information on a data store before deciding the destination of a flow. The last application is the *Device Manager*, a non-trivial application that tracks devices in the network and builds an information base that other applications can rely on.

4.2.6 Workload Generation

For the first phase of our study we emulated a network environment in Mininet [23] and connected it to our prototype. We use Mininet to send the appropriate OF data plane messages from the switch to the controller, triggering an access to the data store (see Fig. 4.23). We record all communication between the controller and the data store.
Our network environment consists of a single switch and at least a pair of host devices. After the initialization of the test environment (e.g., creation of a switch table, configuration of the Load Balancer application, etc.) we generated ICMP requests between two devices. The goal was to create OF traffic (`packet-in` messages) from the ingress switch to the controller. Then, for each OF request, the controller performs a variable, application-dependent number of read and write operations, of different sizes, in the data store (i.e., the *workload*). In the controller (the data store client), each data store interaction is recorded entirely (i.e., request and reply size, type of operation, etc.) and associated with the data plane event that has caused it.
Table 4.3 contains all the captured workloads for each application we considered: Learning Switch (`ls`), Load Balancer (`lb`), and Device Manager (`dm`). The initial message sizes for each operation recorded are displayed in column `init`. There is one column for each optimization described in Section 4.2.3 (Cross References - `cref`; Versioning - `vers`; Columns - `cols`; Micro Components - `micro`).

4.2.6.1 Learning Switch

The Learning Switch application emulates a layer 2 switch forwarding process based on a switch table that associates MAC addresses to switch ports. The switch is able to populate this table by listening to every incoming packet that, in turn, is forwarded according to the information present in the table.

Workload	Operation	Type	(Request size, Reply size)				
			init	cref	vers	cols	micro
ls-bcast	1) Associate source address to ingress port	W	(113,1)	-	-	-	-
ls-ucast	1) Associate source address to ingress port	W	(113,1)	-	-	-	(56,6)
	2) Read egress port for destination address	R	(36,77)	-	-	-	
lb-arp	1) Get VIP id of destination IP	R	(104,8)	(104,509)	(104,513)	(62,324)	-
	2) [Get VIP info (pool)]	R	(29,509)				-
	1) Get VIP id of destination IP	R	(104,8)	(104,509)	(104,513)	(62,324)	-
	2) [Get VIP info (pool)]	R	(29,509)				-
lb-vip	3) [Get the chosen pool]	R	(30,369)	-	(30,373)	-	
	4) [Conditionally replace pool]	W	(772,1)	-	(403, 1)	-	(11,4)
	5) [Read the chosen Member]	R	(32,221)	-	(32,225)	(44,4)	
	1) Get source key	R	(408, 8)	(408,1274)	(408,1278)	(486,1261)	(28,1414)[a]
	2) [Get source device]	R	(26,1444)				
dm-known	3) [Update timestamp]	W	(2942,0)	(2602,0)	(1316,1)	(667,1)	(36,0)
	4) Get target key	R	(408,8)	(408,1199)	(408,1203)	(416,474)	Not needed
	5) [Get target device]	R	(26,1369)				
	1) Read source key	R	(408,0)	-	-	(486,0)	(28,201)[b]
	2) [Increment counter]	W	(21,4)	-	-	-	
	3) [Update device table]	W	(1395,1)	(1225,1)[b]	-	(1183,1)	
dm-unknown	4) [Update MAC table]	W	(416,0)	-	-	-	
	5) [Get from IP index]	R	(386,0)	-	-	-	
	6) [Update IP index]	W	(517,0)	-	-	-	
	7) Get target key	R	(408,8)	(408,1208) [c]	(408,1212)	(416,474)	Not needed
	8) [Get target device]	R	(26,1378)				

Table 4.3: Learning Switch, Load Balancer and Device Manager operations and respective sizes (in bytes) across different optimizations. Operations under brackets are executed only in certain conditions. Operations with dashed entries translate into no improvement from the respective optimization. Legend: a) This operation also fetches the target device; b) This operation also fetches the destination device; c) Differences in sizes caused by a marshalling improvement.

The two significant workloads we consider for this application are related with the type of packet observed by the controller.

Broadcast Packet Workload (`ls-bcast`)—The operations (`init` column) in Table 4.3 show that for the purpose of associating the source address of the packet to the ingress switch-port where it was received, the Learning Switch application performs one write (W) operation with a request size (Request) of 113 bytes and reply size (Reply) of 1 byte.

Unicast Packet Workload (`ls-ucast`)—This workload creates an additional operation to the previous one, since for every unicast packet we must also fetch the known switch port location of the destination address. The operations (`init` column) at Table 4.3 shows that this second operation comprises a 36-bytes request and a 77-bytes response, which contains the known switch port.

4.2.6.2 Load Balancer

The Load Balancer application employs a round-robin algorithm to distribute the requests addressed to a Virtual IP (VIP) address across a set of servers. Again, we consider two workloads for this

application.
ARP Request (`lb-arp`)—This workload (see column `init` in Table 4.3) shows the operations that result from an OpenFlow' `packet-in` message caused by an ARP request querying the VIP MAC address. In the first operation, the Load Balancer application attempts to retrieve the `vip-id` for the destination IP. If it succeeds, then the retrieved `vip-id` is used to obtain the related VIP entity in operation #2 (we surround the operation description with brackets to mark it as optional—it is only executed when the first succeeds). Although only the MAC address is required to answer the ARP request, the VIP entity is read entirely. Notice that the size (509 bytes) is two orders of magnitude larger than a standard MAC address (6 bytes).
Packets to a VIP (`lb-vip`)—This workload (see column `init` of Table 4.3) shows the detailed operations triggered by IP packets addressed to a VIP. The first two operations fetch the VIP entity associated with the destination IP address of the packet. From the VIP we obtain the `pool-id` used to retrieve the *Pool* (operation #3). The next step is to perform the round-robin algorithm by updating the `current-member` attribute of the retrieved *Pool*. This is done locally. Afterwards, the fourth operation aims to replace the data store *Pool* by the newly update one. If the *Pool* has changed between the retrieve and replace operation this operation fails (reply equal to 0) and we must try again by fetching the *Pool* one more time (repeating operation #3 and #4). In order to check if the versions have changed, the replace operation contains both the original and updated Pool to be used by the data store. In order to succeed, the original client version must be equal to the current data store version when processing the request. If successful (reply equal to 1), we can move on and read the chosen *Member* (server) associated with the `member-id` that has been determined by the round-robin algorithm.

4.2.6.3 Device Manager

The Device Manager application tracks and stores host device information such as the switch-ports attachment points (ports the devices are connected to). This information is retrieved from all OpenFlow messages the controller receives.
Known Devices Workload (`dm-known`)—When a packet from a known device is received, a `packet-in` request triggers the operations seen in column `init` of Table 4.3 on the data store. The first two operations read the source device information. Then an update is required to update the "last seen" timestamp of the device generic `entity`. Notice that the size of this request is nearly twice that of a device (1444 bytes). This is due to the fact that this is a standard replace containing both the original device (fetch in step #2) and the updated device. This operation will fail if other data store client has changed the device. If so, the process is restarted from the beginning. Otherwise, the last two operations can fetch the destination device.
Unknown Device Workload (`dm-unknown`)—This workload is triggered in the specific case in which the source device is unknown and the OF message carries an ARP reply packet. The first operation reads the source device key. Being that it is unknown (notice, in the table, that the reply has a size of zero bytes corresponding to `null`) the application proceeds with the creation of the device. For this, the following write (second operation) atomically retrieves and increments a device unique `id` counter. Afterwards, the third and fourth operation updates the `devices` and `macs` tables respectively. Then, since the `ips` table links an IP to several devices, we need to first collect a set of devices (operation #5) in order to update it (operation #6). If successful, the Device Manager has created the new device information and can, finally, move to the last two operations that fetch the destination device information. If unsuccessful, the process is repeated from step #5.

4.2.7 Performance Evaluation

After obtaining the workloads for each application, we executed a series of experiments to evaluate the performance of the data store considering all workloads (including optimizations). The objective

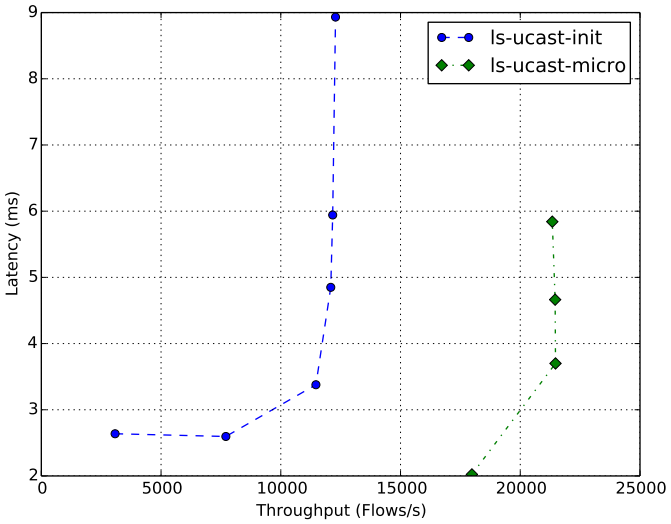


Figure 4.24: Learning Switch performance (ls-ucast workload)

here is to shed light on the data store performance (latency and throughput) when subject to realistic workloads, as this is the bottleneck of our consistent distributed control plane architecture.

4.2.7.1 Test environment

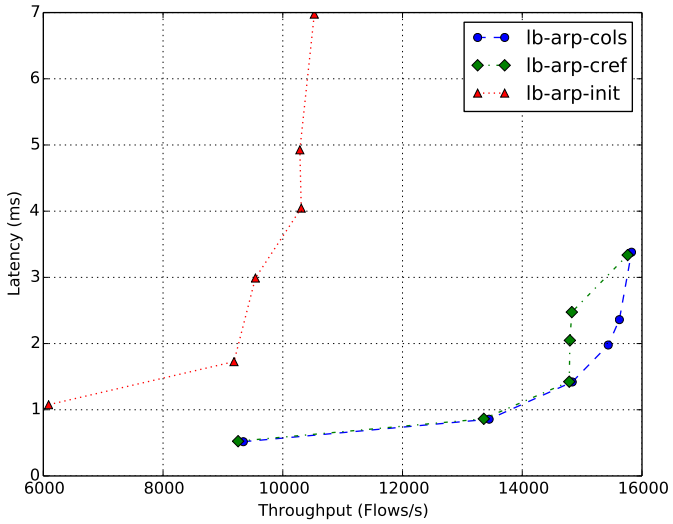
We execute our experiments in a four-machine cluster, one for the data store client (responsible for simulating multiple controllers), and three for the data store (to tolerate one crash fault, $f = 1$).
Each machine has two quad-core 2.27 GHz Intel Xeon E5520 and 32 GB of RAM memory and were interconnected with Gigabit Ethernet. The software environment was Ubuntu 12.04.2 LTS with Java SE Runtime Environment (build 1.7.0_07-b10), 64 bits.
The data store client runs as a single Java process, but executes multiple threads that replay a simulation of the recorded workload with an equal number of messages and payloads (i.e., same message type and size). We emphasize that in order to replay a workload composed of $op_1, op_2, ...op_n$ operations, a thread must first send operation op_1, wait for a reply from the data store and only after, send operation op_2 (and so on until op_n).
This simulation is repeated for a variable number of concurrent data store clients (representing different threads in one controller and/or different controllers). From the measurements we obtain throughput and latency benchmarks for the data store under different realistic loads.
Each workload was run 50 thousand times, measuring both latency and throughput. The values shown in this section are the 90th percentile of all measurements.

4.2.7.2 Learning Switch

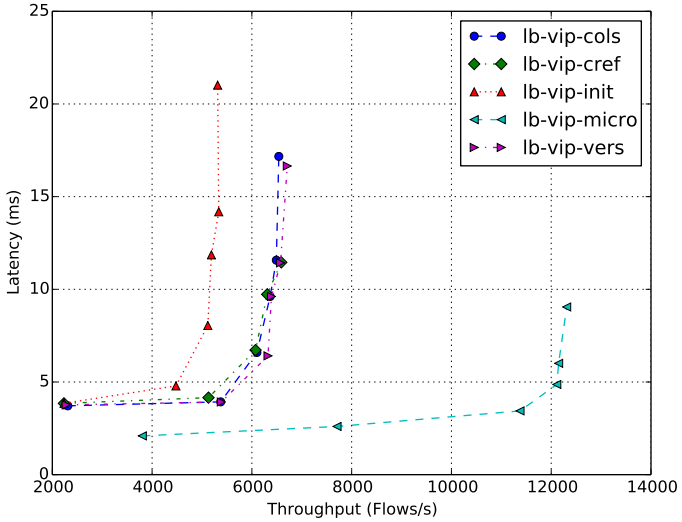
Fig. 4.24 shows that the unicast workload prior to being optimized (`ls-ucast-init`) leads to a throughput of 12k Flows/s, with a 6 ms latency. However, when using the `ls-ucast-micro` optimized workload, the throughput increases to 22k Flows/s while the latency decreases to less than 4 ms. The reason behind this improvement in both latency and throughput is due to a decrease in the number of messages exchanged with the data store.

4.2.7.3 Load Balancer

In Fig. 4.25 we present the results of our experiments considering different load balancer workloads.



(a) lb-arp workload.



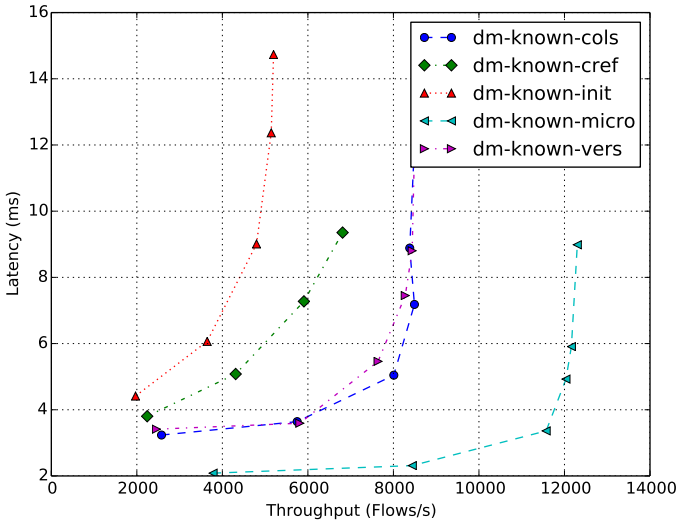
(b) lb-vip workload.

Figure 4.25: Load Balancer performance.

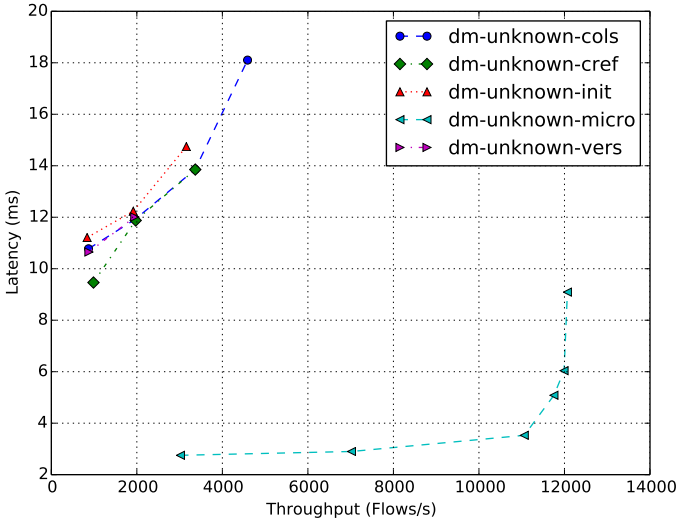
When considering the lb-arp workload (Fig. 4.25a), the reduction from two to one data store interaction of the optimizations improves the throughput by up to 60%. However, the message size reductions from `cref` to `cols` (see row lb-arp in Table 4.3) have little to no effect in the performance of the system under this workload.
For the lb-vip workload (Fig. 4.25b) these data store optimizations are less effective, improving the peak throughput from 5.4k Flows/sec to 6.5k Flows/sec (less than 20%). The improvement is much more significant when microcomponents are used (`micro`): the throughput more than doubles and latency decreases by 50%, when compared with the non-optimized data store (`init`).

4.2.7.4 Device Manager

Fig. 4.26 presents the performance results from the data store when considering the two Device Manager workloads.
Our results confirm that the most significant improvement comes from using a micro component to



(a) dm-known workload.



(b) dm-unknown workload.

Figure 4.26: Device Manager performance.

create a device (`dm-known-micro` in Fig. 4.26a and `dm-unknown-micro` in Fig. 4.26b). This optimization reduces the latency penalty significantly while increasing the data store throughput from 5k and 3k Flows/s to more than 12k Flows/s, a three to four-fold increase in throughput, with a significant latency reduction in both workloads. Additionally, under adequate workloads the data store has a significantly low latency penalty: the use of micro-components shows a steady latency penalty of less than 4 ms for a throughput of more than 11k Flows/s in both workloads.

4.2.7.5 Cache

In the workloads shown in the previous sections, the applications perform all operations in the data store. However, it is possible to perform some of the operations of each workload locally (in the controller) by integrating the applications with our cache interface, as explained before.
In this section we show how we modified the workloads with the cache integration, the effect that it can have on the staleness of the data used by the clients (i.e., the applications), and if any consis-

Workload	Operation	Type	(Request,Reply)	Case	Throughput (kFlows/s)	Latency (ms)
ls-ucast	1) Associate source address to ingress port	W	(29,1)	best	∞	0
cache	2) Read egress port for destination address	R	(27,6)	worst	21.5	4.7
lb-vip	1) Get VIP pool for the destination IP*	R	(62,324)	best	21.4	4.8
cache	2) [Round-robin pool and read chosen Member]*	W	(21,4)	worst	11.4	3.4
dm-known	1) Get source and target devices*	R	(28,1414)	best	21.4	3.6
cache	2) [Update "last-seen" timestamp of source device]*	W	(36,0)	worst	11.1	3.5

Table 4.4: Cache optimized workloads operations and sizes (in bytes). Operations in gray background are cached.

tency problems can arise. We conclude with a theoretical analysis of the performance of the cache optimization for the considered workloads.

Learning Switch. The Learning Switch is a single writer, single reader application,[6] so it is possible to introduce caching without affecting the consistency semantics or the staleness of the data. To clarify, a cached entry in the Learning Switch application is *always* consistent with the data store since no other controller modifies that entry. Therefore, with cache we can potentially avoid the data store while processing data plane events, thus avoiding the two operations in the unicast packet workload (ls-ucast).

The `ls-ucast-cache` workload in Table 4.4 shows the operations that can be cached (in gray background). Note that it was based on `ls-ucast-msg` workload as opposed to `ls-ucast-micro`, since our current implementation of the cache is based on the former.

First, we avoid re-writing the source address to source port association when we already know it, because it is present in cache (operation #1). Second, we can also avoid the read of operation #2, which queries the egress port of the currently processed packet, if that entry is available in cache. With this improvement, we no longer have to read values from the data store as long as they are available in cache, and we still get consistent values because when we update a value we also update the cache. Note, however, that the cache is also limited in size, thus entries are refreshed over time. In the case of cache misses (i.e., entry is not available in cache), the operation is performed in the data store.

Load Balancer. In the Load Balancer case we use the cache to maintain VIP entities locally. Only the first operation can be cached since it is the only read. For the write, we must rely on the data store to accurately perform the round-robin algorithm and return the address of the next server chosen. Otherwise, consistency problems may arise (i.e., conflicts). We make use of this mandatory access to the data store to evaluate the staleness of the VIP present in the cache. If the VIP changed between the time it was added to the application cache and the time the write is performed, then the data store aborts the operation and the application can restart from scratch. This time the value is obtained from the data store.

Device Manager. The Device Manager workload `dm-known-cache` was based on `dm-known-micro` from Table 4.4. Operation #1 reads the source and target devices based on the IP addresses present in the packet. If any of the two are not available in cache, the application fetches both from the data store. Since this operation is based on a micro component the implementation is trickier because the client (the Device Manager application) implements the logic to either fetch both values from the cache (source and target devices) or invoke the micro component (through the cache interface that knows the semantic of the reply and updates the cache with both the source and target devices). Also, notice that we rely on the second operation (the write updating the timestamp) to validate the cached data, but in our current implementation, this operation only validates the source device. If the cached

[6]For each switch table only a single thread, in a particular controller (the one responsible for the switch) reads and performs writes in the data store.

source device has been modified in the data store, the operation fails and the process must be repeated. If repeated then the first operation forcibly fetches values from the data store. Of course, this validity check could be expanded to include the destination device, thus narrowing the inconsistency window of all the cached information used to install flows.

Analysis. The last three columns of Table 4.4 show the results of the performance analysis considering the use of caching. The best case of each workload refers to when all the cache-enabled operations are performed locally. In contrast, the worst case refers to when all operations that compose the workload are performed in the data store. Of course, these values can only be used as a broad reference to understand the impact of caching. The true results may be far from the best case, since the frequency of cache-hits is dependent of the accepted staleness, the frequency of data plane events, the size of the cache, etc.

Regarding the results, in the `ls-ucast-cache` workload we show that the best case has an infinite throughput and zero latency since no operation is performed in the data store. These values merely mean that the throughput and latency are limited by the controller.

The best case results of the device manager and load balancer are very similar since they have identical workloads. This best case of each workload achieved a twice as high throughput when compared to the worst case. This was expected since the best case in each workload reduces the number of messages sent to the data store by half.

4.2.8 Related Work

The need for scalability and dependability has been a motivating factor for distribution and fault-tolerance in SDN control planes. We consider it therefore no surprise the most successful SDN controller to date to be Onix [30], the first distributed, dependable, production-level solution that considered these problems from the outset. As the choice of the "right" consistency model was also perceived as fundamental by its authors, Onix offered two data stores to maintain the network state: an eventually consistent and a strong consistent option. In terms of performance, the original consistent data store supported up to 50 SQL queries per second without batching. With batching (grouping more than one operation in a single request), the data store increased its performance to (only) 500 operations/s. The eventual consistent data store used in Onix could support 22 thousand "load attribute" updates/s, considering 5 replicas (33 thousand with 3 replicas). These values are equivalent to our data store, despite the fact that we support a strong, consistent view of the network state, contrary to Onix's eventual data store. Onix distributed capabilities and transactional data store have been subject of improvements recently, but not much information exists to date on its current performance [29]. The closed-source nature of Onix and lack of information prevents us from investigating it further.

Unlike Onix, ONOS [6] is an open-source solution based on an optimistic replication technique complemented by a background gossip protocol to ensure eventual consistency to manage the network state. The published performance results show that ONOS is able to achieve a throughput of up to 18.000 path installments per second in an experiment similar to ours. These values are on pair with the throughput results we obtained using a strongly consistent data store. Although our tests do not consider the overhead of the interaction between the data and control planes, the limiting factor of our architecture is the interaction with the data store, therefore we are assured to achieve a level of performance of the same order.

OpenDaylight [47] is another open, industry-backed project that aims to devise a distributed network controller framework. Similarly to ONOS, OpenDaylight runs on a cluster of servers for high availability, uses a distributed data store where it employs leader election algorithms. However, the OpenDaylight clustering architecture is still evolving and its performance is reported to be several orders or magnitude below our results (cf. the data store drop-test at [54]).

Recent work on SDN has explored the need for consistency at different levels. Network programming languages such as Frenetic [20] offer consistency when composing network policies (automatically solving inconsistencies across network applications' decisions). Other related line of work proposes

per-packet and per-flow abstractions to guarantee data-plane consistency during network configuration updates [39, 50]. Contrary to our work, the aim of these systems is to guarantee consistency *after* the policy decision is made by the network applications. In the same line of research, Software Transactional Networking (STN) [12] offers an abstraction that guarantees consistency on the data plane in a concurrent multi-controller scenario. STN is based on the assumption that controllers can perform read-modify-write atomic operations to the switches. As their solution requires each controller to communicate with all other controllers, the solution does not scale. Our architecture differentiates from these proposals by targeting the problem of consistency of the global network view. In other words, our concern is in guaranteeing consistency *before* the policy decisions are made by the (distributed) controllers. The solution is also fully distributed – and therefore scalable – with each controller communicating and controlling only the subset of switches of its domain. Despite the differences, this line of work on SDN consistency is complementary to our work. We believe that the combination of our solution with these mechanisms will enable an integrated solution that guarantees all packets (or flows) will always follow the same network policy and therefore avoid network anomalies in *any* scenario (centralized or distributed).

4.2.9 Conclusions

The introduction of distribution, fault tolerance and consistency in the SDN control plane has a cost. Adding fault tolerance increases the robustness of the system, while strong consistency facilitates application design. But it is undeniable: these mechanisms will affect system performance. By understanding and accepting the inevitability of this cost, our objective in this work was to show that, for network applications considered representative, this cost may be attainable and the overall performance of the system can remain under acceptable bounds.
In this section we proposed a distributed SDN control plane centered on a data store that offers strong consistency and fault tolerance for network (and applications) state. Our data-centric approach leads to a simple and modular architecture. As the bottleneck of our architecture is the data store, we have proposed a set of optimization techniques specifically tailored for SDN environment, achieving acceptable performance.

Chapter 5 Self-management of network security

Based on a short survey on security incidents and abuses affecting public cloud providers carried out in Deliverable D4.1, we concluded that the use of public cloud services by malicious software has been increasing for the last six years, although providers have attempted to limit the extent of these incidents. Additionally, we have observed that criminals are increasingly moving additional components of their malicious infrastructures into the cloud making attacks harder to correlate and characterize. Finally, our survey proved, to some extent, that cloud providers have not improved in detecting and handling incidents and abuse cases occurring in their infrastructure. In this chapter, we introduce a network security management for SUPERCLOUD that is able to address these observations, with the following characteristics: (a) user-centric security service composition allowing users to respond to the very different categories of attacks affecting their cloud services, as presented in Sect. 5.2; (b) security monitoring for users to detect malicious services and behaviors, such as a service being moved towards a different VM and trigger response, as presented in Sect. 5.3; (c) cross-layer and cross-provider feature and alert correlation in order to characterize ongoing attacks, as well as the state of the provider network, and policy management to decide how to react to the threat, as presented in Sect. 5.4. Such network security management will be able to achieve *autonomic security* of the SUPERCLOUD, as illustrated in Fig. 5.1.

5.1 Concepts and Requirements

As stated in Deliverable D4.1, an essential requirement of SUPERCLOUD is its ability to provide *autonomic security*, i.e., SUPERCLOUD should be able to monitor and react to security incidents automatically. This is motivated by the fact that, as SUPERCLOUD will leverage on heterogeneous network infrastructures, from both public cloud providers and private infrastructures, each provider (including both public and private ones) would have only a partial view of security incidents that may affect the SUPERCLOUD users. For example, an attack that leverages SUPERCLOUD resources may share the same artefacts across multiple heterogeneous providers. Therefore, it would be easier and more efficient to characterize this attack when correlating observations across all these providers. Besides, a comprehensive approach for attack remediation may require to coordinate actions such as VM quarantine or network migration across multiple cloud providers. Security incidents may be manifested in two different ways. On the one hand, SUPERCLOUD users may be affected by cyberattacks such as DDoS and malware infections. On the other hand, security incidents also include abuse of SUPERCLOUD resources that may occur when malicious SUPERCLOUD users resort to the heterogeneous multi-cloud environment to coordinate or to launch attacks against third parties.
To address such challenges, the autonomic security management framework offers data plane monitoring and proactive attack detection across the different cloud providers contributing to the SUPERCLOUD infrastructure. It enables to characterize and classify security incidents that may occur within the SUPERCLOUD, and implement customizable reaction policies that would segregate and isolate attack traffic.
In Deliverable D4.1, we have highlighted three important requirements to build such a framework:

Req. 1 Service Composition: *Since the security incidents that may affect public cloud services belong to very different categories of attacks, the users of the SUPERCLOUD should be able to compose*

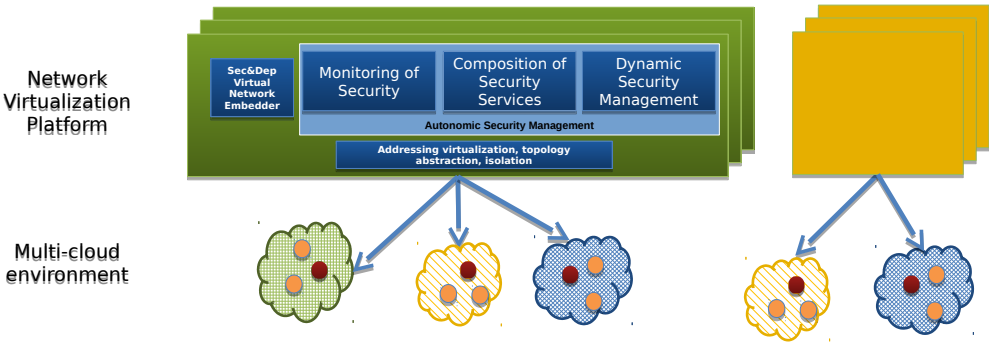


Figure 5.1: SUPERCLOUD Autonomic Security Management

the security services that are most appropriate according to the context and security requirements of their use cases that are hosted on the SUPERCLOUD.

Req. 2 Multi-provider cooperation: *Since attackers are moving additional components of their malicious infrastructures in the cloud, and when such components end up on different cloud providers, users of the SUPERCLOUD should be able to correlate features and alerts across multiple cloud providers to better characterize ongoing attacks and abuses.*

Req. 3 Migration Monitoring: *Since malicious servers tend to frequently migrate within the same address space in the cloud, users of the SUPERCLOUD should be provided with appropriate monitoring applications that enable them to monitor a service each time it migrates towards a different VM. Moreover, SUPERCLOUD users should be also provided with appropriate tools that enable them to promptly react upon detection of malicious services.*

To address these functional requirements, the autonomic security management framework of SUPERCLOUD will satisfy two design specifications, as proposed in Deliverable D4.1:

Spec. 1 *An appropriate SDN security control application will be provided on top of the network abstraction layer. It allows users to compose their own security services in the data plane, and to tune these services on a per-flow (e.g., web traffic) or per-destination (e.g., towards a database service) basis.*

Spec. 2 *Users will be able to define their own security management procedures that will be automatically triggered by the SUPERCLOUD in response to attacks and security incidents.*

5.2 Composition of Security Services

The security management component of the SUPERCLOUD network virtualization infrastructure enables users to select and compose their network security services on a per-flow or per-destination basis. In particular, it leverages the data plane topology and the placement of security services (both physical or virtual appliances) in order to compute and deploy routing policies that take into account multiple constraints such as the user-defined security services, cost, sovereignty, and the length of network paths. We implement this component as an SDN application that uses the northbound REST API for the Floodlight controller.
The security management component includes four main modules, as illustrated in Fig. 5.2.

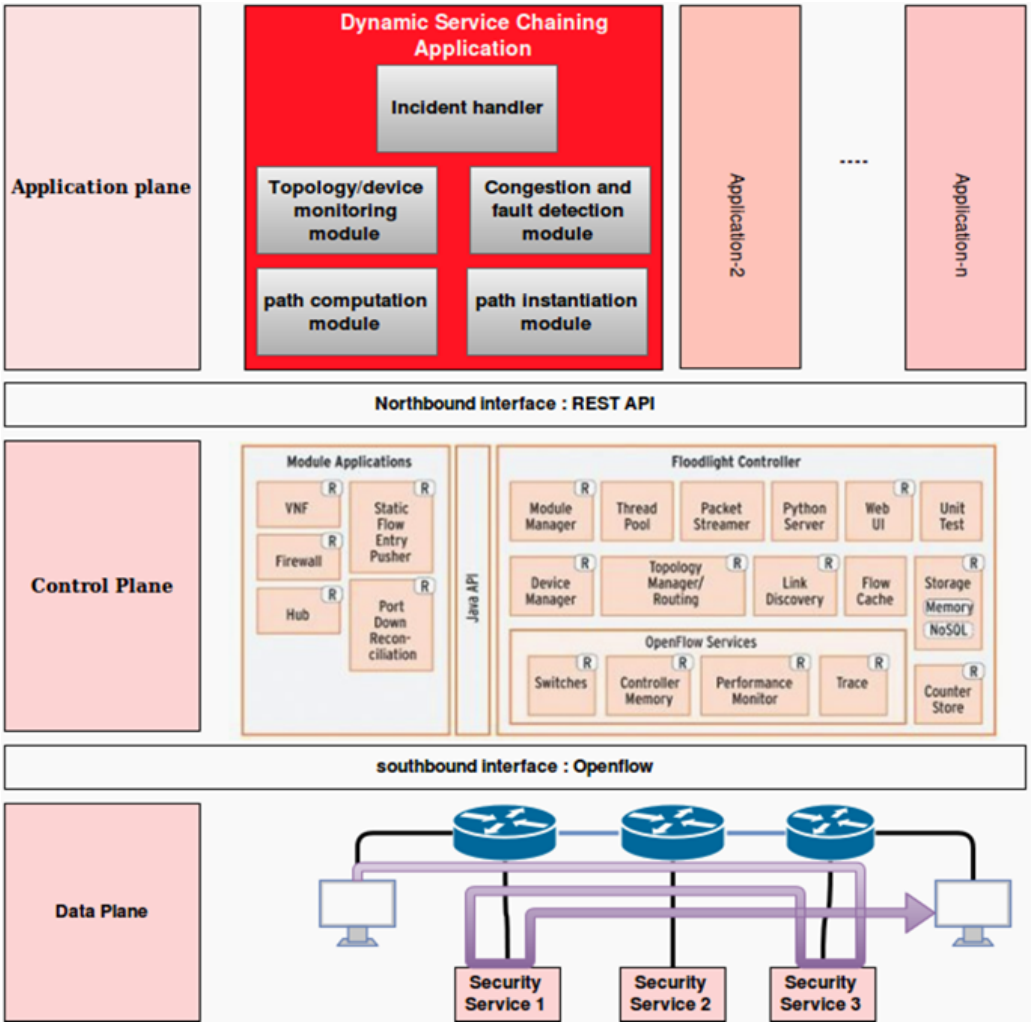


Figure 5.2: SUPERCLOUD Composition of Security Services Component

5.2.1 Topology management module

The topology management module uses the REST API provided by the Floodlight controller. It keeps a consistent graph-based representation of the network topology, which is further used as input to the path computation module.
The network topology is represented as an undirected labelled graph where nodes represent OpenFlow devices, and edges represent active network links. Link labels characterize the nominal capacity for each link. Network security services are also represented in this topology. They are characterized through service nodes being attached in the graph to their corresponding OpenFlow device. Note that the topology management module also tracks host migrations, device or link outages that may have an impact on the routing paths established by the path computation module.

5.2.2 Path computation module

The path computation module computes optimal routing strategies taking into account both user-defined constraints (e.g. security, QoS) and the routing and deployment cost. It implements a mathematical optimization model that uses as input a graph-based representation of network topology provided by the topology management module. It computes an optimal routing policy, taking into account multiple user-defined constraints, as follows.

Security services chaining: The security management component offers the ability for SUPERCLOUD users to select and compose network security services and attach these services to specific network flows. Users may constitute service chains that are considered as input to the path computation module.

For example, a SUPERCLOUD user may need to set up a public web application, and would like to subscribe to a Web Application Firewall (WAF) service and a Data Leak Protection (DLP) service offered by the SUPERCLOUD. All network flows originating from the Internet towards the public web application need to be routed by the SUPERCLOUD through the WAF and DLP services, both implemented as third-party appliances available as part of the network virtualization infrastructure. The path computation module uses such service chaining constraints for all network flows and computes optimal routing paths for each network flow, taking into account the set of security services that are attached to this flow. This module also manages compatibility issues between security services, e.g., placing the DLP service behind the WAF service to avoid redundant alerts and false positives.

Link capacity: The path computation module avoids link congestion by taking into account the nominal capacity of network links when computing the optimal routing policies. Link capacity is both used by the optimization model at the route setup phase, and further used by the congestion and fault avoidance module in order to reroute traffic in case of congestion and link or device failures.

Service capacity: The path computation module also takes into account the nominal capacity of security services available in the data plane. It makes sure that a given security service is not supplied with network traffic exceeding its nominal capacity.

Sovereignty: The SUPERCLOUD network virtualization infrastructure is built on top of multiple distributed cloud providers. SUPERCLOUD users may constrain the SUPERCLOUD provider to use, or to avoid certain cloud providers for hosting their applications. Moreover, SUPERCLOUD users may have specific constraints depending on their environment and their data hosted on the SUPERCLOUD. Legal constraints include network traffic not to be routed through specific cloud providers, yet also outside specific countries or geographic area (e.g. traffic should stay within Europe). The optimization model implemented by the path computation module enables to take such sovereignty requirements into account and computes routing paths that strictly abide by these constraints.

Cost: The path computation module also optimizes routing costs when the bandwidth cost is not the same for all underlying cloud providers. More specifically, cloud providers may have different pricing policies, including different per-MB rates for traffic routed across their infrastructure. The path computation module takes these costs into account for computing optimal routing policies, aiming to reduce deployment costs both for the SUPERCLOUD provider and for end users.

5.2.3 Path instantiation

This module translates and instantiates the optimal routing strategy using low-level OpenFlow commands. For each network flow, it instantiates network paths using a list of network devices and security services that are attached to this flow. It uses for this purpose the Floodlight native circuit pusher application. This application utilizes the Floodlight REST APIs to create a bidirectional circuit, i.e., permanent flow entry, on all switches in route between two devices based on IP addresses with a specified priority.

5.2.4 Congestion and fault avoidance

This module interfaces with the native Floodlight *topology monitor* module. It detects changes in the network topology, which may be attributed to host migrations, link or device failures, and network congestions. Upon the detection of a fault or a topology change, this module behaves as follows.
First, when the fault does not concern any device or network link that are involved in established security services chains (see Sect. 5.2.2), this module only notifies the topology management module to update the current topology and take this fault into consideration for further optimization tasks.
Alternatively, the fault may concern some already established security services chains. The congestion and fault avoidance module iterates over all installed circuits that use the faulty link or device, and computes alternative paths that satisfy the user-defined constraints (see Sect. 5.2.2). When another optimal route may be found, this module removes the old circuit and installs the new circuit over the new optimal path. When no other route alternative that meets the user-defined constraints could be identified, this module notifies the security monitoring component (see Sect. 5.3), and hands over the faulty circuit to the native forwarding module of the Floodlight controller.

5.3 Monitoring of Security

The security monitoring component of the SUPERCLOUD network virtualization infrastructure enables users to define and manage their security contexts based on security incidents and OpenFlow statistics collected from the data plane. We implement the security monitoring component as an application module within the Floodlight controller. The architecture of the security monitoring component is illustrated in figure 5.3, and includes five main modules.

5.3.1 Topology handler

The topology handler collects OpenFlow statistics from network devices and provides the security monitoring component with a near real-time view of the data plane topology. To do so, it constantly polls the network devices using a native Floodlight service that enables to acquire network statistics from the managed OpenFlow devices. Raw network data collected by the topology handler is further processed in order to provide contextual topology information to the context handler module. For example, raw `Rx` and `Tx` fields are used to compute the usage rate and bandwidth of a given network link. The topology handler also offers a notification registration service that may be customized in such a way that the context handler can be notified only for specific topology changes that are relevant to the context of a specific SUPERCLOUD user.

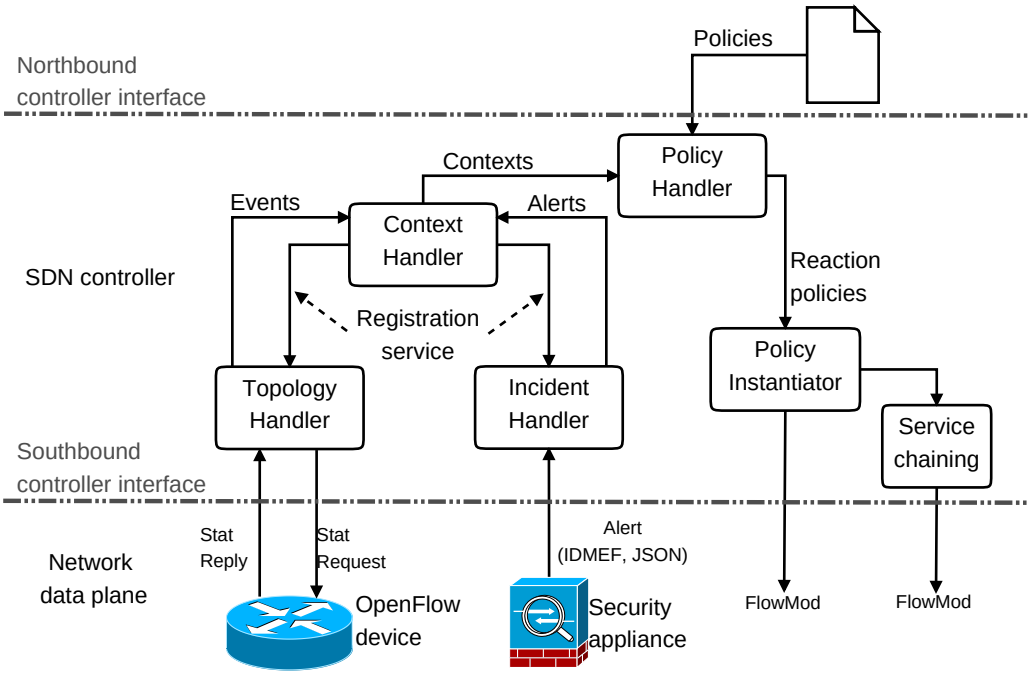


Figure 5.3: SUPERCLOUD Network Security Monitoring Component

5.3.2 Incident handler

The incident handler enables to collect alerts and incidents from security appliances that are available on the network data plane. It is implemented as an internal module of the Floodlight controller, and extends the controller ability to collect security alerts that are not natively supported by OpenFlow. In particular, it offers a REST API that is used by security appliances to notify the controller about security incidents that occur in the data plane.
The incident handler supports two standard formats for security alerts: IDMEF and JSON. Real-time alerts are pushed to the incident handler through POST requests. When a new alert is pushed to the incident handler, a parser plug-in extracts relevant alert data and further sends this data to the context handler module. The incident handler also offers a notification registration service that may be customized in such a way that the context handler can be notified only for specific incidents that are relevant to the context of a specific SUPERCLOUD user.

5.3.3 Context handler

The context handler manages all user-defined contexts and keeps an updated list of all security contexts that are activated for a given user. These contexts may be further associated to specific reaction policies, which may be triggered by the policy handler upon activation of the appropriate contexts. The context handler manages either or both security-related contexts such as attacks being detected by the incident handler, or topology-related contexts such as a link outages or the migration of a given terminal.
For example, a DDoS context may be defined as follows. It indicates that the DDoS context should be activated both when a DDoS alert is pushed to the controller, and when the target link experiences a packet drop rate exceeding 90%.

$$DefContext\ DDoS : (Alert.Classification, "DDoS") \ and \ (Link.drop\_rate, "0.9")$$

The context handler offers an API that allows SUPERCLOUD users to define their own contexts and to associate these with specific incidents or topology changes. Therefore, a context is defined as a combination (either conjunctive or disjunctive) of multiple conditions that may apply to relevant

alert data or topology information. The context handler constantly monitors the conditions that are appropriate for a given context. It updates the list of activated contexts each time it is notified by the incident handler or the topology handler about new incidents and topology changes, and provides an updated list of all activated contexts as input to the policy handler.

5.3.4 Policy handler

The policy handler manages the reaction policies on behalf of the SUPERCLOUD users. It enables the users of the SUPERCLOUD to customize their own reaction policies in response to dynamic network incidents and attacks. Network policies in the context of SUPERCLOUD are context-based. Thus, a reaction policy is triggered only when the appropriate context had been activated. Therefore, the policy handler is constantly provided with the list of all active contexts that are triggered by the context handler. It instantiates the reaction policies that are associated with a given context, and further notifies the policy instantiator module for reaction enforcement.

5.3.5 Policy instantiator

The policy instantiator module supports two sets of actions. First, it may act directly on OpenFlow devices, using for example FlowMod commands, in order to set routing policies such as for example to reroute traffic through a given security device, quarantine a terminal, or disable a network link. Second, it may also compose dynamic chains of security services that are specified by the SUPERCLOUD users in response to specific security contexts.

5.4 Dynamic security management

One of the objectives of the SUPERCLOUD project is to build a resilient system that self-adapts to both adversary threats and network and system failures. Self-healing mechanisms such as proactive and reactive recovery may be useful to ensure the perpetual and unattended operation of the multi-cloud environment, and will therefore be also subject to consideration in the design of our solution. A policy-based approach is the adequate solution for the management of a multi-cloud environment as it enables adaptability to dynamic changes on the network and security levels. It allows as well the application of policy rules to the heterogeneous components of the monitoring and network planes. Finally, the high level of abstraction of the policy model eases the modelling and management of complex security policies.

5.4.1 Framework architecture

Figure 5.4 illustrates the main components of the network security management:

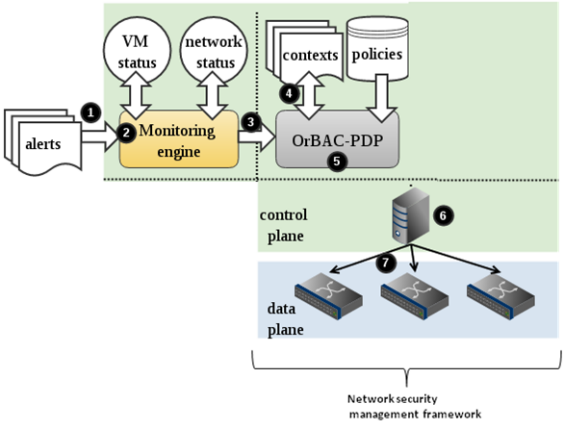


Figure 5.4: Network security management hypervisor.

- The Policy Decision Point (PDP) is a decision entity. It is the architecture engine that will receive the requests and the notifications. Based on them, it will activate and deactivate the adequate policies and rules.
- The Policy Enforcement Point (PEP) is the entity executing the configurations implementing the current policy. PEPs generally comprise the OpenFlow switches and the SDN controller. The PEP will enforce automatically the selected policy by the PDP using the OpenFlow scripts.

5.4.2 Workflow description

In order to achieve a user-centric management of network security, SUPERCLOUD proposes to provide an SDN security control application on top of the network abstraction layer. The proposed framework integrates the PDP as an application of the controller. The PDP provides an abstraction that allows to model all security services available in the data plane (e.g., IDS, proxies, honeypots, anti-DDoS devices). Thanks to the OrBAC policy model, the PDP dynamically manages the routing policies based on information collected by the monitoring component, as well as, requirements expressed by the SUPERCLOUD users. For example, as a path is allocated to the service of a given application, our proposed framework is able to fulfil topological requirements in order to build a resilient system that self-adapts in the face of not only potential threats, but also in the case of network or system failures.
Figure 5.4 illustrates the workflow of our approach. The monitoring component, as described in Sect. 5.3, continuously supervises the data plane (2) to detect any failures, be it system or network-based. This component updates the virtual machines' status, as well as the network status, based on alerts received from local or external sources (1). When resources are critically damaged (failure, error, etc.), the OrBAC-PDP is notified (3). This policy engine activates contexts based on the status of the network or the virtual machines (4). Appropriate remediation policies are then selected to be pushed to the data plane in order to preserve the availability of the targeted services (5). The chosen policy will be mapped to one or more OpenFlow scripts (6). Finally, the controller will deploy these scripts to the SDN switches by pushing or deleting some rules (7).
This framework is being implemented along the two network services presented in the following section.

5.4.3 Network security services

The network security self-management component will orchestrate the provisioning, deployment and support of a number of services to accommodate tenants' needs, with respect to agreed SLAs, and to defend against common threats, both external and internal. In this section, we describe some of the foreseen services offered by the security component, including availability management and incident management and mitigation.

5.4.3.1 Availability management

Availability today is an important issue in a multi-cloud environment. According to a study commissioned by Global Switch, IT service interruptions cost 440,000 euros per hour for companies in Europe. Therefore, the definition of a strategy to limit failures of a service is compulsory. In SUPERCLOUD, we aim to define an SDN strategy to solve this issue. Indeed, the use of SDN technology, that permits to dynamically program a network reaction, will be extremely useful. In the following, we detail two possible availability services that may be deployed in the SUPERCLOUD project.

- The first service (see Figure 5.5) leverages replication through a replicated server with degraded quality of service (QoS). In this scenario, the provider pledges to preserve the availability of the service without necessarily maintaining the quality of the network. The replica is deployed to receive live copies of user requests, but does not honour them as long as the main server is

running. However, if the connectivity to the main server is lost or get degraded, the replica takes over the main server and sees its QoS increased to act as the main service.

During the whole life-cycle of the service, two servers are kept alive, the main server, with the requested QoS and the backup or replica, with degraded QoS. The role of the availability policy is to manage the performance of the networks and to redirect the flows to the replica with the modification of its QoS in case of a connection problem with the main server. In other words, even if the main server cannot provide the required QoS, new paths are provided with the needed performance for the replica to guarantee the requested QoS.

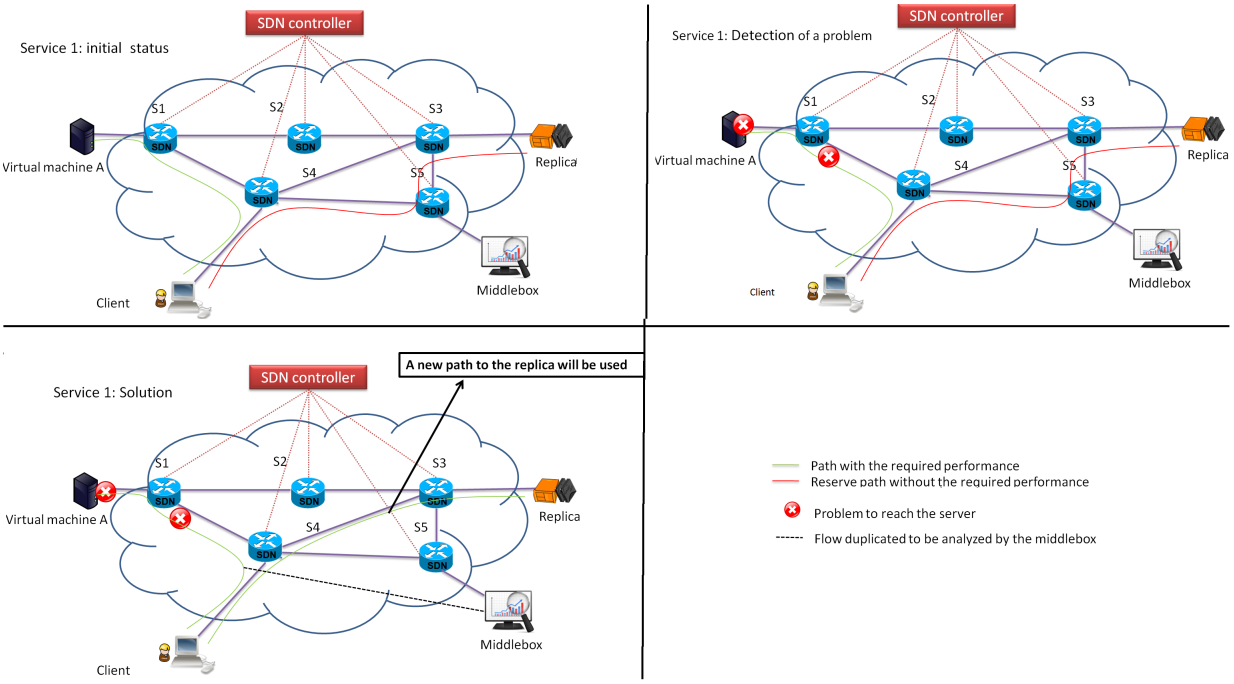


Figure 5.5: Possible status of the first availability service

- The second service (see Figure 5.6) proposes load balancing on demand. At the beginning, only one server provides a service to the customers. Next, based on the network status, a second server is provided to achieve the required QoS. This policy provides network resilience for the customer's service by categorizing the network status into three different states. The NORMAL state, where the service can answer requests properly and does not suffer from bandwidth nor computation power shortage. The MEDIUM state corresponds to a service with several simultaneous connections or less than 50 % of its available bandwidth remaining. Finally, the CRITICAL state characterizes an offline system. In this scenario, the second machine is solicited depending on the state of the main machine. As we defined previously the different states that represent the network status, our solution acts according to the following algorithm:

Algorithm 1 Service Backup Algorithm

if $Status = NORMAL$ **then**
  REDIRECT_TRAFFIC_TO(Main_Server)
else if $System\_Status = MEDIUM$ **then**
  SET_NEW_CONNECTIONS_TO(Second_Server)
else$System\_Status = CRITICAL$
  INCREASE_NETWORK_PERFORMANCE(Backup_Server)
  REDIRECT_TRAFFIC_TO(Backup_Server)

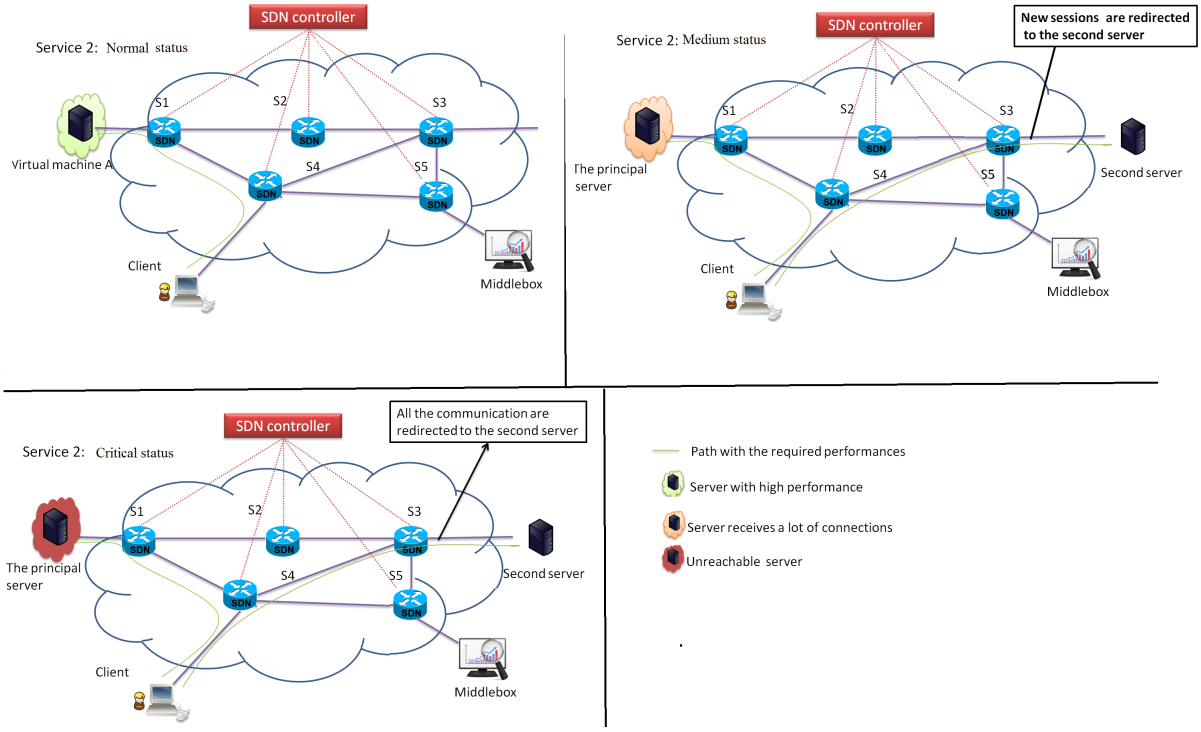


Figure 5.6: Possible status of the second availability service

This service has been implemented into a physical testbed featuring a number of OpenFlow switches and hardware hosts to emulate a data center network. We are actually running a number of experimentations to evaluate the framework and this service.

5.4.3.2 Incident management and mitigation

Cyber-attacks cause considerable losses not only for end-users but also Cloud Providers (CPs). They are fostered by myriads of infected resources and mostly rely on network resources for whether propagating, controlling or damaging. There is an essential need to address these numerous attacks by efficient defence strategies. These solutions involve a detection process, as detailed in Sect. 5.3, completed by mitigation actions.
In this part, we will describe our second network service. It will be complementary to the monitoring engine. The lessons learned from the past attack mitigation designs clearly indicate that the following requirements deserve careful consideration:

- Easy deployment and operation, avoiding to use special purpose software or hardware devices.
- Effective information exchange between the different parties involved (i.e., the different CPs and the customers).
- Timely handling of alert notifications and enforcement of appropriate attack mitigation policies.
- Management of the entire life-cycle of the attack mitigation scheme in a scalable, adaptive, and automated way, involving only trivial manual effort.

Unfortunately, the intrinsic characteristics (for example, tight coupling between control plane and data plane) of today's legacy networking infrastructure make a majority of attack mitigation schemes fail to meet the given requirements. Thanks to the recent development of SDN technologies, we are given a promising opportunity to re-examine such designs requirements [1,2], by taking advantage of the clear separation of network control plane and data plane, as well as the programmability of SDN controllers.
For the SUPERCLOUD project, we aim to design a service respecting these properties:

- Optimization: With an overall vision of the network infrastructure, reaction policies can now be optimized to reduce their impacts on other equipments.
- Simple dynamicity: The dynamic deployment of security policies is performed thanks to SDN, with little to no human configuration.
- Automation: Our framework will provide an automatic mitigation process based on the deployed policy.

Some technical challenges should be tackled in order to achieve our objectives:

- Scalable flow management: the CP core switches require a large number of flow entries for forwarding. The reactive nature of the SDN technology makes it difficult for an CP to provide attack mitigation as a service to their customers. Every time an OpenFlow switch receives a flow for which it does not have the entry in its flow table it forwards that flow to the SDN controller to get the policy for forwarding that flow. Since it causes processing overhead on the SDN controller so it becomes difficult for the CP to provide attack mitigation as a service to its customer.
- Preserving the consistency of forwarding policy: NAT devices, which are widely used in networks, involve packet header information modification. Such alteration may have an impact on how flow packets are forwarded, violating the CP network's forwarding policy for these flows.

Regarding implementation, this service is actively being specified. The service will be implemented and integrated in the SUPERCLOUD network hypervisor in the next phase.

Chapter 6 Conclusions

In this deliverable we have presented the architecture of the SUPERCLOUD network virtualization platform:

- We have started by an overview of the architecture, including the specification of its design requirements, a high level view of all its components, and a preliminary evaluation of its basic funcionality.
- Then, we presented the main components of the network hypervisor: the modules for address virtualization, topology abstraction, and isolation; and secure and dependable virtual network embedding.
- Third, we presented the design and implementation of a fault-tolerant and a distributed controller to guarantee that the platform responds to the resilience and scalability requirements.
- Finally, we detailed the autonomic security management services for the virtualization platform.

As future work we will integrate all components of the proposed architecture and evaluate its overall functionality and performance.

Glossary

AWS Amazon Web Services.

EC2 Elastic Compute Cloud.

GRE Generic Routing Encapsulation.

MILP Mixed Integer Linear Program.

MST Minimum Spanning Tree.

OVS Open vSwitch.

SDN Software-Defined Networking.

TCAM Ternary Content-Addressable Memory.

TPM Trusted Platform Module.

VLAN Virtual Local Area Network.

VM Virtual Network.

VN Virtual Network.

VNE Virtual Network Embedding.

Bibliography

[1] Vine-yard.
`http://www.mosharaf.com/ViNE-Yard.tar.gz`.

[2] Fischer A. et al. Virtual network embedding: A survey. *IEEE Communications Surveys and Tutorials*, 15(4):1888–1906, Fourth 2013.

[3] A. Al-Shabibi et al. OpenVirteX: Make your virtual SDNs programmable. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, 2014.

[4] M. Alaluna, F. M. V. Ramos, and N. Neves. (literally) above the clouds: Virtualizing the network over multiple clouds. In *2016 IEEE NetSoft Conference (NetSoft)*, June 2016.

[5] M Faizul Bari, Arup Raton Roy, Shubhajit Roy Chowdhury, Qi Zhang, Mohamed Faten Zhani, Rizwan Ahmed, and Raouf Boutaba. Dynamic controller provisioning in software defined networks. In *Network and Service Management (CNSM), 2013 9th International Conference on*, 2013.

[6] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, 2014.

[7] Alysson Bessani, Marcel Santos, Joao Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proc. of the USENIX Annual Technical Conference (ATC 2013)*, June 2013.

[8] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362, 2014.

[9] F. Botelho, T. A. Ribeiro, P. Ferreira, F. M. V. Ramos, and Alysson Bessani. Design and implementation of a consistent datastore for a distributed sdn control plane. In *The 12th European Conference on Dependable Computing (EDCC'16)*, September 2016.

[10] Fabio Botelho, Alysson Bessani, Fernando M. V. Ramos, and Paulo Ferreira. On the design of practical fault-tolerant SDN controllers. In *Third European Workshop on Software Defined Networks*, 2014.

[11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer, 2nd edition. edition, February 2011.

[12] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proceedings of the IEEE INFOCOM*, INFOCOM '15, 2015.

[13] Mark Cavage. There's just no getting around it: You're building a distributed system. *Queue*, 11(4), April 2013.

[14] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Trans. Netw.*, 20(1), February 2012.

[15] N. M. M. K. Chowdhury et al. Virtual network embedding with coordinated node and link mapping. In *INFOCOM 2009, IEEE*, 2009.

[16] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proc. of the 10th ACM European Systems Conference – EuroSys'15*, April 2015.

[17] D. Drutskoy, E. Keller, and J. Rexford. Scalable network virtualization in software-defined networks. *Internet Computing, IEEE*, 17(2):20–27, March 2013.

[18] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Kobus van der Merwe. The Case for Separating Routing from Routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, September 2004.

[19] Andreas Fischer and Hermann De Meer. Position paper: Secure virtual network embedding. *PIK-Praxis der Informationsverarbeitung und Kommunikation*, 34(4):190–193, 2011.

[20] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, 2011.

[21] GLPK. Gnu linear programming kit. `http://www.gnu.org/software/glpk/`, 2008.

[22] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[23] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, 2012.

[24] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[25] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, 2010.

[26] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, 2013.

[27] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015.

[28] Hyojoon Kim, Mike Schlansker, Jose Renato Santos, Jean Tourrilhes, Yoshio Turner, and Nick Feamster. Coronet: Fault tolerance for software defined networks. In *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*, ICNP '12, 2012.

[29] T. Koponen et al. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, 2014.

[30] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[31] D. Kreutz, F. M. V. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, HotSDN '13, 2013.

[32] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C.E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: a comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, January 2015.

[33] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), May 1998.

[34] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized? state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, 2012.

[35] A. Li et al. Cloudcmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, 2010.

[36] Barbara Liskov. From viewstamped replication to byzantine fault tolerance. In Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 121–149. Springer Berlin Heidelberg, 2010.

[37] S. Liu, Z. Cai, H. Xu, and M. Xu. Security-aware virtual network embedding. In *2014 IEEE International Conference on Communications (ICC)*, 2014.

[38] R. Los, D. Shackleford, and B. Sullivan. The notorious nine cloud computing top threats in 2013. In *Cloud Security Alliance*, 2013.

[39] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, 2015.

[40] N. McKeown et al. OpenFlow: enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[41] L. Nonde, T. E. H. El-Gorashi, and J. M. H. Elmirghani. Energy efficient virtual network embedding for cloud networks. *Journal of Lightwave Technology*, 33(9):1828–1849, May 2015.

[42] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[43] Open Network Foundation. OpenFlow Switch Specification (version 1.2) [opennetworking.org], December 2011.

[44] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047 (Informational), December 2013.

[45] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, 2015.

[46] Floodlight Project. Floodlight project `http://www.projectfloodlight.org/`.

[47] OpenDaylight Project. Opendaylight project `http://www.opendaylight.org/`.

[48] Muntasir Raihan Rahman, Issam Aib, and Raouf Boutaba. Survivable virtual network embedding. In *International Conference on Research in Networking*, 2010.

[49] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[50] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, series = SIGCOMM '12, title = Abstractions for network update, year = 2012.*

[51] Francisco Javier Ros and Pedro Miguel Ruiz. Five nines of southbound reliability in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[52] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), December 1990.

[53] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[54] OpenDaylight Performance Tests. Opendaylight performance tests `https://wiki.opendaylight.org/view/CrossProject:Integration_Group:Performance_Tests#Helium_CBench_Results`.

[55] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2012.

[56] Wai-Leong Yeow, Cédric Westphal, and Ulaş Kozat. Designing and embedding reliable virtual infrastructures. In *Proceedings of the Second ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, 2010.

[57] H. Yu, V. Anand, C. Qiao, and G. Sun. Cost efficient design of survivable virtual infrastructure to recover from facility node failures. In *2011 IEEE International Conference on Communications (ICC)*, 2011.

[58] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008.

[59] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, 1996.

[60] L. Zheng et al. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.