

# Monitoring Service Chains in the Cloud

Emad Nikkhoy

M.Sc. Thesis  
UNIVERSITY OF HELSINKI  
Department of Computer Science

Helsinki, May 21, 2016

Tiedekunta — Fakultet — Faculty		Laitos — Institution — Department	
Faculty of Science		Department of Computer Science	
Tekijä — Författare — Author			
Emad Nikkhoy			
Työn nimi — Arbetets titel — Title			
Monitoring Service Chains in the Cloud			
Oppiaine — Läroämne — Subject			
Computer Science			
Työn laji — Arbetets art — Level		Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis		May 21, 2016	82
Tiivistelmä — Referat — Abstract			
<p>Service chaining in the cloud is a new trend that network operators are moving towards. Service chaining in the cloud, is the process of virtualizing various services in the cloud instances and linking them together in order to create a chain of services. Aside from the benefits that it provides for the subscribers and network operators, it needs further considerations to be fully applied and utilized. Since service availability is a key concern for network provider and operators, the availability of service chain requires careful attention.</p> <p>The goal of this thesis work is to investigate how to monitor service functions that form the service chain in the cloud. By monitoring the service functions we aim to inspect the running services and report occurrence of abnormality (i.e. heavy work load), to our main monitoring platform, in order to trigger corresponding operations. We believe with monitoring the services we can increase their availability with low overhead. Our main contribution in this work, is to build a platform that can control and monitor the services in the cloud in order to enhance their availability.</p>			
Avainsanat — Nyckelord — Keywords			
service chaining, SDN, cloud, monitoring, OpenStack, OpenDaylight, Docker			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Service Chaining in the Cloud</b>	<b>4</b>
2.1	Software Defined Networking (SDN) . . . . .	4
2.2	OpenFlow . . . . .	6
2.3	Network Function Virtualization (NFV) . . . . .	9
2.4	Service Chaining . . . . .	9
2.5	Service Chaining Challenges . . . . .	12
2.6	Cloud (OpenStack) . . . . .	14
2.7	Linux Containers (Docker) . . . . .	17
<b>3</b>	<b>Service Function Chaining Methodologies</b>	<b>21</b>
3.1	Background and Motivation . . . . .	21
3.2	Service Chaining Methods . . . . .	22
3.2.1	Ericsson: StEERING for Inline Service Chaining . . .	22
3.2.2	Ericsson: Optical Service Chaining . . . . .	27
3.2.3	Huawei: Service Chaining with Service Based Routing	29
3.2.4	FCSC: Function-Centric Service Chaining . . . . .	31
3.3	Service Chaining Methods Comparison . . . . .	33
3.4	Monitoring Cloud: Methods . . . . .	34
3.4.1	PCMONS: Private Cloud Monitoring System . . . . .	34
3.4.2	Elastack: Automatic Elasticity in the OpenStack . . .	36
3.5	Monitoring Cloud: Tools (General Purpose) . . . . .	39
3.5.1	Nagios . . . . .	39
3.5.2	Collectd . . . . .	40
3.5.3	Ganglia . . . . .	41
3.6	Monitoring Cloud: Tools (Cloud Specific) . . . . .	42
3.6.1	Amazon CloudWatch . . . . .	42
3.6.2	Azure Watch . . . . .	42
3.6.3	Nimsoft . . . . .	43
3.7	Cloud Monitoring Tools Comparison . . . . .	43
<b>4</b>	<b>Design for Monitoring Service Chain</b>	<b>45</b>
4.1	Monitoring cloud . . . . .	47

4.2	Monitoring Services and OpenFlow Controller . . . . .	49
4.2.1	Monitoring Containers through Control Groups (cgroups)	50
4.2.2	Monitoring OpenFlow Controller and its Statistics . .	53
<b>5</b>	<b>Monitoring System Implementation and Evaluation</b>	<b>55</b>
5.1	System Implementation . . . . .	55
5.1.1	Software and Hardware . . . . .	56
5.1.2	RAM and CPU calculation . . . . .	56
5.1.3	Communication Architecture and Messages . . . . .	57
5.2	Monitoring Service Chain in Action . . . . .	59
5.2.1	Service chain with Squid proxy . . . . .	60
5.2.2	Main Monitoring System . . . . .	63
5.3	Evaluation . . . . .	69
5.3.1	Overhead . . . . .	69
5.3.2	Monitoring System Response Time . . . . .	72
<b>6</b>	<b>Conclusion and Future Work</b>	<b>76</b>
	<b>References</b>	<b>78</b>

# 1 Introduction

Software Defined Networking (SDN) has brought managing and designing networks into a new era during the past few years. Private networking companies have kept their devices and the firmware that comes along with it proprietary, which makes networking devices closed source [FRZ13]. Making networking devices proprietary has some drawbacks. First, it will make each device to has its own interface for configuration depending on the manufacturing company, second, it will prevent network services to be elastic. Elasticity here conveys the meaning of hiring a middlebox service for instance deep packet inspection (DPI) and exempt it when its usage is unneeded. Currently the network infrastructures and devices are tightly coupled, which makes troubleshooting and removing or introducing new middleboxes tedious and time consuming.

However, networking and services along with it are evolving and entering a new phase. With the help of Network Function Virtualization (NFV), physical network devices are transforming to virtualized softwares that are basically same as the physical device, with the difference that they can run on any hardware node in a cloud or data center or even users premises. With running different services as a virtualized software, cloud is a good candidate to be the base and host due to its flexibility. In this case each service or middlebox can be running on a virtual machine or linux container and the number of these services can be increased or shrinked whenever is necessary. However reliability of cloud and each of these virtualized services is playing an important role in quality of service and user experience. [CDSI<sup>+</sup>14]

In theory transforming physical network to virtual seems to be easy and straightforward. However in practice there are many issues and obstacles that need to be considered beforehand. Hardware faults, software faults and operator faults are some of the issues that can degrade the quality of network or even break the communication. Due to necessity of reliability in the network middleboxes and services in the cloud, there are many projects that have been working on the cloud's reliability. Prefail [JGS11] is a programmable tool that injects multiple faults to the cloud in order to analyze cloud's behaviour during crisis. Netflix is working on set of tools called The Simian Army [Tse13] (i.e. chaos monkeys, chaos gorilla, chaos kong and latency monkey), which inject different kinds of faults in the cloud

computing platform in order to assess its resiliency.

Internet traffic has been escalated in the last decade, and predictions shows that in the next decade mobile traffic will be increased by 1000 times [LNPW14]. Therefore, by that time 4G, which is used by many users today will not be sufficient enough to satisfy their needs. Researchers are working on 5th Generation (5G) of mobile internet which will be having larger bandwidth, less delay, and better quality of services for large data size (such as high definition (HD) videos), in order to satisfy day to day need of users. Since network functionality through the aid of SDN is moving from hardware to software, mobile network service infrastructure can shift from monolithic to more elastic, which 5G can benefit from it [AP15]. In this approach while services are running on general purpose hardware devices, then expanding or reducing services can be easier.

Service chain in SDN is related to higher level, where before chaining services we have to solve the lower and more fundamental issues. Previously we have mentioned that services can easily expand or shrink on demand due to the nature of NFV and SDN. However, before even expanding or shrinking services, we have to be able to link these services together spontaneously. On the bigger perspective, we have to link these virtual machines or linux containers that are running our services. Linux containers are isolated linux processes that operate on top of linux kernel and their functionality is similar to virtual machines. There has been tremendous amount of research going on to solve the problem of elastic service chaining. Huawei [Hua13], Ericsson [ZBB<sup>+</sup>13], Nokia [Nok15], and many other companies are trying to implement elastic service chain in their network infrastructure. Ericsson is also one of key contributors of Service Function Chaining (SFC) project in OpenDaylight [Odl16], which is an open source SDN platform.

Therefore, by monitoring service chain in the cloud we monitor the base for the services that are operating, including controller, network, compute nodes, and the services running inside the compute nodes. While monitoring this base for services, there are many issues that needs to be addressed. Traffic steering is one of the issues that needs to be considered. By traffic steering, traffic is enforced to take a specific route which is defined by a central controller to meet certain requirements, for instance inspecting the packets before sending them to their destination. Another issue, is the sudden failure of an instance in the cloud, where traffic was enforced to pass through

it, however after the instance failure that is not possible anymore. In this case another copy of the same instance should be started and the incoming traffic should be enforced to pass through it, which requires sudden changes of the route.

To address these issues, we design a new platform that can monitor the foundation of service chain, and monitor the cloud nodes and services running inside it from preventing any catastrophe to happen. For the cloud side, we use OpenStack, and instead of Virtual Machines we use linux containers for the cloud instances, specifically Docker [Doc15] due to its agility. Furthermore, to control the traffic, we utilize OpenDaylight, which is an open source SDN platform. The research methodology in this thesis is driven by measurement. We measure the resource usage (i.e CPU, RAM, network) of cloud components. Measuring the network performance in the cloud and number of packet drops via OpenDaylight can be beneficial in order to alert the main controller in case any of cloud services are getting saturated. The goal for the developed platform is to be robust and reliable in case of service saturation, because in the service chain if one node goes down, the whole chain will be broken.

The rest of this thesis is organized as follows. Chapter 2 describes some fundamental concepts about SDN, NFV, and SFC and also introduces some existing SDN and SFC platforms along with brief explanation of cloud. Chapter 3 illustrates an overview and related works in SDN, Service Function Chaining and cloud monitoring tools and methods. The design architecture of the service chain monitoring platform is explained in chapter 4. Chapter 5 depicts implementation details of developed platform along with measurement results obtained from various studies. Finally, chapter 6 concludes this thesis.

## 2 Service Chaining in the Cloud

In this chapter we discuss the basics of Software Defined Networking (SDN), OpenFlow, Network Function Virtualization (NFV), OpenStack and Linux containers, specifically Docker. In addition, we introduce the well known open source cloud platforms, and compare them.

### 2.1 Software Defined Networking (SDN)

Over the past 20 years, many technologies and their underlying infrastructure have been changed. Early telephony networks had separated data and control plane, which SDN somehow is revisiting that idea in order to facilitate deployment of new services and network management. Moreover, SDN is applying the past research which articulated perspective of programmable networks with affirmation on programmable data planes. [FRZ13]

The evolution which has happened to the networking throughout the past 20 years, can be divided into 3 stages. First stage is active networks that took place from mid-1990s to the early 2000s, which led to greater innovations by introducing network programming functions. Second stage, took place between 2001 to 2007, which open interfaces were developed during this stage and the outcome was separation of data and control plane. Third stage was from 2007 to around 2010, which during that time with the creation of OpenFlow protocol, separation of data and control plane became practical and feasible. Network virtualization, which is going to be discussed in the next section also played an important role throughout the history of SDN. Figure 1 illustrates selected development of programmable networking over the past 20 years.

Conventional networks are not programmable, however active networking community introduced an approach to the network control, which by exposing resources (such as packet queues, processing, storage) on each individual network node, different kinds of functionality could be applied on a subset of packets that are passing through that node. Though, this approach was not pleasant to many Internet community members and defendant of simplicity in the network core, which is the reason that Internet to be successful. Therefore, active network community pursued for other approaches. Capsule model and programmable router/switch model, are two models that active network community focused on. In capsule model, the code which needs to be



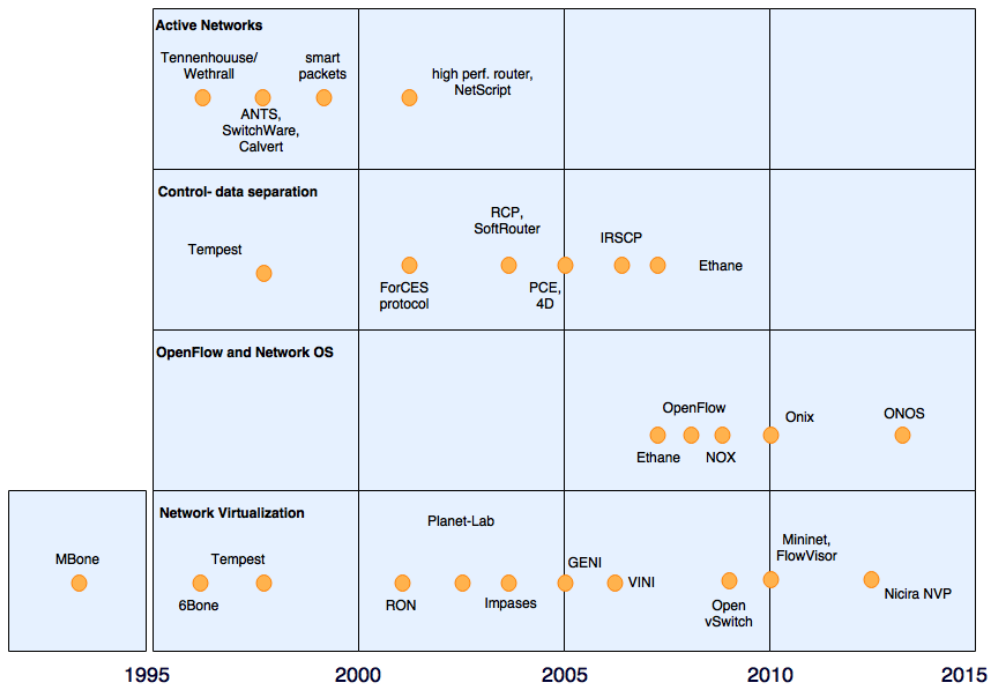


Figure 1: 20 years perspective of Selected Development in Programmable Networking [FRZ13]

executed on the node was carried in the data packets (in-band) and caching was used for efficiency of code distribution. In programmable router/switch model, out-of-band mechanism was used in order to execute code on each node.

In early 2000s, network operators were looking for better solutions to control the traffic path (traffic engineering). The anxiety that network operators had, was due to increment in traffic and significance of network reliability. Control and data plane in conventional routers and switches were tightly coupled. Therefore, this tight coupling made network management tasks such as controlling or predicting routing behavior or even debugging configuration issues extremely tough and challenging. Thereupon, to address this issue, different attempts to separate data and control plane started to flourish.

## 2.2 OpenFlow

In the mid 2000s, funding agencies and researchers were attracted to do network experimentations at scale. However, during that time Stanford university came up with a program, in which its idea was distinctive and its focus was more local and tractable rather than big scale (e.g. campus network). Before OpenFlow, there were two goals in order to make deployment of the SDN globally. First one was fully programmable network and the second one was pragmatism. OpenFlow tried to satisfy these two goals by having a balance between them. Therefore, building on existing switch hardware and enabling more functions than previous controllers satisfied both goals. Even though, founding OpenFlow on top of existing switches limited its flexibility, but it made OpenFlow promptly deployable and allowed SDN development to be both pragmatic and bold.

Almost all of the modern ethernet routers and switches use flow-tables to forward packets from one point to another. Different vendors build different switches and routers with different flow-table structure. However, there are a big set of common functions which are the same in different vendor's devices, which OpenFlow tries to exploit and take advantage of them. Network administrators can divide traffic into two production and research flows, where it gives the ability to the researchers in order to experiment new network protocols in realistic environment to gain enough confidence which is needed to deploy these new network protocols globally. [NTH<sup>+</sup>08]

The datapath of an OpenFlow switch comprises a flow table and also a task which is attached with each flow entry. Moreover, an OpenFlow switch has a flow table, a secure channel and OpenFlow protocol. There is an action attached to each flow entry which tells the switch how to process the flow. The job of secure channel is to connect controller to switch that allows packets and commands to be transferred between switch and controller using OpenFlow protocol. OpenFlow protocol is a standard interface that helps the controller to communicate with the switch. Therefore, researchers do not need to program the switch directly, rather entries in the flow table can be defined externally. Figure 2 illustrates an example of OpenFlow Switch.

In OpenFlow switch, a remote control process defines where packets should be forwarded. Flow can be a TCP connection or all the packets from specific IP or MAC address, or even all the packets from same switch port

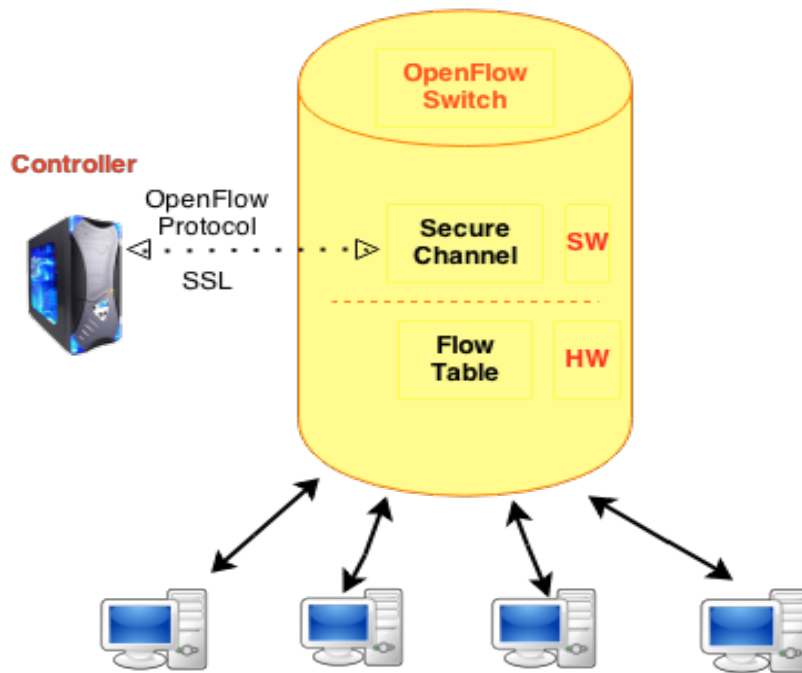


Figure 2: OpenFlow Switch Example

or same VLAN tag. There are three fields in an entry of Flow-Table. The first field is the packet header which defines the flow. The second field is the action that specifies how the packets should be processed. Lastly, the third field is statistics that tracks the bytes and number of packets of each flow in order to facilitate finding inactive flows for removal.

As mentioned earlier each flow entry has an action associated with it, and these actions can be concluded into four basic ones. The first action is forward the flow's packet to specific port(s), which permits packets to be routed in the the network and in most of switches this action is expected to take place at line-rate. The second action is to encapsulate the flow's packets and forward them to a controller through a secure channel. Usually this action is taken for the first packet in the flow, so the controller can determine whether this flow should be added to the flow table or not. However, this action also can be used in some experiments where all the packets are sent to the controller for the processing. The third action is to drop all the packets, which can be used for some security reasons to prevent denial of service attacks, or to decrease counterfeit broadcast discovery traffic coming from end-hosts. And lastly the fourth action is to forward the flow's packets to

the switch's ordinary processing pipeline.

OpenFlow enabled switch is a commercial switch (router, or access point) that supports OpenFlow protocol, flow table and secure channel. Basically, OpenFlow protocol and secure channel will be ported on the switch's operating system to run, and flow table will be using the existing hardware on the switch such as TCAM. Figure 3 illustrates an example of OpenFlow enabled switches, and access points. In this example, one controller manages all the flow tables; for purpose of good performance and robustness, OpenFlow protocol permits a switch to be controlled by multiple controllers. Controller job is to add or remove flow entries from the flow table. There are two different types of static and dynamic controllers. In the static, the controller statically add flows to the flow table and for the duration of experiment multiple computers get connected to each other and the experimental flows are isolated in order to do not interfere with the production network. A more complex controller, can add or remove the flows dynamically during the experiment, therefore the researcher can control how flows should be processed and control entire OpenFlow switches.

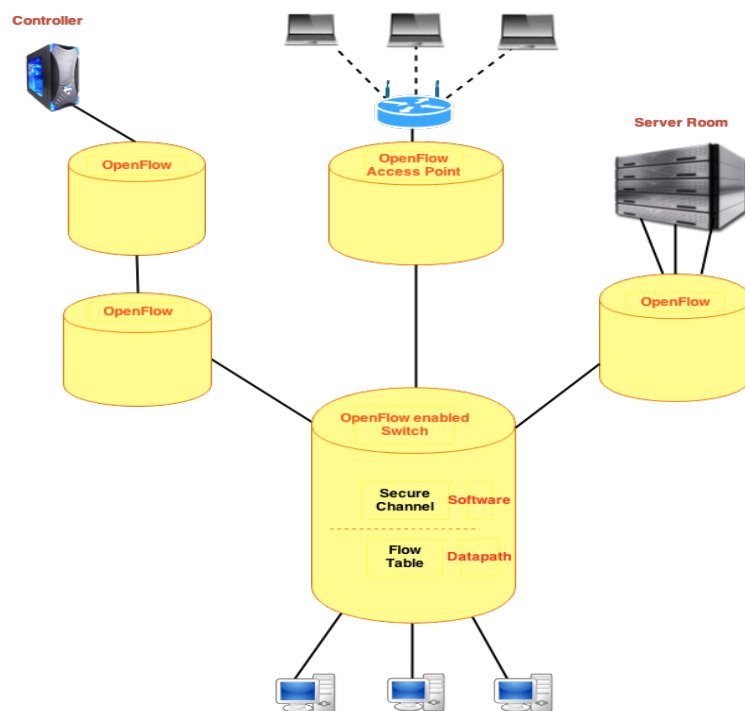


Figure 3: Example of OpenFlow enabled switches and access point

### **2.3 Network Function Virtualization (NFV)**

Commonly, network services (i.e. firewall, DPI, load balancer, etc.) are operating on proprietary hardware boxes. Running services on proprietary hardware takes big portion of expenditure for CAPital EXpenditure (CAPEX) and OPerational EXpenditure (OPEX). The process of moving services functionalities from hardware boxes to software and making them virtualized is called Virtualized Network Functions (VNFs). Therefore, instead of running services or network entities on specialized hardware, an implemented software version of the same service will be running on commodity hardware which already exist in data centers, cloud providers or even end-user office or home. [CDSI+14]

Resiliency to failure in VNF is crucial for network operators. If the service is not reliable, then users and subscribers would abandon it, in this case the cost of failure would be incredibly high for the operators. Software faults, hardware faults and operator faults are three major failures that can impact service delivery of network operators. In software level, different levels, such as virtual machine, hypervisor, host OS or even the VNF instance itself can cause failure in the system. In hardware level, using commodity servers can cause failure. Also, configuration mistakes by operator can break the system.

Evaluation and building an application for reliability evaluation of VNF, is not easy. This is because, lack of information for the internal structure of virtualized technologies like virtual machines provided by third parties, make it challenging to evaluate and test their reliability. Since NFV itself is still under development, therefore reliability of network function virtualization infrastructure (NFVI) and suitable measures and metrics for its reliability should be analyzed. Lastly, integration and interoperability of different hardwares and softwares together results in complexity, which consequently makes finding and resolving issues in the infrastructure troublesome.

### **2.4 Service Chaining**

The current network infrastructure with its current middleboxes (such as firewall, caching, network monitoring, DIS/PIS and etc.) are fixed, which is advantageous in term of service quality that has served very well the telecommunication industry up to now. However, in the current market and technology this infrastructure is inflexible, and hinders the network service

providers (NSPs) to cope with the technology in order to form pioneer and full service chains at will. [JPA<sup>+</sup>13]

Changing the current network infrastructure needs careful engineering due to the interdependencies among the functional components and high quality expectations. Moreover, adding new component or functionality in the network that has been already deployed, is complicated, expensive, and requires a lot of time. This stiffness in the network infrastructure, hinders the network from emerging to new revenue sources and other possible opportunities such as retooling the network.

Configuration and management tasks currently are done more in manual and tradition way, however with increasing automation, both CAPEX (capital expenditure) and OPEX (operational expenditure) can be reduced dramatically. By deferring network resource investments such as refactoring and optimizing the use of existing resources, reduction in CAPEX can be attained. By reducing network touch points which can possibly lead to less configuration errors, OPEX reduction can be achieved.

Each middlebox in the service chain is stateful and has very limited but specialized functionality which is closed hardware and purpose-built. With the existence of many middleboxes in the network, they play an important role in the network ossification, which take considerable part of the network OPEX/CAPEX. However, because each middlebox is designed to provide a single service, network operators cannot reuse them, and if one box is taken out the whole chain will break. Though, with the emergence of SDN and NFV in the operator network, orchestration, flexible allocation and management in the layer 2 to 7 of network functions and services became easier. Therefore, SDN and NFV provide a good foundation for dynamic service chains.

Dynamic service chaining bring carrier-grade process for persistent delivery of services, where carrier-grade in this context means designing the whole process for high availability and fast failure recovery that in each step of process reliable testing capability is integrated. Moreover, Persistent delivery means, utilizing automated (re-)deployment and network function orchestration in order to improve the operational efficiency. Figure 4, depicts how data traverse from source to destination with and without usage of dynamic network service chaining.

As Figure 4 illustrates, service chaining provides the means that the

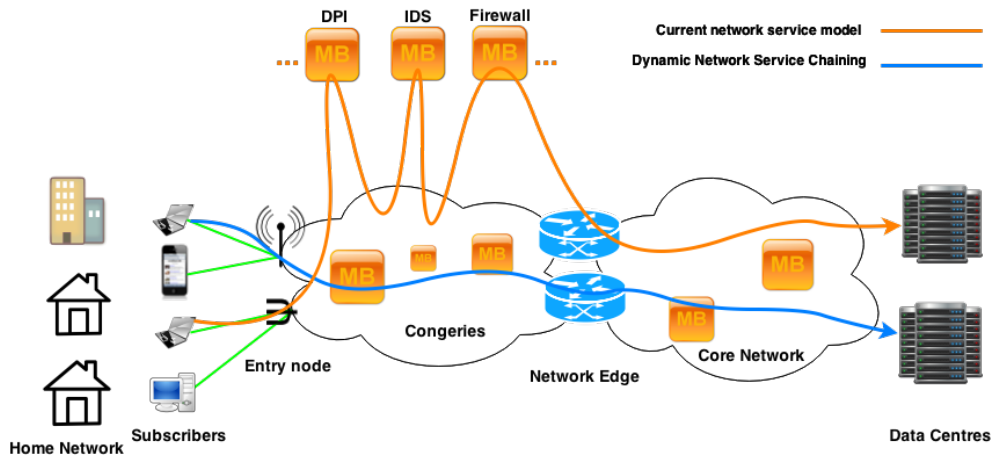


Figure 4: Static Vs. Dynamic Service chaining

flows go from source to the destination without any disturbance by different services which are located at different nodes. Deploying various physical network elements can be avoided in such way that different services are implemented as member of dynamic chain where each flow is processed through different service functions.

Figure 5 shows another benefit of dynamic service chaining. In order to provide better fairness and security for the end users in a network, data passes through various policy enforcement points such as traffic schedulers, load balancers, security gateways, etc., that depending to the size of network, it can be in hardware or software. However, if the data needs to pass different networks, then extra operator investments in terms of hardware and software are needed.

As an example we can mention provider edge (PE) that comprises different policy elements for software and various hardware elements for routing and forwarding traffic. In this case, because data requires to pass through various numbers of policy elements, there will huge deterioration in terms of delay, which depends to the load of cross-domain network. However, with the aid of dynamic service chaining, which needs to be implemented in every one of the network domains, this degradation in the delay can be bypassed. Therefore, network service chaining functionality should be implemented on PE elements, which can provide traffic performance acceleration with its

intelligent traffic steering.

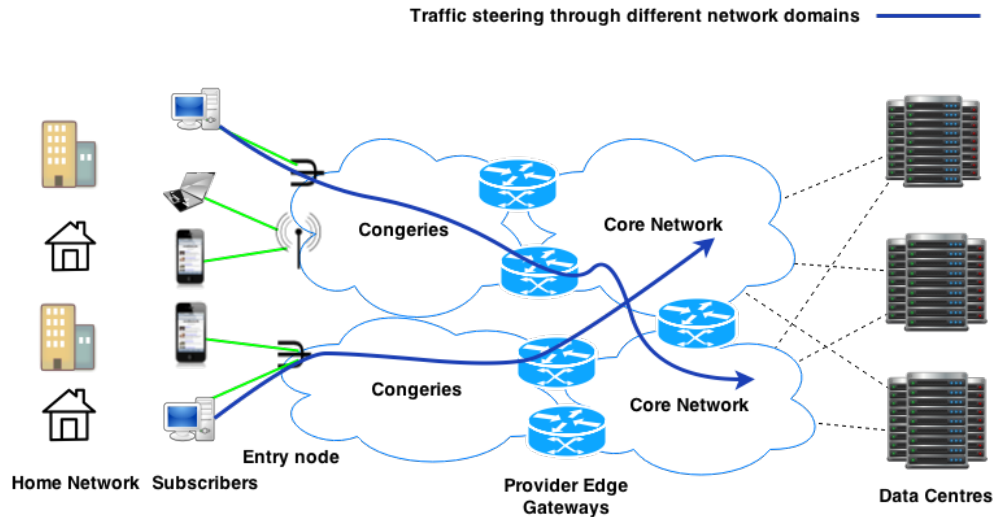


Figure 5: Traffic Steering

## 2.5 Service Chaining Challenges

With all the benefits and advantages that dynamic service chaining introduces to the network, it comes with its own challenges. Among various challenges in this field we can name, coupled topology, configuration complexity, solid ordering of service functions, application of service policy, imposed high availability, transport dependency, elastic service delivery, limited end-to-end service visibility, traffic selection criteria, per-service function (re)classification, symmetric traffic flows and multi vendor service functions. In the next subsections, these challenges will be discussed briefly. [PQ14]

**Coupled topology:** Usually, network service deployments are dependent to the network topology. Therefore, this dependency prevents the network operators to fully utilize service resources, which restricts redundancy, scalability and capacity across network resource. Moreover, when there is a need for extra service function, topology has to be modified in order the new service to fit in, which result complex device configuration and network changes.



**Configuration complexity:** A direct impact of coupled topology is the complication of the whole configuration, specially in deploying dynamic service function chains. Simple changes in the chain such as introducing new service or removing it, causes a change in the ordering of the services that requires to change the topology and consequently changing the configuration of some or all of the services in the chain. This is the reason network operators try to avoid changing the topology once they installed, configured and deployed it in the production, due to probable misconfiguration and resultant downtime.

**Solid ordering of service functions:** Services in the service chain are not related to each other, and there is no concept that specific service should be installed before another one. Although, for the network administrator there is a restriction to put the services in a special order to receive the best performance. Service function chains today are built based on manual configuration processes which makes them error prone and slow. With the emergence of new service deployment models the policy and control planes provide services that can be utilized optimally and provide connectivity state.

**Transport dependence:** Service functions are usually deployed in the network (overlays and underlays) with a confine transports. However, service functions should support different transport encapsulations or have a transport gateway function to be present when service functions are tightly coupled with the topology.

**Traffic selection criteria:** Traffic selection is rigid, therefore when traffic traverse from one segment to another, it should goes through all the service functions whether it needs the service enforcement or not. This rigidity is due to the topological behaviour of service deployment since the forwarding topology forces the data to pass through the service functions that it imposes. In some cases there are some flexible traffic selection, which access control filtering and policy routing is used to achieve that. However, this results operationally complicated configuration which is comparably inflexible.

**Limited end-to-end service visibility:** Both service-specific and network-specific expertise are required in order to troubleshoot and find service related

issues, which is a complicated process. Moreover, this complexity is increased when the service chains expand into multiple administrative boundaries. In addition, due to differences in virtual and physical environments in term of topology the challenge for finding the network issue will be higher.

**Per-service function (re)classification:** Each service does the classification individually and it has its own method to do the classification. Therefore, services do not take advantage of their previous service that did the classification before.

**Symmetric traffic flows:** Each service in the service chain depending to its functionality, might need to be unidirectional or bidirectional. For instance services such as firewall or DPI, need to be bidirectional to ensure consistency. Therefore, existing service deployment models have static approach which has complex configuration for each network device in the service chain.

## 2.6 Cloud (OpenStack)

OpenStack is an open-source cloud platform which is founded by NASA and Rackspace on July 2010. Over the years, it has become mature gradually and currently it has more than 4500 members and most of well-known enterprises such as Dell, IBM, HP, SUSE, Citrix, NASA, Rackspace, NetApp, Cisco, Nexenta and many more companies are supporting and contributing in it. OpenStack currently comprises of nine projects which we discuss about each one of them briefly [KGC<sup>+</sup>14] [WGL<sup>+</sup>12].

- **Nova (Compute)** is the computing component in OpenStack, its job is to handle the computing tasks by deploying and managing large amount of virtual machines and other kinds of instances.
- **Swift (Object Storage)** is scalable object storage infrastructure in OpenStack, which is used mostly for permanent type of data that need to be stored, retrieved and updated. Its key roles in OpenStack are secure storage for large data size, archival ability with redundancy, and media streaming. Swift comprises of these components: Container Server, Object Server, Account Server, Proxy server and the Ring.
- **Cinder (Block Storage)** is a persistent block storage provider for the guest virtual machines. Cinder can provide the backup of virtual

machines volumes by collaborating with swift. With the aid of API, supplied volumes for nova instances can be manipulated to provide the ideal type of volume and volume snapshot. [RB14]

- **Keystone (Identity Service)** is identity service in OpenStack. With keystone, different users are mapped to OpenStack in order to access the services they are authorized. Keystone provides tokens in order to give authorization to different users, and each token by default is valid for an hour.
- **Neutron (Networking)** is the manager of network in OpenStack. It connects different components of OpenStack together and at the same time provides static internet protocol (IP), dynamic host configuration protocol (DHCP) and virtual area network (VLAN). Neutron also provides advance policy and topology management.
- **Horizon (Dashboard)** is the user interface in OpenStack. Through the web user interface it provides easy provisioning and automating cloud-based resources.
- **Glance (Image Service)** is the image service in OpenStack which provides a template for new instances during the deployment. It provides essential services for retrieving, discovering and registering virtual images through API.
- **Ceilometer (Telemetry)** is the telemetry service to provide cloud providers billing service for the users. It keeps track of usage of different services by the user in order to provide the bill information for them.
- **Heat (Orchestration)** is the service which stores necessary information for a cloud application in a file that specifies which resources are required for that specific application.

All the above mentioned components can be installed on different servers, and there is no need for them to reside on the same machine. This is achieved by Queue Server, so messages from different services are transported through Advanced Message Queue Protocol. Managing cloud is possible through Nova-API which make it possible for developers to create applications managing OpenStack. Figure 6 shows the Architecture of OpenStack.

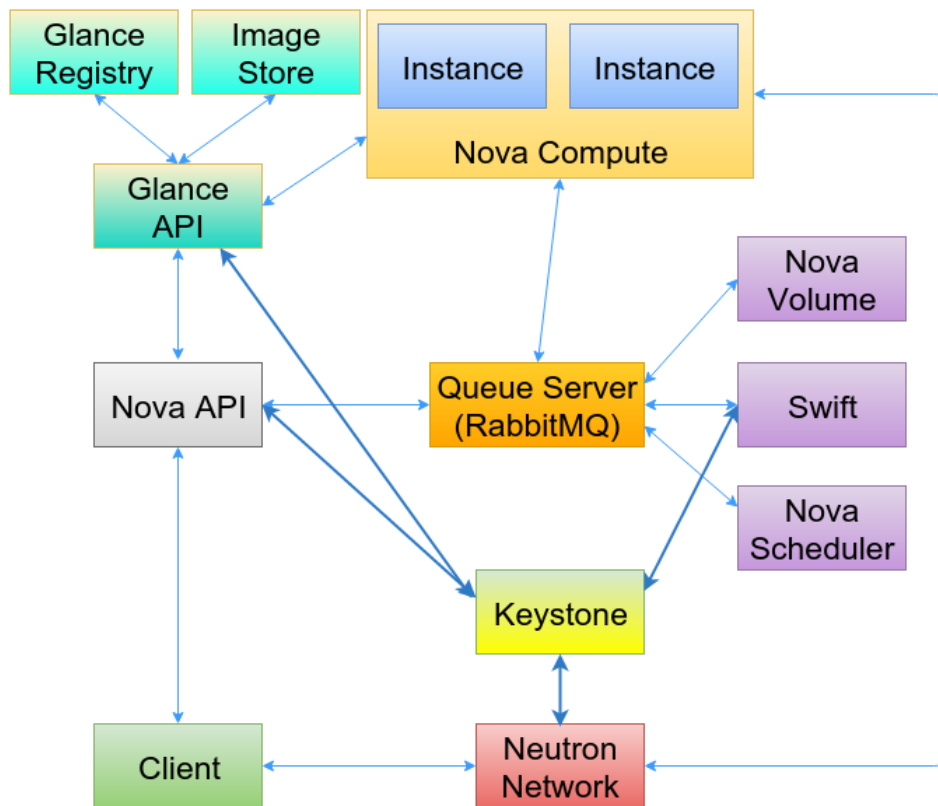


Figure 6: OpenStack Architecture

Currently there are different Open-Source cloud platforms which are used by different communities. Each of these platforms have their own characteristics which make them unique in every aspect. Among different cloud platforms we can name OpenStack, OpenNebula, Eucalyptus and CloudStack. Table 1 shows comprehensive comparison between these cloud platforms.

Table 1: Open Source Cloud platforms Comparison

<b>Name</b>	<b>OpenStack</b>	<b>Eucalyptus</b>	<b>OpenNebula</b>	<b>CloudStack</b>
<b>Origin</b>	Rackspace and NASA	University of California	European Infrastructure Grants	cloud.com
<b>Open-Source License</b>	Apache 2.0 License	GPLv3	Apache 2.0 License	Apache 2.0 License
<b>Programming Language</b>	Python	Java and C	Java and Ruby	Java
<b>Public Cloud Compatibility</b>	Amazon EC2, S3	EC2, S3, EBS, AMI [HP115]	Amazon EC2	Amazon EC2, S3
<b>Hypervisors</b>	Xen, KVM, HyperV, Xen Server, VMware, LXC	Xen, KVM, VMware [Euc15a]	Xen, KVM, VMware, vCenter	KVM, Xen Server, HyperV, VMware [Clo15]
<b>Networking Model</b>	Flat DHCP, VLAN DHCP	VLAN DHCP [Euc15b]	VLAN	VLAN
<b>Cloud Implementation</b>	Public and Private	Private	Hybrid, Private and Public	Public and Private
<b>Operating System Support</b>	Most Linux Distributions	Linux (images of Windows and Linux)	Most Linux Distributions	Most Linux Distributions
<b>Database</b>	PostgreSQL, SQLite3, MySQL	PostgreSQL	SQLite (some versions), MySQL	Mainly MySQL
<b>VM Migration Support</b>	Yes	No	Yes	Yes

## 2.7 Linux Containers (Docker)

Linux containers are linux processes, residing on single machine, sharing the same operating system kernel, yet separated from the rest of machine processes. So in this manner, other linux processes have no permission to interfere with linux container processes. Containers put all the necessary dependencies, code, and system tools that are needed for running a software in a package. Therefore, it is guaranteed that running the software will always be the same without the necessity for any modification or extra configuration.

Virtual machines and linux containers are sharing the same goal which is resource isolation and allocation, however they use different architecture and approach. Figure 7 shows the architecture of linux containers and virtual machines, which illustrates linux containers are more efficient and portable compared to virtual machines since they do not require to operate on top of guest operating system. [Doc15][ISK<sup>+</sup>14]

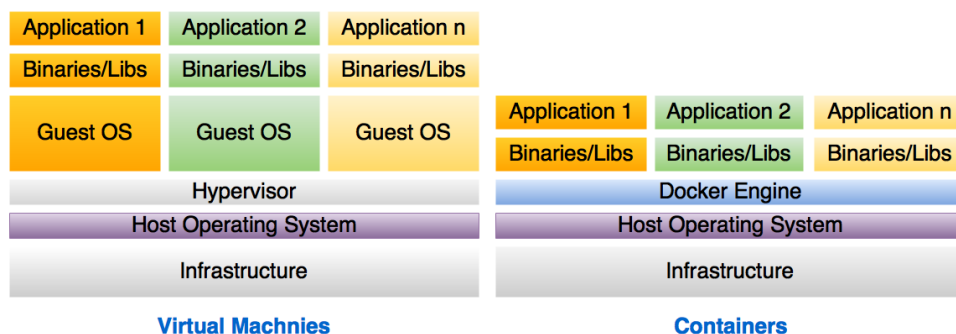


Figure 7: Virtual Machines vs. Containers Architecture

Docker is an open source software system, which makes utilizing linux containers simple and efficient. Docker containers compare to virtual machines are lighter, memory efficient and boot-up instantly. Docker exerts layered file system for images, thus various images can share common files, which can make disk usage and image downloads efficiently. Docker has an online repository where images can be pushed to the cloud, and downloaded on another machine simply and hassle-free.

Virtual machines are good for what they are designed for, which is abstracting the underlying hardware and permitting multiple operating systems (OS) to run on the same hardware. This lowers the cost, however virtual machines are not ideal for every situations. For instance, virtual machines take minutes to boot-up completely the OS, therefore gives the opportunity to hackers to exploit known vulnerabilities during boot-up [ISK<sup>+</sup>14]. Virtual machines are not suitable for microservices since even the simplest process requires its own virtual machine. It requires a lot of effort to manage virtual machine's lifecycle and apply patches to it. The reason is because every virtualized application has to operate with two operating systems which is hypervisor and the guest OS inside the virtual machine.

On the contrary, linux containers lowers the cost and improve agility. Each virtual machine can handle multiple containers inside it, which are all sharing the same operating system kernel, leading to simpler management due to fewer operating systems. Since docker containers leverage layered file system, application patching is matter of adding another layer. For instance, Apache web server, java runtime system, and Redis for caching can be different layers for a web application image. While operators are not able to see the workload inside a virtual machine, from container host environment they can look inside a container and detect unused containers to retire them if necessary to optimize resources. Since containers have smaller payload and do not carry the overhead of guest OS and hypervisor, booting and restarting them is quicker. Faster boot time, consequently can lead to less downtime that can reduce cost for organizations who utilize public cloud services to handle their sensitive tasks.

Linux containers provide as much isolation as virtual machines through using linux namespaces, Security Enhanced Linux (SELinux) and control groups (cGroups). However, there are some security issues with containers. For instance, kernel exploit at the host operating-system level endangers all the containers running on that host. Though, by enforcing compulsory access control eliminating this flaw is feasible. Ultimately, securing applications by containerizing them is easier, since smaller payload lessens the surface area for security glitch, and rather patching the operating system it is possible to update it. Moreover, isolating security module of an application from the application and putting them into separate containers, makes it easier for security team to debug and update the security module since they do not need to touch the application's logic.

Containers are more suitable with microservices compare to virtual machines. This is because, for developing an application, instead of using monolithic architecture, developers can develop each component with the best suitable programming language. Thus, each component can be packaged inside a container independently. In case of scaling, instead of duplicating the whole application, only those components that are necessary can be replicated, which can save space and other resources. In addition, CPU, RAM, and other resources can be scaled individually. Aside from microservices, below is a short list of some advantages of linux containers:

- **PaaS (Platform as a Service):** permits PaaS frameworks to interoper-

ate.

- **Policy-aware networks:** by giving priority to the containers that have acute services running on them, application experience can be improved.
- **Development and Testing:** applications can be replicated and deployed on any host machine which reduces the dependency issues.
- **Network application containers:** add new functionality to the foundation of network operating system by running third-party applications in containers.
- **Intercloud portability:** efficiently move application components among clouds.

IBM has done an extensive comparison between virtual machine (KVM) and linux containers (Docker) [FFRR14]. In this report they have conducted different comparisons such as memory bandwidth using Stream benchmark, CPU Linpack performance, Random Memory Access using RandomAccess benchmark, Network bandwidth using nuttcp tool, Network latency using netperf tool, NoSQL redis and MySQL performance. In this report it is concluded that in every comparison test which is conducted, Docker is equal or better than KVM in every case. KVM is not suitable for scenarios when workloads are sensitive to latency, due to the overhead that KVM adds to every I/O operation which can vary from negligible to significant depending on I/O operation. It has to be mentioned that Docker's NAT also adds overhead to workloads that have high packet rate. Therefore there is tradeoff between performance and each of management that needs to be investigated in disparate cases.



### 3 Service Function Chaining Methodologies

In this chapter we will focus on the problem of service function chaining (SFC) and related research works that are done thus far. We will present few various methods for elastic service chaining and later in this chapter we will discuss about different methods and tools for monitoring cloud and resources.

#### 3.1 Background and Motivation

Current infrastructure of many Internet Service Providers (ISPs) and mobile network carriers, is tightly coupled with services. Services or middleboxes in the network provide various functionalities such as deep packet inspection (DPI), Intrusion Prevention System (IPS), video enhancement, and many more. The cost for maintenance, and also the trouble of introducing new service into the network, forced network operators, organizations and research institutes to endeavor for solving this fundamental issue.

When an internet subscriber is streaming a video, in a tight coupled infrastructure that all the routes are predefined, all the traffic goes through firewall to reach to the user. This is costly for the ISPs due to wasting resources for processing every packet through the firewall. It is also inefficient and time consuming to process packets which are not in need of processing. By eliminating firewall during the session when user is streaming video, resources can be saved and processing of traffic can be accelerated.

By the rise of OpenFlow and SDN controllers, researchers have been attempting to eliminate the current restrictions in the network. In dynamic service chaining different services are connected when it is needed, and traffic from one or more of these services is bypassed when their presence in the network is unnecessary. In case a service is over saturated by traffic, a duplicate of the same service can start running dynamically, in order to reduce the workload from the saturated service.

Steering is the building block for dynamic service chaining that has been investigated for the past few years, and different methodologies have been proposed for it. In steering, traffic is forced to go through different service nodes, instead of taking the default route to reach its destination. Steering currently can be done in various methods, using Network Service Header (NSH), policy-based routing, or policy aware switching [ZBB<sup>+</sup>13].

## 3.2 Service Chaining Methods

### 3.2.1 Ericsson: StEERING for Inline Service Chaining

Ericsson [ZBB<sup>+</sup>13] attained service chaining, which is efficient, flexible, scalable and open. One feature in this method that makes it outstanding compared to similar approaches, is its multi-dimensionality for handling different rules through various tables. Therefore, various rules can expand when number of subscribers and applications increase. Different routing tables communicate with each other through specific type of metadata, which makes tables to manage a set of services independently, for instance adding or removing services. Scalability in StEERING, makes it suitable for broadband networks, which enables operators to handle large number of subscribers and applications.

StEERING infrastructure consists of two different switches, OpenFlow switches and inner switches. OpenFlow switches are located on perimeter of network, and they are connected to middlebox services or gateway nodes. These OpenFlow switches are responsible to classify the incoming traffic and steer it to the next middlebox or service in the chain. Inner switches are connected to each other as well as OpenFlow switches and they are responsible for forwarding traffic using layer 2 switching.

As shown in Figure 8 in StEERING architecture there are two modules that are controlling the logic, which are OpenFlow controller and an algorithm for checking the best possible position for placing the services. OpenFlow controller used in StEERING is NOX, which its job is to set up the table entries for the OpenFlow switches. The OpenFlow rules that are set in the switches, perform classification for the incoming traffic and based on the subscriber, application, or the ordering policies assign different path to it. Afterwards, the traffic is forwarded to the next service based on its current location in the chain.

If the policies cannot be resolved through the packet header, then they can be determined via the packet payload. In order to determine the policies through the payload, deep packet inspection (DPI) is needed. In Figure 8 the yellow dotted lines indicate the DPI interface, which is between OpenFlow controller and inline services. Therefore, a notification is sent via the DPI to the OpenFlow controller, once it has resolved the policies of a flow.

In order to achieve scalability and prevent exploding switches with tremen-

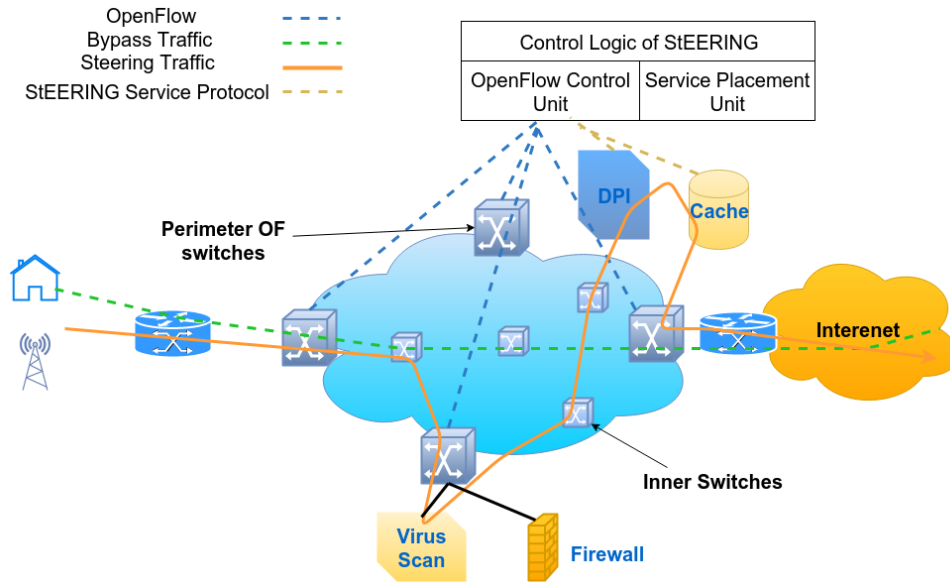


Figure 8: StEERING Architecture

dous amount of rules, four different functionalities are considered in StEERING to reduce pressure on each switch. Multiple tables are utilized in this method in order to break down multi-dimensional policies. Different ports on switches are assigned to have different tasks, where for each direction that traffic is going, different port will be dedicated. Microflow tables are also utilized in this method, which handle dynamically generated rules. Lastly, metadata is used for communicating the information between tables and associated actions.

Putting all the information such as subscriber, application, and ports in a single table would prevent the scalability. Therefore, separating different information into multiple tables would result scalability in each separate table linearly. Following Figure 9 shows six coercive tables along with one optional table which is dashed rectangle, and Table 2 explains responsibility of each of these tables.

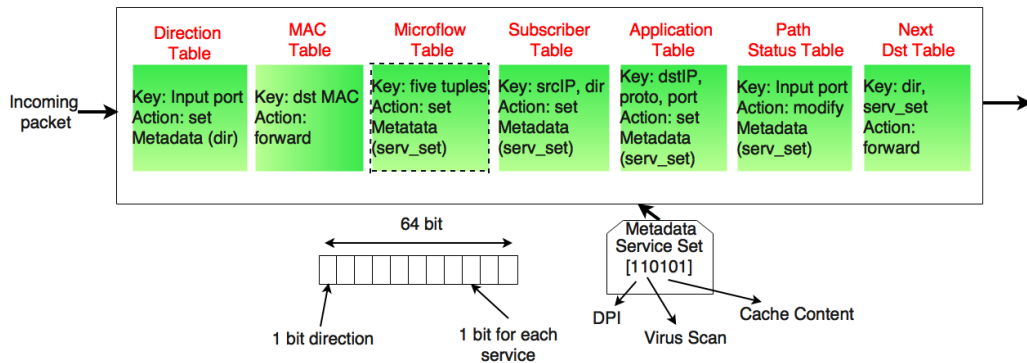


Figure 9: StEERING Tables

	Key	Responsibility
<b>Direction Table</b>	input port	<ul style="list-style-type: none"> <li>Identify direction of packet (upstream or downstream)</li> <li>type of the port packet received (node or transit port)</li> <li>set the metadata field (called dir)</li> </ul>
<b>MAC Table</b>	destination mac address	<ul style="list-style-type: none"> <li>Forward or Drop the pack</li> </ul>
<b>Microflow Table</b>	five tuples	<ul style="list-style-type: none"> <li>Handles dynamically generated rules</li> <li>Set the metadata field with services that flow should go through</li> </ul>
<b>Subscriber Table</b>	source ip direction bit	<ul style="list-style-type: none"> <li>Default service set for each subscriber</li> <li>Set the metadata field with services that flow should go through</li> </ul>
<b>Application Table</b>	destination ip protocol port	<ul style="list-style-type: none"> <li>Modifies default service set for a subscriber</li> <li>Set the metadata field with services that flow should go through</li> </ul>

<b>Path Status Table</b>	input port	<ul style="list-style-type: none"> <li>• Determines next service in the chain and services that have already been traversed</li> <li>• Modifies the metadata service set in order to omit precede services that the flow traversed already</li> </ul>
<b>Next Destination Table</b>	direction service set	<ul style="list-style-type: none"> <li>• The highest priority bit in the service set will be chosen for the next destination</li> </ul>

Table 2: Responsibilities of tables in StEERING

Perimeter switches in StEERING have two different types of ports, transit port and node ports. Transit ports are linked to the other perimeter switches or inner switches, while node ports are connected to the gateway nodes and services. Packets that are coming through transit ports, are forwarded to the destination base on MAC address, and packets that are coming through node ports are forwarded to the next service or gateway based on their assigned service path. Figure 10 depicts the port direction in StEERING architecture. OF1, OF2, and OF3 are perimeter switches, while SW1 is an inner switch. Node ports are divided into upstream and downstream, and in this figure black colored ports are upstream-facing, and red colored ports are downstream-facing. All the packets that arrive on upstream port, are travelling downstream and vice versa. Moreover, packets arriving on transit ports can be traveling in each of two directions.

Policies are usually pre-configured for switches, based on subscribers and applications. However, in some cases operators need to install new rules/policies dynamically. Microflow tables give the ability of introducing new rules dynamically to the network. Based on results of one middlebox (e.g. DPI), new policies can be added in order to meet specific requirement. The key in microflow table is a direction bit, and five tuples which are, IP protocol field, source and destination IP addresses, and TCP/UDP source and destination port numbers. Microflow tables have higher priority compared to application and subscriber tables. Therefore, if there is a hit in direction table the next table to call in will be microflow table.

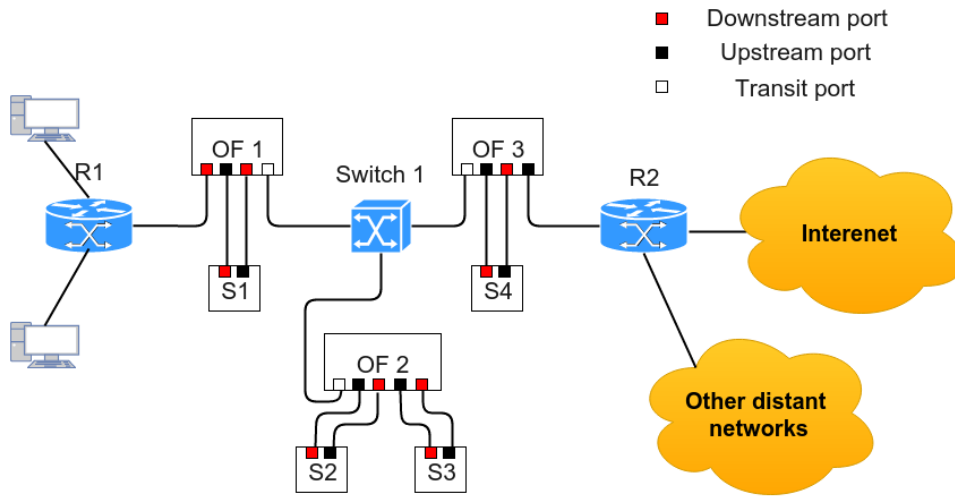


Figure 10: StEERING Port Direction

StEERING method works with OpenFlow 1.1, which uses metadata to communicate information and related actions among tables. Direction bit and service set are two types of metadata that are used in this method. The direction bit, indicates the direction of flow, and service set represents the set of services that requires to be applied on the flow, which is under process in the chain. The metadata supported in OpenFlow 1.1 is 64 bit. Thus, the first bit is used for the direction and another 63 bits are used for different services. Nonetheless, more complex features in metadata can be employed to attain better performance in the chain, for instance, load balancing.

Figure 11 depicts encoded metadata for attaining load balancing. In this case, instead of using a separate load balancer in the chain, traffic can be distributed in multiple instances with the help of metadata. By extending the basic format of metadata, from 1 bit for each service to multiple bits for one service, multiple instances for a service can be achieved. Thus, if  $n$  bits are assigned for a service then  $2^n$  instances can be represented. For instance if 4 bits are dedicated to a service, 1 bit will be an apply bit, and 3 bits will be instance bits, which will be 8 instances for that service. However, there is a tradeoff between number of instances and number of services that can be represented in the chain. The more instances assigned for a service, the less services can be used in the chain.

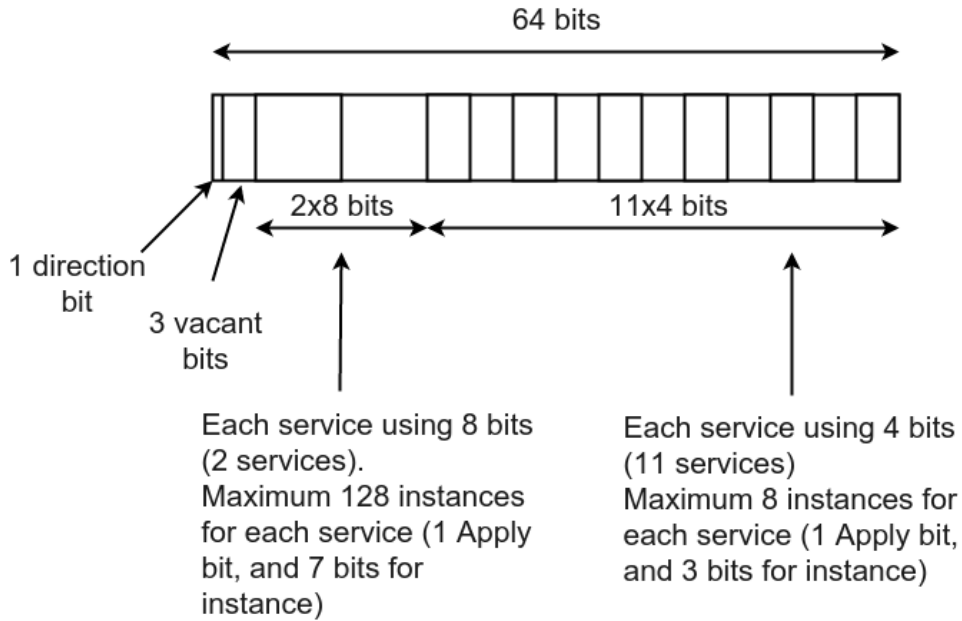


Figure 11: StEERING encode metadata for load balancing

### 3.2.2 Ericsson: Optical Service Chaining

Ericsson researchers Xia et al. [XSZ<sup>+</sup>15], believe that packet-based traffic steering is not efficient with aggregated flow, and it is only suitable for small volume of traffic due to its high configuration complexity and energy consumption. As the number of flows increase, configuration of flow matching rules become more complex and fallible. Therefore, a new traffic steering method is introduced, where instead of packets, wavelengths are switched to achieve coarse-grained optical steering. However, optical steering is less flexible and agile compare to packet-based methods. Though, it is suitable for service chains, where high capacity network functions (NFs) and traffic aggregation exist.

This method, similar to other traffic steering approaches consists of different modules and interfaces such as operations support system/business support system (OSS/BSS) module, cloud manager, and SDN controller. Service chaining rules and policies are defined and enforced by OSS/BSS module. On the other hand, cloud manager and SDN controller are responsible for provisioning of resources. There is a southbound interface between the

SDN controller and optical steering domain, which optical circuit switching can be done with the aid of proprietary interfaces provided by hardware vendors or using OpenFlow v. 1.4 protocol that has support for optical circuit configuration. Service chain can be achieved by both optical domain and packet domain. Figure 12 depicts the architecture of optical service chaining.

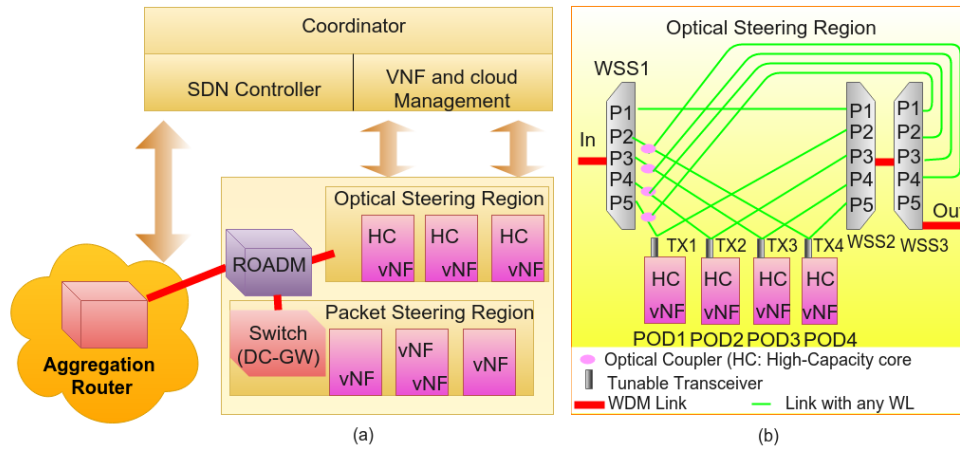


Figure 12: Ericsson Optical Service Chaining Architecture a)General Architecture b)Domain of Optical Steering [XSZ<sup>+</sup>15]

In this method, aggregator routers are residing at the edge of operator’s access/metro network, which is part of forwarding plane. These routers come with optical module in order to change aggregated traffic into wavelength flows that later are multiplexed into a fiber link to the data center. Aggregated flows from aggregator routers go through the high-capacity (HC) Network Functions (NFs). Moreover, aggregation can be achieved by OpenFlow matching rules on packet fields or using multiprotocol label switching (MPLS) [XSZ<sup>+</sup>15].

Figure 12a shows that in this architecture, data center/cloud has capability of both optical and packet steering, which makes it hybrid. Flows are entered to the data center and first go through Reconfigurable Optical Add/Drop Module (ROADM). SDN controller configures ROADM, therefore flows based on their size are forwarded to different domains. Aggregated flows are forwarded to the optical steering domain, and small size flows are dropped into packet steering domain for fine-grained processing. After a



wavelength gone through all the required virtual network functions (vNFs) then it is forwarded back to ROADM. From ROADM it can get dropped for further fine-grained processing or leave the data center.

Figure 12b, depicts how optical steering is attained by existence of different modules in this architecture. Wavelength-selective switch (WSS) is a switching device, which has one common port on one side as an input and multiple branches of ports on its other side for output. By configuration, every wavelength that enters WSS can be switched to one of the output ports without interfering with other wavelength channels. Tunable optical device, is responsible for the conversion of optical to electrical signals and vice versa. SDN controller is responsible to configure WSS, so wavelength flows that are coming from WSS output ports are connecting to the next vNFs in the service chain via optical couplers and fiber links. Optical coupler is a passive device which has 2 inputs and one output (2 x 1). It allows optical signal to enters from one of its input ports, though combining two or more input wavelength signals and extracting one output out of these signals results with significant losses. Hence, SDN controller should be responsible to prevent wavelength conflict when flows are looped back.

This method is flexible, since WSS takes few hundreds of milliseconds to tune, therefore it is considerably fast to provision on demand NFV service chain. optical steering can adapt traffic growth, since wavelength switching is independent from transmission rate, which makes it scalable. Lastly, power efficiency is another advantage of this method. Based on the simulation conducted by Xia et al, optical steering is more power efficient compared to packet-based steering.

### **3.2.3 Huawei: Service Chaining with Service Based Routing**

Huawei [Hua13] achieved service chaining using service based routing (SBR). The SBR method operates based on Policy and Charging Enforcement Function (PCEF) or Traffic Detection Function (TDF). Three different modules cooperate in SBR to attain service chaining, SBR controller (SBR-C), Traffic Classifier (TC) and SDN controller. Service routing policies are managed by SBR-C, which is configured locally or received dynamically through Policy and Charging Rules Function (PCRF). TC is responsible to classify the traffic using deep packet inspection (DPI), and report the result to the SBR-C. Based on the specified service routing policy, the classified

traffic is binded to the related service chain through SBR-C. Therefore, SBR-C dictates the forwarding rules to the SDN controller in order to forward these rules to OpenFlow enabled switches. The classified traffic steer through required middleboxes based on specified forwarding rules. Figure 13 illustrates huawei SBR solution using SDN.

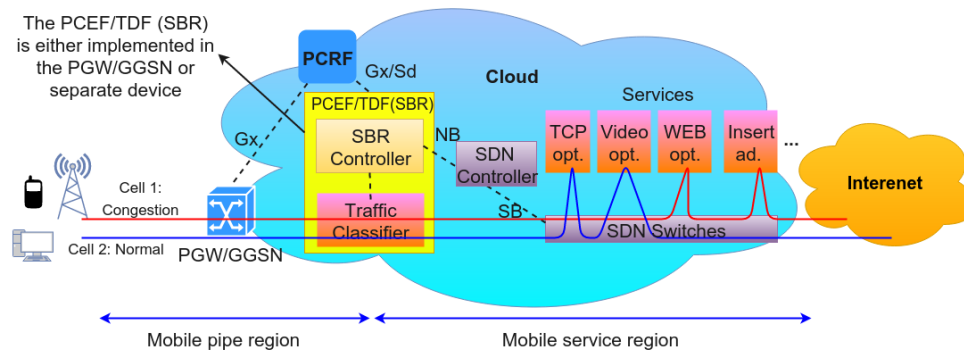


Figure 13: Huawei SBR with SDN solution

The SBR solution can be deployed in two different methods, which are standalone and embedded. Standalone SBR solution is suitable for operators that have already deployed PGWs/GGSNs (PDN Gateway/Gateway GPRS Support Node) and do not want to replace them. Mobile pipe zone remains unchanged in this method, where PCEF/TDF (Policy and Charging Rules Function/Traffic Detection Function) is placed behind the Gi/SGi firewall interface of existing PGW/GGSN. However, compared to embedded deployment, standalone SBR requires more configuration and more policy control signaling. On the other hand, embedded SBR deployment is suitable for network operators who prefer to replace their existing PGW/GGSN with the new one. In the embedded SBR, service routing policy control can take effect directly at the mobile gateway, which can be based on user context and radio access information. This makes, deployment and management of service provisioning simpler.

Huawei SBR solution is capable of multi-dimensional service routing policy control. Multi-dimensional policy control, chooses the best awareness option, which consists of radio access awareness, user awareness, and service type awareness. Based on the conditions that these awareness methods

provide, different path can be taken, and traffic can be steered through different middleboxes.

Radio access awareness in SBR, gathers information such as radio access type and user location from mobile gateway or PCRF. Based on collected information, SBR can steer the traffic through different middleboxes. For instance, if the user connection is in 3G mode, due to low bandwidth of 3G, video traffic steers through a video optimizer for enhancing the video streaming. While, if the user has 4G connection, then video traffic goes directly from internet to his device without the need of optimization.

SBR is user aware and it can hold information of each user in order to determine the steering based on this data. Each user has a profile and the routing policy is defined based on this profile. Traffic is classified into different user categories based on different subscriptions that users have. For example, a user with the VIP subscription gets better video quality because his traffic passes through video optimizer, while normal user traffic goes directly to the internet without any video enhancement.

Service type awareness in SBR, analyze the traffic from layer 3 to 7 and identifies the service that it requires. The Value Added Service (VAS) policies can be based on this awareness method in order to utilize different services for different traffic. For instance, video streaming traffic can be forwarded for video optimization service, while web traffic can be forwarded to web cache service.

#### **3.2.4 FCSC: Function-Centric Service Chaining**

Arumathurai et. al. [ACM<sup>+</sup>14] believe that current existing service chaining methods are not efficient due to the coupling of routing with the policy. In the existing approaches, SDN controller decides the services that needs to be applied for a flow. They setup the route for the flow which needs to go through the specified services, and for that they need to setup the state for intermediate switches. This solution makes the service chain, inflexible, undynamic, and unscalable.

Information-Centric Network (ICN), is a new approach that decouples the location of specific service from the function it provides. Arumathurai et. al. proposed Function-Centric Service Chaining (FCSC) which is a novel approach based on ICN. FCSC, decouples the services that needs to be applied for a flow from the location of these services (middleboxes), and does

the routing through a naming layer. Decoupling can simplify the dynamic alteration of services needed for a flow, which can also result in balancing the load. In this approach, instead of putting flow state in the switches, they will be deployed in the packet header. This can reduce the amount of state stored in the network and result in better scalability.

As mentioned earlier current SDN approaches have lack of dynamic, scalability, flexibility and reliability. Real-time decision for SDN controllers, is an issue to reroute the traffic for balancing the load on the network or a service. This issue can exacerbate if the number of flows increase, which shows low level of dynamicity. SDN controllers install rules on each switch, and the number of this rules is proportional to the number of flows. This hinders the network for scalability, if the number of flows or the network itself grows. If the result of DPI, indicates that the flow should go through another service for further processing, controller is responsible for building the new path. This, can cause extra control overhead in both latency and communication for every flow whenever the set of services are modified, which can degrade flexibility. Lastly, if a service or middlebox fails, switches should rely on the controller to build a new path for the flow, which increases the time for handling such crisis, and diminishes the reliability.

To eliminate the lack of dynamic, flexibility and reliability, FCSC uses naming layer for the current SDN infrastructure. In the naming layer approach, instead of requesting the controller to build a path for flow, only the name of services that are required for a flow is needed. Network can forward the packet to any service that provide the same functionality, which can cause load balancing, and also fast recovery when a service fails. To achieve this, FCSC incorporates hierarchical naming approach from ICN in order to represent the list of services, and longest-prefix matching in the forwarding information base (FIB) to forward the packets.

In order to provide scalability in the network system, FCSC utilizes packet header to carry the flow-state instead of using switches. In current method of SDN service chaining, switches hold the flow-state to forward the packet based on 5-tuple, to the next service. The number of these flow-states in the switches increase as the network expands or number of flows increase, which is not suitable for scalability. FCSC, uses ICN naming convention in order to put list of the services that packet needs to traverse. After each packet is processed with the service, the name of applied service is removed

from the list in the header and will be forwarded to the next service. In this method, switches need to hold forwarding information, based on per-service rather than per-flow. Therefore the number of states stored in the switches are proportional to the number of services, which facilitates scalability.

The FCSC, uses extended version of ICN for naming. Naming in the chain for instance is like: /DPI/cache/R5. Therefore, after packets go through DPI, the prefix DIP is popped from the list, then packet goes to the next service which is cache and then exits the network from egress port R5. Services that provide the same functionality have the same name, and they can have distinctive ID, that can be added as prefix in the service list. For instance, if there are two firewalls in the network they can have prefix `_A` or `_B` and naming list can be like /Firewall/\_A and /Firewall/\_B. Figure 14 depicts an example of the name changing of a packet in FCSC.

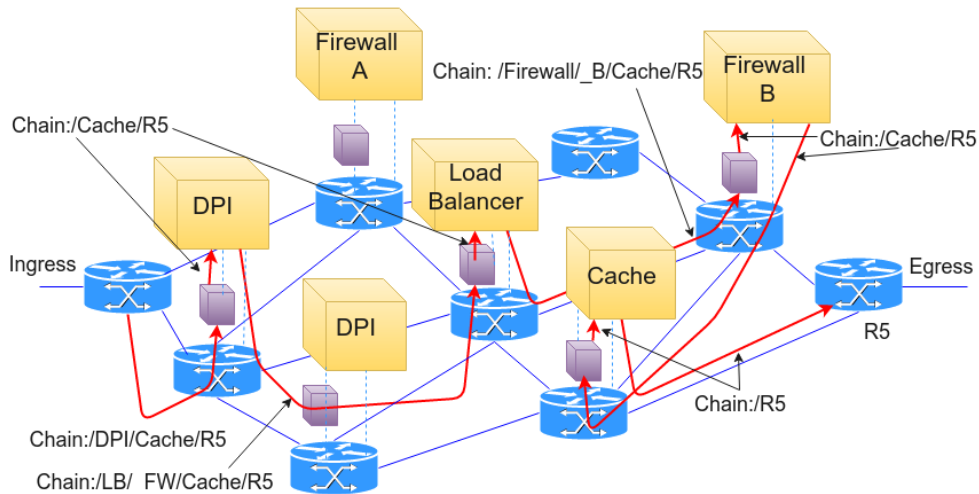


Figure 14: FCSC packet name changing example

### 3.3 Service Chaining Methods Comparison

Table 3 illustrates briefly the main features of different service chaining methodologies, which were discussed in the previous section. This table briefly shows what each method is suitable for. For instance, StEERING method, is suitable for cases where we do not want to have different elements in the system to take care of load balancing, or decision making for placing

services. Optical service chaining is good when we have a lot of data to process, where normal packet based steering is not efficient, and system breakdown might occur. SBR solution is appropriate where we have different set of users with different service needs. And FCSC is best when there are a lot of flows with the need of real-time decision making.

Table 3: A brief Comparison between different service chaining methodologies

	<b>Features</b>	<b>Steering Method</b>	<b>Goal</b>
<b>StEERING</b>	Multi-dimensional tables Algorithm to check best place to put a service Load balancing through the Metadata	StEERING (Packet Based)	Scalability Flexibility Efficiency
<b>Optical Service Chaining</b>	Suitable for aggregated data	Optical (Wavelength Switching)	Flexibility Power efficiency
<b>Service Based Routing (SBR)</b>	Multi-dimensional service routing User aware (profile)	Service Based Routing (Packet Based)	Easy deployment and Management
<b>FCSC</b>	Naming layer approach to apply services	ICN Naming in packet header (Packet Based)	Scalability Reliability Dynamic

### 3.4 Monitoring Cloud: Methods

Reliability and consistency in the cloud plays an important role for quality of service. There are various methods and tools that are used in the cloud to prevent or detect any unusual behaviour such as heavy traffic on specific node, low disk space, low RAM, or low CPU. In this section we will discuss some of well-known methods and tools for monitoring cloud.

#### 3.4.1 PCMONS: Private Cloud Monitoring System

PCMONS [DUW11] is a general purpose monitoring system for private cloud. Its architecture consists of three layers, which are infrastructure layer, integration layer and view layer. The infrastructure layer in this system has

the basic network equipments and software tools such as various hypervisors, cloud platform and operating systems. Since the infrastructure layer consists of heterogeneous resources, the goal of integration layer is to provide a common interface for this inconsistency. For instance, different hypervisors such as Xen and KVM or various cloud platforms like OpenNebula and Eucalyptus require different actions for monitoring. Thus, integration layer is responsible for abstracting any infrastructure details. The view layer, provides information such as service level agreement (SLA) and policies, where the users in this layers are mainly interested to check available service levels and virtual machine images. Different views are implemented for different users based on their needs, for example network administrators view is distinct from business managers view. Figure 15 depicts the PCMONS system architecture.

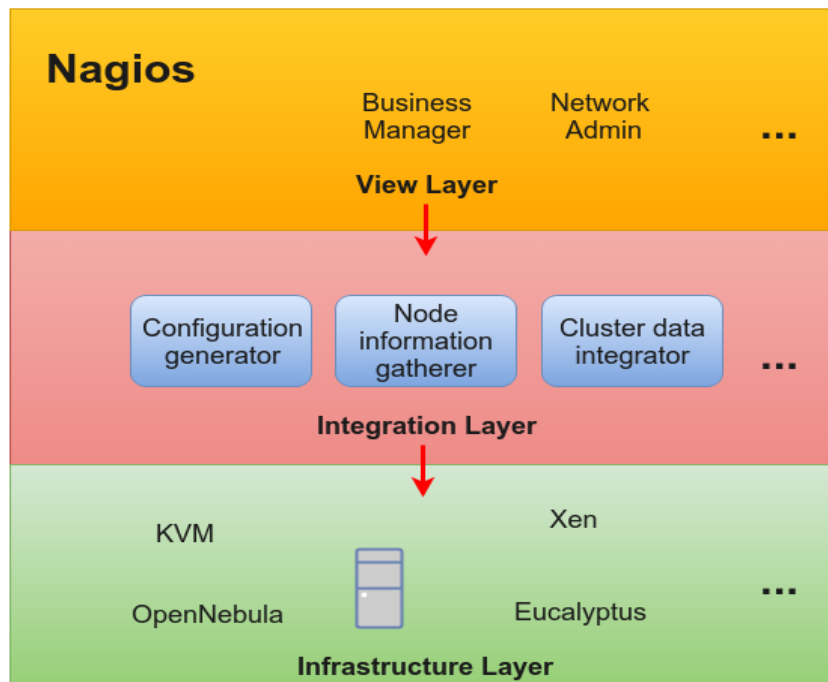


Figure 15: PCMONS Architecture

PCMONS is highly relying on integration layer for gathering information, and transforming these information for visualization. The monitoring is divided into different modules, which makes future adaptation to particular tools simple and effortless. Below is the list these modules and their responsibilities, and Figure 16 illustrates all the modules in action.

- **Node Information Gatherer:** gathers local information about cloud virtual machines and sends it to cluster data integrator.
- **Cluster Data Integrator:** gathers necessary data from cluster to prepare it for the next layer, and avoid dispensable data transfer from nodes to monitoring data integrator.
- **Monitoring Data Integrator:** gathers the data from cloud and saves it into the database for archival and also providing this data for the configuration generator.
- **Configuration Generator:** fetches the information from the database and prepares configuration file appropriate for visualization tools.
- **VM Monitor:** uses script running on virtual machines, and get information such as CPU and memory usage.
- **Monitoring Tool Server:** receives monitoring data from different modules and take various actions such saving the data into the database for archival purposes.
- **Database:** by getting support from monitoring data integrator, and configuration generator, saves the data into database.
- **User Interface:** Nagios user interface is used in this system.

### 3.4.2 Elastack: Automatic Elasticity in the OpenStack

The key element for elasticity is to monitor instances and check them regularly in order to get alerted when they have heavy load or when they are idle. Elastack [BMVO12] is a system introduced by Beernaert et. al., which performs monitoring of instances for cloud platforms that collects data from them, which can be used to manage the infrastructure.

In order to make Elastack scalable with its beneath infrastructure, a monitoring daemon is running on each nova-compute node to monitor the instances running on that node. Serpentine is an adaptive middleware for complex heterogeneous distributed systems that permits a system to adapt to changes which might happen in a production environment without the need of human interference. Components of Serpentine do not pertain on a



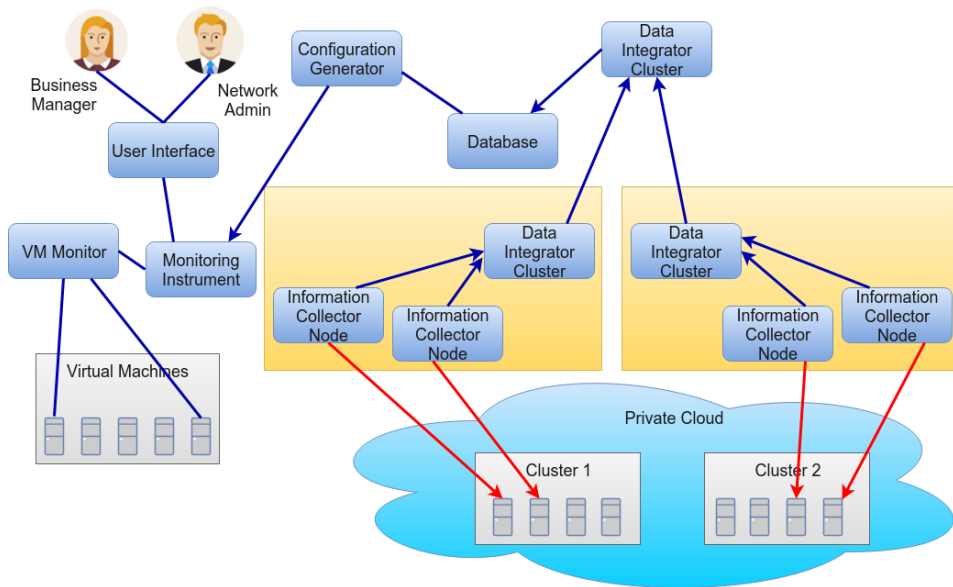


Figure 16: PCMONS Typical Deployment Scenario

persistent state since it was initially made to be scalable, which enables the system for macro and micro-management. Management policies are defined through different scripts which are applied to the system. Serpentine plays the role of node monitoring in the Elastack system which is shown in Figure 17.

Elastack is made of three main components: controller daemon, monitor daemons and serpentine script. When instances launch or terminate there are events which are propagated by the queue server (1), and instance monitors are responsible for monitoring instances and collect data periodically (2). The behavior of system is defined by serpentine script, which first a configuration file should be fed to it in order to connect all the available monitor daemons and controller daemon (3,4). The controller daemon is a background process that should be running on the same host as controller node ideally. Its main job is to extract the underlying cloud service and to separate the remaining Elastack components from it to ensure that it stays cloud infrastructure agnostic. With the communication of controller daemon through nova-api on the controller node the system can create and terminate instances in OpenStack (5).

Serpentine script creates and terminates instances and coordinate it with

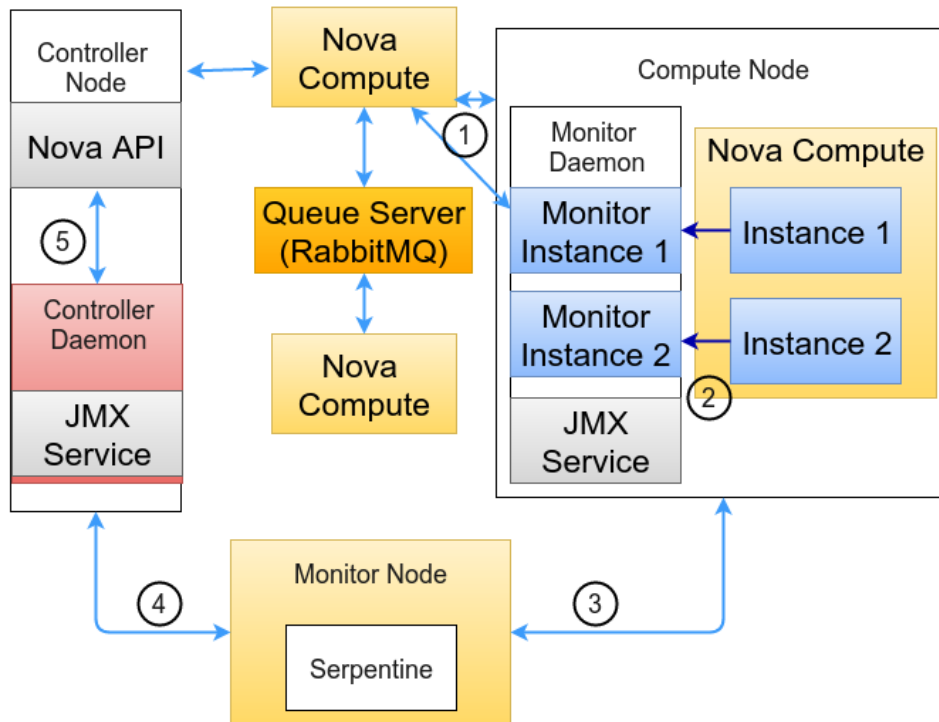


Figure 17: Elstack System Architecture

the load balancer, so when an instance is created it will inform the load balancer. Afterwards, load balancer initiates connection to the client running on that instance, then after establishing connection it will distribute tasks to this instance. In case of terminating an instance, Serpentine informs the load balancer that the instance is signed to be terminated, then the load balancer will stop assigning further tasks to that instance and after all the pending tasks of that instance are finished it will be signed as ready to be terminated. Meanwhile, Serpentine queries the load balancer if that instance is ready for termination, and once it receives positive reply then it will issue termination for that instance. New instances are created when  $x \geq (90.0 \times n)$ , where  $x$  is the total cpu usage of all instances and  $n$  is the number of instances. Moreover, Instances are deleted when the total cpu usage can be distributed to all the instances.

In order to not create or terminate an instance sequentially for example,

creating instance, removing it and then again creating it, the system requires each decision to be made twice before acting accordingly. Therefore, each time when the script is executed the decision will be stored, and after second time if the decision is the same then it will proceed otherwise the first decision will be marked as invalid.

### **3.5 Monitoring Cloud: Tools (General Purpose)**

In this section some of well-known general purpose tools for monitoring cloud and data centers will be discussed. All of these tools are open source and free to use.

#### **3.5.1 Nagios**

Nagios [Nag15] is an open source platform for monitoring network, services, applications, operating systems, infrastructure components and system metrics. Using plug-ins, any organization or individual can develop their own, or use pre-developed plug-ins to customize monitoring behavior of nagios in order to regulate it in the best possible way to suit their needs. Nagios is highly scalable [IPMM12], which can support up to 100,000 hosts and 1,000,000 services.

The scheduler daemon in Nagios is responsible for checking network and other related services and inform the administrator in case of failure or emergency by sending email or instant messages. By the help of Nagios Remote Plug-in Executor (NRPE) it is also possible to check CPU load and memory usage of local resources. Nagios Event Broker (NEB) is another module in nagios, which provides an API for developers to add extra work flow to nagios events. Therefore, a developer for instance can add extra feature to save results of a plug-in execution in a database.

Nagios configuration is text-based, which makes configuring it troublesome by remembering various configuration options and complex structure. Since the configuration file is text-based, configuration files can be created with any text editor. However, this makes it error prone especially for large network enterprises. Open source applications from nagios community make the process of creating configuration files easier. Some of these applications that can create simple configuration files are, Fruity and Lilac, which have PHP-based web interface. And for creating more complex configuration files,

NConf and NagiosQL can be useful to assist creating configurations for large network topology.

Nagios has a web-interface that gives the ability of monitoring servers, and services from anywhere and anytime. Nagios Fusion, which was released on 2014, unifies all nagios products and gives additional visual info to the user. Upgraded highcharts and advance dashlets are implemented in this release. This release, enhances visualizing operational status, which enables faster trouble resolution for an entire IT infrastructure.

### **3.5.2 Collectd**

Collectd [Col15], is another open source monitoring tool, which is a daemon that collects system performance statistics regularly and provide various ways for storing the collected data. From the collected statistics, one can extracts system performance and bottlenecks, and do future or capacity planning for the system. Collectd takes advantages of graphs, which provide tremendous amount of information for analyzing different system components.

Collectd is written in C language, for the performance purposes and also feasibility of running it without scripting language or cron daemon. This makes collectd to handle more than thousands of data sets without any issue or performance degradation. Moreover, collectd has a community that actively develop and update, and also provide a proper documentation for it. There are more than 90 plugins developed for collectd, which are ranging from basic cases to more specialized and complex topics.

Notifications can be sent through the daemon and this allows for threshold checking. There has been a development for a module called “collectd-nagios”, which allows collectd to be integrated into nagios for more sophisticated system monitoring. Figure 18 shows the architecture of collectd.

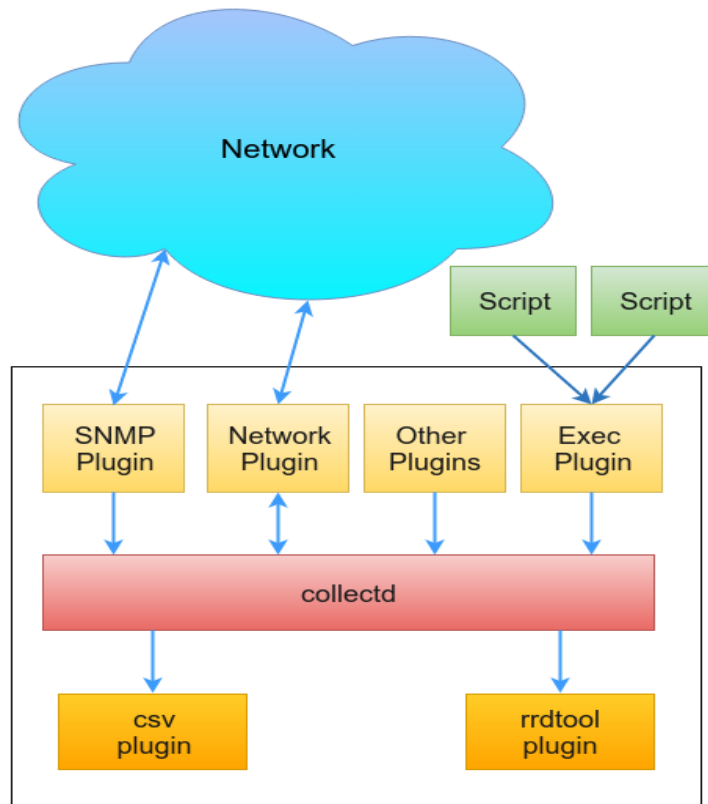


Figure 18: Collectd Architecture

### 3.5.3 Ganglia

Ganglia [Gan15], is also another open source monitoring system which targets clusters and grids. Similar to many other monitoring systems, it allows the user to check the system performance by checking CPU load, network utilization and other system attributes. It operates based on multicast protocol (listen/announce), to aggregate the state of systems that are monitored. Ganglia uses XML for data presentation, RRDTool for storing data and representation, and XDR for serializing and compacting the data.

The ganglia monitoring daemon (gmond) is multi-threaded daemon, which runs on every node of cluster and has four primary responsibilities. It monitors nodes for changes and notifies the user for any critical situations that needs attention. It responds to request with XML representation to describe the node state in the cluster. And last but not least, it listens to the states of other nodes in the cluster through multicast or unicast channel.

## **3.6 Monitoring Cloud: Tools (Cloud Specific)**

In this section we will discuss about some of tools, which are specifically made for monitoring cloud. Among all, we will briefly discuss about Amazon Cloud Watch, Azure Watch and Nimsoft.

### **3.6.1 Amazon CloudWatch**

Amazon CloudWatch [Ama16] is a proprietary monitoring tool for amazon cloud services. It can be used with AWS services such as Amazon RDS DB, Amazon EC2, and Amazon DynamoDB instances. It can collect logs, track various metrics (i.e. CPU utilization, disk usage, data transfer, etc), or respond to different changes automatically.

Aside from the metrics that Amazon CloudWatch provides, it is possible to create custom metrics in the application and use a simple API to request Amazon CloudWatch to monitor them. These metrics can be stored in order to spot trends or troubleshoot the issues that the application might encounter. Similarly, logs can also be stored with the Amazon CloudWatch, or existing logs can be sent to it to monitor them in almost real-time manner.

Furthermore, Amazon CloudWatch provide another feature, which is called Alarm. Alarms can be set in order to take automatic actions whenever is necessary. For instance, if CPU usage of one Amazon EC2 instance exceeds certain threshold, the alarm module can send a notification to the admin, or automatically add new EC2 instance to bring down the load and distribute the jobs.

CloudWatch also provide a graphical user interface in order to check the graphs. It can store metrics up to two weeks, which are shown on different graphs, that can help to detect issues at a glance. Amazon CloudWatch can also save all the metrics in the application owner's account for search and easy access.

### **3.6.2 Azure Watch**

Similar to Amazon CloudWatch, Azure watch [Azu16b] is a proprietary tool for monitoring microsoft cloud services. It can monitor different metrics which by default are disk read/write, Data in/out, available and used memory and CPU percentage. However, by configuring verbose monitoring, more metrics can be stored. By default each 3 minutes, metrics are sampled and

stored. However by turning on the verbose configuration, these intervals can be changed to 5 minutes, 1 hour, or 12 hours. Metric data is stored for 10 days, and after that will be removed. Different metrics can be selected to plot them on graph which is provided by Azure Watch.

Azure Watch also can send alerts when certain metrics goes above predefined threshold. For enabling alerts, a rule should be specified. If the rule meet certain criteria it can send a notification based on an event or aggregated events. Notifications can be sent as email to service admin, or any specified person. If the problem is resolved, another email will be sent as well notifying that the issue has been resolved.

Azure Watch can also auto scale [Azu16a] similar to Amazon CloudWatch. This can be achieved based on different attributes, such as admin predefined upper and lower limits, schedules, epochal performance, at the beginning or end of specific hour, or any combinations of these attributes.

### 3.6.3 Nimsoft

Despite from Amazon CloudWatch and Azure Watch that are proprietary tools for Amazon and Microsoft, nimsoft [Nim16] cloud monitoring tool, can be used for any cloud platform. This tool has subscription fee, and it is not free. However it provides wide ranges of features for monitoring the cloud.

Aside from basic features that Amazon CloudWatch and Azure Watch provide such as storing various metrics and representing them graphically, it has different features that make it unique compare to other monitoring tools. For instance, it can notify the service admin via SMS, RSS feed, instant messenger, twitter, email and many more. For testing availability and transaction times, it tests the availability from 60 different locations in 40 different countries.

Nimsoft is SaaS, which does not need to be installed, and can be accessed through web and set up easily. However, based on all the good features that Nimsoft provide, we have mention that this monitoring tool is only suitable for websites and web services, since they are only focusing on that.

## 3.7 Cloud Monitoring Tools Comparison

Fatema et. al, have done a comprehensive survey of various monitoring tools, which here we put some of their findings. Table 4 demonstrates a

brief comparison between monitoring tools that we discussed in the previous sections of this chapter (i.e. general purpose and cloud specific).

Table 4: A brief comparison between different monitoring tools

<b>Tools</b>	<b>Monitored Resources</b>	<b>Drawbacks</b>	<b>OS Support</b>
<b>Nagios</b>	Network Applications System resources Services Sensors	Unable to bind to service after VM migration Difficult configurations Unable to operate at high sampling interval	Linux/Unix Windows through proxy
<b>Collectd</b>	Network Applications System resources Services Sensors Databases	No Graphical interface	Linux/Unix Mac OS
<b>Ganglia</b>	System Resources	Rigid to customize Overhead at both networks and hosts	Linux/Unix Mac OS Solaris Windows
<b>Amazon CloudWatch</b>	AWS resources	Security threats because of non-efficient utilization Limited only to AWS resources	Linux Windows
<b>Azure Watch</b>	Azure resources	Limited only to Azure resources Supports only Windows	Windows
<b>Nimsoft</b>	Network Applications Different cloud resources	Not fault tolerant No support for SLA agreement for data location Limited mostly to Web sites and services	Linux/Unix Mac OS Windows Netware



## 4 Design for Monitoring Service Chain

Monitoring service chain is essential in order to check the performance and health of every module in the system. In the service chain which is operating under cloud services, if any component goes down, then the whole service chain breaks, and the entire service will be unavailable for the customer. The consequence of this unavailability will be distrust from service subscriber towards the service provider and eventually damage to the reputation of its service and loss of income.

As discussed in the previous chapter, there are different monitoring solutions available in the market, however some of them are proprietary like Amazon CloudWatch, or Azure watch, and other open source systems are not tuned for monitoring the service chain. Due to this reason, we are introducing our own design for monitoring the service chain. Our monitoring system divides the monitoring into two subsystems. The first part is monitoring the cloud components and the network (i.e. flow controller and its tables) since the services are operating under the cloud system and communicate over OpenFlow switch. The second part is responsible for monitoring the services running inside our compute nodes, which provide the required service for the user.

Figure 19 shows our overall design for the cloud service and its integration with OpenFlow controller. As it is illustrated, our cloud system consists of three nodes, controller, network and compute nodes. Moreover, there is an OpenFlow controller responsible for controlling the flows in the OpenFlow switch. Instances or services that are running inside the compute node are all connected to an OpenFlow Switch, and their traffic is controlled by the OpenFlow controller.

Controller node is responsible for managing the network node and compute node. For the compute node, controller can create, delete, start, restart or shutdown an instance. For the network node, controller is responsible for creating router, managing subnets, assigning floating ip to the instances, etc. Aside from controlling the compute and network node, controller node is responsible for managing the databases, identifications, message broker, tenants, security groups, and operating system (OS) images that compute node needs for booting an instance. In general, controller, as it is clear from its name, is responsible for controlling the whole cloud system.

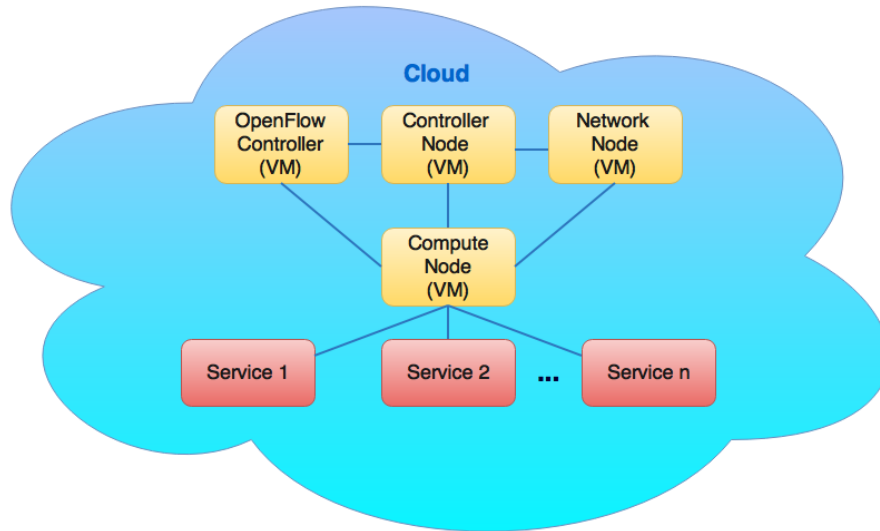


Figure 19: Overall Service Chain Design

The network node has the networking plug-in, layer 2 and 3 agents to control the network. Layer 2 agent is responsible for provisioning the tunnels and virtual networks. Layer 3 agent is responsible for the provisioning and operations of tenants network, and its services include routing, NAT, and DHCP. Network node is also responsible for handling the external network connectivity of virtual machines to the internet.

The compute node runs the hypervisor part of compute that operates tenants instances or virtual machines. By default the hypervisor in compute node is KVM, however in our design, we use linux containers. The compute node has the networking plug-in, and the layer 2 agent, which controls the tenant networks, and implements security groups. Since the compute node, is responsible for handling the instances/services, after limited amount of instances, it gets overloaded. And the exact amount of services that can operate simultaneously on a compute node is proportional to the hardware configuration and the amount resources dedicated to the compute node. However, compute node can get expanded horizontally to multiple compute nodes in different host machines.

OpenFlow controller is responsible for managing the flows through its

tables. Tables have the necessary rules for managing the service chain. Based on deployment architecture of service chain, these rules may vary. The OpenFlow controller also populates statistics for each port on the OpenFlow switch, and with these statistics, we can get how much traffic passed from specific port, and how many packets have dropped, which can help us in our monitoring system.

#### 4.1 Monitoring cloud

For monitoring cloud, the design should be in such a way that, the main monitor system takes control of every module in the cloud in order to prevent any service in the cloud to become unavailable or unresponsive. There are two key elements in monitoring that needs to be considered carefully. The first element is overhead produced by the monitoring system in the network, and the second element is availability of the monitoring system.

Many monitoring systems in order to check availability of the monitored system, use ICMP messages to ping the services. In this case, if there is any response from the service then they conclude that the service is available, otherwise they assume the service is down. Therefore, these systems mostly use the pulling mechanism to get informations from the service that they are monitoring. However, despite from functionality of this mechanism and its popularity, a lot of overhead is introduced in the network. Since, all the time the monitoring system pulls the information from the machines.

This mechanism works flawlessly if only couple of machines are monitored. However if it comes to hundreds or thousands of machines, then the network bandwidth and other resources are wasted in order to periodically pull information from every single machine to check whether they are healthy or not. By default each ICMP message is 32 bytes, if this message is sent to 1000 machines, then 32,000 bytes is sent to the machines requesting and almost the same amount is coming back to the requester as the reply, which in total can be around 64,000 bytes. This is only for each time pingging the machines, and if it is one time request then it would not have any impact on the network. However, if this ping message is sent every 5 seconds depending to the configuration of monitoring system, then in 24 hours 1105920000 bytes (1.1 Gigabytes) are sent in the network only to check if there is a response from the machines or not. This is just an example of 1000 machines, however cloud providers usually have more machines that need to be monitored.

Pull mechanism imposes a huge burden on the CPU of managers, and that is because of repetitive requests to the agents regarding their status. Likewise, the repetitive requests to the agents can cause the network overhead [Mar98]. Thus, reducing overhead is crucial in the network in order to have an optimized system for monitoring the service chain. For this reason, our design uses the pushing mechanism due to its efficiency and low overhead for the network and CPU. In this approach, there will be a light application running on each module of cloud as a daemon, monitoring the resources of that machines, (i.e. CPU usage, RAM usage, Network, etc.). If the usage of each of these resources exceed certain threshold, for a certain period of time, then it will send a message to the main monitoring system, that it is getting overloaded and backup is needed.

However, this question might arise, what if the whole virtual machine goes down suddenly, so the monitoring daemon will not have enough time to send any messages to the main monitoring system? This might be unlikely, since the whole point of monitoring is to prevent such disasters and inform the main controller before any possibility of failure. However, just to make sure nothing goes wrong, we have extended our design. Since all of our cloud components are running inside different Virtual Machines (VMs), then there will be a monitoring daemon running on the host machine, monitoring the VMs. Therefore, if any of these VMs go down, the monitoring daemon on the host machine will send an alert about the situation to the main monitoring system. Figure 20 depicts the design for the monitoring system.

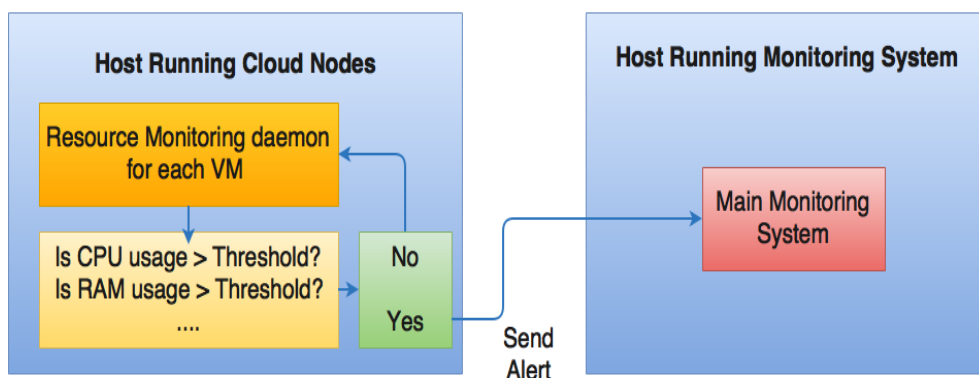


Figure 20: Monitoring Cloud Nodes Design

The monitoring daemon, which is running on the host machine, can

also monitor health of hard disk, RAM, and any other component that its failure can cause the whole host machine become unavailable. There is a trade-off between network overhead and checking the availability of hosts. If any of VMs go down, with the monitoring daemon on the host machine, a notification will be sent to the main monitoring system. However, if the host itself goes down, and we use push mechanism instead of pulling, then we are unable to detect it. This failure can only happen, if certain machine gets power shortage. Thus, power shortage should be detected through another mechanism and hardware, which is out of scope of this paper.

## 4.2 Monitoring Services and OpenFlow Controller

Monitoring services in the compute node that are running to provide a specific service for the customer is momentous. When services are linked together, they create a service chain. However, if one of the service nodes goes down and become unavailable then the whole chain breaks. Thus, by monitoring the services our goal is to minimize such tragedy.

The design idea behind our service chain is static, however in reality many networking companies are shifting towards dynamic service chaining. Since our goal here is not to create a new method for service chain, we decided to implement static chain for the simplicity of implementation, which has the similar fundamental nature as dynamic service chain, that forwards packets from one service to another and finally to the destination. Figure 21 shows our design for the service chain.

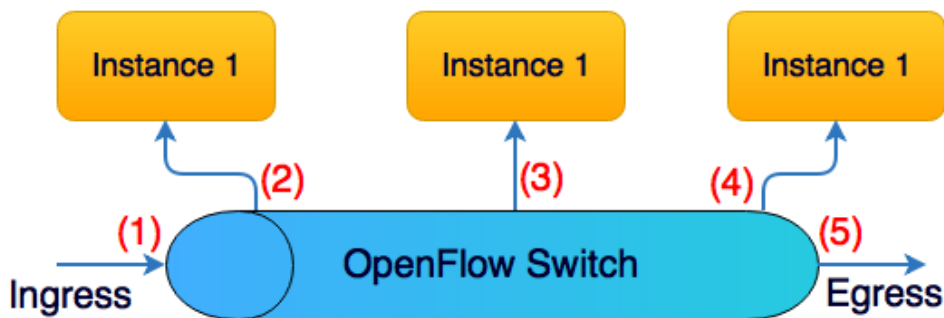


Figure 21: Service Chain Design

Based on this design, packets which are destined to the destination are take the path to the services that the user is subscribed for, then after

packet processing is done in each service the packets are forwarded to the destination. We use Type of Service (ToS) [Alm92] in IP header, in order to steer the traffic to the services, before sending them to the destination. In this method, based on the services that the user is subscribed, the port number and mac address is changed to the mac address of desired service node, in order to steer the traffic to that node, after the process is done, packets are sent back to the OVS, and if another service needs to process the packets, then the packets are steered through that service, otherwise the packets will be sent to the destination.

#### 4.2.1 Monitoring Containers through Control Groups (cgroups)

Linux control groups (cgroups) partition tasks into different groups that have specific behaviour, which affects their children, if any, in the hierarchy [Men15]. Linux containers, are process groups in linux, where each container is an isolated process with its own cgroup. In our design for the monitoring services we use the method of monitoring cgroups from [KISdL15] to monitor the containers that services are running on them for few different reasons.

The first reason is that, even if the container itself is unresponsive, we still can find the necessary information that we want, for instance we can check if our IPS process id is still in the cgroup or not. The second reason is, we do not need to install any special software with different required libraries on every container node to monitor them. And the third reason is that, even if the container instance gets compromised, still we can monitor it, since we are collecting the data outside of the container. However, the only case that we cannot monitor the container is, when the container gets compromised and get access to the host context.

Through this approach we gather resource utilization of each container periodically. However, containers that are monitored should be mapped to the corresponding cloud instance. If the resource usage is above specified threshold then the monitoring system sends an alert that specific service is overloaded and backup service is needed to balance the load. The monitoring system receives the alert, investigates the type of service that requires load balance, finds suitable image which has that specific service installed on it, and starts the service, or creates the service if there are not any services available. After new service is created, necessary actions should be taken through the OpenFlow controller to direct the flows to the new initiated

service. Figure 22 illustrates the flowchart for monitoring daemon for the services in the compute node. And Figure 23 demonstrates the flowchart of main monitoring system, which takes care of automatic scale up and down for the services.

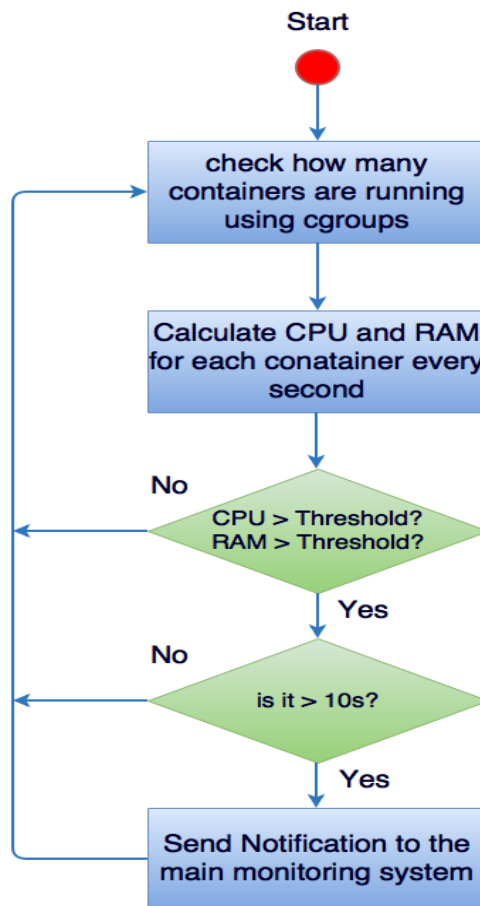


Figure 22: Monitoring Services Flowchart

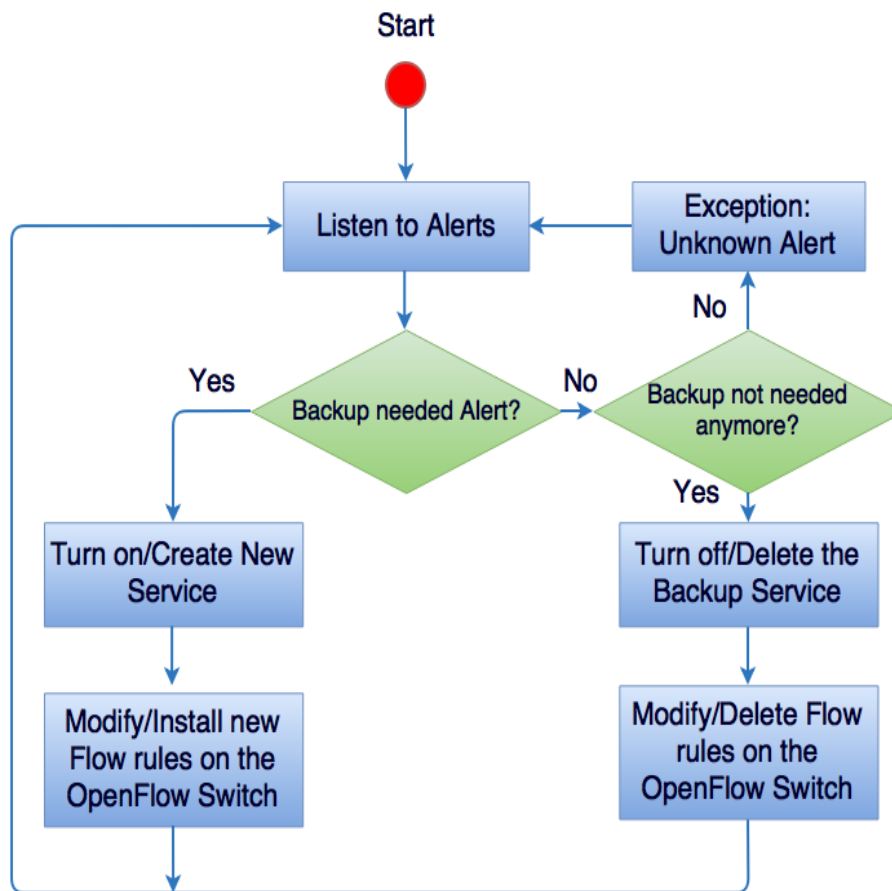


Figure 23: Main Monitoring Service Flowchart for Auto Scale up/down

However, if the new backup service is idle for certain period of time, then it will send an alert to the monitoring system that it has been idle for certain amount of time, with very low CPU usage. The monitoring system then will request from main service (requester service) its current resource usage. Therefore, if monitoring system realizes that the main service is near the heavy load threshold, it will keep the backup service on, otherwise it will shut it down to save resources. Figure 24 shows the flowchart for turning off an unneeded backup service.



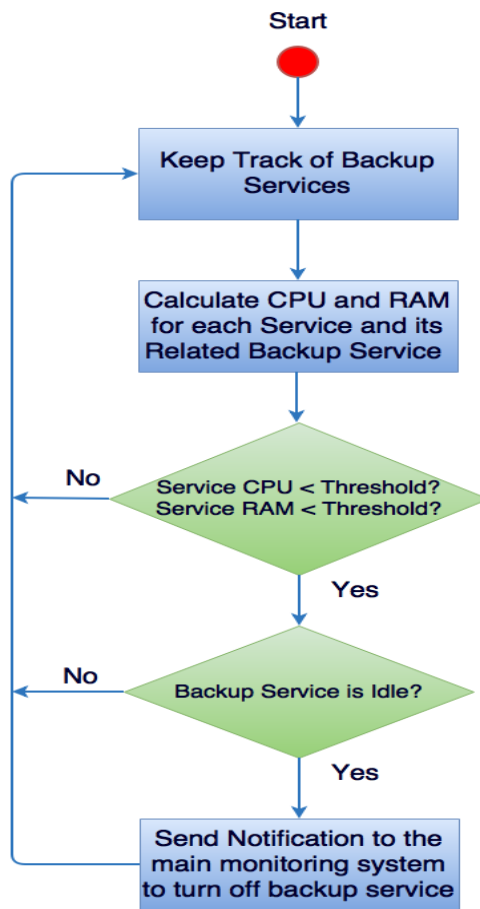


Figure 24: Turning off a service Flowchart

#### 4.2.2 Monitoring OpenFlow Controller and its Statistics

OpenFlow controller is responsible for the OpenFlow rules and managing them. With the rules, we can specify that packets go through specific nodes before forwarding them to the destination. Therefore, monitoring the OpenFlow controller is essential, because if the controller becomes unresponsive or its entire node goes down, then whole packets will drop or take the default route without being steered to specified service nodes.

For monitoring the controller itself, our design, takes the similar approach as monitoring cloud nodes. Therefore, we measure CPU usage, RAM usage and Network traffic in the OpenFlow controller node, and if resource usage goes above certain threshold for certain amount of time then we notify the main monitoring system about the situation and send an alert to it. Main

monitoring system can take the necessary action to handle the situation.

One desirable function in the OpenFlow controllers is that they provide easy access to the statistics of OpenFlow switch through their APIs. With these statistics we can monitor the network that our services are running in the compute node. We can extract the information that how much traffic has passed from specific port in the OVS. In addition, we can observe how many packet drops each port have had so far, in order to make a prediction that for certain hour in the day we need extra backup services to balance the load. Consequently, we pull the statistics from the OVS less frequently, after certain period of time, that we have gathered information about behavior of our network and services.

## 5 Monitoring System Implementation and Evaluation

In this chapter we discuss the implementation of designed platform for the monitoring service chain. First, we illustrate that how we implemented this platform, and what tools we have used, then we demonstrate different screen shots of working system in different scenarios. Lastly we evaluate our system by showing the latency, uptime and the overhead that it imposes on the cloud components and instances.

### 5.1 System Implementation

Our system contains three core components, which are main monitoring system, services monitoring daemon (i.e docker instances), and KVM monitoring daemon for each one of the KVM nodes. The main monitoring system has all the logics, and has control over the cloud. KVM monitoring daemon which is running on cloud nodes (i.e. compute, network, and controller) as well as OpenFlow controller (i.e OpenDaylight) node is responsible for monitoring the resources on each of these KVM nodes. Furthermore, service monitoring daemon is responsible for monitoring the resource utilization of docker containers through docker API, which access cgroups. Figure 25 demonstrates the basic structure of this system.

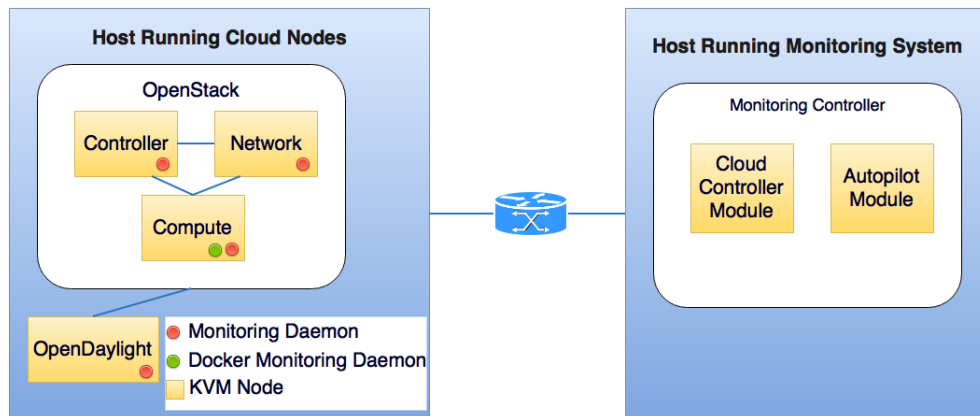


Figure 25: Overall System Implementation

As it is illustrated in the Figure 25 OpenStack components are running

on three KVM nodes, as well as OpenDaylight on a separate node. The autopilot module on the monitoring system is responsible for turning on new instances, which we will discuss more in upcoming sections. The cloud controller module is responsible for connecting to the OpenStack controller node, in order to take control over the various operations.

### 5.1.1 Software and Hardware

For the cloud side we use OpenStack since it is open source with the huge community support. Based on the hardware that we had for this development, we put all the OpenStack components on a single machine including the OpenDaylight component. The machine is running ubuntu 14.04 LTS, with 16GB of RAM and intel core i7-4790 CPU 3.60GHz. The main monitoring system is running on a HP laptop running ubuntu 14.04 LTS, with 8GB of RAM and intel core i5-2540 2.60GHz

We use KVM for running the cloud components as well as OpenFlow controller, and each machine comes with its own configuration for the resources that it is dedicated to it based on its requirements. For the controller node we assigned 3GB of RAM and 2 cores out of 8. Compute node has 3.5GB of RAM with 2 CPU cores. The network node comes with 3GB of RAM and 1 CPU core. And OpenDaylight node is tuned to have 3GB of RAM with 2 CPU cores.

All the applications main development language is Java, with Spring framework. For the OpenDaylight and OpenStack KVM nodes, we use Sigar API in order to get resource usages (i.e RAM and CPU). For the compute node, we use nova-docker module instead of KVM for the service virtualization, which enables OpenStack to run docker containers. Therefore, for their resource calculation we use docker API to get the data, which is from cgroups.

### 5.1.2 RAM and CPU calculation

The heart of our monitoring system is the RAM and CPU usage, which via these data we can determine if a virtual machine or cloud instance is under a lot of pressure or not. For the resource utilization of docker containers, we have referred to their "docker stats" git repository [Doc16], in order to find the formula behind "docker stats" command, which is used for displaying

the docker resource usages (i.e. CPU and RAM). Here is the formula for calculating the CPU and RAM usage in percentage:

$$cpuPercentage = \frac{cpuDelta}{systemDelta} \times 100 \quad (1)$$

$$ramPercentage = \frac{memoryUsage}{memoryLimits} \times 100 \quad (2)$$

Here *cpuDelta* is the current total CPU usage minus *previousTotalCPUusage*, depending how regularly we are checking the CPU. And *systemDelta* is current system CPU usage minus *previousSystemCPUusage*. Moreover, in the second formula, *memoryLimits*, refers to the memory limitation that is imposed to each container, in order to use up to certain amount of memory. In our system we check CPU and RAM usage of each container in the interval of 1000 milliseconds (1 second).

In the developed system, if the CPU or RAM usage exceeds to more than 80% for more than 10 seconds, then the monitoring daemon will send an alert notification to the main monitoring system. The reason for waiting 10 seconds is to make sure the service is under persistent pressure and not just a sudden temporary burst. In the next subsection we discuss about the messaging mechanism between monitoring daemons and the main monitoring system.

### 5.1.3 Communication Architecture and Messages

For messaging and communication between all the client and servers we use REST architecture due to its efficiency and simplicity to implement. REST web architecture first was introduced by Roy Fielding. REST uses URI in order to manipulate, or fetch the data from a web server. To do that, it uses HTTP interfaces which are limited to GET, PUT, POST, and DELETE. With the GET request data from web server can be obtained for processing or displaying it to the user. PUT messages are responsible for creating data entry in the web server. In addition, POST messages are similar to PUT messages, with the difference that POST can update the data in the web server as well as creating it. Lastly, DELETE messages are used for deleting the data from the web server. REST uses XML or JSON data format for sending and receiving the data to/from the web server. [He03]

Currently almost all applications have support for REST and they pro-

vide REST-API for programmers in order to have unified procedure to communicate with their application. In our environment, all the components including OpenStack, OpenDaylight, Docker, monitoring daemon, and main monitoring system have support for REST-API which we utilize it as primary communication method. Table 5 illustrates the messages that goes from monitoring system to OpenStack controller node. Table 6 shows the message which goes from monitoring daemon to the main monitoring system. And Table 7 illustrates the messages that goes from the docker monitoring daemon to the main docker server and the main monitoring system.

Table 5: Messages from Monitoring System to OpenStack Controller

Message	Argument(s)	Type	Description
AUTHENTICATION	username password tenant name	POST	Sends authentication request to the OpenStack controller and receives token in response
SERVER_DETAILS	token admin_url	GET	Gets the details of all the instances
SWITCH_INSTANCE	token instance_address action	POST	Turns on or off an instance in the cloud
CREATE_INSTANCE	token name network_id image_id flavor_id	POST	Sends a request to create new instance
NETWORK_ID	token	GET	Gets the network id of OpenStack controller
IMAGE_ID	token	GET	Gets the glance image IDs of OpenStack controller
FLAVOUR_ID	token	GET	Gets the flavour IDs of OpenStack controller
DELETE_INSTANCE	token instance_id	DELETE	Deletes specified instance

Table 6: Message between Monitoring System and Monitoring Daemon

Message	Argument(s)	Type	Description
ALERT_CONTROLLER	authentication vm_hostname vm_cpu_ percentage vm_ram_ percentage	POST	Send an alert message to the monitoring system that a VM has high CPU or/and RAM usage

Table 7: Messages between Monitoring System, Docker server and Docker Monitoring Daemon

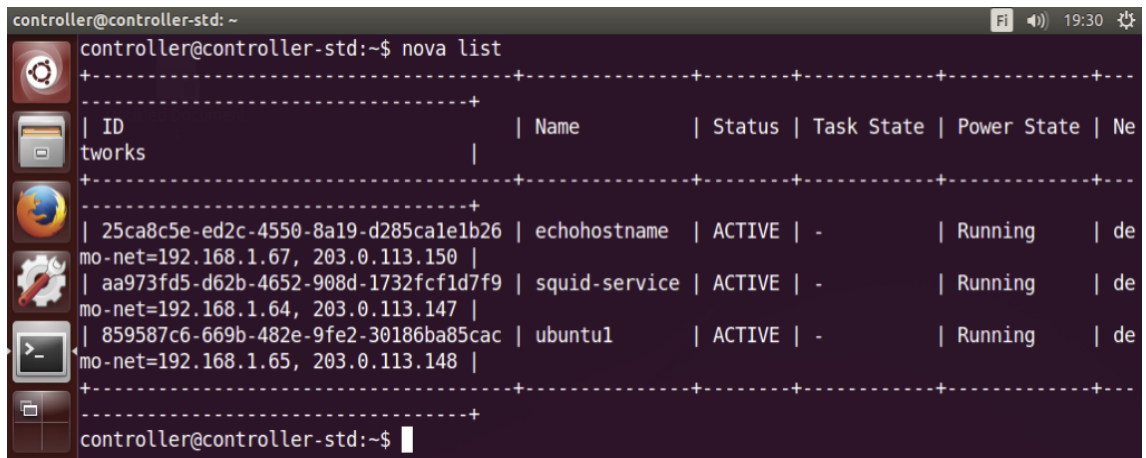
Message	Argument(s)	Type	Description
ALERT_CONTROLLER	authentication docker_instance_ hostname docker_instance_ cpu_percentage docker_instance_ ram_percentage	POST	Send an alert message to the monitoring system that a docker instance of OpenStack has high CPU or/and RAM usage
CONTAINER_DETAILS	-	GET	Gets details of all the docker instances which are running, from the docker server
COTAINER_STATS	container_id	GET	Gets the raw CPU and RAM data to calculate the RAM and CPU usage in percentage from the docker server

## 5.2 Monitoring Service Chain in Action

In this section we show our system in action, therefore we illustrate the platform through various screen shots that have been taken during project runtime. First we show how our service chain operates with different screen shots that are taken during packet forwarding from one service to another. Then later, we show the main monitoring system in action specially the platform's reaction when it receives an alert from docker monitoring daemon.

### 5.2.1 Service chain with Squid proxy

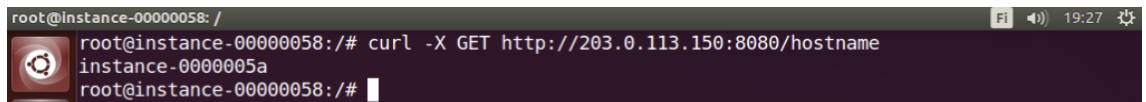
In order to achieve service chaining we have developed a small application that whenever receives a message requesting host name, it will respond with the host name of the machine that it is running on. We have three instances that are running with the naming of *ubuntu1*, *squid-proxy* and *echohostname*. *ubuntu1* requests the host name of *echohostname* and it replies its host name to the *ubuntu1*. Figure 26 demonstrates the list of instances which are running in our cloud along with their IP addresses.



```
controller@controller-std: ~
controller@controller-std:~$ nova list
+-----+-----+-----+-----+-----+-----+
| ID                | Name          | Status | Task State | Power State | Ne
+-----+-----+-----+-----+-----+-----+
| 25ca8c5e-ed2c-4550-8a19-d285ca1e1b26 | echohostname | ACTIVE | -          | Running     | de
mo-net=192.168.1.67, 203.0.113.150 |
| aa973fd5-d62b-4652-908d-1732fcf1d7f9 | squid-service | ACTIVE | -          | Running     | de
mo-net=192.168.1.64, 203.0.113.147 |
| 859587c6-669b-482e-9fe2-30186ba85cac | ubuntu1       | ACTIVE | -          | Running     | de
mo-net=192.168.1.65, 203.0.113.148 |
+-----+-----+-----+-----+-----+-----+
controller@controller-std:~$
```

Figure 26: List of running docker instances in the OpenStack

However to show how service chain works in action, with the aid of OpenDaylight we force all the incoming packets from *ubuntu1* port go to the *squid-proxy* instance instead of taking the default route. Therefore in this situation, identity of the instance that is requesting host name will be masked by the identity of *squid-proxy* instance. Figure 27 illustrates the terminal message that goes from *ubuntu1* instance and its response from *echohostname*.



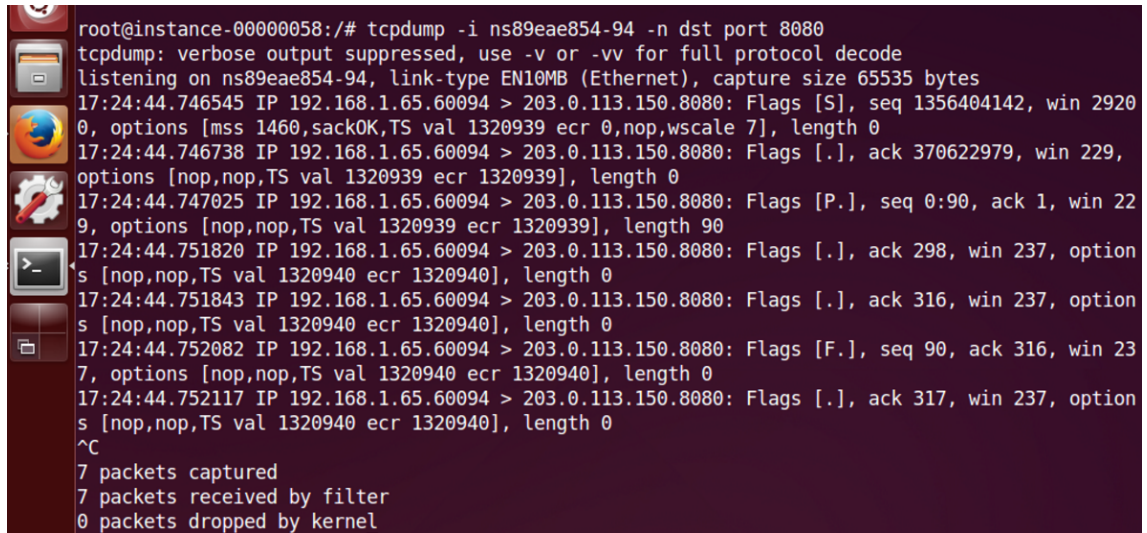
```
root@instance-00000058: /
root@instance-00000058:/# curl -X GET http://203.0.113.150:8080/hostname
instance-0000005a
root@instance-00000058:/#
```

Figure 27: Ubuntu1 requesting host name

Figure 28 demonstrates that the packets are sent from 192.168.1.65

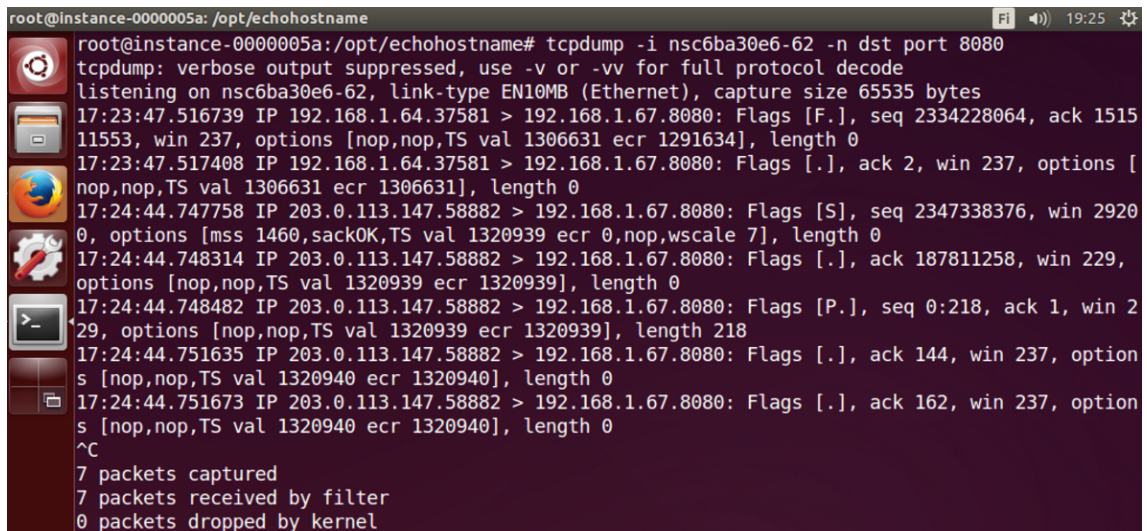


which is the private IP address of "ubuntu1". However tcpdump messages of "echohostname" in Figure 29 shows that the incoming packets are coming from 203.0.113.147 which is the IP address of "squid-proxy". Moreover, Figure 30 shows the log of messages that are sent to it.



```
root@instance-00000058:/# tcpdump -i ns89eae854-94 -n dst port 8080
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ns89eae854-94, link-type EN10MB (Ethernet), capture size 65535 bytes
17:24:44.746545 IP 192.168.1.65.60094 > 203.0.113.150.8080: Flags [S], seq 1356404142, win 2920
0, options [mss 1460,sackOK,TS val 1320939 ecr 0,nop,wscale 7], length 0
17:24:44.746738 IP 192.168.1.65.60094 > 203.0.113.150.8080: Flags [.], ack 370622979, win 229,
options [nop,nop,TS val 1320939 ecr 1320939], length 0
17:24:44.747025 IP 192.168.1.65.60094 > 203.0.113.150.8080: Flags [P.], seq 0:90, ack 1, win 22
9, options [nop,nop,TS val 1320939 ecr 1320939], length 90
17:24:44.751820 IP 192.168.1.65.60094 > 203.0.113.150.8080: Flags [.], ack 298, win 237, option
s [nop,nop,TS val 1320940 ecr 1320940], length 0
17:24:44.751843 IP 192.168.1.65.60094 > 203.0.113.150.8080: Flags [.], ack 316, win 237, option
s [nop,nop,TS val 1320940 ecr 1320940], length 0
17:24:44.752082 IP 192.168.1.65.60094 > 203.0.113.150.8080: Flags [F.], seq 90, ack 316, win 23
7, options [nop,nop,TS val 1320940 ecr 1320940], length 0
17:24:44.752117 IP 192.168.1.65.60094 > 203.0.113.150.8080: Flags [.], ack 317, win 237, option
s [nop,nop,TS val 1320940 ecr 1320940], length 0
^C
7 packets captured
7 packets received by filter
0 packets dropped by kernel
```

Figure 28: TcpDump messages in "ubuntu1" terminal



```
root@instance-0000005a: /opt/echohostname
root@instance-0000005a:/opt/echohostname# tcpdump -i nsc6ba30e6-62 -n dst port 8080
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on nsc6ba30e6-62, link-type EN10MB (Ethernet), capture size 65535 bytes
17:23:47.516739 IP 192.168.1.64.37581 > 192.168.1.67.8080: Flags [F.], seq 2334228064, ack 1515
11553, win 237, options [nop,nop,TS val 1306631 ecr 1291634], length 0
17:23:47.517408 IP 192.168.1.64.37581 > 192.168.1.67.8080: Flags [.], ack 2, win 237, options [
nop,nop,TS val 1306631 ecr 1306631], length 0
17:24:44.747758 IP 203.0.113.147.58882 > 192.168.1.67.8080: Flags [S], seq 2347338376, win 2920
0, options [mss 1460,sackOK,TS val 1320939 ecr 0,nop,wscale 7], length 0
17:24:44.748314 IP 203.0.113.147.58882 > 192.168.1.67.8080: Flags [.], ack 187811258, win 229,
options [nop,nop,TS val 1320939 ecr 1320939], length 0
17:24:44.748482 IP 203.0.113.147.58882 > 192.168.1.67.8080: Flags [P.], seq 0:218, ack 1, win 2
29, options [nop,nop,TS val 1320939 ecr 1320939], length 218
17:24:44.751635 IP 203.0.113.147.58882 > 192.168.1.67.8080: Flags [.], ack 144, win 237, option
s [nop,nop,TS val 1320940 ecr 1320940], length 0
17:24:44.751673 IP 203.0.113.147.58882 > 192.168.1.67.8080: Flags [.], ack 162, win 237, option
s [nop,nop,TS val 1320940 ecr 1320940], length 0
^C
7 packets captured
7 packets received by filter
0 packets dropped by kernel
```

Figure 29: TcpDump messages in "echohostname" terminal

```
root@instance-0000057: /var/log/squid3
root@instance-0000057:/var/log/squid3# cat access.log
1456505867.509      1 192.168.1.65 TCP_DENIED/403 3556 GET http://203.0.113.150:8080/hostname -
HIER NONE/- text/html
1456506475.101      0 192.168.1.65 TCP_DENIED/403 3556 GET http://203.0.113.150:8080/hostname -
HIER NONE/- text/html
1456506637.828      2 192.168.1.65 TCP_DENIED/403 3556 GET http://203.0.113.150:8080/hostname -
HIER NONE/- text/html
1456506986.818      5 192.168.1.65 TCP_MISS/200 315 GET http://203.0.113.150:8080/hostname - HI
ER_DIRECT/203.0.113.150 text/plain
1456507002.854      4 192.168.1.65 TCP_MISS/200 315 GET http://203.0.113.150:8080/hostname - HI
ER_DIRECT/203.0.113.150 text/plain
1456507094.240      5 192.168.1.65 TCP_MISS/200 315 GET http://203.0.113.150:8080/hostname - HI
ER_DIRECT/203.0.113.150 text/plain
1456507200.715      4 192.168.1.65 TCP_MISS/200 315 GET http://203.0.113.150:8080/hostname - HI
ER_DIRECT/203.0.113.150 text/plain
1456507259.210      4 192.168.1.65 TCP_MISS/200 315 GET http://203.0.113.150:8080/hostname - HI
ER_DIRECT/203.0.113.150 text/plain
1456507367.527      3 192.168.1.65 TCP_MISS/200 315 GET http://192.168.1.67:8080/hostname - HI
ER_DIRECT/192.168.1.67 text/plain
1456507484.751      4 192.168.1.65 TCP_MISS/200 315 GET http://203.0.113.150:8080/hostname - HI
ER_DIRECT/203.0.113.150 text/plain
```

Figure 30: Squid Log

For redirecting the flows to *"squid-proxy"*, we had to add new flow to the flow table of Open vSwitch, and we achieve this by sending REST PUT message to the OpenDaylight controller. Figure 31 shows the XML code that we put in the flow table. *"in-port"* is the port number that the packets are coming from which is *"ubuntu1"*, *"output-node-connector"* is the port number that we want packets to go there which is the port number of *"squid-proxy"* and *"address"* is the mac address of our *"squid-proxy"* instance.

```

<?xml version="1.0" encoding="UTF-8"?>
<flow
  xmlns="urn:opendaylight:flow:inventory">
  <priority>35000</priority>
  <flow-name>Foo</flow-name>
  <match>
    <ethernet-match>
      <ethernet-type>
        <type>2048</type>
      </ethernet-type>
    </ethernet-match>
    <in-port>9</in-port>
  </match>
  <id>1</id>
  <table_id>0</table_id>
  <instructions>
    <instruction>
      <order>0</order>
      <apply-actions>
        <action>
          <order>0</order>
          <set-dl-dst-action>
            <address>fa:16:3e:bb:4a:f8</address>
          </set-dl-dst-action>
        </action>
        <action>
          <order>1</order>
          <output-action>
            <output-node-connector>
              12
            </output-node-connector>
          </output-action>
        </action>
      </apply-actions>
    </instruction>
  </instructions>
</flow>

```

Figure 31: OpenDaylight XML code for redirecting packets

### 5.2.2 Main Monitoring System

In this section we demonstrate how our system operates, and what actions it takes in order to avoid the service chain to break down. Figure 32 shows the application user interface, which is divided into two columns. The left column includes Authentication, Properties, Create Instance and Attach IP. The right column will display all the instances and responses that comes

from the OpenStack controller.

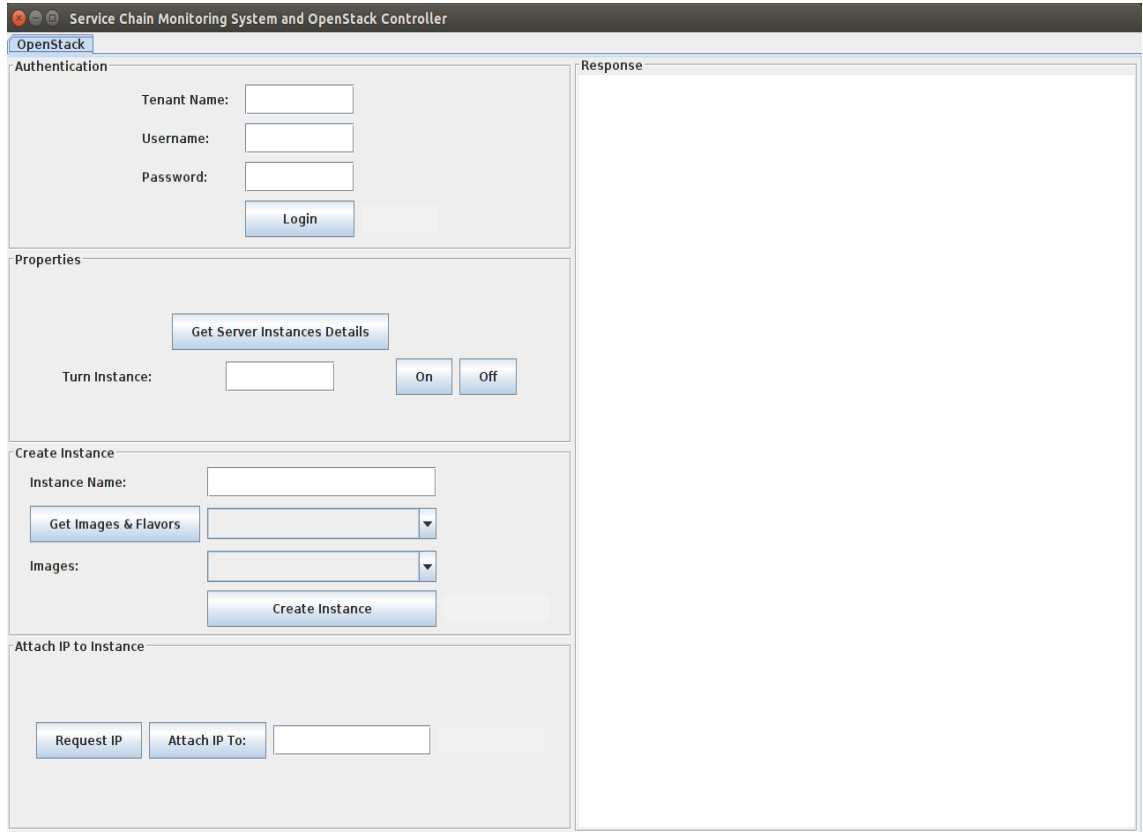


Figure 32: Platform User Interface

In order to communicate with the OpenStack controller, we have to login with the credentials, which are all *"demo"* in this case. After login we are able to do any operations such as listing all the available instances, turning on/off an instance, or creating new instance. Figure 33 illustrates the Response after we login and request for list of available instances. In this case we have only one instance running with the name of *"ubuntu1"*.

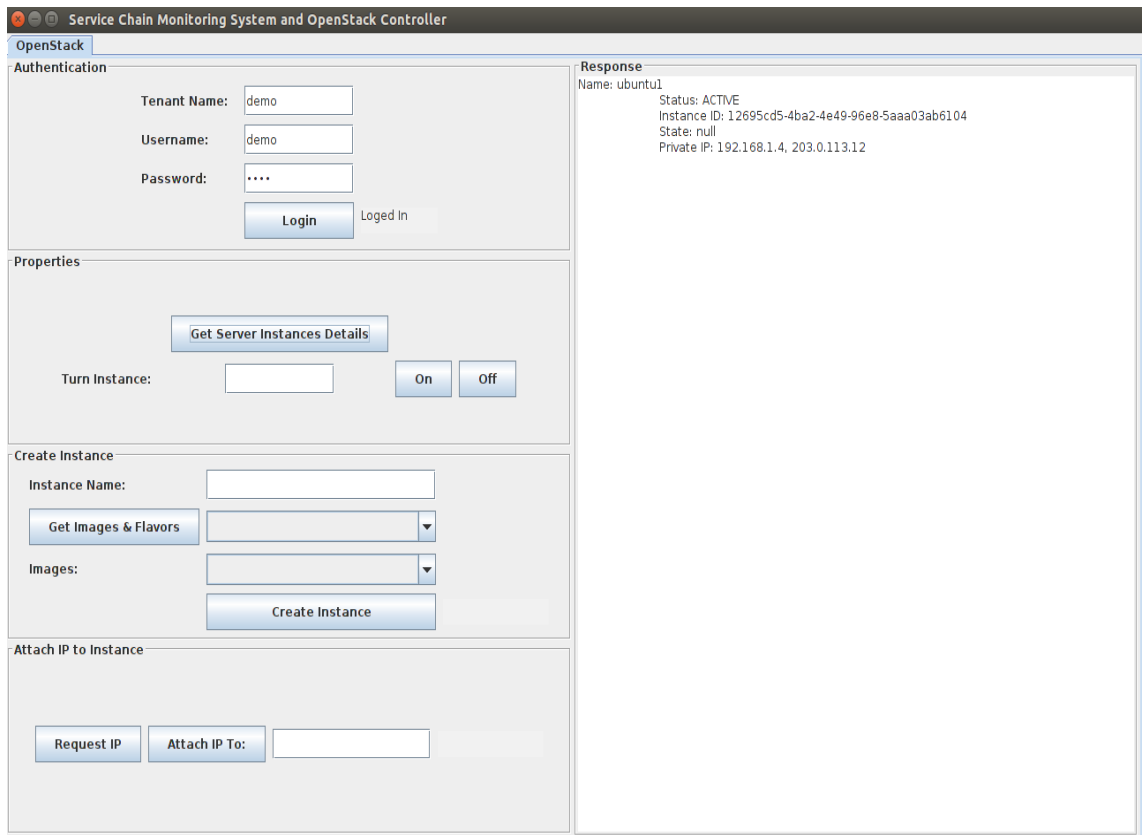


Figure 33: Listing available instances

With the system we can create new instances remotely and assign a public IP to them. For doing that we have to choose a name for the new instance, and choose what flavor and image type we want it to boot from, and then create the instance. Here for our OpenStack flavors, we assigned 512MB of RAM and 1GB of storage space to the "m1.tiny" which we use it for creating all the instances during this demo. Figure 34 shows creating new instance and assigning public IP to it in action.

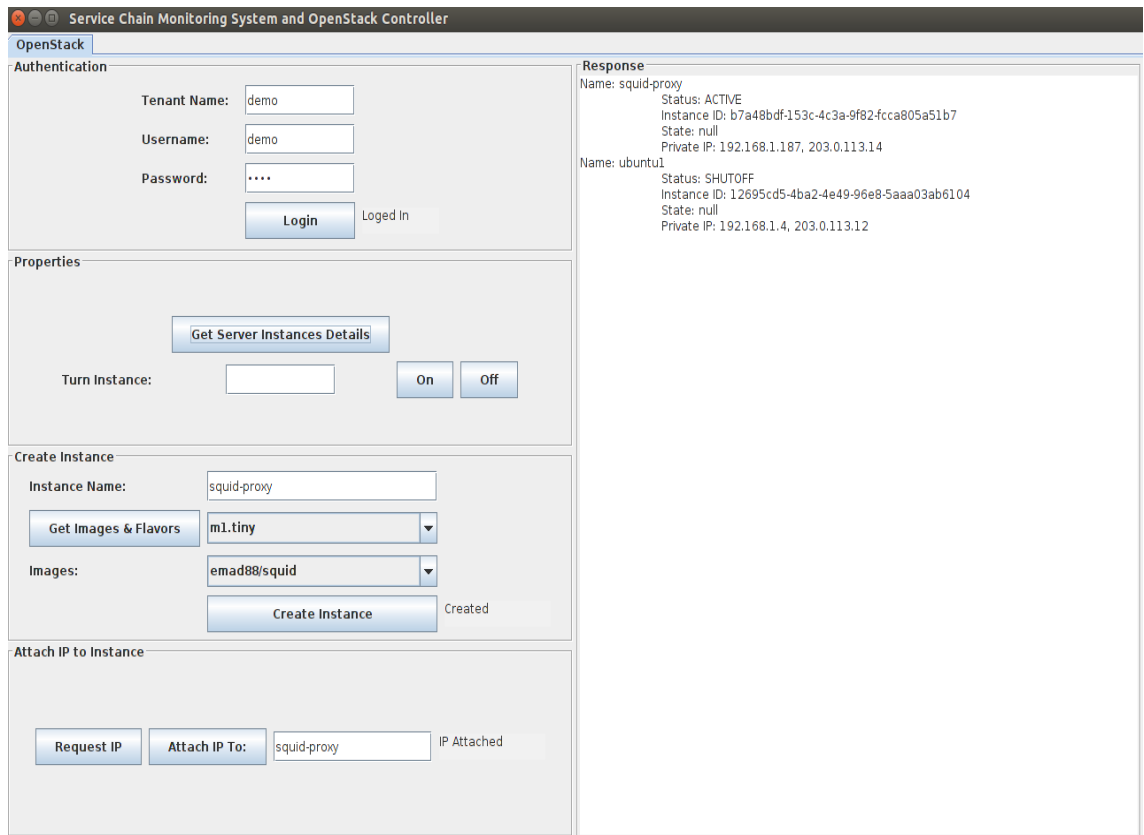


Figure 34: Creating new instance and assigning public IP to it

The module that makes this platform responsive to alerts from instances and virtual machines, is called "AutoPilot". With the aid of this module, whenever help is needed from our services in the cloud, the system provides assistance for them automatically. Therefore, in this case, the main monitoring system receives a message from the docker monitoring daemon that a specific service is seeking help. The monitoring system's console shows which instance needs help, which is shown in Figure 35, the id that needs help belongs to "ubuntu1".

```
Run ApplicationMain
2016-03-09 16:46:12.298 INFO 12026 --- [nio-8080-exec-1] o.a.c.c
2016-03-09 16:46:12.299 INFO 12026 --- [nio-8080-exec-1] o.s.web
2016-03-09 16:46:12.309 INFO 12026 --- [nio-8080-exec-1] o.s.web
Instance id requesting help: 12695cd5-4ba2-4e49-96e8-5aaa03ab6104
```

Figure 35: System's console showing the ID of the service that needs help

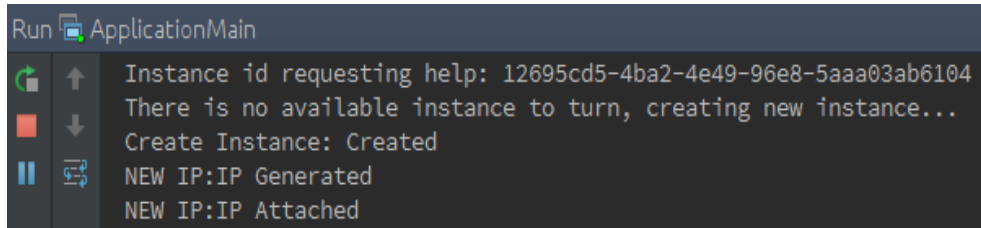
After the system receives this message, it will investigate what kind of flavor and image type this instance is using, in order to find suitable instance to turn on. If the system finds any instance which is turned off (available to use), and have the same type of flavor and image, then it will turn on that instance to decrease the load of the instance which needs help. As Figure 36 illustrates, the application found that "ubuntu2" has the same image and flavor type as "ubuntu1" and it turns it on.

```
Response
Name: ubuntu2
  Status: SHUTOFF
  Instance ID: 7bad8a91-a141-41b0-9c3d-99240b78000b
  State: powering-on
  Private IP: 192.168.1.188,
Name: squid-proxy
  Status: SHUTOFF
  Instance ID: b7a48bdf-153c-4c3a-9f82-fcca805a51b7
  State: null
  Private IP: 192.168.1.187, 203.0.113.14
Name: ubuntu1
  Status: ACTIVE
  Instance ID: 12695cd5-4ba2-4e49-96e8-5aaa03ab6104
  State: null
  Private IP: 192.168.1.4, 203.0.113.12
```

Figure 36: Turning on new instance automatically

Though, if the system cannot find any suitable or available instance to turn on, it will start creating new instance identical to the instance requesting help. After the instance totally gets ready and boot up, the system will automatically assign a public IP to it. Figure 37 shows the system's console, when it receives message from an instance requesting for help. After the

system finds that there is no available instance to turn on, it displays in the console that there is no available instance, and it is processing to create new instance. Consequently Figure 38 displays the new created instance that has *"-ubuntu2"* as its suffix, to show this instance was created based on the request of *"ubuntu2"*.



```
Run ApplicationMain
Instance id requesting help: 12695cd5-4ba2-4e49-96e8-5aaa03ab6104
There is no available instance to turn, creating new instance...
Create Instance: Created
NEW IP:IP Generated
NEW IP:IP Attached
```

Figure 37: System's console showing there is no available instance



```
Response
Name: autoCreated1539-ubuntu2
  Status: ACTIVE
  Instance ID: e5bd0c41-e454-412f-b49b-dd97e07a4276
  State: null
  Private IP: 192.168.1.189, 203.0.113.15
Name: ubuntu2
  Status: ACTIVE
  Instance ID: 7bad8a91-a141-41b0-9c3d-99240b78000b
  State: null
  Private IP: 192.168.1.188,
Name: squid-proxy
  Status: SHUTOFF
  Instance ID: b7a48bdf-153c-4c3a-9f82-fcca805a51b7
  State: null
  Private IP: 192.168.1.187, 203.0.113.14
Name: ubuntu1
  Status: ACTIVE
  Instance ID: 12695cd5-4ba2-4e49-96e8-5aaa03ab6104
  State: null
  Private IP: 192.168.1.4, 203.0.113.12
```

Figure 38: Response panel of system, showing the new created instance

In addition, if one of our cloud nodes (i.e. controller, compute, network) or OpenFlow controller node (i.e. OpenDaylight) due to some reason have high CPU or RAM usage, the monitoring daemon will send an alert message to the main monitoring system. However due to time limitation that we had for this project, we just show the message, and not implemented any



module to take care of this situation. Figure 39 demonstrates the notification message that our system shows in case one of the KVM nodes goes under heavy load. For simulation purpose of heavy load, we use an application in ubuntu which is called "*stress*". With "*stress*" the user can define how long and how much CPU and RAM should be used to put the system under the pressure.

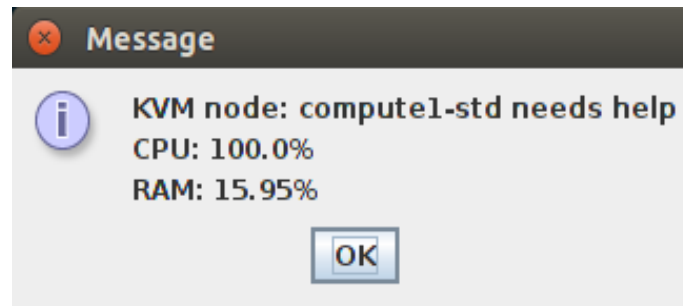


Figure 39: System showing a message that a KVM node is heavy loaded

### 5.3 Evaluation

In this section we evaluate our system by different metrics such as the overhead that monitoring daemons impose on the KVM nodes, and how long it takes for our system to react to the situations where a service needs help.

#### 5.3.1 Overhead

It is essential that the daemons that are running in each KVM node to have low overhead, so we would not have any trade off between performance and reliability of nodes. Therefore, we check their resource consumption when they are running on each node in order to investigate if they utilize great deal of resources or not. For comparison, we decided to compare our developed java monitoring daemon with *collectd* which is a popular and open source monitoring system. In order to check the overhead of each of these systems we use two simple linux commands, first we use "*ps aux | grep SYSTEM\_NAME*" to get the process ID (PID) of running daemons, then we use "*top -p PROCESS\_ID*" command to get the CPU and RAM usages of the daemons in percentage. Figure 40 demonstrates two running systems and their resource usage.

```

compute1@compute1-std: /tmp/collectd-5.4.1
top - 16:27:54 up 54 min, 5 users, load average: 0,07, 0,20, 0,30
Tasks:  2 total,   0 running,  2 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1,7 us,  1,3 sy,   0,0 ni, 95,5 id,  1,5 wa,   0,0 hi,   0,0 si,   0,0 st
KiB Mem:  3511436 total, 2537772 used,  973664 free, 179892 buffers
KiB Swap: 5116924 total,    0 used, 5116924 free. 1264836 cached Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 15123 root        20   0  914212  6288  5236  S   1,3   0,2   0:08.94 collectd
 15489 compute1    20   0 1985300 70568 15784  S   0,3   2,0   0:01.60 java

```

Figure 40: Getting resource usages of running daemons

For the purpose of fair comparison, we disabled all the unnecessary plugins of collectd and just kept memory and CPU monitoring plugins enabled. Both systems are configured to get the CPU and memory usage in interval of one second. We recorded the memory and CPU consumption of both systems for 60 minutes. Figure 41 shows the CPU usage of both systems. As it is demonstrated our developed system has lower CPU consumption and it is more steady compare to collectd.

The average CPU usage of collectd in one hour is 1.04% and the average CPU usage of developed system is 0.29%. However, as it is illustrated in Figure 42, our system has more memory consumption compare to collectd. The average memory usage of collectd is 0.2% compared to developed system which is 2.0%. The reason that our system consumes more memory is because it is written in java while collectd is written in C programming language which has lower memory consumption compare to java since JVM requires more memory.

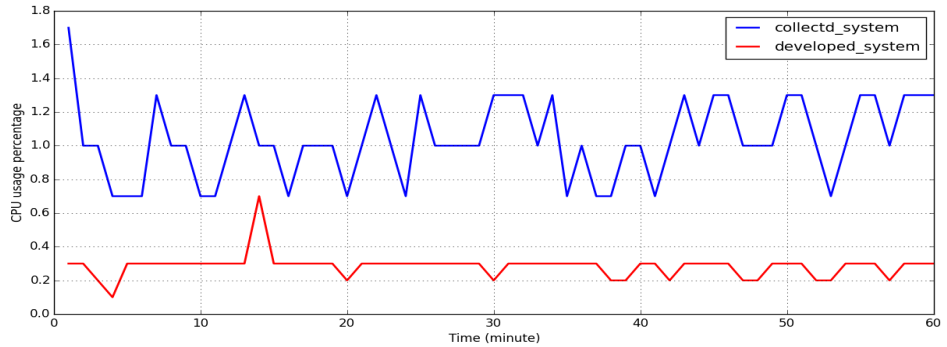


Figure 41: CPU usage of collectd vs. developed system

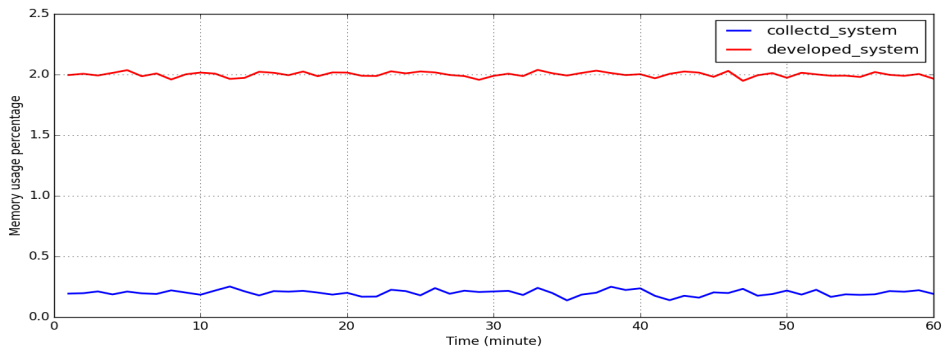


Figure 42: Memory usage of collectd vs. developed system

Advantage of our system is that it can be started easily on any machine that supports java without the need for configuration and hassle for installation. However, collectd needs many libraries and configuration to work properly. Collectd is not designed to handle OpenStack instances and report their behaviour if they use excessive resources, and plugin development might be needed for that purpose. On the other hand, our system is developed to work with any operating systems that supports java as well as docker containers. Collectd is more suitable for general environments which just monitoring resource consumption and reporting it to the admin is sufficient.

### 5.3.2 Monitoring System Response Time

In this section, we discuss the response time that the monitoring system takes in case of receiving alert message from one of the services in the cloud. At first we evaluate the situations where we have available instances and we only need to turn them on. Later, we evaluate the system in situations where there is no available instance, and the system should create a new one. The reason for evaluating the response time is to investigate how quick our system can react to prevent services from failure and breaking the chain.

For the purpose of this evaluation, we repeat the process of turning on an instance and creating new instance for 250 times, where the standard deviation becomes steady. Our computing node with its current configuration can handle only 9 instances to run simultaneously. Figure 43 shows that the standard deviation for turning on an instance, when we only want to turn on 1 instance which is shown with blue dashed line. When we have 8 instances running and one of the instances request for help to turn on the 9th instance, dashed magenta line is used to represent that.

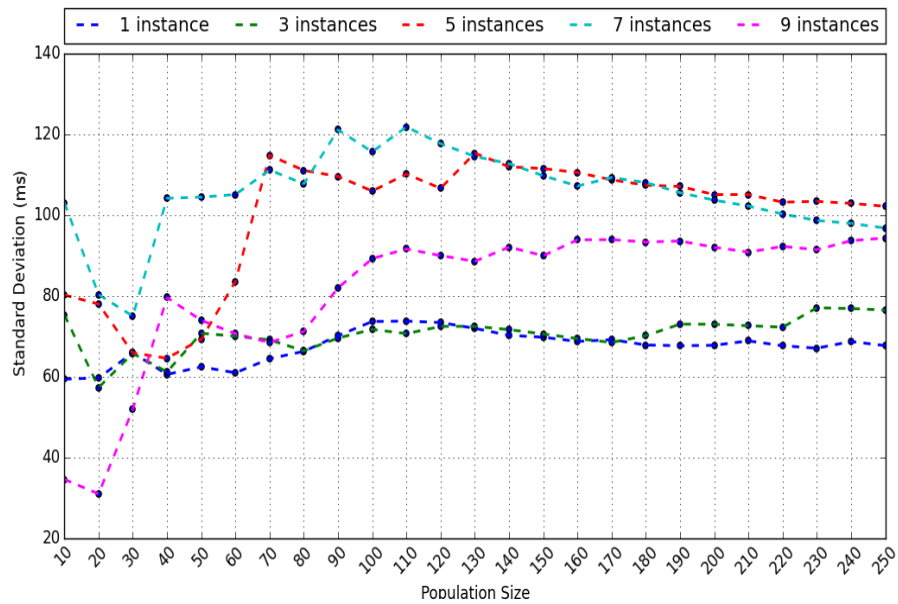


Figure 43: Standard deviation of turning on an instance from 1 up to 9 instances

The average time to turn on instances is displayed in Figure 44, which

shows that the turning on time increases linearly which starts from 783 millisecond for turning on one instance, and ends at 911 milliseconds for turning on the 9th instance. The linear increasing of average time is because the more services running on the compute node the less memory, CPU and disk space will be available for new instances to start.

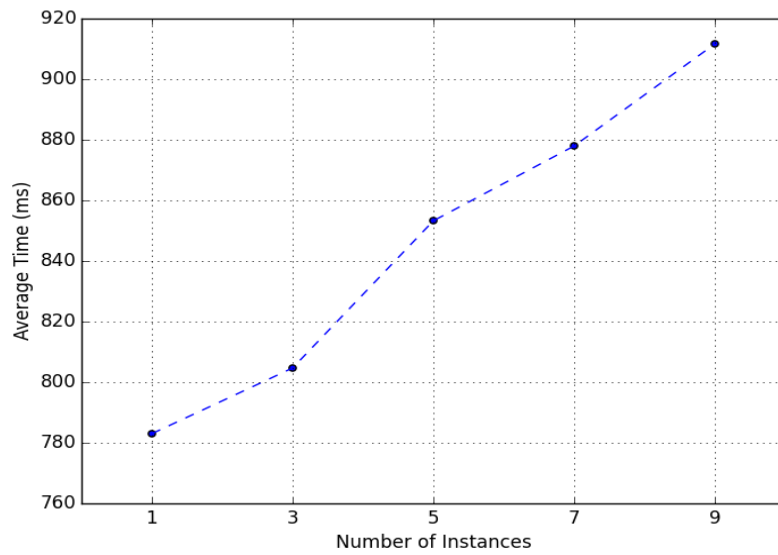


Figure 44: Average time for turning on an instance from 1 up to 9 instances

Similarly we have done the same evaluation for creating a new instance. The time recorded, is the time taken from the point the system receives an alert message, until it finishes creating new instance and wait for the instance to spawn and boot completely. Figure 45 illustrates the standard deviation for population of 250 recorded data, where blue dashed line shows when we already had 1 instance running, and that instance requested for help, and the magenta dashed line, demonstrates when we already had 7 instances running, and one of these running instances asked for help.

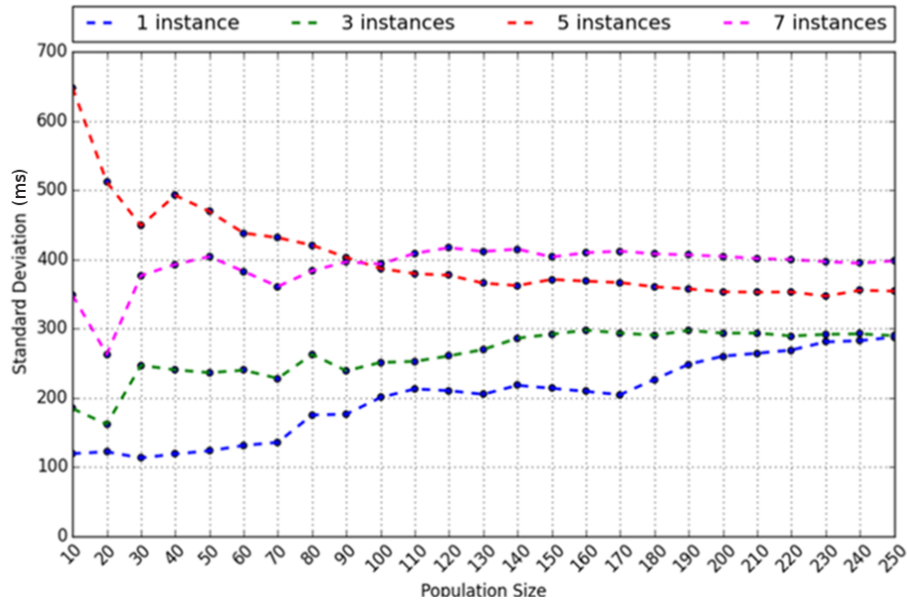


Figure 45: Standard deviation of creating an instance

The average time to create a new instance is similar to the average time to turn on an instance, which is linearly increasing. However with the difference that the time taken to create new instance is greater compare to turning on an already existing instance. The reason is due to the time taken for the main monitoring system to investigate what image and flavour it should use and after preparing proper configuration send the request to the OpenStack controller to spawn and boot the new instance, which requires more time. Figure 46 shows that the system took around 2848 milliseconds to create the 2nd instance when there was only 1 instance running. And, around 3170 milliseconds were taken to create the 8th instance when there were already 7 instances running.

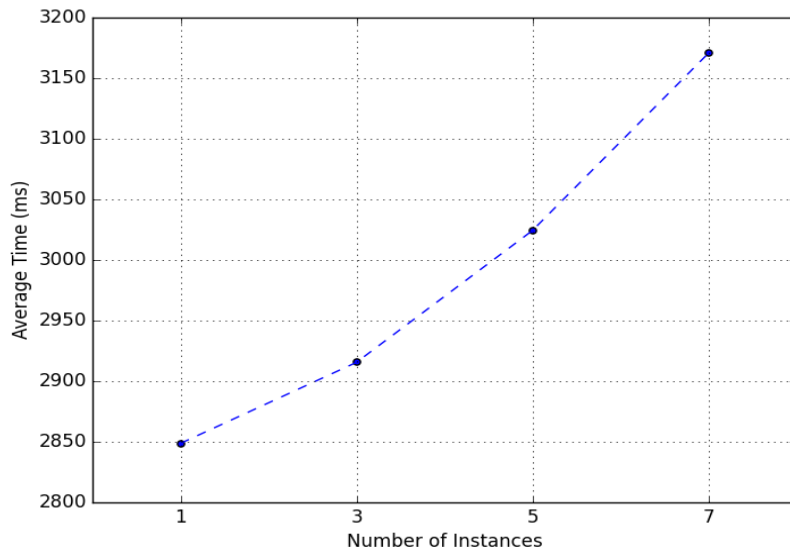


Figure 46: Average time for creating an instance

## 6 Conclusion and Future Work

Monitoring service chain is essential and important, since failure of any components of the cloud and its services can break the service chain. Many network communication companies are investing on service chain, and transforming their hardware based networking infrastructure to NFV and SDN. In addition dynamic service chain is one the key elements in 5G network that operators are working on it.

In this report we studied some of the well known techniques for service chaining as well as tools for monitoring the cloud. We compared these techniques and tools to discover which tool or technique is suitable for what kind of environment and situation. Later we have explained briefly the techniques and terms that are used in the entire of this report to provide a better understanding for the reader.

For the purpose of service chain monitoring, we developed a system with different modules to monitor the cloud nodes, and services. Main monitoring system is responsible for controlling the cloud and creating/turning on new instances automatically whenever is needed. Moreover, main monitoring system has the ability to control the services manually to turn them on/off, create/delete them as well as assigning floating IPs to them. Monitoring daemons are responsible to monitor the CPU and memory usages of KVM nodes as well as docker services and report when there is a heavy load on any of the services.

For the evaluation, we compared the developed monitoring daemons with the *collectd* monitoring system in order to evaluate how much overhead our system imposes on each node. We found out that our system consumes less CPU compare to *collectd*. However, it utilizes more memory, because our system is developed in java which runs through JVM, and *collectd* is written in C language which is more memory friendly. In addition we evaluated the system reaction time to alerts, and identified that the system response time increases linearly as the number of services increase.

However, due the time limitation that we had, we could not implement some of the features that we discussed in the design chapter, which are reserved for the future work, including:

- Decision making logic to turn off or delete an unused instance in order to save resources.



- Turning on new compute node automatically if the first compute node cannot handle more services, which in our case our compute node could handle only 9 services simultaneously.
- Requesting OpenDaylight to divide the flows between two or more services when the system creates new instance for help, in order to bring down the load of a saturated service.
- Some extra features for the UI in order to handle more jobs, such as deleting an instance, creating a router, creating different tenants, etc. However these features consider a plus, and have no impact on the monitoring which was the purpose of this system.

## References

- [ACM<sup>+</sup>14] Arumathurai, M., Chen, J., Monticelli, E., Fu, X. and Ramakrishnan, K. K., Exploiting icn for flexible management of software-defined networks. *Proceedings of the 1st international conference on Information-centric networking*. ACM, 2014, pages 107–116.
- [Alm92] Almquist, P., Type of service in the internet protocol suite. RFC 1349, Internet Engineering Task Force, July 1992. URL <https://tools.ietf.org/html/rfc1349>. Visited 2015-11-03.
- [Ama16] Amazon cloudwatch, 2016. URL <https://aws.amazon.com/cloudwatch/>. Visited 2016-03-17.
- [AP15] Akram, H. and Pascal, B., Leveraging SDN for the 5g networks: Trends, prospects and challenges. *CoRR*.
- [Azu16a] About azure, 2016. URL <https://www.paraleap.com/AzureWatch>. Visited 2016-03-18.
- [Azu16b] Microsoft azuzre, how to monitor cloud services, 2016. URL <https://azure.microsoft.com/en-us/documentation/articles/cloud-services-how-to-monitor/>. Visited 2016-03-18.
- [BMVO12] Beernaert, L., Matos, M., Vilaça, R. and Oliveira, R., Automatic elasticity in openstack. *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, New York, NY, USA, 2012, ACM.
- [CDSI<sup>+</sup>14] Cotroneo, D., De Simone, L., Iannillo, A., Lanzaro, A., Natella, R., Fan, J. and Ping, W., Network function virtualization: Challenges and directions for reliability assurance. *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, Nov 2014, pages 37–42.
- [Clo15] Apache cloudstack, 2015. URL <http://cloudstack-release-notes.readthedocs.org/en/latest/compat.html>. Visited 2015-09-11.

- [Col15] collectd – the system statistics collection daemon, 2015. URL <https://collectd.org/index.shtml>. Visited 2015-10-09.
- [Doc15] What is docker, 2015. URL <https://www.docker.com/whatisdocker>. Visited 2015-09-14.
- [Doc16] docker stats, 2016. URL <https://github.com/docker/docker/blob/0d445685b8d628a938790e50517f3fb949b300e0/api/client/stats.go>. Visited 2016-02-04.
- [DUW11] De Chaves, S., Uriarte, R. and Westphall, C., Toward an architecture for monitoring private clouds. *Communications Magazine, IEEE*.
- [Euc15a] Eucalyptus 3.1, 2015. URL [https://www.eucalyptus.com/docs/eucalyptus/3.1/ig/installing\\_hypervisors.html](https://www.eucalyptus.com/docs/eucalyptus/3.1/ig/installing_hypervisors.html). Visited 2015-09-11.
- [Euc15b] Eucalyptus networking modes, 2015. URL [https://www.eucalyptus.com/docs/eucalyptus/3.2/ig/planning\\_networking\\_modes.html#planning\\_networking\\_modes](https://www.eucalyptus.com/docs/eucalyptus/3.2/ig/planning_networking_modes.html#planning_networking_modes). Visited 2015-09-11.
- [FFRR14] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J., An updated performance comparison of virtual machines and linux containers. *technology*, 28, page 32.
- [FRZ13] Feamster, N., Rexford, J. and Zegura, E., The road to sdn. *Queue journal*.
- [Gan15] Ganglia monitoring system, 2015. URL [www.ganglia.info](http://www.ganglia.info). Visited 2015-10-09.
- [He03] He, H., What is service-oriented architecture. *Publicação eletrônica em*, 30, page 50.
- [HP115] Hpe helion eucalyptus, 2015. URL <http://www8.hp.com/us/en/cloud/helion-eucalyptus.html>. Visited 2015-09-11.
- [Hua13] Enabling agile service chaining with service based routing. Technical Report, 2013. URL [http://www.huawei.com/ilink/en/download/HW\\_308622](http://www.huawei.com/ilink/en/download/HW_308622). Visited 2015-07-12.

- [IPMM12] Issariyapat, C., Pongpaibool, P., Mongkolluksame, S. and Meesublak, K., Using nagios as a groundwork for developing a better network monitoring system. *Technology Management for Emerging Technologies (PICMET), 2012 Proceedings of PICMET '12*, July 2012, pages 2771–2777.
- [ISK<sup>+</sup>14] Ivan, M., Sandeep, P., Ken, O., Kiran, T., Sailesh, Y., Lars, H., Mark, C., Joe, F., Kimberly, C. and Dan, J., Linux containers: why they're in your future and what has to happen first. Technical Report, 09 2014.
- [JGS11] Joshi, P., Gunawi, H. S. and Sen, K., Prefail: A programmable tool for multiple-failure injection. *SIGPLAN Not.*, 46,10(2011), pages 171–188.
- [JPA<sup>+</sup>13] John, W., Pentikousis, K., Agapiou, G., Jacob, E., Kind, M., Manzalini, A., Risso, F., Staessens, D., Steinert, R. and Meirosu, C., Research directions in network service chaining. *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, Nov 2013.
- [KGC<sup>+</sup>14] Kumar, R., Gupta, N., Charu, S., Jain, K. and Jangir, S. K., Open source solution for cloud computing platform using open-stack. *International Journal of Computer Science and Mobile Computing*, pages 89–98.
- [KISdL15] Koller, R., Isci, C., Suneja, S. and de Lara, E., Unified monitoring and analytics in the cloud. *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015, USENIX Association.
- [LNPW14] Li, Q., Niu, H., Papathanassiou, A. and Wu, G., 5g network capacity: Key elements and technologies. *Vehicular Technology Magazine, IEEE*.
- [Mar98] Martin-Flatin, J., Push vs. pull in web-based network management. *CoRR*.
- [Men15] Menage, P., cgroups, 2015. URL <https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>. Visited 2015-12-15.

- [Nag15] Nagios features and capabilities, 2015. URL <https://www.nagios.org/about/features/>. Visited 2015-10-08.
- [Nim16] Ca nimsoft cloud monitor, 2016. URL <http://www.ca.com/~media/files/lpg/ca-nimsoft-cloud-monitor-datasheet.pdf>. Visited 2016-02-04.
- [Nok15] Flexible service chaining. Technical Report, 2015. URL [http://networks.nokia.com/sites/default/files/document/whitepaper\\_flexible\\_service\\_chaining.pdf](http://networks.nokia.com/sites/default/files/document/whitepaper_flexible_service_chaining.pdf). Visited 2015-09-13.
- [NTH<sup>+</sup>08] Nick, M., Tom, A., Hari, B., Guru, P., Larry, P., Jennifer, R., Scott, S. and Jonathan, T., Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38, pages 69–74.
- [Odl16] Opendaylight, platform overview, 2016. URL <https://www.opendaylight.org/platform-overview-beryllium>. Visited 2016-04-07.
- [PQ14] P. Quinn, T. N., Service function chaining problem statement. Technical Report, August 2014. URL <http://tools.ietf.org/html/draft-ietf-sfc-problem-statement-09>. Visited 2015-10-20.
- [RB14] Rosado, T. and Bernardino, J., An overview of openstack architecture. *Proceedings of the 18th International Database Engineering & Applications Symposium*. ACM, 2014, pages 366–367.
- [Tse13] Tseitlin, A., The antifragile organization. *Commun. ACM*, 56,8(2013), pages 40–44.
- [WGL<sup>+</sup>12] Wen, X., Gu, G., Li, Q., Gao, Y. and Zhang, X., Comparison of open-source cloud management platforms: Openstack and opennebula. *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*. IEEE, 2012, pages 2457–2461.

- [XSZ<sup>+</sup>15] Xia, M., Shirazipour, M., Zhang, Y., Green, H. and Takacs, A., Optical service chaining for network function virtualization. *IEEE Communications Magazine*, 53,4(2015), pages 152–158.
- [ZBB<sup>+</sup>13] Zhang, Y., Beheshti, N., Beliveau, L., Lefebvre, G., Manghir-malani, R., Mishra, R., Patneyt, R., Shirazipour, M., Subrahmaniam, R., Truchan, C. and Tatipamula, M., Steering: A software-defined networking for inline service chaining. *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, Oct 2013.