



UNIVERSIDADE ESTADUAL DO CEARÁ
CENTRO DE CIÊNCIAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

ALEX FERREIRA RAMIRES TRAJANO

BALANCEAMENTO DE CARGA DE CACHES DE IMKVS

FORTALEZA – CEARÁ

2016

ALEX FERREIRA RAMIRES TRAJANO

BALANCEAMENTO DE CARGA DE CACHES DE IMKVS

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Marcial Porto Fernandez

FORTALEZA – CEARÁ

2016

Dados Internacionais de Catalogação na Publicação

Universidade Estadual do Ceará

Sistema de Bibliotecas

Trajano, Alex Ferreira Ramires.

Balanceamento de carga de caches de IMKVS
[recurso eletrônico] / Alex Ferreira Ramires Trajano.
- 2016.

1 CD-ROM: il.; 4 ¾ pol.

CD-ROM contendo o arquivo no formato PDF do
trabalho acadêmico com 63 folhas, acondicionado em
caixa de DVD Slim (19 x 14 cm x 7 mm).

Dissertação (mestrado acadêmico) - Universidade
Estadual do Ceará, Centro de Ciências e Tecnologia,
Mestrado Acadêmico em Ciência da Computação,
Fortaleza, 2016.

Orientação: Prof. Ph.D. Marcial Porto Fernandez.

1. Caching. 2. Balanceamento de Carga. 3.
Datacenter. I. Título.

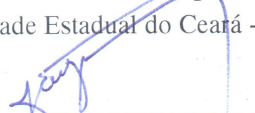


**ATA DA OCTOGÉSIMA SEGUNDA DEFESA PÚBLICA
DE DISSERTAÇÃO DE Mestrado**

Ao quarto dia do mês de março de dois mil e dezesseis, no miniauditório do prédio de Pesquisa e Pós-Graduação em Computação, do Mestrado Acadêmico em Ciência da Computação – MACC, realizou-se a sessão pública de defesa da dissertação de ALEX FERREIRA RAMIRES TRAJANO, aluno regularmente matriculado no Mestrado Acadêmico em Ciência da Computação – MACC, intitulada: “**BALANCEAMENTO DE CARGA DE CACHES DE IMKVS**”. A Banca Examinadora reuniu-se no horário de 15h às 17 horas, sendo constituída pelos Professores Doutores **Joaquim Celestino Júnior** (Orientador/UECE); **Leonardo Sampaio Rocha/UECE**; **Marcial Porto Fernandez/UECE** e **Javam de Castro Machado** da Universidade Federal do Ceará-UFC. Inicialmente o mestrando expôs seu trabalho e a seguir foi submetido à arguição pelos membros da Banca, dispondo cada membro de tempo para tal. Finalmente a Banca reuniu-se em separado e concluiu por considerar o mestrando APROVADO COM LOUVOR por sua dissertação e sua defesa pública. Eu, **Professor Dr. Marcial Porto Fernandez**, Orientador e Presidente da Banca, lavrei a presente Ata que será assinada por mim e demais membros da Banca. Fortaleza, 04 de março de 2016.


Prof. Dr. Marcial Porto Fernandez
Orientador – UECE


Prof. Dr. Leonardo Sampaio Rocha
Universidade Estadual do Ceará - UECE


Prof. Dr. Joaquim Celestino Júnior
Universidade Estadual do Ceará-UECE


Prof. Dr. Javam de Castro Machado
Universidade Federal do Ceará-UFC



Dedico a todos que acreditam que ciência pode ser feita por qualquer pessoa, em qualquer lugar, até mesmo debaixo de um cajueiro...

AGRADECIMENTOS

Agradeço à minha família, sobretudo meus pais, Rosana e Francisco, por desde muito cedo terem me incentivado aos estudos, por terem me proporcionado um ambiente em que eu pudesse me desenvolver como pessoa, cidadão e profissional, apesar das poucas condições que tiveram acesso no passado. Não fosse a certeza deles de que a educação seria fundamental no meu desenvolvimento, talvez eu não tivesse a oportunidade de estar onde estou e de construir esse trabalho. Também agradeço ao meu irmão, Davi, que me suportou durante todo o período em que eu precisei usar três computadores e todo o nosso link de "banda larga" para desenvolver esse trabalho :).

Agradeço à minha namorada, conselheira e melhor amiga, Morgana, que pôde compreender a minha ausência nos períodos em que tive que desenvolver este trabalho, além de ter suportado todo o stress do mundo. Ela foi a maior responsável por ter me dado uma nova visão do mundo, visão essa que ajudou o meu crescimento pessoal e profissional. Os últimos 4 anos foram os melhores de minha vida graças a ela. Muito obrigado por tudo, o mérito por esse trabalho também é seu!

Agradeço ao meu orientador, Prof. Marcial, por ter me dado a confiança e a liberdade que foram extremamente fundamentais para produzir esse trabalho. Graças a sua atuação bastante precisa, pude me desenvolver como pesquisador e também pude conhecer um pouco mais desse mundo em que vivemos. Obrigado ao Prof. Celestino, que, em sala de aula, me apresentou uma visão diferente de como aprender e criticar novas tecnologias. Sou grato aos professores do MACC que me ajudaram a obter o conhecimento necessário para que eu possa deixar minha contribuição à humanidade.

Agradeço aos colegas de trabalho, Alexandre, Jackson, Rubens, João Paulo e Alan, que, mesmo sem saber, puderam contribuir para o desenvolvimento desse trabalho. As longas discussões arquiteturas do nosso sistema foram de suma importância para muitas decisões de projeto dos trabalhos que venho desenvolvendo. Obrigado pela confiança da Trixlog em me ceder algumas horas para a dedicação ao mestrado.

Agradeço a todos amigos que participaram dos momentos dessa fase importante de minha vida. Afinal, de que servem as conquistas sem alguém pra compartilhar?

“There is a huge need and a huge opportunity to get everyone in the world connected, to give everyone a voice and to help transform society for the future. The scale of the technology and infrastructure that must be built is unprecedented, and we believe this is the most important problem we can focus on.”

(Mark Zuckerberg)

RESUMO

Redes sociais e outros tipos de aplicações em nuvem requerem respostas rápidas da infraestrutura de seus datacenters. Uma das técnicas que tem sido amplamente utilizada para alcançar tal requisito é o emprego de In-Memory Key-Value Storage (IMKVS) como mecanismo de caching, a fim de melhorar a experiência do usuário. Memcached e Redis são dois exemplos de aplicações que seguem a abordagem IMKVS. Geralmente, clientes de IMKVS utilizam Consistent Hashing para escolher onde armazenar um determinado objeto, o que pode causar desbalanceamento da carga na rede de computadores. Além do mais, tais clientes operam apenas na camada de aplicação, o que faz com que as condições da rede de computadores não seja levada em consideração enquanto distribuindo as requisições dos usuários. Este trabalho tem como objetivo propor uma nova arquitetura de caching em que é utilizado um balanceamento de carga em duas fases a fim de melhorar a performance de caches IMKVS. Tal arquitetura faz uso extensivo dos conceitos de Software-Defined Networking e Network Function Virtualization para gerenciar o mecanismo de balanceamento de carga. A proposta foi avaliada no Mininet, uma ferramenta de emulação de redes de computadores, e os resultados mostram que a proposta melhora o desempenho geral do sistema, ajudando a reduzir os custos operacionais das redes de datacenter.

Palavras-chave: Caching. Balanceamento de Carga. Datacenter.

ABSTRACT

Social networks and other clouding applications should require a fast response from datacenter's infrastructure. One of the techniques that has been widely used for achieving such requirements is the employment of In-Memory Key-Value Storage (IMKVS) as caching mechanisms in order to improve overall user experience. Memcached and Redis are applications that use IMKVS approach. Commonly IMKVS clients use Consistent Hashing to decide where to store an object, which may cause network load imbalance. Furthermore, these clients work only at the application layer, so network conditions are not considered to distribute user's accesses. This paper proposes a new cache architecture with two phases load balancing to improve IMKVS performance, which has adopted the Software-Defined Networking and Network Function Virtualization concepts to design an architecture to manage the load balancing mechanism. The proposal was evaluated in the Mininet emulation environment and shows an improvement on load balancing, helping to reduce operational costs of datacenter networks.

Keywords: Caching. Load Balancer. Datacenter.

LIST OF FIGURES

Figure 1 – The OpenFlow architecture	22
Figure 2 – The OpenFlow Flow Entry	22
Figure 3 – The NFV architectural components proposed by ETSI	24
Figure 4 – Two cached objects on a Consistent Hashing ring	27
Figure 5 – The "look aside" strategy used to fill Memcached caches	29
Figure 6 – Facebook's servers architectural overview	29
Figure 7 – An arbitrary deployment scenario for the two-phase load balancing	33
Figure 8 – Network communication while forwarding IMKVS traffic	34
Figure 9 – <i>uLoBal</i> architectural modules	35
Figure 10 – <i>kvsKeeper</i> architectural components on two VMs	43
Figure 11 – Facebook's "4-post" network topology	49
Figure 12 – Network Topology for the <i>uLoBal</i> experiments	51
Figure 13 – SDN load balancing compared to a traditional SPF forwarding strategy	53
Figure 14 – Message delivery delay when using SDN load balancing modes	54
Figure 15 – Servers load levels when using SDN load balancing modes	55
Figure 16 – IMKVS commands execution time comparison	55
Figure 17 – Execution time of mget command with multiple <i>kvsKeeper</i> instances	56
Figure 18 – Servers and network loads comparison	57

LIST OF TABLES

Table 1 – <i>uLoBal</i> API methods and parameters	36
Table 2 – <i>uLoBal</i> operational modes	37

LIST OF ALGORITHMS

Algorithm 1 – Consistent Hashing	27
Algorithm 2 – PacketIn handling algorithm	38
Algorithm 3 – RR algorithm	39
Algorithm 4 – IPH algorithm	39
Algorithm 5 – NSL algorithm	40
Algorithm 6 – Dispatcher’s <i>set</i> algorithm	45
Algorithm 7 – Dispatcher’s <i>get</i> algorithm	46
Algorithm 8 – Dispatcher’s <i>mget</i> algorithm	46
Algorithm 9 – Dispatcher’s <i>delete</i> algorithm	46
Algorithm 10 – gBstSrv procedure	47
Algorithm 11 – Replication algorithm	48

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CAN	Content Addressable Network
CDN	Content Delivery Network
CH	Consistent Hashing
CPU	Central Processing Unit
DHT	Distributed Hash Table
ETSI	European Telecommunications Standards Institute
ForCES	Forwarding and Control Element Separation
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IMKVS	In-Memory Key-Value Storage
IPH	IP Hashing
LRU	Last Recently Used
NAL	Network-Assisted Lookups
NAT	Network Address Translation
NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualization
NFVI	NFV Infrastructure
NFVMANO	NFV Management and Orchestration
NSL	Network and Server Load
P2P	Peer-to-Peer
QoE	Quality of Experience
REST	Representational State Transfer
RR	Round-Robin
SDN	Software-Defined Network
SPF	Shortest Path First
VM	Virtual Machine
VNF	Virtualized Network Function

LIST OF SYMBOLS

δ	uLoBal network path cost
ζ	Available bandwidth percentage of a network enlace
ν	Percentage of transmission errors of a network enlace
γ	kvsKeeper instance load
ε	Available Central Processing Unit (CPU) percentage
ξ	Available memory percentage
η	Available network interface bandwidth percentage

CONTENTS

1	INTRODUCTION	16
1.1	PROPOSAL	18
1.2	CONTRIBUTIONS	19
1.3	WORK STRUCTURE	19
2	BACKGROUND	21
2.1	SOFTWARE-DEFINED NETWORKING	21
2.2	NETWORK FUNCTION VIRTUALIZATION	24
2.3	CONSISTENT HASHING	25
2.4	IN-MEMORY KEY-VALUE STORAGES	28
3	RELATED WORKS	30
3.1	LOAD BALANCING ON SOFTWARE-DEFINED NETWORKS	30
3.2	LOAD BALANCING OF IN-MEMORY KEY-VALUE STORAGES	31
4	TWO-PHASE LOAD BALANCING OF IMKVS CACHES	33
4.1	ULOBAL: ENABLING GENERIC LOAD BALANCING ON SDN	35
4.1.1	Management Module	36
4.1.2	Network Monitoring Module	37
4.1.3	Load Balancer Module	37
4.2	KVSKEEPER: ON THE LOAD BALANCING OF IMKVS	40
4.2.1	Architectural Components	42
4.2.2	Index	43
4.2.3	Dispatcher Module	44
4.2.4	Replication Module	47
5	EVALUATION	49
5.1	EXPERIMENT 1	50
5.2	EXPERIMENT 2	51
5.3	EXPERIMENT 3	51
5.4	EXPERIMENT 4	52
6	RESULTS	53
6.1	EXPERIMENT 1	53
6.2	EXPERIMENT 2	55
6.3	EXPERIMENT 3	56

6.4	EXPERIMENT 4	57
7	CONCLUSION AND FUTURE WORK	58
	Bibliography	60

1 INTRODUCTION

Traditional small web sites often rely on a web server software and a database that are hosted on the same physical server, which means that most of the network communication is the user traffic coming from the Internet. However, recent cloud applications like social networks are accessed by hundreds of millions of users every day, imposing computational, network and I/O demands that a traditional architecture would struggle to satisfy. In order to support the massive workloads related to such applications, it is necessary to design scalable and reliable infrastructures that are able to provide the resources and the technology for processing millions of user requests per second.

Most of cloud applications must allow near real-time communication, while aggregating content on-the-fly from multiple sources whose access cost may be high. Besides, it is necessary to provide access to popular shared content. Facebook, for instance, has a deployment scenario where a front-end web server is responsible for deliver requested content to users through Hypertext Transfer Protocol (HTTP). These servers must handle the requests and fetch data from different cache, databases and back-end servers in order to render the final web page. It has been reported that a single HTTP page request required 88 cache lookups (consuming 648 KB), 35 database lookups (consuming 25.6 KB) and 392 back-end remote calls (consuming 257 KB), taking just a few seconds to the page be completely loaded on the user's screen (FARRINGTON; ANDREYEV, 2013).

In order to support the storage and the processing of large amounts of data, many cloud applications have adopted a simple but effective caching infrastructure that rely on In-Memory Key-Value Storage (IMKVS). These simple storages are able to provide fast access to any type of data that can be mapped by a key. Its in-memory placement of data helps to avoid slow and expensive access to persistent storages on disk. Thus, IMKVS is often used to store and supply information that is cheaper to cache than to re-obtain, such as commonly accessed results of database queries or the results of complex computations. Several IMKVS cache implementations have been developed and deployed in large scale cloud services, including Dynamo at Amazon (DECANDIA et al., 2007); Redis at GitHub, Digg, and Blizzard (SANFILIPPO; NOORDHUIS, 2016); Memcached at Facebook, Zynga, and Twitter (FITZPATRICK, 2004); and Voldemort at LinkedIn (SUMBALY et al., 2012). Facebook has reported that there are tens of thousands of Memcached instances operating on their datacenters (NISHTALA et al., 2013).

These simple applications are basically Hash Maps capable of storing any kind

of data mapped by an unique key, usually a string of variable length. IMKVS can be used to form huge caching layers designed to operate in a distributed and independently fashion over unstructured networks, being an important component on the architecture of applications with diverse load levels. In such deployments, the IMKVS instances do not know about the existence of other IMKVS instances, making their clients applications to manage all aspects of data partitioning and load balancing.

Most of IMKVS clients Application Programming Interface (API) use a technique called Consistent Hashing (CH) (KARGER et al., 1997). It consists in distributing a set of keys uniformly among a set of servers in such a way that neither servers addition nor removal causes large impacts on the keys distribution, also avoiding the creation of hot spots in the network. CH has been used successfully in several kinds of applications, like caching (NISHTALA et al., 2013) and storage (LAKSHMAN; MALIK, 2010). Although CH is a very efficient technique, it is, basically, a special hash function that uses a key and a set of servers to select where to store data. However, it may incur load imbalance in large production networks, producing hot spots, since CH does not consider any environmental aspect, object's characteristics, congestion nor object's popularity while distributing objects.

Moreover, the growing complexity and workload of current computer networks often require large infrastructure investments in order to support new demands. However, it is not feasible to increase the network capacity at the same rate as the demand grows, requiring a set of techniques that aims a more efficient use of network resources. One of the most-used techniques is to perform load balancing, either by application layer algorithms or network orchestration, in order to optimize network traffic.

In fact, over the last years, it has been common to find specialized hardware appliances or applications capable of performing traffic load balance of specific types of service. Nevertheless, the load balancing task should not be coupled to specialized infrastructure items, since it should be an embedded feature of the network itself in order to ensure maximum performance. In-network load balancing is a way of providing more flexibility and near optimal performance. Besides, there is a wide set of Internet services that can be boosted using an in-network load balancing technique, which opens the opportunity for developing generic solutions focused on the adherence to current and future services at no cost.

Over the last few years, Network Functions Virtualization (NFV) (HAN et al., 2015) has been becoming one of the most promising study areas for developing new computer network

technologies and architectures. NFV poses a novel way to develop network services, by using software and virtualization aiming the replacement of proprietary hardware appliances that run network functions, like Network Address Translation (NAT), Intrusion Detection System (IDS), caching, etc. In NFV, these services, called Virtualized Network Function (VNF), are implemented through software and deployed in Virtual Machine (VM)s, allowing new and efficient ways of network deploying. It also allows customization and real time management of such services, enabling a tremendous cost saving and more agility to serve the daily changes that every computing network is susceptible.

Simultaneously with NFV, another networking approach that has been gaining space is the Software-Defined Network (SDN) (KREUTZ et al., 2015). SDN allows network management by creating an abstraction of the lower-level functionalities of the networking, by separating the control plane (decision making) from the data plane (forwarding). SDN is often confused with OpenFlow (MCKEOWN et al., 2008) since it is the most popular and promising SDN protocol. OpenFlow, beyond have the data and control planes separated, have a centralized architecture that simplifies the development of different kinds of network services, which consists in creating a unique programmable controller that can be deployed to commodity servers, maintaining TLS channels to all switches within the network, programming all aspects of packet forwarding of the network. Although SDN and NFV have many common aspects, both are neither competitors nor conflicting. When both are used together, the whole network tends to benefit from it, since it has the best aspects of data and control plane separation coexisting with the virtualization power, allowing a more efficient control of the network. SDN contributes to better traffic orchestration while NFV focuses on service delivery.

1.1 PROPOSAL

Since the major issue of CH is not taking any environmental metric into account, it is needed to develop a new technique that is able to perform load balancing of IMKVS requests. This technique must be capable of truly load balance the datacenter network traffic, in such a way that the IMKVS servers will handle the incoming traffic according to its current load state. In order to do that, this work proposes a two-phase load balancing mechanism that make intensive use of VNFs to manage IMKVS traffic dynamically over the network, providing a scalable and resilient alternative to CH. This mechanism is divided into two modules, *uLoBal* and *kvsKeeper*, being each one responsible for a specific task at a specific placement on the network.

The first phase of the load balancing happens on the SDN controller, through the use of *uLoBal*, a system capable of dynamically load balance arbitrary services through the use of different forwarding approaches. The second phase is made by *kvsKeeper*, a VNF that can be deployed in multiple VMs spread over the network, being responsible for decide where to forward IMKVS packets and performing orchestration on the data. Futhermore, *kvsKeeper* is also capable of replicate highly popular data, helping to mitigate the appearance of hot spots over the network.

The main idea is to use the SDN controller to select the best VM that executes a *kvsKeeper* process and forward the incoming IMKVS packets to there. Once a *kvsKeeper* process receives the traffic, it will decide to which IMKVS server the incoming packets should be forwarded, according to network and server metrics and based on an index shared across the *kvsKeeper* VMs. Both *uLoBal* and *kvsKeeper* select the best destination to forward requests by considering the load on both network and the available servers, helping to reduce the load created by new requests.

By load balancing the traffic in two phases considering the load on network and servers and replicating popular data across multiple servers, it is expected that the two-phase mechanism outperforms CH yet improving the use of expensive datacenter resources.

1.2 CONTRIBUTIONS

During the research process of this work, two related papers were published.

The first, presented at the 20th IEEE Symposium on Computers and Communications (ISCC), was the preliminary work about the two-phase load balancing technique employed on IMKVS systems that is presented here (TRAJANO; FERNANDEZ, 2015).

The second, presented at the 15th International Conference on Networks, was developed as an abstraction of the first, proposing an generic load balancer for arbitrary Internet services (TRAJANO; FERNANDEZ, 2016).

1.3 WORK STRUCTURE

The rest of this work is structured as follows. In Chapter 2, it is presented the background concepts used in the work. Chapter 3 shows some related works about SDN load balancing and IMKVS load balancing. Chapter 4 presents the load balancing architecture. In

Chapter 5, the experimental scenarios are described and in Chapter 6 the results are presented. Finally, Chapter 7 concludes the work and some future research is described.

2 BACKGROUND

In this Chapter the concepts related to this work are going to be presented. In Section 2.1 the Software-Defined Networking will be introduced in conjunction with the OpenFlow protocol and its architecture. Then, in Section 2.2, the proposed architecture for Network Function Virtualization will be presented and in Section 2.4 the In-Memory Key-Value Storages will be shown. Finally, in Section 2.3, the Consistent Hashing approach will be discussed.

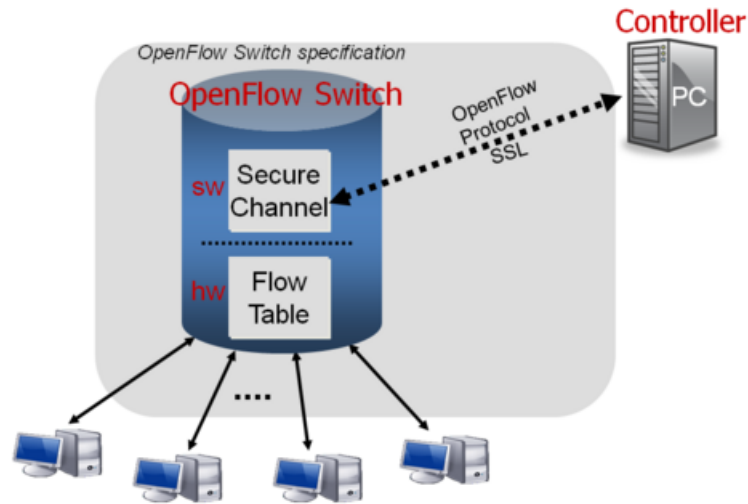
2.1 SOFTWARE-DEFINED NETWORKING

SDN is an approach to network control and management that allow administrators to manage network services by an abstraction of lower network protocol functions. This is done by decoupling the control plane, that takes decisions about forwarding traffic through the network infrastructure; from the data plane, the users traffic itself. The objective is to simplify the network control in high speed traffic. SDN requires a protocol for the control plane that is able to communicate with devices. The most known protocol is the OpenFlow, often misunderstood to be equivalent to SDN, but other protocols could also be used, such as Forwarding and Control Element Separation (ForCES) (DORIA et al., 2010) and Network Configuration Protocol (NETCONF) (ENNS et al., 2011). In our work, we will consider only the OpenFlow architecture.

OpenFlow has two main components: the controller, an unique programmable remote control, and the network devices. These two components work together through the OpenFlow Protocol. The main idea is to keep network devices as simple as possible in order to reach better forwarding performance, having no complex decision-making process within the devices, delegating such a task to the network controller. Figure 1 shows how these components are organized on the OpenFlow architecture.

The OpenFlow Controller is the centralized controller of an OpenFlow network. It sets up all OpenFlow devices, maintains topology information and monitors the overall status of entire network. The OpenFlow Device is any OpenFlow-enabled device in a network, such as a switch, router or access point. Each device maintains a Flow Table that indicates the processing that must be applied to any packet of a certain flow. The OpenFlow Protocol works as an interface between the controller and the switches, setting up the Flow Table and exchanging information about the network state. The controller updates the Flow Table by adding and removing Flow

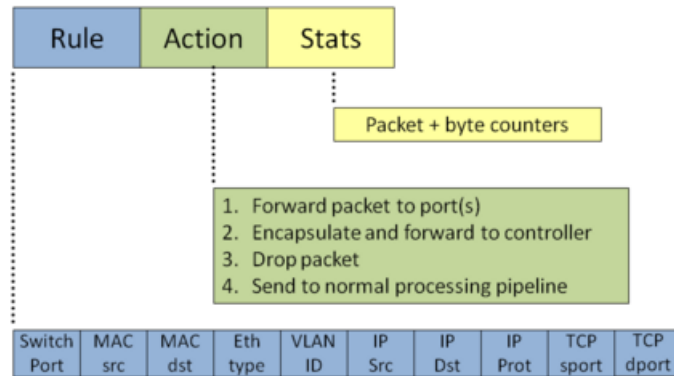
Figure 1 – The OpenFlow architecture



Source: MCKEOWN et al., 2008

Entries using the OpenFlow Protocol. The Flow Table is a database that contains Flow Entries associated with a set of actions that commands the switch for apply these set of actions on the packets of a certain flow. Some possible actions are: forward, drop and encapsulate. Figure 2 shows the structure of a Flow Entry.

Figure 2 – The OpenFlow Flow Entry



Source: Open Networking Foundation, 2015

Each OpenFlow device has a Flow Table with Flow Entries. A Flow Entry has three fields: Rule, Action and Stats. The Rule field is used to define the set of conditions to characterize the packets that will match that specific flow. The Action field defines the set of actions that must be applied to a packet if it matches the conditions defined on the Rule field. The Stats maintains a set of counters that are used to monitor flow’s statistics, which can be used for management purposes. Each incoming packet is matched against the entries of the Flow Table. If a set of Flow

Entries matches a packet, the device will select the entry with the highest priority and the actions defined on the entry will be executed, having its statistics updated at the end of the process. If there is no matching entry for an incoming packet, the device sends a *PacketIn* message to the controller, wrapping the unmatched packet.

Once the controller receives a *PacketIn* message, it can take some actions on that packet, such as send *FlowMod* messages to the network devices in order to install new flow entries that match the incoming packet, or send a *PacketOut* message to "manually" forward the packet, or even ignore that packet. Besides, the controller can send messages to query statistics on every OpenFlow-enabled device within its network. An example is the *StatsRequest* message, which aims to query flow, port or queue statistics at the devices, which is useful for determining the current state of the network. Each *StatsRequest* message is sent asynchronously, so the controller must listen for a *StatsResponse* message that will gather the requested data.

The Openflow Controller presents two operational behaviors: reactive and proactive. In the reactive approach, the first packet of flow received by a device triggers the controller to insert flow entries in each OpenFlow switch of network. This approach presents the most efficient use of existing flow table memory, but every new flow incurs in a small additional setup time. This approach may turn the devices highly dependant on the controller, since if the connection between a device and the controller is lost, such device would have limited utility. On the other hand, the proactive approach is based on the fact that the controller pre-populates the Flow Table of each switch on the network. This approach has zero additional flow setup time because the forward rule has already been defined. Now, if the switch loss the connection with controller it does not disrupt traffic. However, the network operation requires a hard management, since it would be necessary to aggregate rules to cover all routes, for instance. Both reactive and proactive behaviors can be used simultaneously according to business needs.

The controller performance is a central issue of an OpenFlow architecture. A controller can support only a limited number of flow setups per second. The work (TAVAKOLI et al., 2009) shows that a single NOX controller can handle a maximum of 30K flow setup per second maintaining a flow install time below 10 ms. From the network side, the work (KANDULA et al., 2009) has measured the creation of 100K flows per second in a 1500-server cluster datacenter, implying a need for, at least, four OpenFlow controllers.

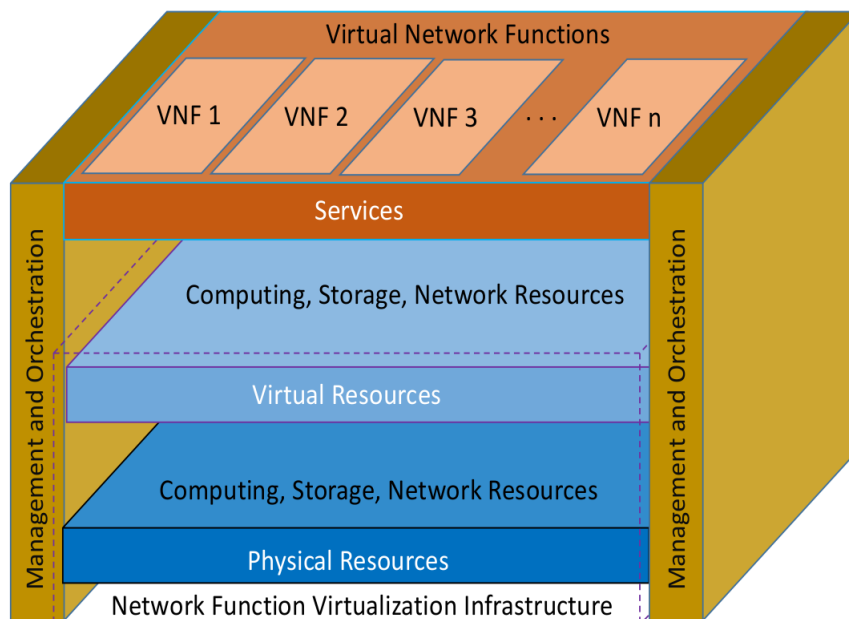
Several approaches have proposed new OpenFlow Controller architectures and implementations. One of the first approaches was the NOX Controller (GUDE et al., 2008),

a centralized controller that implements a reactive and proactive approach. The Floodlight (ERICKSON, 2012) is Java-based OpenFlow Controller, forked from the Beacon controller developed at Stanford. The Floodlight controller is an open-source software Apache-licensed, supported by a community of developers. It offers a modular architecture that makes easy to extend and enhance.

2.2 NETWORK FUNCTION VIRTUALIZATION

NFV offers a new network architecture to design, deploy and manage network functions and services in a virtualized infrastructure. NFV decouples the network functions, such as firewalls, intrusion-detection systems, load balancers, Network Address Translation (NAT), etc., from proprietary dedicated hardware appliances. It is designed to deploy networking components to support a virtualized infrastructure, including virtual machines, servers, storage, etc. The basic idea was proposed in 2012 by a group from the European Telecommunications Standards Institute (ETSI). The ETSI proposal for NFV is based on three key components: NFV Infrastructure (NFVI), VNF and NFV Management and Orchestration (NFVMANO). Figure 3 shows how these components are organized.

Figure 3 – The NFV architectural components proposed by ETSI



Source: MIJUMBI et al., 2015

NFVI is composed by the hardware and software necessary to build the environment

where the VNF will be deployed. Usually, the resources used on NFVI rely on commercial-off-the-shelf hardware, which helps to reduce cost and increase flexibility. Virtualization is the key concept of NFV, so the network functions are deployed on VMs, which are instantiated on the physical hardware resources that compose the NFVI. In other words, the NFVI is the infrastructure behind the virtualized resources (MIJUMBI et al., 2015).

VNF is the functional component of the NFV architecture. A network function is a component that has well defined external interfaces and well defined functional behavior within the network. Firewalls, NAT, load balancers and IDS are some examples of network functions. Thus, a VNF is a network function that was developed to be deployed on the virtualized resources provided by the NFVI. The VNF deployment scenarios are extremely flexible, allowing the network to work with as many VM instances as needed, allowing better scalability and easier management, following diverse business needs and user constraints (MIJUMBI et al., 2015).

NFVMANO is the NFV component responsible for providing the mechanisms that handle the management and orchestration of both infrastructure and network functions. The tasks related to configuration, resource allocation, provisioning, lifecycle management, monitoring and storage of data models are all performed by the NFVMANO. Furthermore, it also can provide an user interface and expose an API to enable service integration with other softwares over the network (MIJUMBI et al., 2015).

NFV is a complementary approach to SDN. While both intend to manage networks, they rely on different point of view. While SDN separates the control and forwarding planes to offer a centralized view of the network, NFV focuses on implement network services on a unique network configuration infrastructure, e.g., based on SDN. There are inherent benefits in leveraging SDN to implement a NFV infrastructure. Basically, when we were looking the management and orchestration of VNF features in a multi-vendor SDN infrastructure.

2.3 CONSISTENT HASHING

Consistent Hashing arose from the limitations that were observed in web caches (KARGER et al., 1997). One of its main objectives is to avoid the presence of *hot spots* on the network, which may lead the network to congest and services to be unavailable for users. The concept of hot spot has been defined by Karger et al. as:

Hot spots occur any time a large number of clients wish to simultaneously access data from a single server. If the site is not provisioned to deal with all

of these clients simultaneously, service may be degraded or lost. Many of us have experienced the hot spot phenomenon in the context of the Web. A Web site can suddenly become extremely popular and receive far more requests in a relatively short time than it was originally configured to handle. In fact, a site may receive so many requests that it becomes “swamped,” which typically renders it unusable. Besides making the one site inaccessible, heavy traffic destined to one location can congest the network near it, interfering with traffic at nearby sites (KARGER et al., 1997).

In that context, it is common to distribute a set of objects to several cache servers in order to balance access load to different network segments. An object is any kind of data that is used to serve users requests, so, it could be from HyperText Markup Language (HTML) files to large database queries results.

An well known technique for objects distribution is to use a hashing function and a modulo operation. The objective is to evenly store the objects on servers, by only considering the number of servers available on the network. To describe the technique, let O be an arbitrary object and n the amount of available servers on the network. The result of (2.1) maps O to the i -th available server S_i .

$$i = \text{hash}(O) \bmod n \quad (2.1)$$

This solution works satisfactorily if the set of available servers does not change over time. When either inclusion or exclusion of servers occurs, it is necessary to remap all objects to the new set of servers, since n has changed. In a production environment, it could cause a massive increase on network traffic.

In order to avoid such problems, CH is an alternative capable of evenly distribute an object set to a server set in such a way that neither addition nor removal events could cause a total cache remapping. When a new server is added to the caching layer, the objects of its neighbors are shared with the new server. When a removal happens, the server’s objects are shared back with its neighbours. This approach minimizes the impacts caused by network changes, by making local changes within the caching layer, avoiding a complete object redistribution on these events. The way CH can do that is by using a hash function on both objects and servers, mapping an object to a range of servers, then selecting one of these servers to store the data. The basic idea is shown in Algorithm 1.

Following the Algorithm 1 with an arbitrary hash function, Figure 4 shows an example of how two random objects would be mapped to some caching servers. O_1 would be

Algorithm 1: Consistent Hashing

Data: A server set C_s , an object O and a hash function H

Result: A server S that O must be stored

Initialize a circular list L ;

foreach server S in C_s **do**

$H_{sv} := H(S)$;

$T := \{h : H_{sv}, s : S\}$;

 insert T into L ;

end

sort L based in $T.h$ values;

$H_{ov} := H(O)$;

if L contains any $T.h = H_{ov}$ **then**

return $T.s$;

end

else

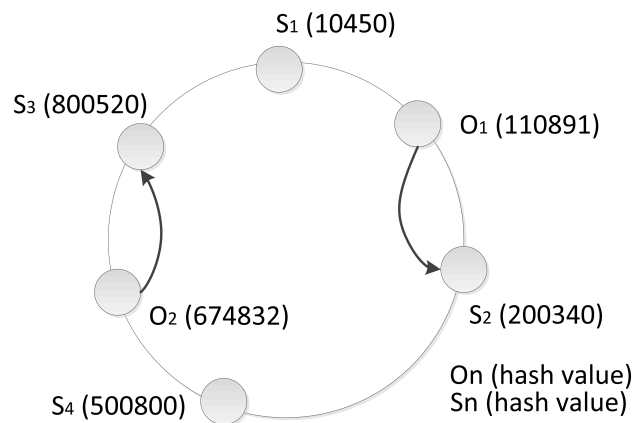
 insert H_{ov} into L in a sorted fashion, comparing H_{ov} against $T.h$ values;

return the H_{ov} rightmost $T.s$ value;

end

mapped to S_2 and O_2 to S_3 . If S_3 were removed from the caching layer, O_2 would be mapped to S_1 or S_4 because they are the current S_3 neighbours. If S_5 were added to the ring, and it was placed in between S_2 and S_4 , the objects held by them would be shared with S_5 .

Figure 4 – Two cached objects on a Consistent Hashing ring



Source: The author

Since CH shows great results in object distribution across several servers (KARGER et al., 1997), it is important to notice that it does not take everything into consideration that can cause network hot spots. Datacenters handle a huge amount of objects in order to serve either user and application complex requests (NISHTALA et al., 2013), thus any algorithm that aims to reduce the probability of emerge a hot spot have to take both network's indicators and objects'

characteristics into consideration while distributing objects to servers.

2.4 IN-MEMORY KEY-VALUE STORAGE

As said before, Memcached and Redis are two examples of IMKVS. These softwares behave as simple Hash Maps, by mapping content through keys. There are some IMKVS that offers a more robust API, being more than a simple map data structure, since they are able to either perform manipulation within the caching layer or store nested data structures, though the basic functionality is the Key-Value relationship (FITZPATRICK, 2004). The basic commands that can be executed in these applications are described as follows ¹:

void set (key, value) Ask a specific server to store a given value identified by a given key.

value get (key) Retrieve the value identified by the given key stored in a specific server. If there is no value associated to the key, then *null* is returned and a miss happens.

value[] mget (key[]) Retrieve the values identified by the given keys that are stored in a specific server. If there is no value associated to one of the keys, then a *null* is included into the response, and a miss happens.

void delete (key) Delete the value identified by the given key of a specific server.

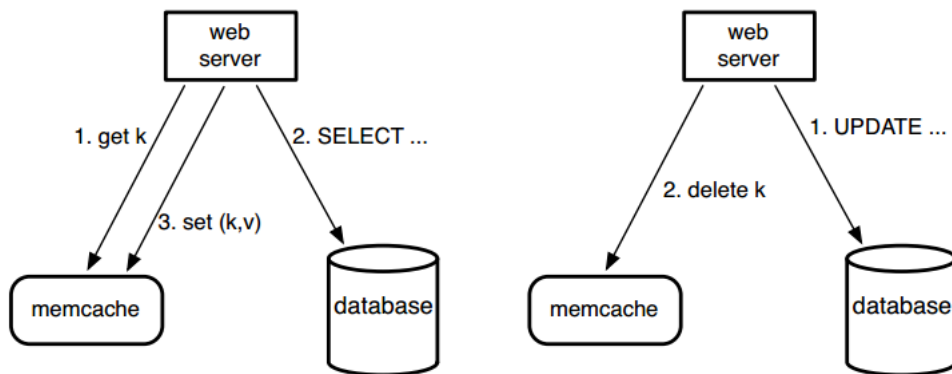
Every IMKVS implementation can have its own commands. However, all of them will rely these four basic commands. For instance, it is common to have others *set* command variations, which can accept a third parameter for defining a timeout which the given object will be stored into the memory.

In today's datacenters, IMKVS are mostly used to serve as a caching layer that holds complex and time consuming tasks, like large database queries results. The most common approach followed by the applications that uses these caching layers consists in these three basic steps: (1) when a request arrives, check if the related object has already been stored in the cache, if find it, return it, otherwise; (2) run the database query and process whatever is needed, and; (3) store the result into the cache for future requests. Such strategy is called "look aside", which is used by Facebook (NISHTALA et al., 2013) and most of complex cloud applications. Figure 5 shows the "look aside" strategy, where the left side represents the read path and the right side the write path.

It is easy to notice that IMKVS is a crucial piece of today's datacenters applications, whereas these systems need to be capable of serving a huge amount of application requests that

¹ Format: return type, command name, parameters

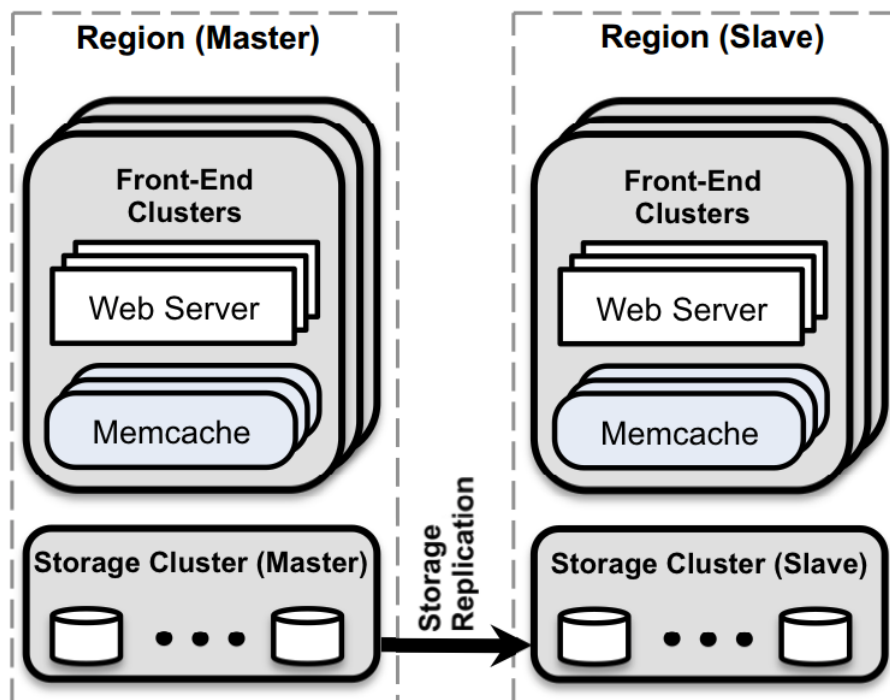
Figure 5 – The "look aside" strategy used to fill Memcached caches



Source: NISHTALA et al., 2013

aims to serve users' requests. Following the approach described above, the simplest user request may trigger multiple requests to the caching layer, which makes these caching layers be heavily accessed over time. Besides, due to the huge complexity involving cloud applications, it is usual to set up datacenters with tens of thousands instances of IMKVS systems. Figure 6 shows how Facebook sets up their application servers in conjunction with the database and the Memcached instances. Obviously this structure must be replicated thousands of times to represent the actual size of the infrastructure needed to support the social network systems.

Figure 6 – Facebook's servers architectural overview



Source: NISHTALA et al., 2013

3 RELATED WORKS

The problems related to SDN load balancing have been discussed by some researchers and some related works are briefly discussed in Section 3.1. Then, in Section 3.2, some closely related works about load balancing of IMKVS will be discussed in order to give an idea of similar problems were addressed by other researchers.

3.1 LOAD BALANCING ON SOFTWARE-DEFINED NETWORKS

The work (LI; PAN, 2013) have proposed an OpenFlow-based load balancer for Fat-Tree networks that supports multipath forwarding. Their proposal aims to recursively find the current best path from a source to a destination, load balancing the network by enabling the use of alternate paths at runtime, minimizing network congestion. Their algorithm works only on networks that operate on the Fat-Tree topology and use network metrics for choosing the best path.

The work (WANG et al., 2011) have proposed an interesting load balancing approach that aims to proactively load balance traffic from clients to servers by slicing the IP address space into trees that isolates a set of clients to a set of servers. The work uses the concept of server weighting, which defines a fixed portion of the clients to a server on the network. To do so, it is proposed the extensive use of wildcards, which may reduce forwarding performance and create management issues, as can be seen in (BATISTA et al., 2014). Furthermore, the proposed solution requires that, at certain conditions (network topology changes or server weight updates), a part of the network traffic passes through the controller, which could cause serious scalability problems that may lead the network controller to collapse. Network metrics are not considered.

The work (KOERNER; KAO, 2012) proposes an architecture that enables in-network load balancing of multiple services using OpenFlow. Their proposal relies on a set of SDN controllers on top of a FlowVisor instance (SHERWOOD et al., 2009), where each controller is responsible for load balancing the traffic of a specific service. The authors have focused on the architecture, so there is no information about particular service implementation, while the performed experiment does not fit real-world scenarios. The idea of using a set of controllers to handle exact services might be interesting in some specific cases, but has the drawback of not permitting multiple services to be handled by a single controller, which is the most common case of SDN deployment.

The work (HANDIGOL et al., 2009) shows Plug-n-Serve, a module that reside within an OpenFlow controller that is capable of perform load balancing over unstructured networks, aiming to minimize average response time of HTTP servers. Plug-n-Serve load balance HTTP requests by gathering metrics about CPU consumption and network congestion on the network links, which makes its load balancing algorithm to select the appropriate server to direct requests to, while controlling the path taken by packets on the network.

3.2 LOAD BALANCING OF IN-MEMORY KEY-VALUE STORAGES

The work (CESARIS et al., 2014) proposes Network-Assisted Lookups (NAL), a method to do rapid load balance of key-value storages through the existing IP infrascructure. The proposed solution consists in assign multiple IP addresses to each server of the caching layer, being each IP address mapped to a bucket of objects. The NAL Controller is responsible for collect the load of the buckets in order to notice performance degradation of servers. Since the controller recognize that a bucket is an issue, through pre-configured threshold values, it fires the migration process in order to move the bucket to the less loaded server into the caching layer. Insofar as each bucket is assigned to a IP address, the server that holds the recently moved bucket is also assigned to its IP address, in order to guarantee that further requests to the bucket's objects will be successfully accomplished. NAL was compared to CH and results shown that NAL achieves a balanced state close to half of the time that Consistent Hashing does.

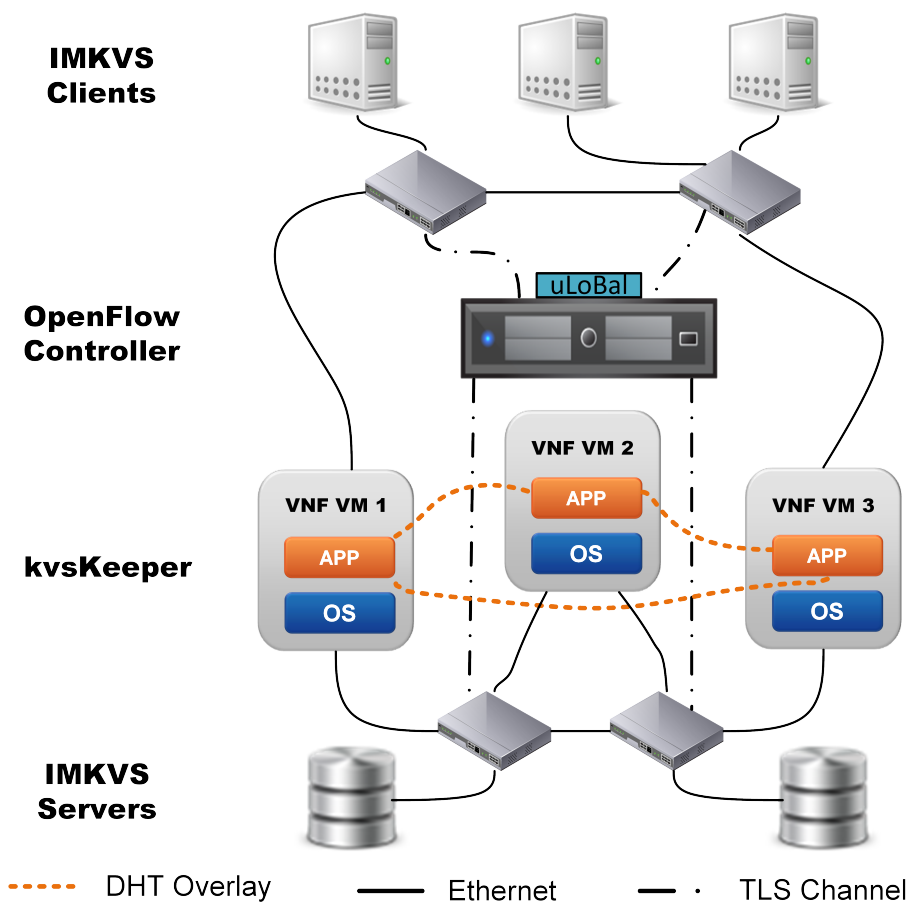
The work (ZHANG et al., 2014) proposes MemSwitch, a proxy capable of interpreting and redirecting Memcached requests. The proxy was used to serve as a load balancer and caching layer to a set of Memcached servers. MemSwitch was built upon the DPDK libraries that allow userspace network IO, minimizing the overhead of packet copying between the VM's OS-Kernel and the proxy application. The preliminary results showed that MemSwitch had better results than Twitter's TwemProxy (RAJASHEKHAR, 2012), reducing the request latency by half and increasing throughput by eight. Despite the good results shown in the study, there are some points to be aware of: (1) the proxy is application aware, making the proxy's location a negative characteristic for the deployment in real networks; (2) the study showed the proxy running only for a single Memcached server, while most of today's datacenters have tens of thousands of servers; (3) *set* requests can not have their size greater than a packet's size, limiting harshly Memcached's clients, since most of the packets that travel over the network have a size in order of bytes or kilobytes; and, (4) MemSwitch focuses only in handling requests made

by the UDP protocol, supporting TCP only when redirecting to a single server. This could be acceptable only if the clients tolerate UDP fails by considering it as a cache miss, otherwise the packet recovery would generate an overhead that would not justify the use of UDP instead of TCP. Finally, the authors have not shown how MemSwitch would handle *mget* requests.

4 TWO-PHASE LOAD BALANCING OF IMKVS CACHES

This Chapter will introduce the two-phase load balancing mechanism that aims to improve IMKVS cache performance. As said previously, the proposed solution consists in two systems, *uLoBal* and *kvsKeeper*, that are responsible for specific tasks of the load balancing process. *uLoBal* was developed to serve as a generic SDN load balancer that can operate with three distinct load balancing algorithms, which can be configured according to the type of service that should be load balanced. On the other hand, *kvsKeeper* is a VNF specialized in IMKVS traffic management and popular data replication that can be deployed on unstructured networks aiming the replacement of the CH algorithm. The next sections will introduce each system in details. Figure 7 shows how an arbitrary network would be with the presence of both *uLoBal* and *kvsKeeper*.

Figure 7 – An arbitrary deployment scenario for the two-phase load balancing



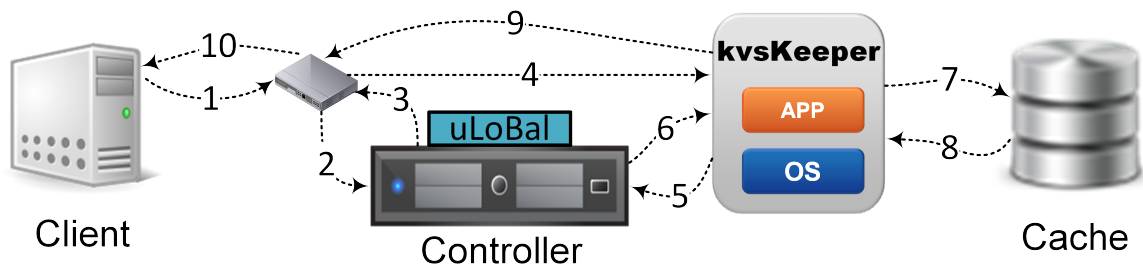
Source: The author

The first phase of the load balancing is made by *uLoBal*, a system that is deployed

atop the SDN controller, managing the network flows that are destined to a configured type of service, following a given load balancing operational mode (or algorithm). The *uLoBal* architecture was designed to provide high flexibility and to fit multiple and diverse scenarios and needs. As described previously, it is essential that an in-network load balancer can address different types of service, not being focused on specific scenarios, allowing high flexibility and helping to reduce operational costs over time, so *uLoBal* follows such design directive. Furthermore, *uLoBal* introduces a method for performing in-network load balancing that considers the load conditions from both network and servers, which makes the system to be dependant of updated network statistics, allowing a more efficient use of network resources.

The second phase of the load balancing is made by *kvsKeeper*, a VNF responsible for handling IMKVS traffic in order to forward commands to the best available IMKVS server, reducing both network and server overload. As said previously, *kvsKeeper* is also able to replicate popular data stored on servers. By performing replication on objects, *kvsKeeper* can forward *get* commands to a less loaded replica server. It is possible to have multiple VMs executing a *kvsKeeper* process in order to allow elasticity, so, it is necessary to create a mechanism that allows each instance to share their internal forwarding state, ensuring that neither addition nor removal of VMs cause data inconsistency. Thus, it is used a Distributed Hash Table (DHT) (STOICA et al., 2001) to hold the forwarding state across the VMs. Figure 8 shows how *uLoBal* and *kvsKeeper* work together while load balancing IMKVS traffic.

Figure 8 – Network communication while forwarding IMKVS traffic



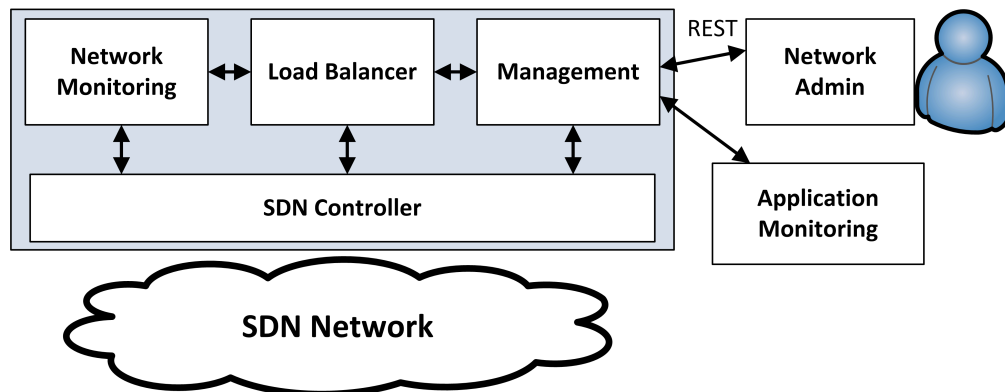
Source: The author

According to Figure 8, when the IMKVS client sends a command to any IMKVS server on the network, the client will send the packets a switch on the network (1). At the moment the switch receives the first packet of the IMKVS flow, it will send a *PacketIn* message to the OpenFlow controller (2), which will trigger the *uLoBal* management to install a new flow on the switch (3) that will be responsible to forward the traffic to the best *kvsKeeper* instance

that is available on the network (4). Once a *kvsKeeper* instance receives a IMKVS command, it asks the controller the current network conditions (5) and the most up-to-date statistics are delivered to *kvsKeeper* (6), so, *kvsKeeper* can decide to which IMKVS the arriving command will be forwarded. After the internal forwarding state has been updated, *kvsKeeper* forwards the command to the cache server (7) and process its response (8). After that, the command response is forwarded back to the client (9 and 10). It is important to notice that the steps (2), (3), (5) and (6) does not happen at every IMKVS command sent. Steps (2) and (3) will happen according to the presence of a flow entry on the switch's flow table, while steps (5) and (6) will happen periodically independently of arrival of commands.

4.1 ULOBAL: ENABLING GENERIC LOAD BALANCING ON SDN

Figure 9 – *uLoBal* architectural modules



Source: The author

The *uLoBal* architecture was designed to provide high flexibility and to fit multiple and diverse scenarios and needs. As described previously, it is essential that an in-network load balancer can address different types of service, not being focused on specific scenarios. The *uLoBal* follows such a design directive, aiming to identify a set of servers of a service using a unique identifier that will set the load balancing mode for that service. Furthermore, the *uLoBal* needs to be aware of network statistics in order to enable a load balanced forwarding, allowing a more efficient use of network resources. To this extent, *uLoBal* has three main modules: (1) the network monitoring module; (2) the load balancing module; (3) the management module, a Representational State Transfer (REST) API that allows management by the network administrator and integration with other monitoring tools. All these components are embedded on the SDN controller in order to avoid unnecessary communication with external services.

Figure 9 shows how these components are connected.

In an environment where there is a set of services that can be accessed through multiple endpoints, it is possible to perform in-network load balance in order to allow an even distribution of requests. The main objective behind *uLoBal* is to enable such service load balancing yet performing network load balancing. In networks where there are multiple paths between clients and endpoints, it is possible to use alternate paths as the network workload grows, mitigating problems related to networking congestion and reducing end-to-end latency. *uLoBal* can load balance the servers of the services by algorithms that use static and dynamic approaches that account for recent load information on both servers and network. Further subsections will give detailed information about each module.

4.1.1 Management Module

The *uLoBal* Management API can be accessed through a REST service, in order to allow administration and integration with external systems that can give updated load information of the services' servers. As the SDN controller cannot perform complex load monitoring at the servers, it is necessary to expose such service in order to allow an external monitoring tool to provide such information to the load balancer. Table 1 shows the *uLoBal* API methods.

Table 1 – *uLoBal* API methods and parameters

#	Method	Parameters	Used By
1	insertServiceEndpoint	(ServiceId, IP, Port)	Network Admin
2	deleteServiceEndpoint	(ServiceId, IP, Port)	Network Admin
3	updateServerLoad	(ServiceId, IP, Port, Load)	Monitoring System
4	changeLBMode	(ServiceId, Mode)	Network Admin

The *uLoBal* uses a tuple of three values to identify an endpoint (or server) of a service: the *ServiceId*, *IP* address and transport *port* values. The *ServiceId* value is any string that uniquely identifies the provided service on a set of servers, being each server identified by the *IP* address and transport *port* values. Any kind of service that uses the TCP or UDP protocols can be addressed using these values if all of its endpoints are accessed on the same transport port, which is the most common case. The methods 1 and 2 of the API defined in Table 1 uses exactly this tuple in order to add or remove endpoints to/from the load balancing. The method 3 is used by an external monitoring system that updates the load information of each server that provides access to the service, being the *Load* value the last load measure of a server normalized within

the interval $[0, 100]$. The *Load* is a generic value that can be calculated using any application's specific metric. In order to allow the network administrator to change the load balancing mode that must be used for a given *ServiceId*, the *Mode* value must be informed on the method 4 using one of three possible values: *ServerRoundRobin*, *ServerIpHash* or *NetServerLoad*, making the load balancer to change the balancing algorithm.

4.1.2 Network Monitoring Module

Since *uLoBal* uses network load information in order to load balance the service traffic on multiple forwarding paths, it is necessary to account such load data to perform the load balancing. The network monitoring consists of two steps: (1) collect statistics on every port of every switch of the SDN at predefined time intervals; (2) calculate the spanning tree of the network graph using the collected statistics as the cost metric.

The collecting process is made through the use of *StatsRequest* messages sent from the SDN controller to all switches on the network. Adrichem et al. describe a similar process in (ADRICHEM et al., 2014). When the load statistics are collected, the Dijkstra algorithm is used to compute the spanning tree that will be internally cached to be queried by the load balancing component. The cost metric used to compute the tree is given by $LinkCost = b + e$, where b is the percentage of the used link's bandwidth and e is the percentage of packets that have suffered of either drops or transmission errors. The *LinkCost* must be normalized within the interval $[0, 100]$ before the spanning tree is calculated.

4.1.3 Load Balancer Module

The *uLoBal* load balancer module is responsible for load balancing requests based in three operational modes: Round-Robin (RR), IP Hashing (IPH) and Network and Server Load (NSL), as shown in Table 2. Each one is identified in the REST API by *ServerRoundRobin*, *ServerIpHash* and *NetServerLoad*, respectively.

Table 2 – *uLoBal* operational modes

Name	Behavior
Round-Robin	Static
IP Hashing	Static
Network and Server Load	Dynamic

The load balancing mechanism is based on the principle of SDN, where the controller can push flows on the switches when there is no matching flows for an arriving packet. At this moment the controller receives a *PacketIn* message, that will be handled by the load balancer module if the destination IP and port match some previously inserted endpoint. The handling algorithm will depend on the configured load balancing mode for the matching *ServiceId*, with the NSL mode as the default mode. Algorithm 2 shows how the *PacketIn* message is handled by the controller.

Algorithm 2: PacketIn handling algorithm

Data: A *PacketIn* message

if *Packet's destination IP and port belongs to any ServiceId* **then**

sId := get the matching *ServiceId*; *lbMode* := get the configured load balancing mode for *sId*;

switch *lbMode* **do**

case *ServerRoundRobin*

 | call *RR(PacketIn, sId)*;

end

case *ServerIpHash*

 | call *IPH(PacketIn, sId)*;

end

case *NetServerLoad*

 | call *NSL(PacketIn, sId)*;

end

endsw

 Send the packet on a *PacketOut* message;

end

else

 | Ignore the packet, not interfering the normal processing;

end

The *uLoBal* provides two approaches for load balancing service requests. The first is static, an approach that does not consider either network or server metrics in order to make decisions to where forward incoming traffic, while the second approach is dynamic and uses this metrics in order to make traffic orchestration. The static approach has the advantage of less overhead since there is no need to be aware of such metrics, which may be useful to networks where there is little congestion and to services that need some predictability about which server will handle a given request. On the other hand, the dynamic approach enables better network traffic orchestration, using resources according to their most up-to-date metric information, which can help to reduce network congestion and, consequently, improve the users Quality of

Experience (QoE). Algorithms 3 and 4 uses the static approach, while the Algorithm 5 uses the dynamic approach.

Algorithm 3: RR algorithm

Data: A *PacketIn* message and the *sId*
 Increment the *RR* packet counter for the given *sId*;
 $pktC :=$ the *RR* packet counter for the given *sId*;
 $sLen :=$ get the amount of servers that belongs to *sId*;
 $sIndex := pktC \bmod sLen$;
 From the list of servers of *sId*, get the server *dstSrv* stored at the *sIndex* position;
 Get the less costly network path from the packet's source to *dstSrv* and send *FlowMod* messages to the switches on the path;

Algorithm 3 is a basic function that simply forward requests by choosing the destination server through the Round-Robin algorithm. Once it chooses the server, it gets the currently cheapest network path from the source to the destination in order to load balance the network. Its main characteristic is that the servers constantly receive a similar amount of requests, which can be useful for services that the costs of the requests are always the same.

Algorithm 4: IPH algorithm

Data: A *PacketIn* message and the *sId*
 Create a circular list *srvCirLst*;
foreach server *srv* that belongs to *sId* **do**
 | Calculate the hash *h* of the *srv* IP;
 | Insert *h* into *srvCirLst*;
end
 Get the packet's source IP and calculate its hash *srcH*;
 Insert *srcH* into *srvCirLst* and get its index *srcIdx*;
 Get the server *dstSrv* whose hash is stored at the $srcIdx + 1$ position on *srvCirLst*;
 Get the less costly network path from the packet's source to *dstSrv* and send *FlowMod* messages to the switches on the path;

Algorithm 4 aims to map a set of clients to the same endpoint following a CH approach (KARGER et al., 1997). The main goal is to always forward requests made by an user to the same endpoint, which may be useful for services that need to fetch context information before serving the request, since this context information can be locally cached.

Algorithm 5 was designed for considering the current network and server loads in order to choose the destination server. It works by selecting the endpoint that can be accessed with the minimum cost, being the cost calculated through the geometric mean of both server and

Algorithm 5: NSL algorithm

Data: A *PacketIn* message and the *sId*
 Create a Hash Map *cstMap* capable of storing multiple values mapped by a single key;
foreach *server srv* that belongs to *sId* **do**
 Get the less costly network path *nPth* from the packet's source to *srv*;
 netCst := the cost of *nPth*;
 srvCst := the current *srv* cost value;
 cost := $\sqrt[2]{netCst \times srvCst}$;
 Insert the *nPth* on *cstMap* mapped by *cost*;
end
 Get the minimum key *k* from *cstMap*;
 Get a random entry *nPth* mapped by *k*;
 Send *FlowMod* messages to the switches on the path *nPth*;

network costs. As a dynamic approach, it is not easy to predict which requests are going to reach a determined server, since it will depend exclusively on the current load from both servers and network.

In Algorithm 5, the cost of a network path, called *netCst*, can be calculated in many ways, depending on the target network, though the experimental version of *uLoBal* has used (4.1) to define the cost of a given path. Since a network path is composed by a set of links connecting a pair of network devices, let δ be the sum of the metrics of each link on the path.

$$\delta = \sum_{x=1}^n \left(100 - (\zeta - \nu) \right) \quad (4.1)$$

Note that Algorithms 3, 4 and 5 sends *FlowMod* messages to the switches within the network path from the source to the selected endpoint. Each *FlowMod* message consists in a header that matches the packet and two actions, (1) rewrite the packet's destination/source address; and (2) send the packet to the next hop. Furthermore, note that even though Algorithms 3 and 4 performs a static server load balancing, the chosen network path remains dynamic, since the selection process is based on the network metrics collected by the monitoring module. The flows generated by *uLoBal* use a *soft timeout* approach in order to set the duration of flows on switches, configuring the inactivity timeout to 1 second.

4.2 KVSKEEPER: ON THE LOAD BALANCING OF IMKVS

When clients send commands to IMKVS servers, if there's no path configured within the switch in which the packet has reached, the switch asks the SDN controller to where to

forward the packets. Once the request arrives at the controller, *uLoBal* will forward the packets to a *kvsKeeper* instance by using the NSL load balancing mode. To this extent, each *kvsKeeper* instance must be integrated with the *uLoBal* management module, by periodically sending load data through the method *updateServerLoad* on the REST API. The load of a *kvsKeeper* instance is defined in (4.2).

$$\gamma = (100 - \sqrt[3]{\varepsilon \times \xi \times \eta}) \quad (4.2)$$

The main reason that justifies the deployment of *kvsKeeper* as a VNF is the handling process of the connection between the IMKVS's client and server. Suppose that a client C_1 starts a request to a server S_1 and this server has a IMKVS instance that handles connections through TCP. The first thing that must happen is a three-way handshake, which consists in a negotiation between the hosts in which three packets need to be exchanged: *SYN*, *SYN + ACK* and *ACK* (POSTEL, 1981). To both C_1 and S_1 establish the connection, all these packets need to reach each other, forcing the switch and the controller to forward these packets without any intervention. When the packets arrive the switch, no entry flows will match with them, which will make the switch send a *PacketIn* message to the controller. Once the packet arrives at the controller, it must send a *PacketOut* message with the original packets' destination. After C_1 and S_1 establishes the connection, the controller could analyse further packets in order to extract the *key* which the command refers to, and only at this moment the controller could send a *FlowMod* to the switch in order to forward further packets directly. In other words, it is not possible to make a deep inspection of the packets until the connection be established.

Besides, there are a few considerations about this approach. First, when a command finishes, would be necessary to remove the recently installed flow, since the load balancing must be *per key*, not *per connection*. Second, the controller would be flooded with *PacketIn* messages since the time of life of a flow is too short, making the controller behave like the only network's switch. Third, it would not be possible to handle *mget* commands, due to the fact that if the load balancing is *per key*, then there is no guarantees that all the requested keys are stored on the same server, increasing the cache misses. Finally, it would create serious scalability problems, due to the high amount of packets being handled by the SDN controller.

Said that, it is crucial to move the task of inspecting packets to a VNF, which has to handle connections to both clients and servers. In the proposed solution, when a client opens a

connection to a server, the *uLoBal* is going to forward the packet to a *kvsKeeper* instance. At this point, *kvsKeeper* have to establish the connection with the client and start to listen its commands in order to forward it to the best IMKVS server at the moment. The use of *uLoBal* for performing the first load balancing is a crucial task to avoid the own *kvsKeeper* of becoming a bottleneck, since all the IMKVS would always be forwarded to the same point in the network.

The VNF is designed to deal with IMKVS requests by distributing them evenly across the caching servers. To do this, *kvsKeeper* must be aware of the specific IMKVS protocol to ensure that it will be compatible with all the available commands. When a IMKVS client tries to save an object into the caching layer, it will use a *set* command. In this case, there are two possible situations: the object whether exist or not in the cache layer. When the object does not exist, then *kvsKeeper* need to select the best caching server and forward the request to it. On the other hand, when the object exists, *kvsKeeper* need to overwrite the object on the servers that it has already been stored. All other IMKVS commands supposes that the requested object has already been stored into the cache. To this extent, when *kvsKeeper* handles *get*, *mget* and *delete* commands, it supposes that the requested object has already been stored into some cache server, so *kvsKeeper* just need to check which servers hold the object and then forward the command to the best IMKVS server. If the object has not been stored anywhere, so, *kvsKeeper* can send back a cache *miss* to the requester client, saving network communication by not forwarding a request that certainly would not be successful.

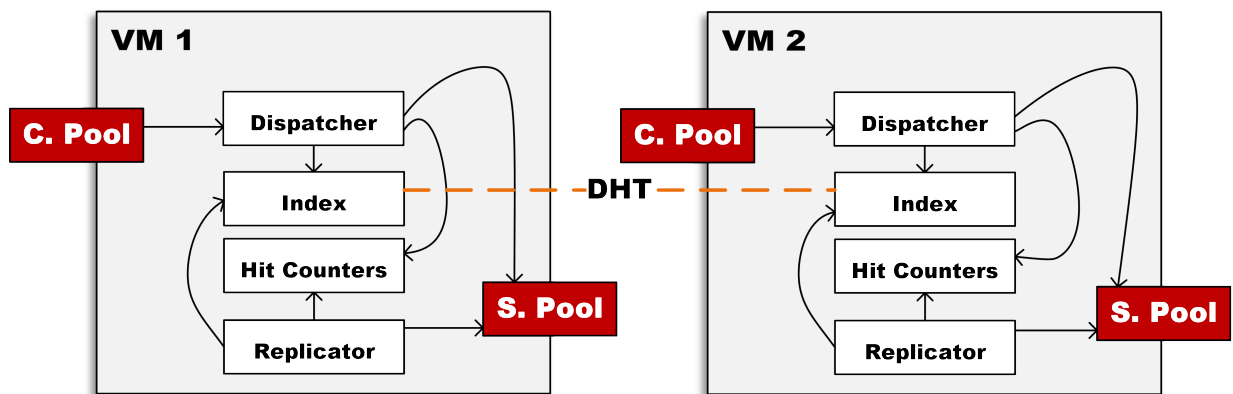
To accomplish the requirements of IMKVS commands, it is necessary to build an index that will be used to acknowledge where such objects were stored. Since every object is mapped by an unique key, it is possible to use a Hash Table that maps a key k to a server S . However, it is not possible to use a basic Hash Table to accomplish such a task, because *kvsKeeper* may have multiple instances and the requests that reach a specific instance are not forwarded by its keys. Hereupon, the index must be shared with all *kvsKeeper* instances in order to satisfy such requirements, and this is made by using a DHT.

4.2.1 Architectural Components

Each *kvsKeeper* instance has four basic components, as described in Figure 10. Arrows indicate that a component A depends of component B ($A \rightarrow B$) and inter-VM links indicates network communication. Both *C. Pool* and *S. Pool* are connection pools to both clients and servers, respectively. *Dispatcher* and *Replicator* components are the most important

ones. The first is responsible for handle and forward requests from clients to servers, according to the load balancing algorithms. The second is capable of getting the most popular cached objects, through the *Hit Counters*, and create copies of them across the caching layer, in order to minimize the appearance of hot spots within network. The *Index* maintains updated and distributed information about the location of each object into the caching layer, by mapping an object's key to a set of IMKVS servers.

Figure 10 – *kvsKeeper* architectural components on two VMs



Source: The author

These components are going to be described in details on the next subsections.

4.2.2 Index

In a DHT, every node that is part of its *overlay network* is responsible for holding a portion of the table. These distributed data structures have a set of algorithms for providing *lookup* operations and a series of mechanisms to fail tolerance. DHT is one of the basis of several distributed systems and Peer-to-Peer (P2P) applications, being capable of handling millions of requests swimmingly (RISSON; MOORS, 2006). The main reason of choosing a DHT instead of a unique centralized Hash Table is that by doing this the system would have only a single point of failure, what could be extremely dangerous in a datacenter environment. There are several protocols that implements DHT, but four are considered to be the first ones: Chord (STOICA et al., 2001), Content Addressable Network (CAN) (RATNASAMY et al., 2001), Pastry (ROWSTRON; DRUSCHEL, 2001) and Tapestry (ZHAO et al., 2004). The *kvsKeeper* shared index will use the Chord protocol, although it could be any other DHT protocol. Chord has been chosen due to ease of development and flexibility, but other DHT protocols should be

supported as well.

Chord provides *lookup* operations in logarithmic time, needing $O(\log N)$ messages to find any object, where N is the amount of nodes into the network (STOICA et al., 2001), which makes it scalable. Supposing that each *kvsKeeper* instance can handle 500 requests per second in a datacenter with 10^6 IMKVS requests per second, then the whole datacenter would need at least 2000 *kvsKeeper* instances to handle all these requests. This means that, in such simplistic environment, *kvsKeeper* would need only 11 lookup messages to find an indexed value. Although the logarithmic time of Chord lookup, it is important to avoid networking while handling a huge amount of IMKVS requests. Supposing that there are W switches between each *kvsKeeper* VM, then it would be necessary at least $2 \times W \times N$ packets traveling over the network just for a single index lookup.

Considering a datacenter environment, the network topology is one important feature to consider. Taking Facebook's "4-post" network topology (FARRINGTON; ANDREYEV, 2013) as an example, if there are some DHT nodes spread across some different clusters, a lot of network communication would be forwarded to both cluster and aggregation switches. It is preferable to perform intra cluster switching rather than inter cluster switching, avoiding high traffic at aggregation switches.

The main reason that IMKVS servers are deployed as caching layer is the high throughput that such systems are capable of provide, so, is necessary to use an internal Hash Table over Chord, avoiding networking in each IMKVS request, providing some improvement on the basic DHT algorithm. So, this internal index is a minor portion of the main DHT shared index. It works as follows. For each incoming *lookup* request on the index, *kvsKeeper* checks if the key is into the internal index, checking the shared index only if the key could not be found into the internal index. Once the shared index returns the requested value, it is replicated into the internal index for future *lookup* requests. When either *insert* or *delete* request is made, *kvsKeeper* sends it to both internal and shared indexes. By doing this *kvsKeeper* reduces expensive network communication, providing faster lookups in time $\Omega(1)$. Thus, both internal and shared indexes store an object that contains an arbitrary key k that maps a set of servers S ($k \mapsto S$).

4.2.3 Dispatcher Module

The main component of *kvsKeeper* is the Dispatcher, which contains all the VNF central logic. It is responsible to handle and forward requests from clients to servers, according

to the load balancing policy. This subsection is going to give a view of what happens when a request arrives into a *kvsKeeper* instance.

When a request arrives at a *kvsKeeper* instance, the dispatcher is going to inspect the command sent by the client to some server. The first step made by the dispatcher is to extract the key(s) whose command refers to. If the command has more than one packet (what depends on the IMKVS protocol), it will be necessary to read only the p first packets needed to extract the key, ignoring the inspecting process of further packets of the same command. Once the dispatcher has both key(s) and command type (*set*, *get*, *mget* or *delete*) it can do specific actions according to the command type. Algorithms 6, 7, 8 and 9 must be executed according to the command type.

Algorithm 6: Dispatcher's *set* algorithm

```

Data: A command  $C$  and a key  $k$ 
 $S := \text{Index.get}(k)$ ;
if  $S$  is not empty then
  | foreach server  $s$  in  $S$  do
  |   | Open a new thread and forward  $C$  to  $s$ ;
  | end
  | When the first thread finish forwarding, returns its response to the requester client;
end
else
  |  $s := \text{gBstSrv}()$ ;
  | Forward  $C$  to  $s$  and return its response to the requester client;
  |  $\text{Index.put}(k, s)$ ;
end

```

Algorithms 6 and 7 are the most important dispatcher's algorithms. The basic idea behind them is first check where the content was stored, by looking up into the index, and if the contents are not stored anywhere, then one of the available servers is selected based in their current load. Algorithm 7 can receive a *miss* from the target server even if the key is into the index, due to some cache eviction policy, like Last Recently Used (LRU). Also, it is possible that some IMKVS have a time to live set on some objects, causing the object removal without any client intervention. In addition to that, there is a global hit counter and *per-key* hit counter.

Algorithm 8 splits a *mget* request into several *get* requests. At first, it could be a bad idea because it is better to do a bulk request than multiple individual requests, but the problem here is that when a server handles a high amount of keys per request, the server processing can be a bottleneck, as described in (RAINDEL; BIRK, 2013). Since each sub request is made in

Algorithm 7: Dispatcher's *get* algorithm

Data: A command C and a key k
 $S := \text{Index.get}(k)$;
if S is not empty **then**
 | $s := \text{gBstSrv}(S)$;
 | Forward C to s and get resp ;
 | Return resp to the client;
 | **if** resp is a miss **then**
 | | Delete k from the index;
 | **end**
 | **else**
 | | $\text{GlobalHitCounter} += 1$;
 | | $\text{KeysHitCounter}[k] += 1$;
 | **end**
end
else
 | Return a *miss* to the client;
end

Algorithm 8: Dispatcher's *mget* algorithm

Data: A command C and a key set K
foreach key k in K **do**
 | Open a new thread and call $\text{get}(C, k)$;
end

parallel, the time to finish the whole *mget* command will depend on network conditions.

Algorithm 9: Dispatcher's *delete* algorithm

Data: A command C and a key k
 Delete k from the index;
 Send a *deleted* message to the requester client;

Algorithm 9 aims to reduce the request latency by avoiding communication with the servers that hold the requested key. This is not a problem since most of the IMKVS use eviction policies in order to reuse memory.

Algorithm 10 shows the *gBstSrv* procedure used in Algorithms 6 and 7. The main goal is to find the best server to forward an incoming IMKVS command, using the same metrics defined in (4.1) and (4.2), considering the IMKVS servers though.

Algorithm 10: gBstSrv procedure

Data: An optional servers set S (when omitted, consider all available servers)

Result: A best server s

foreach *server* s *in* S **do**

 Get the network cost L_{Net} to reach s ;

 Get the server load L_{VM} of s ;

end

Get the set of servers $bSrvs$ that minimizes $\sqrt[2]{L_{Net} \times L_{VM}}$;

return a random srv from the set $bSrvs$;

4.2.4 Replication Module

Algorithm 11 describes how the replication mechanism is executed within each *kvsKeeper* VM instance. It checks the Hit Counters in order to retrieve the keys whose popularity is greater than a given replication threshold, named R_t . Having all popular keys, the algorithm reset all counters and replicate the related objects to $p \times R_f$ servers, where R_f is the replication factor and p is an object's popularity percentage. For example, if a key was responsible for 80% of the hits in a VM, then the corresponding object would be replicated over 40% of the servers, if the replication factor was set to 0,5. Both parameters allow a fine tuning in the algorithm behaviour, being able to adapt it to diverse environments and needs. The execution of the replication mechanism depends on previous scheduling made by the network administrator, also in order to adapt the mechanism to its needs.

Simultaneous executions of the replication algorithm on several VMs will cause network congestion if there are very popular keys in multiple servers. Said that, would be advisable to use mutual exclusion (RICART; AGRAWALA, 1981; MAEKAWA, 1985; SUZUKI; KASAMI, 1985) between the VMs in order to avoid multiple replication processes running at the same time, or even scheduling Cron (KELLER, 1999) tasks in each VM that does not overlap each other.

Algorithm 11: Replication algorithm

Data: A replication threshold R_t and a replication factor R_f

Get the current VM load L_{VM} ;

if $L_{VM} \leq L_{max}$ **then**

$P := []$;

$g_{hc} := GlobalHitCounter$;

foreach k_{hc} **in** $KeysHitCounter$ **do**

$p := \frac{k_{hc} \times 100}{g_{hc}}$;

if $p \geq R_t$ **then**

 Append $\{key : k_{hc}.key, pc : p\}$ into P ;

end

end

 Reset both $GlobalHitCounter$ and $KeysHitCounter$ to 0;

$S :=$ all IMKVS servers sorted by its loads;

foreach p **in** P **do**

 Get from the index the servers set S_k which k refers;

$S_r := S - S_k$;

$c := p.pc \times R_f$;

 Replicate $p.key$ over the c first S_r servers;

end

end

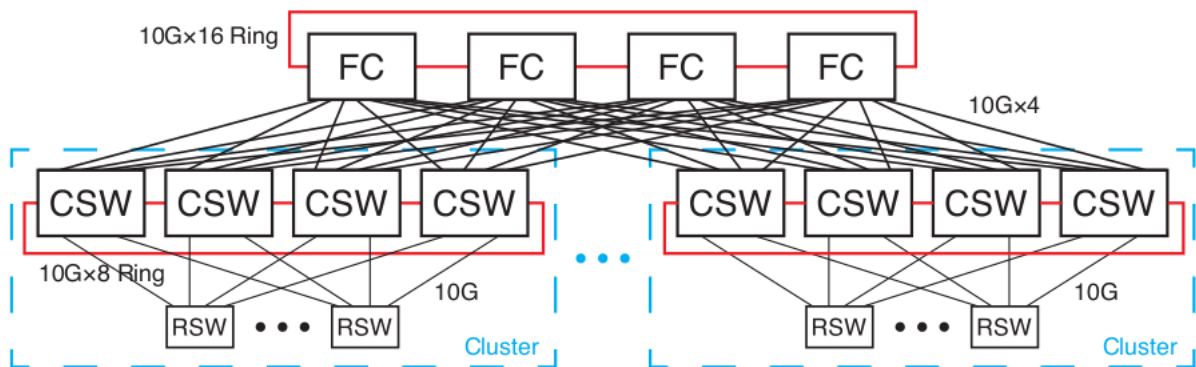
5 EVALUATION

The evaluation of the proposal was made in four experiments. Further sections will describe each experiment in details.

The evaluation environment was based on a virtualized network using Mininet (LANTZ; HELLER, 2015) on top of an Amazon EC2 *c4.8xlarge* instance. The Mininet system permits the specification of a network interconnecting virtualized devices. Each network device, hosts, switches and controller are virtualized and communicate via Mininet. A Python script is used to create the topology in Mininet, and the traffic flow control is made by the OpenFlow controller. Therefore, the test environment implements and performs the actual protocol stacks that communicate with each other virtually. The Mininet environment allows the execution of real protocols in a virtual network. The possibility to set link bandwidth and delay in Mininet allowed the experiment similar to an actual scenario.

The chosen OpenFlow controller was the Floodlight (ERICKSON, 2012), due to its simplicity and development flexibility. The network metrics were collected on Floodlight, according to (ADRICHEM et al., 2014), while the server metrics were collected using native monitoring tools from Ubuntu Server 14.04. All metrics were collected periodically, with a interval of 1 second between successive monitoring calls.

Figure 11 – Facebook's "4-post" network topology



Source: FARRINGTON; ANDREYEV, 2013

Figure 11 shows the Facebook's "4-post" (FARRINGTON; ANDREYEV, 2013) topology, that was used in experiments 2, 3 and 4. Random latencies from 0 to 1 milliseconds and 0 to 3% of packet losses was set in all links in each experiment execution. Bandwidth values was set according to the topology description in (FARRINGTON; ANDREYEV, 2013). IMKVS commands' keys and values were generated randomly according to the description made in (XU

et al., 2014). Memcached (FITZPATRICK, 2004) was chosen to be the IMKVS server.

5.1 EXPERIMENT 1

It is necessary to validate how the SDN load balancing performed by *uLoBal* is going to behave in an arbitrary service in different load balancing modes. To this extent, the experiment 1 was divided in two scenarios.

The first scenario aims to compare how the network load balancing approach affects the perceived delay on clients when requesting some content from a Content Delivery Network (CDN) server that operates through HTTP. This scenario aims to isolate the network load balancing from the server load balancing, allowing better analysis of the proposal behavior. So, the network topology has only a single server that is accessed by several clients spread over the network. Since there is a single server, all the requests are forwarded to a single point of the network, which makes the load balancer to perform swapping on the forwarding paths at runtime, balancing the traffic load. For comparability, the experiment has addressed the use of *uLoBal* operating on the NSL mode and the use of a traditional Shortest Path First (SPF) approach.

The second scenario aims to compare how the different load balancing modes affect (1) the client perceived latency and (2) the server load. Again, the clients are going to request contents on CDN servers that operate through HTTP. In this scenario, it is used three CDN servers, each one providing an endpoint for content delivery, making *uLoBal* to forward requests to one of these servers, following the configured load balancing mode.

Figure 12 shows the Abilene network topology, which has been used to perform the experiment 1. The topology's sources were obtained from the TopologyZoo (KNIGHT et al., 2011) and parsed according to the method described by (GROSSMANN; SCHUBERTH, 2013). In the first scenario, the server was positioned at the network node represented by the Indianapolis city. In the second scenario, the servers were positioned at the New York, Washington and Sunnyvale cities. Neither link latency nor bandwidth has been modified in the experiments. In each city that did not contain any CDN endpoint, a set of 10 clients has been placed in order to make content requests. Each client was configured to perform sequential requests to a randomly chosen server from the available servers of the experiment. Of course, it is the load balancer's job to redirect the request to the proper server, following the load balancing mode.

Figure 12 – Network Topology for the *uLoBal* experiments

Source: QUOTIN, 2016

5.2 EXPERIMENT 2

In order to check how *kvsKeeper* would benefit the applications that rely on operations made on multiple distributed IMKVS caches, the load balancing of these requests must be evaluated. The second experiment aims to evaluate how *kvsKeeper* would impact the time for a IMKVS client execute the four basic operations on a IMKVS server. By evaluating this, it would be possible to check if the overhead of inclusion of a VNF on the network would be worthwhile to the clients.

Said that, 10 clients were configured to trigger 100 operations per second to a set of 10 IMKVS servers. Just a single *kvsKeeper* instance is used, so *uLoBal* becomes useless in balancing load on VNF instances, while the network load balancing is performed normally through the NSL mode. NSL has been chosen due to its capacity of load balancing dynamic workloads, which makes it the better choice for IMKVS services. For comparability, the same amount of clients and servers were configured to use the traditional CH.

5.3 EXPERIMENT 3

Both *uLoBal* and *kvsKeeper* must be set up together in order to evaluate how the whole load balancing would act in an actual datacenter scenario, checking the relationship between caching performance and the number of load balancers on the network.

The third experiment aims to evaluate the relationship between the number of *kvsKeeper* instances with the achieved performance of IMKVS commands. To this extent, the *mget* command execution time was chosen as the baseline performance metric, since the *mget* command is the responsible for most of the traffic IMKVS in most applications. Facebook has reported about 99.8% of read operations at one of its cache pools (XU et al., 2014). This experiment is important to answer the question about how many *kvsKeeper* instances would be needed to achieve a performance metric at different workloads, which is usefull for provisioning and cost planning.

5.4 EXPERIMENT 4

Finally, as an extension of third experiment, it is necessary to check how the loads from both network and servers are affected when the proposed solution is deployed on a datacenter, comparing the results with CH.

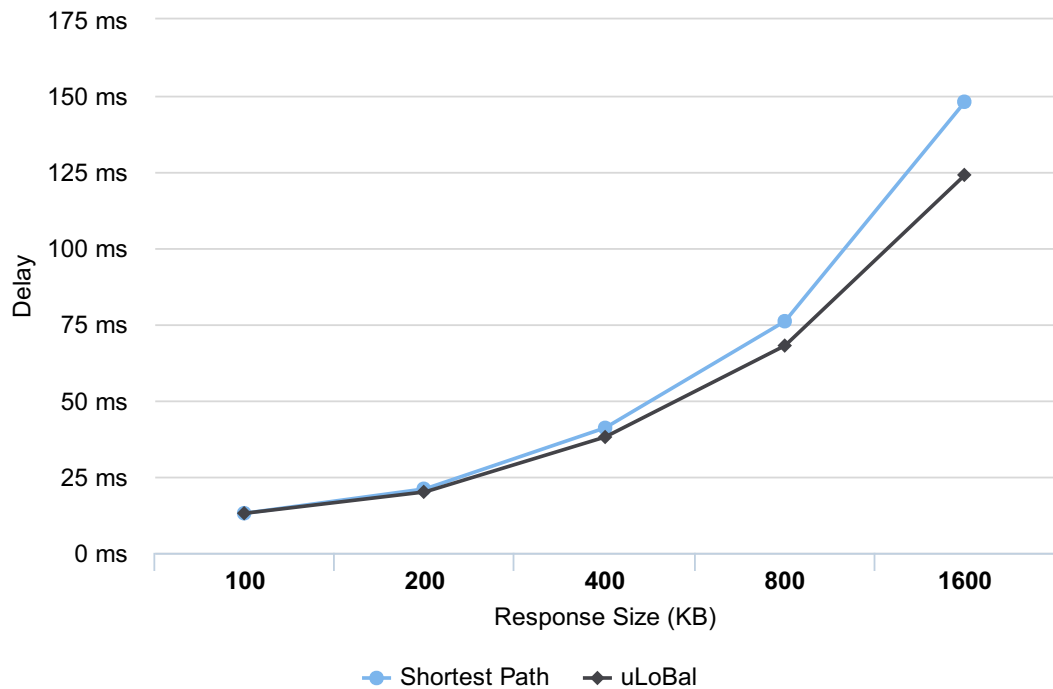
The fourth experiment aims to check how better *uLoBal* and *kvsKeeper* are compared to the traditional load balancing approach, the CH. The results of this experiment could be the most important indicator to be considered while deploying the proposed solution on an actual production network. The network load metrics were collected on the own SDN controller, while the server metrics were collected using the monitoring tools provided by their operating system. Since the results needed to be compared with CH, both keys and objects were generated just once to ensure that the load measures would be made at equal traffic conditions.

6 RESULTS

This Chapter is going to present the results of the experiments described in the last Chapter. Further sections follows the same presentation pattern found on Chapter 5.

6.1 EXPERIMENT 1

Figure 13 – SDN load balancing compared to a traditional SPF forwarding strategy

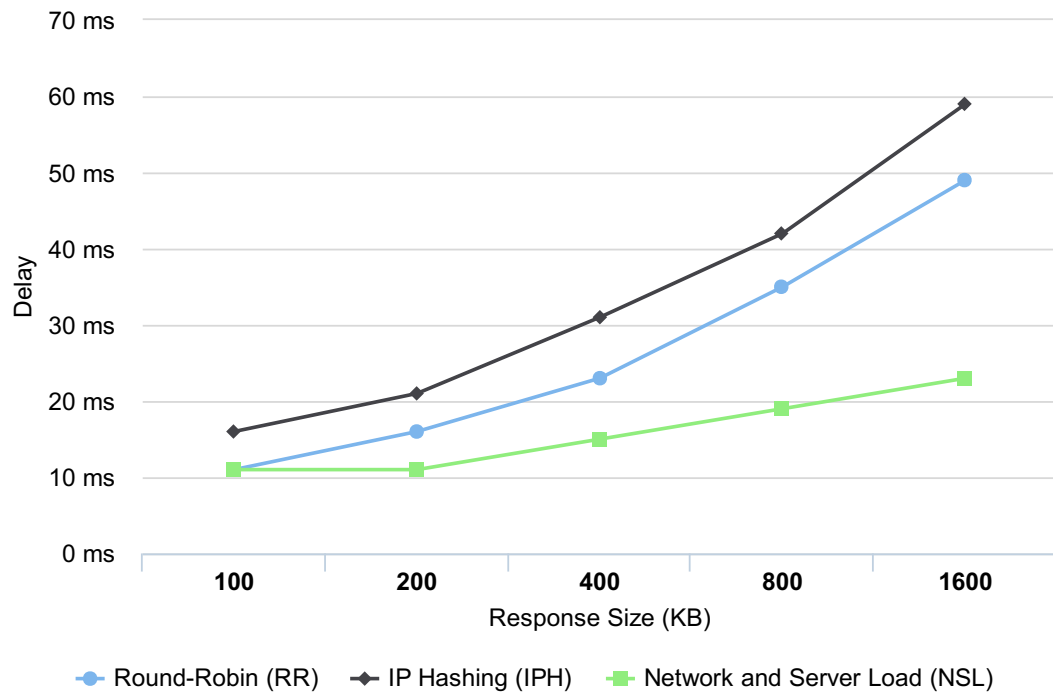


Source: The author

Figure 13 shows the results for the first scenario of the first experiment, where the network load balancing was compared to a SPF approach. By looking at the results, it is possible to notice that at low workloads, these approaches did not show significant differences, suggesting that the proposed network load balancing scheme is not relevant. However, when the workload starts to grow, there is an improvement in order of tens of milliseconds on the clients perceived delay, suggesting that this network load balancing approach can be useful as the workload grows. Besides, it is possible to conclude that at high workloads, the network would benefit from such load balancing mechanism, since congested paths would be avoided.

Figure 14 shows the results for the second scenario of the first experiment, where the load balancing modes can be compared with each other. Both RR and IPH have similar

Figure 14 – Message delivery delay when using SDN load balancing modes

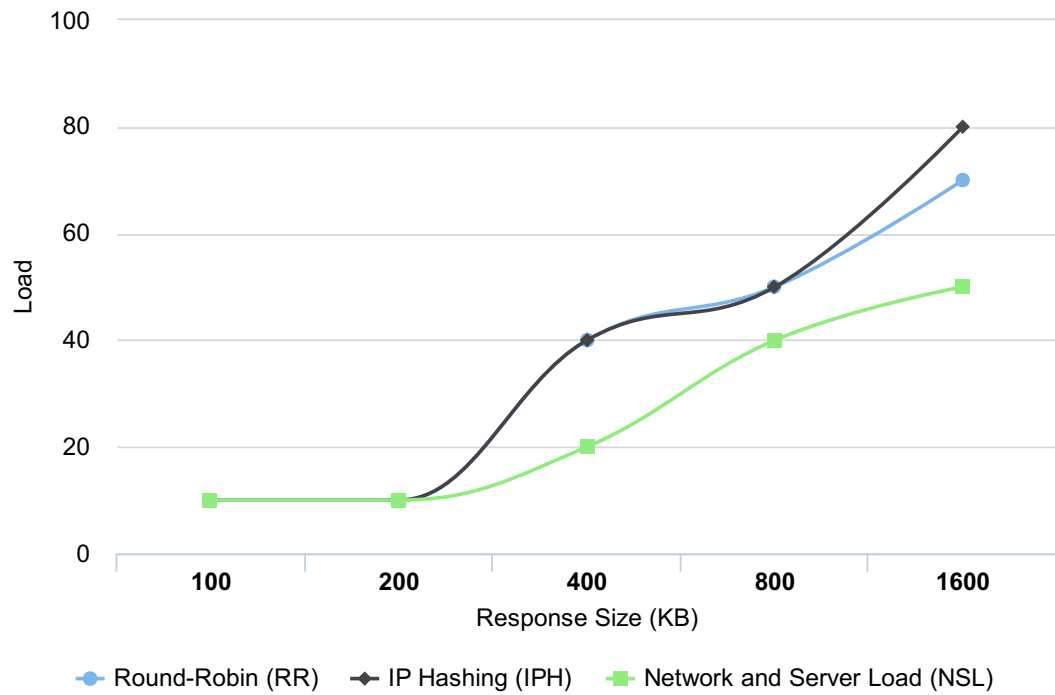


Source: The author

behaviors at different workloads, although the RR shows better latency results. Such results can be explained by the fact that the network nodes are spread over an area of a whole country, with the servers being positioned at the edges of the network, which makes the use of these techniques a bad idea due to the high distance, increasing the end-to-end delay. It is possible to notice that the NSL mode outperforms both RR and IPH modes, which can be explained using both server and network metrics when deciding to which server the requests will be forwarded, considering always the less costly network path at the moment. When the responses' sizes were 1600 KB, the improvement was about 53% and 62%, compared to RR and IPH modes, respectively.

Figure 15 shows the results where the average servers' load can be compared according to the load balancing mode. Again, both RR and IPH modes are similar to different workloads, unless the responses' sizes were 1600 KB. This result was being expected, since these load balancing schemas aim to distribute the requests evenly across the servers, not looking for any external factor on the decision-making process. For this reason, it is also possible to notice that NSL can alleviate the average server load in most of the workloads, suggesting that the proposed load balancer can be useful in different production networks with distinct workloads when configured to operate in this mode.

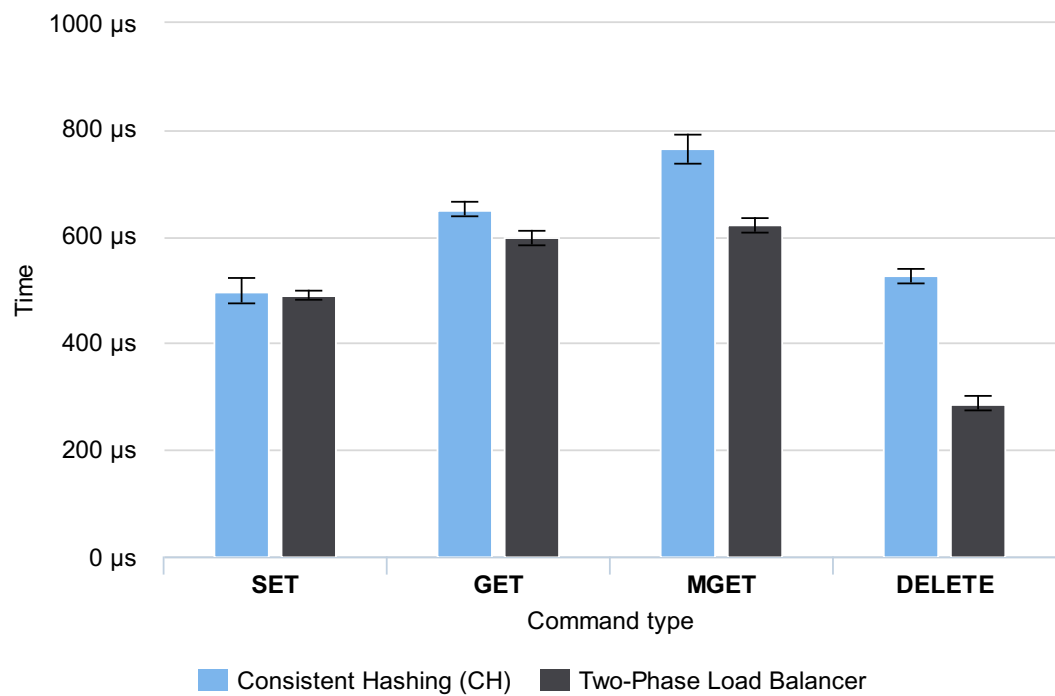
Figure 15 – Servers load levels when using SDN load balancing modes



Source: The author

6.2 EXPERIMENT 2

Figure 16 – IMKVS commands execution time comparison

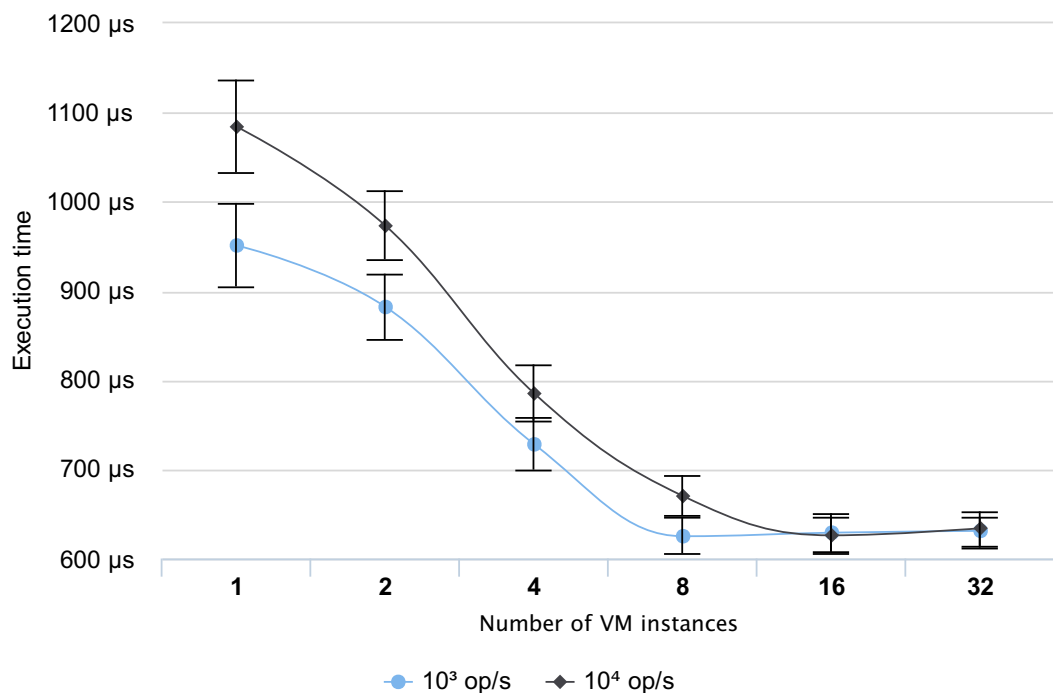


Source: The author

Figure 16 compares the execution time of each IMKVS command. The results show that, in all commands, the proposed load balancer improves the performance of IMKVS requests when compared with CH. The most notable improvement can be seen on the *mget* command, which time was reduced in 18%. Furthermore, the *delete* command had an improvement of 45%. Although *delete* had an improvement greater than *mget*, the last is the most notable improvement, since most of the workload is formed by read commands.

6.3 EXPERIMENT 3

Figure 17 – Execution time of *mget* command with multiple *kvsKeeper* instances

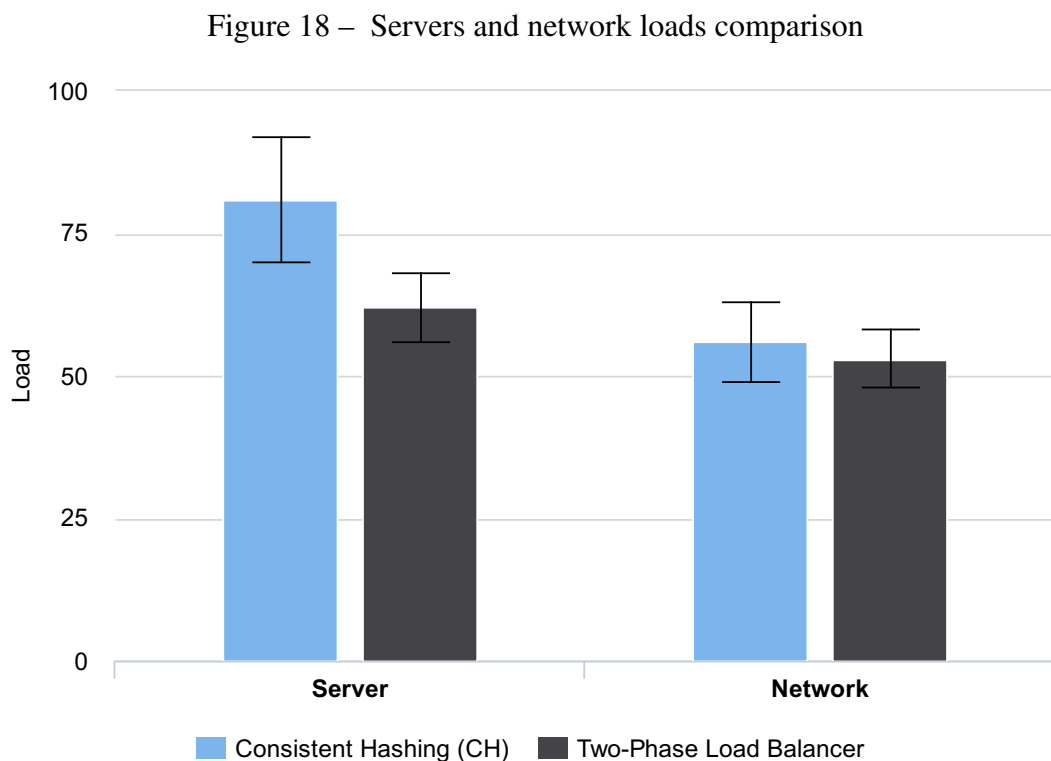


Source: The author

Figure 17 compares the execution time of *mget* command with different numbers of VM *kvsKeeper* instances. Results show that a lower amount of instances with a high op/s rate (underprovisioning) causes performance degradation, while the addition of new instances reduces the load, improving the results. This was an expected result, since it is the job of *uLoBal* to forward requests to *kvsKeeper* considering the reported loads on the VMs. Besides, it is possible to notice that overprovisioning does not improve the overall performance of *mget* commands, since it can be observed that such situation would increase the operation time. This behavior can be explained to the increase in lookup operations on the *kvsKeeper* shared index, which is

implemented on a DHT, so, if the number of instances increases, then more time is needed to find an arbitrary entry. Therefore, the main conclusion from this experiment is that, in production environments, a provisioning mechanism must be deployed in conjunction with the load balancer in order to avoid performance degradation. The NFVMANO is the most suitable component to perform such a task, which could work in conjunction with the SDN controller, since it is fully aware of the current network conditions.

6.4 EXPERIMENT 4



Source: The author

Figure 18 compares the average load of both servers and network when using CH and *kvsKeeper*. Results show that the proposed load balancer reduces the load of both servers and network by 23% and 5%, respectively. Furthermore, the load variation was reduced compared to CH. Such behavior would be expected due to the capacity of selecting the less loaded servers when storing content, and also due to the replication mechanism. Besides, the use of *uLoBal* on the SDN controller in conjunction with multiple *kvsKeeper* instances over the network could improve users QoE while reducing operational costs, when compared to the CH traditional approach.

7 CONCLUSION AND FUTURE WORK

Over the present work, it was shown that the use of caching based on IMKVS is a fundamental aspect of modern cloud applications, since these applications should require an efficient communication from the computer networks of datacenters. Besides, it was presented a technique called Consistent Hashing, that is widely used to manage some aspects of the storage of cached data, which has proved to be not so effective to the current demands imposed by modern cloud applications.

This paper has presented a two-phase load balancer for IMKVS systems. The work has proposed two modules to be deployed as a integrated solution that uses the architectural concepts and practices of SDN and NFV in order to offer a reliable and scalable solution that can be applied to real-world datacenters which support cloud applications. The first, *uLoBal*, a SDN based load balancer, is capable of load balance arbitrary services through the use of different forwarding approaches that address services of several types and nature. The second, *kvsKeeper*, a VNF specialized in IMKVS cache traffic orchestration that balances the placement of objects in the best servers, also providing replication mechanisms in order to relieve load and mitigate the appearance of hot spots in the datacenter network.

Experiments have shown that both modules are capable of providing better use of resources, reducing operational costs and improving users experience. On the *uLoBal* module, results showed that network load balancing was improved when compared to classic network forwarding, while helping to reduce load at distributed servers. On the *kvsKeeper* module, results showed that the proposal outperforms Consistent Hashing, relieving the load on servers by 23% and 5% on the network, while reducing the necessary time for complete the four basic commands of IMKVS systems.

The current work opened some research opportunities that can be addressed in the future. The *kvsKeeper* index is based on the Chord protocol, but the use of other DHT should be evaluated in order to check if Chord really is the best option for *kvsKeeper*. Furthermore, the index uses a strategy based on a small representation of the DHT being held in memory, the internal index. This approach was needed to avoid networking on expensive datacenter network segments, so it could be possible to develop novel DHT algorithms that are specific to some datacenter topologies, avoiding the need of *kvsKeeper*'s internal and shared indexes. Additionally, the use of a proactive approach to replicate popular objects could be proposed, by analyzing the traffic patterns related to a set of cached objects. Finally, the acquired knowledge about the

integration of SDN and NFV was usefull to start a new research project, called ContentSDN, which aims to use SDN and NFV to develop an Information-Centric Networking tool to improve web users' Quality of Experience by creating an in-network cache for HTTP requests.

BIBLIOGRAPHY

- ADRICHEM, N. L. V.; DOERR, C.; KUIPERS, F. et al. Opennetmon: Network monitoring in openflow software-defined networks. In: **IEEE. Network Operations and Management Symposium (NOMS), 2014 IEEE**. Krakow, Poland, 2014. p. 1–8.
- BATISTA, B. L. A.; CAMPOS, G. Lima de; FERNANDEZ, M. Flow-based conflict detection in openflow networks using first-order logic. In: **Computers and Communication (ISCC), 2014 IEEE Symposium on**. Funchal, Portugal: [s.n.], 2014. p. 1–6.
- CESARIS, D. D.; KATRINIS, K.; KOTOULAS, S.; CORRADI, A. Ultra-fast load balancing of distributed key-value stores through network-assisted lookups. In: SILVA, F.; DUTRA, I.; COSTA, V. S. (Ed.). **Euro-Par 2014 Parallel Processing**. Porto, Portugal: Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8632). p. 294–305. ISBN 978-3-319-09872-2.
- DECANDIA, G.; HASTORUN, D.; JAMPANI, M.; KAKULAPATI, G.; LAKSHMAN, A.; PILCHIN, A.; SIVASUBRAMANIAN, S.; VOSSHALL, P.; VOGELS, W. Dynamo: Amazon’s highly available key-value store. **ACM SIGOPS Operating Systems Review**, ACM, New York, NY, USA, v. 41, n. 6, p. 205–220, Oct 2007. ISSN 0163-5980.
- DORIA, A.; SALIM, J. H.; HAAS, R.; KHOSRAVI, H.; WANG, W.; DONG, L.; GOPAL, R.; HALPERN, J. **Forwarding and Control Element Separation (ForCES) Protocol Specification**. IETF, 2010. RFC 5810 (Proposed Standard). (Request for Comments, 5810). Available at: <<http://www.ietf.org/rfc/rfc5810.txt>>. Last accessed: 30 Jan. 2016.
- ENNS, R.; BJORKLUND, M.; SCHOENWAELDER, J.; BIERMAN, A. **Network Configuration Protocol (NETCONF)**. IETF, 2011. RFC 6241 (Proposed Standard). (Request for Comments, 6241). Available at: <<http://www.ietf.org/rfc/rfc6241.txt>>. Last accessed: 30 Jan. 2016.
- ERICKSON, D. **Floodlight Java based OpenFlow Controller**. 2012. Available at: <<http://floodlight.openflowhub.org/>>. Last accessed: 30 Jan. 2016.
- FARRINGTON, N.; ANDREYEV, A. Facebook’s data center network architecture. In: **Optical Interconnects Conference, 2013 IEEE**. Coronado, CA, USA: [s.n.], 2013. p. 49–50.
- FITZPATRICK, B. Distributed caching with memcached. **Linux Journal**, Belltown Media, Houston, TX, v. 2004, n. 124, p. 5–, Aug 2004. ISSN 1075-3583.
- GROSSMANN, M.; SCHUBERTH, S. J. **Auto-Mininet: Assessing the Internet Topology Zoo in a Software-Defined Network Emulator**. Bamberg, Germany: [s.n.], 2013. Available at: <https://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_lehrstuehle/informatik_ktr/Dateien/Publikationen/AutoMininet.pdf>. Last accessed: 30 Jan. 2016.
- GUDE, N.; KOPONEN, T.; PETTIT, J.; PFAFF, B.; CASADO, M.; MCKEOWN, N.; SHENKER, S. Nox: Towards an operating system for networks. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 38, n. 3, p. 105–110, Jul 2008. ISSN 0146-4833.
- HAN, B.; GOPALAKRISHNAN, V.; JI, L.; LEE, S. Network function virtualization: Challenges and opportunities for innovations. **Communications Magazine, IEEE**, v. 53, n. 2, p. 90–97, Feb 2015. ISSN 0163-6804.

HANDIGOL, N.; SEETHARAMAN, S.; FLAJSLIK, M.; MCKEOWN, N.; JOHARI, R. Plug-n-serve: Load-balancing web traffic using openflow. **ACM SIGCOMM Demo**, v. 4, n. 5, p. 6, 2009.

KANDULA, S.; SENGUPTA, S.; GREENBERG, A.; PATEL, P.; CHAIKEN, R. The nature of data center traffic: measurements & analysis. In: **ACM. Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference**. Chicago, IL, USA, 2009. p. 202–208.

KARGER, D.; LEHMAN, E.; LEIGHTON, T.; PANIGRAHY, R.; LEVINE, M.; LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: **Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1997. (STOC '97), p. 654–663. ISBN 0-89791-888-6.

KELLER, M. S. Take command: cron: Job scheduler. **Linux Journal**, Belltown Media, v. 1999, n. 65es, p. 15, 1999.

KNIGHT, S.; NGUYEN, H. X.; FALKNER, N.; BOWDEN, R.; ROUGHAN, M. The internet topology zoo. **IEEE Journal on Selected Areas in Communications**, IEEE, v. 29, n. 9, p. 1765–1775, 2011.

KOERNER, M.; KAO, O. Multiple service load-balancing with openflow. In: **IEEE. IEEE 13th International Conference on High Performance Switching and Routing (HPSR2012)**. Belgrade, Serbia, 2012. p. 210–214.

KREUTZ, D.; RAMOS, F.; VERISSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219.

LAKSHMAN, A.; MALIK, P. Cassandra: A decentralized structured storage system. **ACM SIGOPS Operating Systems Review**, ACM, New York, NY, USA, v. 44, n. 2, p. 35–40, Apr 2010. ISSN 0163-5980.

LANTZ, B.; HELLER, B. **Mininet: rapid prototyping for Software Defined Networks**. 2015. Available at: <<http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet>>. Last accessed: 30 Jan. 2016.

LI, Y.; PAN, D. Openflow based load balancing for fat-tree networks with multipath support. In: **Proc. 12th IEEE International Conference on Communications (ICC'13), Budapest, Hungary**. Budapest, Hungary: [s.n.], 2013. p. 1–5.

MAEKAWA, M. A n algorithm for mutual exclusion in decentralized systems. **ACM Transactions on Computer Systems (TOCS)**, ACM, New York, NY, USA, v. 3, n. 2, p. 145–159, May 1985. ISSN 0734-2071.

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. OpenFlow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 2, p. 69–74, 2008. ISSN 0146-4833.

MIJUMBI, R.; SERRAT, J.; GORRICO, J.; BOUTEN, N.; TURCK, F. D.; BOUTABA, R. Network function virtualization: State-of-the-art and research challenges. **Communications Surveys Tutorials, IEEE**, PP, n. 99, p. 1–1, 2015. ISSN 1553-877X.

NISHTALA, R.; FUGAL, H.; GRIMM, S.; KWIATKOWSKI, M.; LEE, H.; LI, H. C.; MCELROY, R.; PALECZNY, M.; PEEK, D.; SAAB, P.; STAFFORD, D.; TUNG, T.; VENKATARAMANI, V. Scaling memcache at facebook. In: **Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2013. (nsdi'13), p. 385–398.

Open Networking Foundation. **Openflow switch specification, version 1.3.5**. 2015. Available at: <<https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf>>. Last accessed: 30 Jan. 2016.

POSTEL, J. **Transmission Control Protocol**. IETF, 1981. RFC 793 (INTERNET STANDARD). (Request for Comments, 793). Updated by RFCs 1122, 3168, 6093, 6528. Available at: <<http://www.ietf.org/rfc/rfc793.txt>>. Last accessed: 30 Jan. 2016.

QUOITIN, B. **C-BGP Tutorial**. 2016. Available at: <<http://c-bgp.sourceforge.net/tutorial.php>>. Last accessed: 30 Jan. 2016.

RAINDEL, S.; BIRK, Y. Replicate and bundle (rnb) – a mechanism for relieving bottlenecks in data centers. In: **Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on**. Boston, Massachusetts USA: [s.n.], 2013. p. 601–610. ISSN 1530-2075.

RAJASHEKHAR, M. **Twemproxy: A fast, light-weight proxy for memcached**. 2012. Available at: <<https://blog.twitter.com/2012/twemproxy>>. Last accessed: 30 Jan. 2016.

RATNASAMY, S.; FRANCIS, P.; HANDLEY, M.; KARP, R.; SHENKER, S. A scalable content-addressable network. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 31, n. 4, p. 161–172, Aug 2001. ISSN 0146-4833.

RICART, G.; AGRAWALA, A. K. An optimal algorithm for mutual exclusion in computer networks. **Communications of the ACM**, ACM, New York, NY, USA, v. 24, n. 1, p. 9–17, Jan 1981. ISSN 0001-0782.

RISSE, J.; MOORS, T. Survey of research towards robust peer-to-peer networks: Search methods. **Computer Networks**, Elsevier North-Holland, Inc., New York, NY, USA, v. 50, n. 17, p. 3485–3521, Dec 2006. ISSN 1389-1286.

ROWSTRON, A.; DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. **Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms**, Springer Berlin Heidelberg, Heidelberg, Germany, p. 329–350, Nov 2001.

SANFILIPPO, S.; NOORDHUIS, P. **Redis**. 2016. Available at: <<http://redis.io/>>. Last accessed: 30 Jan. 2016.

SHERWOOD, R.; GIBB, G.; YAP, K.-K.; APPENZELLER, G.; CASADO, M.; MCKEOWN, N.; PARULKAR, G. Flowvisor: A network virtualization layer. **OpenFlow Switch Consortium, Tech. Report**, 2009.

STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK, M. F.; BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 31, n. 4, p. 149–160, Aug 2001. ISSN 0146-4833.

SUMBALY, R.; KREPS, J.; GAO, L.; FEINBERG, A.; SOMAN, C.; SHAH, S. Serving large-scale batch computed data with project voldemort. In: **Proceedings of the 10th USENIX Conference on File and Storage Technologies**. Berkeley, CA, USA: USENIX Association, 2012. (FAST'12), p. 18–18.

SUZUKI, I.; KASAMI, T. A distributed mutual exclusion algorithm. **ACM Transactions on Computer Systems (TOCS)**, ACM, New York, NY, USA, v. 3, n. 4, p. 344–349, Nov 1985. ISSN 0734-2071.

TAVAKOLI, A.; CASADO, M.; KOPONEN, T.; SHENKER, S. Applying NOX to the Datacenter. In: **Proceedings of workshop on Hot Topics in Networks (HotNets-VIII)**. New York, NY, USA: [s.n.], 2009.

TRAJANO, A. F. R.; FERNANDEZ, M. P. Two-Phase Load Balancing of In-Memory Key-Value Storages Through NFV and SDN. In: **2015 IEEE Symposium on Computers and Communication (ISCC)**. Larnaca, Cyprus: IEEE, 2015. p. 409–414.

TRAJANO, A. F. R.; FERNANDEZ, M. P. uLoBal: Enabling In-Network Load Balancing for Arbitrary Internet Services on SDN. In: **ICN 2016: The Fifteenth International Conference on Networks**. Lisbon, Portugal: IARIA, 2016. p. 62–67. ISBN 978-1-61208-450-3.

WANG, R.; BUTNARIU, D.; REXFORD, J. et al. Openflow-based server load balancing gone wild. In: **Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 2011)**. Boston, MA, USA: [s.n.], 2011.

XU, Y.; FRACHTENBERG, E.; JIANG, S.; PALECZNY, M. Characterizing facebook's mem-cached workload. **Internet Computing, IEEE**, v. 18, n. 2, p. 41–49, Mar 2014. ISSN 1089-7801.

ZHANG, W.; WOOD, T.; RAMAKRISHNAN, K.; HWANG, J. Smartswitch: Blurring the line between network infrastructure & cloud applications. In: **6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)**. Philadelphia, PA: USENIX Association, 2014.

ZHAO, B.; HUANG, L.; STRIBLING, J.; RHEA, S.; JOSEPH, A.; KUBIATOWICZ, J. Tapestry: a resilient global-scale overlay for service deployment. **IEEE Journal on Selected Areas in Communications**, v. 22, n. 1, p. 41–53, Jan 2004. ISSN 0733-8716.