CONTROLLER LOAD BALANCING SCHEMES IN SOFTWARE DEFINED NETWORKS

by

Hakan Selvi

B.S., Computer Engineering, Boğaziçi University, 2010

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2015

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Prof. Fatih Alagöz for many enlightening suggestions and guidance during the development of this thesis, and helpful comments on the thesis text. Without his good will, help, and support, it would not be possible to finish this thesis.

I am grateful to Prof. Tuna Tuğcu and Assist. Prof. M. Şükrü Kuran for their participation in my thesis committee and for their insightful and constructive comments to improve my thesis work.

I would like to offer my deepest thanks to Gürkan Gür for his mentoring, friendship, kind help, and patience.

I also offer my gratitude to my colleagues who are also my dear friends Orhan Ermiş and Selcan Güner for their friendship, ideas, and support.

Last, I would like to thank my family members for their patience and support through all the phases of my education.

This thesis has been supported by the EUREKA/Celtic-Plus and TÜBİTAK TEYDEB (no. 9130038) project *SDN Concept in Generalized Mobile Network Architectures (SIGMONA).*

ABSTRACT

CONTROLLER LOAD BALANCING SCHEMES IN SOFTWARE DEFINED NETWORKS

Software-defined networking (SDN) has been proposed as a new networking paradigm which separates the control and data forwarding planes. The controllers which are centralized network control entities are crucial for meeting the performance and efficiency requirements of these systems. The placement of these network entities, switch assignment and their load-aware operation is a challenging research question. Although various load-balancing schemes are proposed in the literature, this research topic is yet to be explored adequately. In this thesis, we discuss load-balancing issues in SDN and propose a load-balancing scheme, namely *Cooperative Load Balancing Scheme For Hierarchical SDN Controllers (COLBAS)*. Moreover, we evaluate the performance of this scheme and investigate important trade-offs.

ÖZET

YAZILIM TANIMLI AĞ KONTROLÖRLERİNDE YÜK DENGELEME

Yazılım Tanımlı Ağlar (YTA) kontrol ve veri düzlemlerini ayıran yeni bir ağ yaklaşımı olarak önerilmiştir. Bu sistemlerde merkezi kontrol birimleri olan kontrolörler başarım ve verimlilik hedeflerine ulaşmak açısından çok önemlidirler. Kontrolörlerin yerleşimi, anahtarlara atanması ve yük dengelemesini sağlayarak çalışması çetin bir araştırma sorusudur. Literatürde çeşitli yük dengeleme yöntemleri önerilse de bu konu hala araştırılmaya muhtaçtır. Bu tezde YTA'da yük dengeleme ile ilgili konuları tartışıyor ve Hiyerarşik YTA Kontrolörlerinde İşbirlikçi Yük Dengeleme adında bir yük dengeleme yöntemi öneriyoruz. Ayrıca bu yöntemin başarımını değerlendiriyor ve önemli ödünleşimleri inceliyoruz.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\widetilde{B}_{ij}$	Average number of flow-requests from switch i to switch j
C_s	Cost for controller s
$\mathbb{C}$	Set of controllers
D_{avg}	Minimum average node-to-controller transmission and controller processing delay
D_{is}	Transmission delay from switch i to controller s
D^t_{is}	Transmission delay from switch i to controller s
D^p_{min}	Minimum controller processing latency
D^t_{min}	Minimum node-to-controller latency
D^p_s	Processing delay of controller s
F_c	Current number of flow-requests in the controller C to be processed
L	Imbalance detection threshold
N_h	Number of hosts
N_n	Number of nodes
P^s_{ij}	The number of flow-requests that is about to be handled by controller s
$\overline{Q_s}$	Average number of flow-requests already handled by controller s
R_{ij}	The number of flow-requests in a certain T_c from switch i to switch j
R_{total}	Total number of flow-requests in the network
$\mathbb{S}$	Set of switches
T_c	Controller period
T_o	Timeout period for flow entry in rule table
T_s	Supercontroller period
v_M	Variable that makes controllers' load and propagation delay comparable
w	Weight of B_{ij} at T=-1
z	Weight of B_{ij} at T=0

α	Coefficient for cost function
γ	Delay multiplier for representing the delay for packet batches
λ	Host traffic generation rate (exponential)
μ_s	Packet processing power of controllers

LIST OF ACRONYMS/ABBREVIATIONS

ACL	Access Control List
API	Application Programming Interface
BGP	Border Gateway Protocol
CLI	Command-line Interface
COLBAS	Cooperative Load Balancing Scheme For Hierarchical SDN Controllers
CPU	Central Processing Unit
IP	Internet Protocol
IS-IS	Intermediate System to Intermediate System
MAC	Media Access Control
NIC	Network Internet Cards
OF	OpenFLow
OSGi	Open Service Gateway Initiative
OSPF	Open Shortest Path First
QoS	Quality of Service
SDN	Software Defined Networks/Networking
TCP	Transmission Control Protocol
TLS	Transaction Layer Security
ToR	Top-of-Rack
UDP	User Datagram Protocol
VLSI	User Datagram Protocol
VRF	Virtual Routing and Forwarding

1. INTRODUCTION

Traditional networks consist of a large variety of network nodes such as switches, routers, hubs, different network appliances, and many complicated protocols and interfaces, which are defined in detail through standardization. However, these systems provide limited ways to develop and adopt new network features and capabilities once deployed. Therefore, this semi-static architecture poses a challenge against adaptation to meet the requirements of today's network operators and end-users. To facilitate this network evolution, the idea of programmable networks and Software Defined Networking (SDN) has been proposed [4]. The key rationale behind SDN is to separate the control and data planes in the network as shown in Figure 1.1. The control functionalities are centralized in so called SDN controllers while the switches are basically transformed into very fast packet processors and actuators for packet forwarding. The communication between the packet forwarder and controller is carried using a control signaling protocol, OpenFlow being the most leading one [5].

Although SDN is a very promising paradigm, it also poses some challenges such as scalability, delay and communication overhead. In this thesis, we investigate load balancing schemes for hierarchical SDN controller configurations. For literature analysis, we identify important load balancing works in SDN research and focus on them for the sake of simplicity. We devise a load-balancing scheme detailed in Chapter 4.

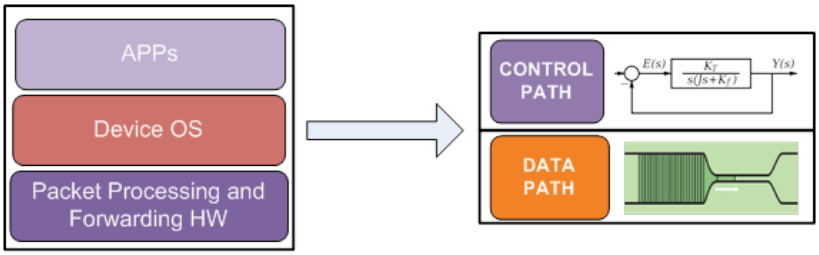

Figure 1.1. The key idea of SDN is to separate the control and data planes.

1.1. Contributions of This Thesis

Our key contributions in our thesis can be summarized as follows:

- A cooperative load-balancing scheme for hierarchical SDN controllers: We devise a load-balancing scheme, Cooperative Load Balancing Scheme For Hierarchical SDN Controllers (COLBAS), for SDN environment.
- A simulation framework based on SimPy for SDN systems: We develop a SDN simulation environment based on SimPy for simulating flow request and controller signaling [6]. This discrete-event simulator is primarily intended for load-balancing research but can be extended for other purposes.
- Investigation of key trade-offs for controller load-balancing schemes: We perform simulation-based experiments for investigating the effects of system parameters and their trade-offs during system operation.

1.2. Thesis Outline

In this thesis, we discuss important aspects of the controller load-balancing problem in SDN and study load-balancing schemes. First, we briefly introduce SDN controller and load-balancing concepts. Then, we discuss related works in the literature. We present our proposed scheme and implement an event-based simulation environment for performance evaluation. Finally, we conclude with key observations and some research directions and open problems for SDN controller load-balancing context.

The outline of this thesis is as follows: First, we introduce SDN and discuss salient points in Chapter 2. Next, in Chapter 3, we review the literature and summarize related works on controller load-balancing.

In Chapter 4, we propose a load-balancing scheme, namely *Cooperative Load Balancing Scheme For Hierarchical SDN Controllers (COLBAS)*. The system model (the environment, the network topology and related parameters) is also detailed for describing the investigated SDN configuration. The performance evaluation is provided

in Chapter 5. Available load-balancing algorithms are applied on the decision making process of controllers. The simulation framework to sample an SDN and control network traffic is also explained.

Finally, Chapter 6 concludes the thesis by summarizing and discussing our results, and rendering some research directions and open problems for SDN controller load-balancing context.

2. SOFTWARE-DEFINED NETWORKS

2.1. Main Concepts

Software-Defined Networking is devised to simplify network management and enable innovation through network programmability. In the SDN architecture, the control and data planes are decoupled and the network intelligence is logically centralized in software-based controllers. A SDN controller provides a programmatic interface to the network, where applications can be written to perform management tasks and offer new functionalities. The control is centralized and applications are written as if the network is a unified system. While this simplifies policy enforcement and management tasks, the binding must be closely maintained between the control and the network forwarding elements [4]. For instance, the OpenFlow Controller sets up OpenFlow devices in the network, maintains topology information and monitors the network status. The controller performs all the control and management functions. The information of location of the hosts and external paths are also managed by the controller. It sends configuration messages to all switches to set the entire path. The port for the flow to be forwarded to or other actions like dropping packets is defined by the OpenFlow Controller [7].

SDN paradigm provides not only facilitation of network evolution via central-

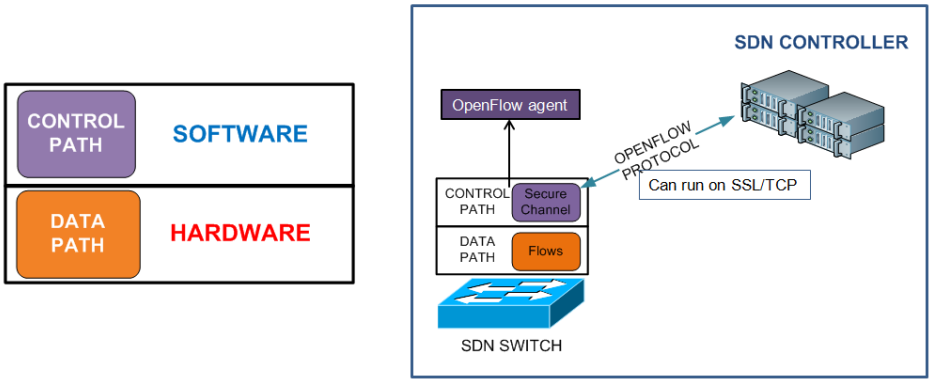

Figure 2.1. The controller and switch communication.

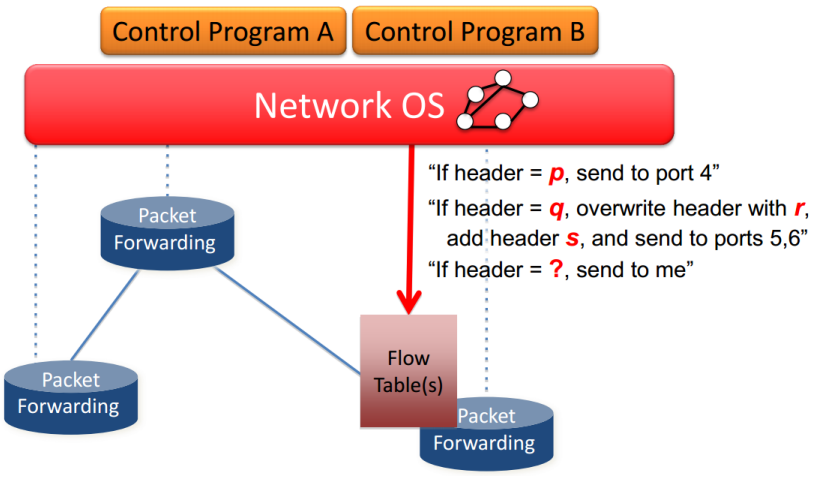

Figure 2.2. SDN control and Rule/Action approach.

ized control and simplified algorithms and programmability by enabling deployment of third-party applications, but also elimination of middle-boxes and rapid depreciation of network devices [8]. Since underlying network infrastructure is isolated from the applications being executed via an open interface on the network devices, they are transformed into uncomplicated packet forwarding devices . The controller related aspects of the software-defined network are paramount for addressing the emerging intricacies of SDN-based systems [4]. Therefore, although SDN paradigm will facilitate new degrees of freedom for traffic and resource management, it will also bring forth profound issues such as security, system complexity, and scalability. The controller load-balancing related challenges also emerge as critical factors on the feasibility of SDN.

A SDN forwarding device contains one or more flow tables consisting of flow entries, each of which determines how a packet performs [4] and the controller updates these flow tables and instructs the switches as to what actions they should take via a programmatic interface called Southbound interface [9] as shown in Figure 2.2. The division of work and some example processes are shown in Table 2.1. The control plane relies on typical VLSI whereas the data plane utilizes purpose-built hardware for wire-speed packet forwarding.

Since the control is centralized and applications are written as if the network is

Table 2.1. Control plane and data plane.

Processing Plane	What it does	Where it runs	How fast	Type of processes performed
Control Plane	Decides how to handle the traffic	Switch CPU	Thousand of packets per second	Routing protocols (OSPF, IS-IS, BGP), Spanning Tree, SYSLOG, AAA, CLI, SNMP
Data Plane	Forwards traffic according to control plane decisions	Dedicated Hardware ASICs	Millions/Billions of packets per second	Layer 2 switching, Layer 3 (IPv4 — IPv6) switching, MPLS forwarding, VRF forwarding, QoS marking, Classification, Policing, Security ACLs

a single system, policy enforcement and management tasks are simplified [10]. The outcome of completed experiments in [11] shows that a single controller has capability to manage excessive number of new flow requests in an unexpected manner. However, in a large-scale network deployment, this centralized approach has some limitations related to the interaction of control and forwarding elements, response time, scalability, infrastructure support and availability. Typically, large amount of network flow originating from all infrastructure nodes cannot be handled by a single controller because of the limited resource capacity. Another work by Voellmy et al. promotes this claim and shows multiple controllers ensure high fault tolerant networks with reduced latency [12]. Therefore, one must clarify following fundamental issues for SDN context [13]:

- How many controllers are needed?
- Where in topology should they go?
- How should they interact?
- How should they share the work?
- How should they serve different switches?

The answers of these essential questions depend on the network topology among other user-imposed requirements. From the latency perspective, a single controller would be mostly adequate. On the other hand, fault-tolerance and scalability concerns impel researchers to consider using multiple controllers in networks [13].

2.2. Performance Objectives

It is harder for fully distributed control planes to fail compared to decoupled planes. However, for controller placement and load sharing, there is trade-off among performance objectives of the decision making algorithm. Some algorithms could distribute the load over controllers to maximize fault tolerance, or some could minimize the propagation delay or distance to send packets through the shortest path between two hosts [13]. The general performance objectives for control allocation and their related effects on the network can be seen in Table 2.2. To minimize the delay of the

Table 2.2. Performance objectives and their effects on networks.

	Scalability	Reliability	Latency	Resilience
Fault Tolerance		X		X
Service Delay	X		X	
Utilization	X	X	X	

network-based services, scalability and latency are considered. For scalability, service delay may be traded while latency minimization directly benefits service delay. For utilization of the network, the effects of the scalability, reliability and latency are taken into account. Fault tolerance is directly affected by reliability and resilience objectives. Load balancing is necessary for minimizing latency while increasing scalability and efficiency.

For networks with more than one controller, a controller may become overloaded if switches mapped to this controller have large number of flows. However, the other controllers may remain underutilized. It is better to shift load across controllers over time depending on the temporal and spatial variation in traffic conditions. Static controller can result in suboptimal performance since no switch will be mapped into less loaded controller. The replacement of the controller can help to improve performance of over-provisioned controllers. Instead of using static mapping, elastic controller architecture can be used to map controller to balance the load as it reflects performance.

If the connection between controller and the forwarding planes is broken because of the network failures or the controller overload, some switches will be left without any controller and will be disabled in SDN-based networks. Network availability should be ensured to assure the reliability of SDN. Also the most important element of a SDN, controller, must maintain its functions for a healthy network. Therefore improving reliability is important to prevent disconnection between controller and the switch or between controllers. To reflect the reliability of the SDN controller and to find the most reliable controller placement for SDN, a metric can be defined as expected percentage of the valid control paths. A control path is defined as the route set between switches

and their controllers and between controllers. Consistency of the network should also be ensured when multiple controllers are in the network.

Each control path uses existing connection between switches. If control path is represented as a logical link, SDN control network is responsible to enable a healthy communication between switches and their controllers, which is a requirement for control paths to be valid. The failure of control paths, which means the connection is broken between switch and its controller or among controllers, results in the case where control network will lose its functionality. In a related work [14], it is assumed that a node is not able to route anymore and becomes practically off if it loses its connection to the controller. However, [15] suppose that in case of a controller is out of order, all the switches assigned to the failed controller can be reassigned to the second closest controller by using a backup assignment or signaling based shortest path routing. Until the last controller survives, all nodes are functional this way.

In addition, if the nodes are assigned to nearest controller using latency as a metric or shortest path distance between node and controller, there may be times that some controller are overloaded due to traffic flow. There can be an imbalance in the number of nodes per controller in the network. The higher the number of nodes attached to a controller, the greater the load on that controller. The increase in number of node-to-controller requests in the network induces additional delay due to queuing at the controller system. Therefore, we must determine a threshold for controller loads and if it exceeds this value, the network must be rescheduled before a failure occurs.

Load balancing in computer and network systems can be traced back to the very early computing research since the utilization and response characteristics of different computing and communicating machinery were to be optimized for different workloads in a spatiotemporal setting.

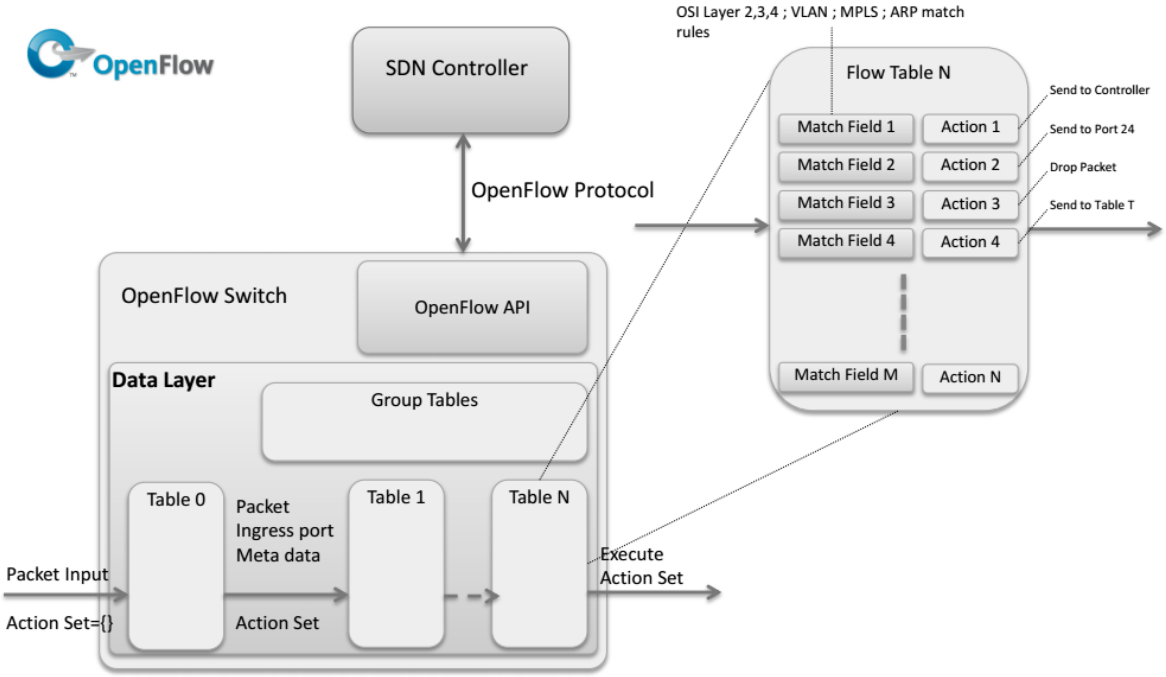

Figure 2.3. OpenFlow protocol.

2.3. OpenFlow

OpenFlow is a communications protocol that allows the control of the forwarding plane on a network device over the network through a well-defined "forwarding instruction set" [5]. It has become the de-facto standard for SDN systems. It is layered on top of the Transmission Control Protocol (TCP), and prescribes the use of Transport Layer Security (TLS). Controllers communicate over TCP port 6653 with OpenFlow supporting switches.

As shown in Figure 2.3, an OF switch has a controlled and secure communication channel with the controller. It has a flow table describing the match patterns, rules and actions. For an incoming flow request, the packet is processed and the table is scanned for a matching rule and thus a corresponding instruction/action. It provides a standard OF API to the controller for control related signaling. The controller provides the rules to the switch in case they are not already available on the switch itself.

3. RELATED WORKS

Load-balancing in SDN networks can be achieved by utilizing different SDN concepts. One way is to handle most of the traffic in data plane. The alternative way is to make decisions on control plane which is more suitable with the nature of SDN. Some important works in the literature are presented in this section.

3.1. Controlling a Software-Defined Network via Distributed Controllers

Many real world data centers handle about 150 million flows per second according to [16]. However, current OF controllers can handle at most 6 million flows per second on a high end dedicated server with 4 cores. Therefore, such data center networks need to handle existing flows by using an a controller with sufficiently many cores or a server cluster where each server consists limited cores. Yazici et al. present a framework providing dynamic controller addition and removal to the cluster in [1].

Process: Figure 3.1 shows clustered architecture of distributed controllers and it designed to work with all existing OF controllers with minimal or in required changes. Controller in the framework communicate with each other *JGroups* notifications and messaging infrastructure [17]. *JGroups* is a robust and flexible group communication library. Controllers select one of them having smallest system load as *mastercontroller* and the *mastercontroller* is responsible for realizing the controller-switch mapping updates. Then each controller checks if it is a master node, if so, it starts executing the regular switch-controller mapping decisions. To avoid frequent master changes, new selection occurs only when the current master becomes unsuitable based on some user-defined criteria.

If the listening master controller detects load imbalance it calls mapping function to generate a new mapping information. Mapping function takes current mapping, IP addresses of each controller and STATS as input parameters. STATS consists of statistics such as link traffic, controller loads, etc., which are collected by the controller

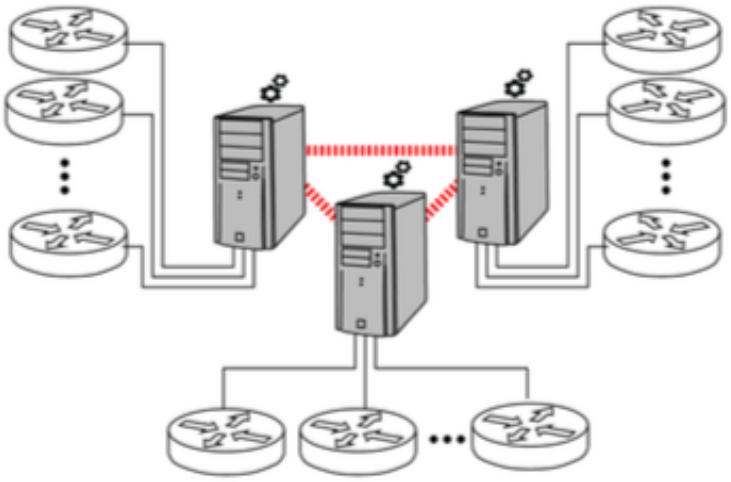

Figure 3.1. Distributed OF architecture [1].

architecture. Moreover, network operators can dictate to a certain flow routes over a certain machine. Newly created mapping information is stored locally by all controllers and this database is automatically synchronized amongst all of them via JGroups with every local update. When a new controller is added to the cluster, the rest of the cluster is instantly notified by the JGroups notification feature and a new call occurs to generate new mapping information. In a similar vein, if controller fails or turned off intentionally due to low traffic, the cluster is notified instantly and new mapping information will be created and installed.

For performance evaluation, they use four Beacon controllers to implement the cluster and add a new OSGi bundle, called *cluster* to them. Controllers run Debian GNU/Linux 6.0.4 (i686) on a system with Dual-Core 2.80GHz CPU, 2GB RAM, and JDK 1.6.0. The cluster power-up using the experimental setup takes approximately 12 seconds for the cluster OSGi bundle to start up initially when no other controllers are active. The start up time is approximately 3 seconds if there is at least one active controller in the cluster. The 9 seconds difference is due to discovery time during JGroups channel initialization. Getting notification during removal/arrival of a member in the cluster lasts under 50 milliseconds.

The result of migrating a group of switches from one controller to another is

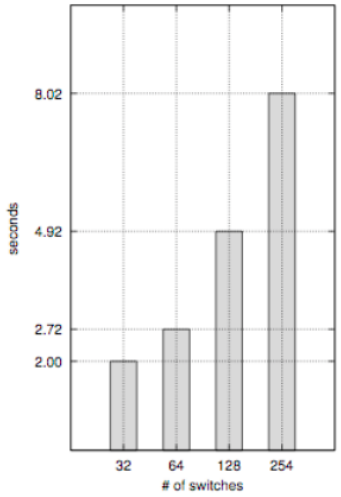

Figure 3.2. Switch migration results in [1].

shown in Figure 3.2. In the figure, they observe that even 254 simultaneous switch migrations clock around 8 seconds approximately 4 of which is due to IP alias relocate operations and this proofs that related work is more efficient than normal network operations in terms of switch migrations latencies.

3.2. DIFANE

Existing techniques for flow based networking rely too heavily on centralized controller software that installs rules reactively, based on the first packet of each flow. In [2], Yu et al. propose DIFANE, a scalable and efficient solution that keeps all traffic in the data plane by selectively directing packets through intermediate switches that store the necessary rules. DIFANE relegates the controller to the simpler task of partitioning these rules over the switches. DIFANE can be readily implemented with commodity switch hardware, since all data-plane functions can be expressed in terms of wildcard rules that perform simple actions on matching packets.

They implemented the DIFANE prototype using a kernel level Click-based OpenFlow switch and compared the delay and throughput of DIFANE with NOX, which is a centralized solution for flow management. They evaluated DIFANE using only one authority switch and then evaluated the throughput with multiple authority switches.Experiments

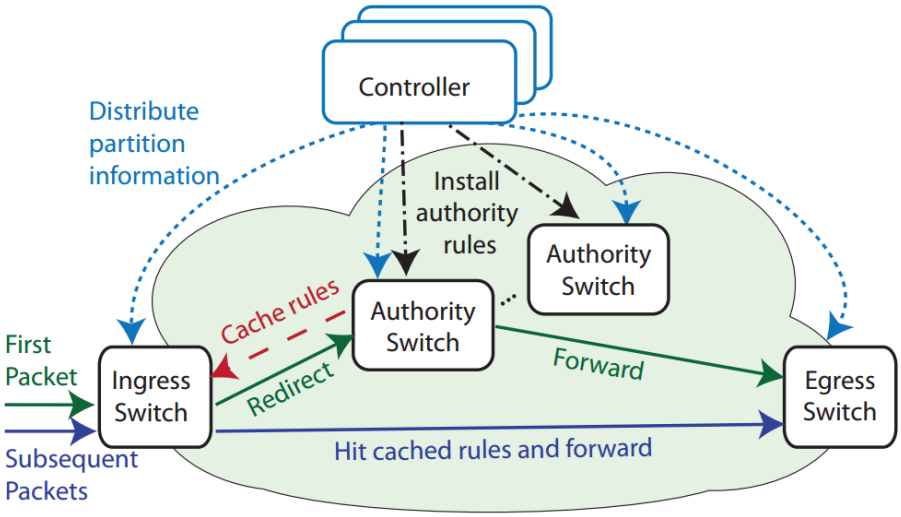

Figure 3.3. DIFANE flow management architecture [2]

show that DIFANE scales to larger networks with richer policies.

3.3. HyperFlow

HyperFlow is the first example of the distributed event-based control plane application which is implemented with C++ for NOX by minor modifications [18]. It localizes decision making into individual controllers and aims to minimize the response time of control plane to data plane requests.All the controllers have a consistent network-wide view and run as if they are controlling the whole network. They all run the same controller software and set of applications. Each switch is connected to the best controller in its proximity. If a controller becomes down, affected switches must be configured to connect to an active nearby controller. It provides network repartitioning flexibility and resilient to component failures.

HyperFlow proactively pushes state to other controllers by using a publish/subscribe messaging paradigm WheelFS to relieve cross-controller communications. Each controller selectively broadcast the events that change the state of the system through this messaging paradigm. An individual event may change the state of several applications, so this paradigm must ensure direct synchronization with the number of applications but in HyperFlow only a very small part of events cause changes to the network-wide

view. The order of events, except those affecting the same switch, does not change the network-wide view. That's why the messaging system should prevent the order of events fired by the same controller and should decrease the cross-controller message traffic to propagate events.

WheelFS is a distributed file system implemented to provide wide-area storage for distributed applications. HyperFlow is resilient to network partitioning because WheelFS is. When a network is partitioned, WheelFS on each partition smoothly lasts to work independently. Controllers on each partition access the advertisements for the controllers on other partitions and assume that they have failed. When they need to reconnect, the WheelFS nodes in both partitions resynchronize and the controllers get notified of all events occured in other partitions.

Process: When a controller starts to process, HyperFlow application starts the WheelFS client and storage services, subscribes to the network's data and control channels, and starts to advertise periodically itself in the control channel. The advertisement message consists of information about the controller with the identifiers of the switches it controls. Then the application captures all the NOX events and replays all the published events, because source controllers filters out them and publish the events if they need to reconstruct the application state on other controllers. HyperFlow application listens for the controller advertisements in the control channel and if a controller does not advertise itself for three advertisement intervals, it is assumed to have failed and fires a *switch leave* event for all switches that was connected to the failed controller and connects them to the another controller. To ensure consistency, and correct operations in all cases, controlapplications must send requests to the *authoritative* controller.

Since WheelFS is not mature during preparation of this paper, they can take better results if the triggered event number is about 1000 per second. The limiting factor is the number of reads since multiple controllers can publish (write) concurrently. WheelFS modification or changing messaging paradigm may be able to improve HyperFlow's performance.

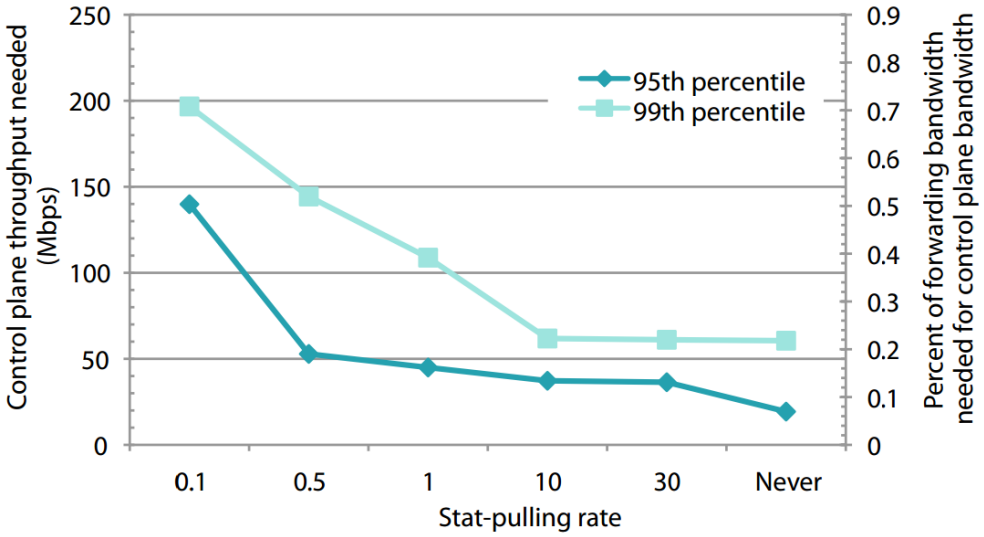

Figure 3.4. The control-plane bandwidth needed to pull statistics at various rates so that flow setup latency is less than 2ms in the 95th and 99th percentiles [3].

For performance evalation, they used ten servers equipped with a gigabit NIC and running WheelFS client and storage node. Each NOX can handle 30k flow installs per second [19]. Events published through HyperFlow only affect controller state and should not fire any interaction with controllers.

3.4. DevoFlow

In [3], Curtis et al. analyze overheads of fine-grained control and visibility of flows, and show that OpenFlows current design cannot meet the needs of high performance networks. They design and evaluate DevoFlow, a modification of the OpenFlow model which breaks the coupling between control and global visibility, in a way that maintains a useful amount of visibility without imposing unnecessary costs. They evaluate DevoFlow through simulations, and find that it can load-balance data center traffic as well as fine-grained solutions, without as much overhead: DevoFlow uses drastically fewer control messages and fewer flow table entries at an average switch. In Figure 3.4, performance regarding the control plane bandwidth necessary for constrained flow setup latency is shown. As expected, the overhead increases with increasing pull frequency.

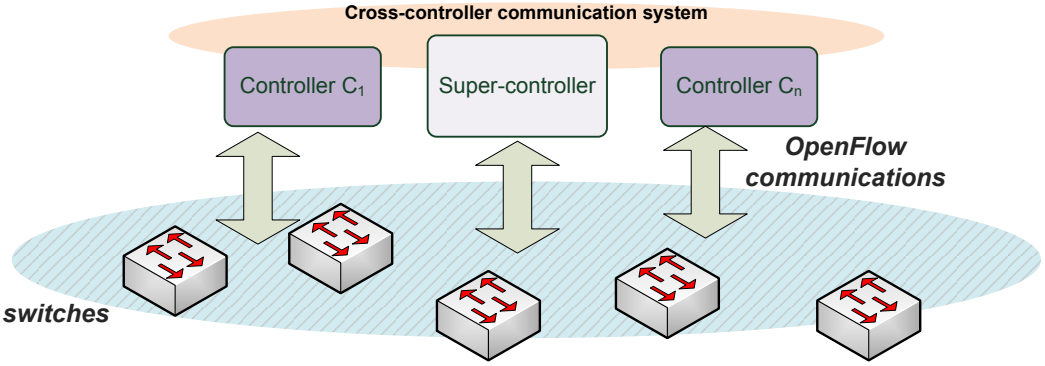

Figure 3.5. BalanceFlow architecture

3.5. BalanceFlow

BalanceFlow is a controller load-balancing architecture by using *Controller X* extension for OF switches and cross-controller communication [20]. It is a hierarchical system as shown in Figure 3.5. The super-controller processes the flow information collected from controllers and detects load imbalances, then acts upon it.

ControllerX: There exist multiple ways of spreading switches' flow requests over multiple controller. The simplest way of this is using hashing. A switch can send each message to one of the controllers, which is determined by the hash of the message but the load of different controllers is totally determined by the hash distribution in the switches with this method. This may lead to a complex switch design and two or more large volumes of flow-requests may collide and end up on the same controller, creating potential controller overload. Alternatively, In BalanceFlow architecture, the switches take initiative for distributing their flow-requests and how to distribute them should be entirely determined at the controller level. With Controller X extension, flow-requests allocation can be dictated by controllers as fine-grained or aggregated flow entries and different flow-requests of each switch thus can be allocated to different controllers. When a new flow arrives at a switch, the first packet of that flow matches one of these entries, and is sent to corresponding controller according to the X in the action part for further process. Upon controller failure, active controllers could simply update the flow entries in the related switches to reallocate the flow- requests.

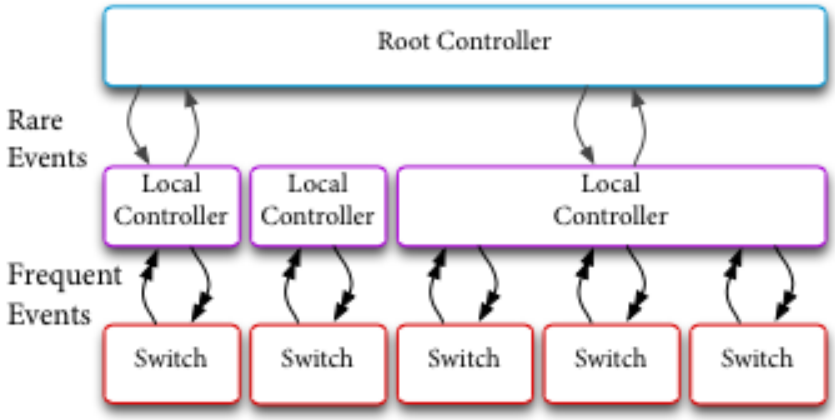

Figure 3.6. Kandoo's two levels of controllers.

BalanceFlow architecture is tested on an event-based simulation and Abilene topology [21] consists of 10 nodes, 13 links is used. Ten hosts are attached to each switch and 3 controllers are deployed in the network. Controllers are placed according to [13] and each switch forwards its flow request to nearest controller to access quick response.

3.6. Kandoo

Kandoo is a framework providing scalability with taking decisions on control plane of OF networks [22]. It consists of two controller layers as distinct from an ordinary OF network as shown in Figure 3.6. *Bottom Layer* is a set of controllers with no intercommunication and they are responsible from only one or a group of switches. They have no idea about the network-wide information and execute only local control applications as close as possible to forwarding elements. *Top Layer* is a logically centralized controller that maintains network-wide state. Most of the frequent events are handled on former and this reduces the load on root controller influentially.

Kandoo is designed to allow network architects to use a deployment model of control plane according to the characteristics of control applications. If there exist software switches in a network, local controllers can be deployed on the same end-host.

If the software of a physical switch is modified, Kandoo can be directly deployed on the switch and finally it can be deployed closest to the switches separately. In addition, with adding more local controllers, it is possible to diminish root controller's load. Since the only required information is a flag representing whether a controller is local (on the bottom layer) or root (on the top layer), its control applications are independent from how they are deployed in the network. In other words, all applications being able to run on a centralized OF controller can run on Kandoo too.

As illustrated with an example in Figure 3.7, Kandoo uses two main control applications: App_{detect} to detect heavy flows and notify $App_{reroute}$ to install or update flow-entries on the switches. An application flagged as non-local can run only on the root controller so local controllers cannot execute $App_{reroute}$ application. On the other hand, App_{detect} is a local application and all controllers can run it. A heavy event being fired by App_{detect} application is relayed to root controller and rif the root controller Local controllers can request data from an application running on root controller by emitting an event, and root controller subscribes it and applications on root controllers send data by replying to that event. If the root controller does not subscribe the event, the local controller will not relay it. This case is the major issue of Kandoo framework since local controllers cannot propagate related event.

App_{detect} asks only top-of-rack(ToR) switches to detect heavy events. Kandoo separates ToR and core switches via a simple link discovery technique: App_{detect} send a query per flow per second and assign related flow as heavy if it has more than 1MB data. $App_{reroute}$ installs new flow entries in all switches for the assigned heavy flow. Besides a simple learning switch application runs on all controllers to setup paths. MAC addresses are linked to ports by this application and it installs respective flow entries on the switch.

For performance evaluation, how the number of heavy flows and network size affect the scale of the control plane is compared to a normal OF network in terms of the number of requests processed by each controller and the bandwidth consumption. Local controllers consume less bandwidth because they handle most of the events locally.

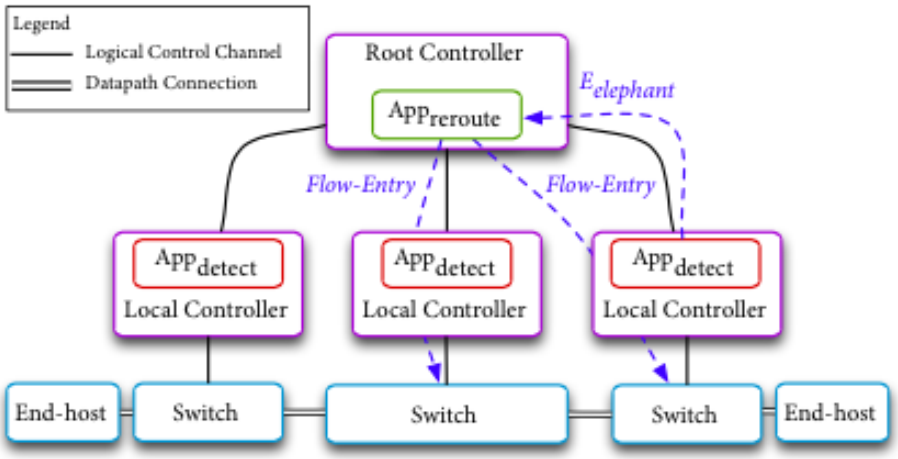

Figure 3.7. Toy example of Kandoo's design.

Moreover, local controller addition when needed is easy in Kandoo, that's why making the root controller more efficient is the only possible bottleneck in terms of scalability.

They used a tree topology of depth 2 and fan-out 6 which means 1 core switch, 6 ToR switches and 36 hosts to compare the effect of proportion of heavy flows in the network. Each host sends a hundred UDP flows to any other host in the network. Kandoo strains with this experiment since most flows are not local. Data center traffic has locality according to [23] so Kandoo would perform better in practise.

In other experiments shows that the control channel usage and the load on the root controller in Kandoo lower than normal OF even if all flows are heavy because it does not require to query ToR switches.

To see the affect of network size on controllers' load, the heavy event is fixed at 20% over all flows and test with networks of different fanouts. Since when the networks become larger, local controllers take more responsibility in Kandoo, it performs better compared to normal OF network because the load of normal OF controllers increments linearly with the size of network.

4. COOPERATIVE LOAD BALANCING SCHEME FOR HIERARCHICAL SDN CONTROLLERS (COLBAS)

The SDN environment that we are investigating consists of as shown in Figure 4.1. We have hosts connected to switches located at nodes on the modified Abilene topology shown in Figure 4.2. The switches are connected to controllers which serve as SDN controllers for these nodes. There is a super-controller which is connected to the lower-tier controllers. This hierarchical structure works in a cooperative manner which entails information exchange for load metrics and partitioning as shown in the figure. The hosts connected to switches generate flow requests according to a Poisson process where packet interarrival times are exponentially distributed. The packet sizes are assumed to be identical.

The system parameters are shown in Table 4.1.

4.1. Algorithm

COLBAS is a controller load-balancing scheme for hierarchical networks. It relies on controller cooperation via cross-controller communication. It assigns one of the controllers *super controller* which can flexibly manage the flow requests handled by each controller. As it is illustrated in Figure 4.1, all controllers work with their own load information and publish this information periodically through a cross-controller communication system. When traffic conditions (e.g. load surge) change, super controller reassigns different flow setups to proper controllers and installs allocation rules on switches for load balancing.

The flow between switch pairs as flow-request information. For this, each controller stores an $S \times S$ matrix, where S is the number of switches in the network, i.e. $S =\| \mathbb{S} \|$. The element in the ith row, jth column, s_{ij} has meaning of the average number of flow requests from switch i to switch j and is denoted as it is calculated

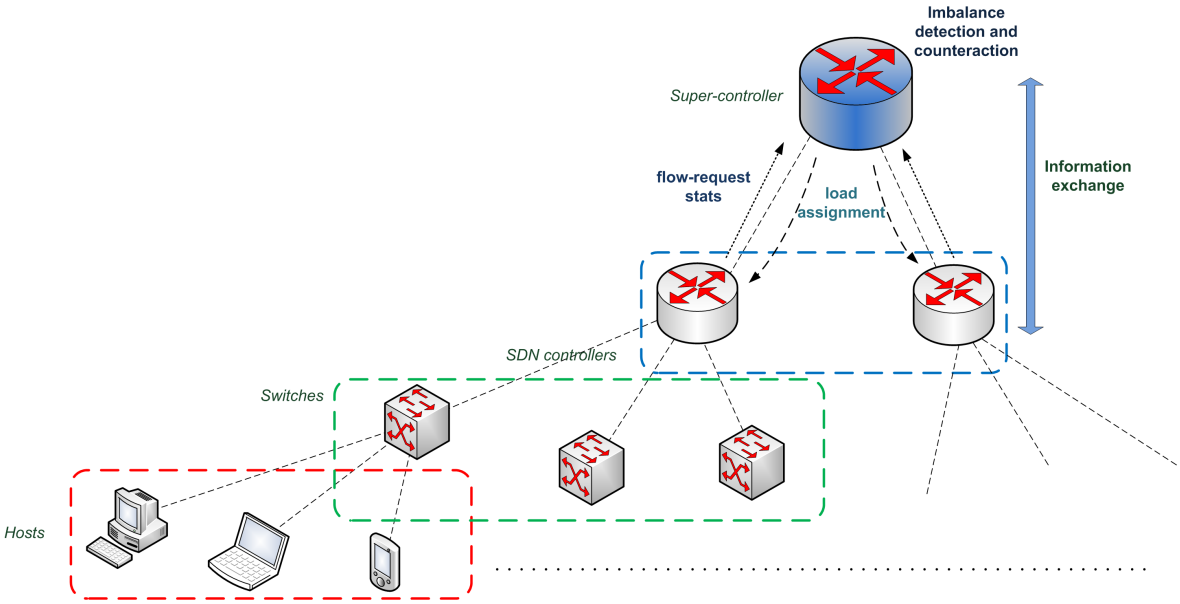

Figure 4.1. System view.

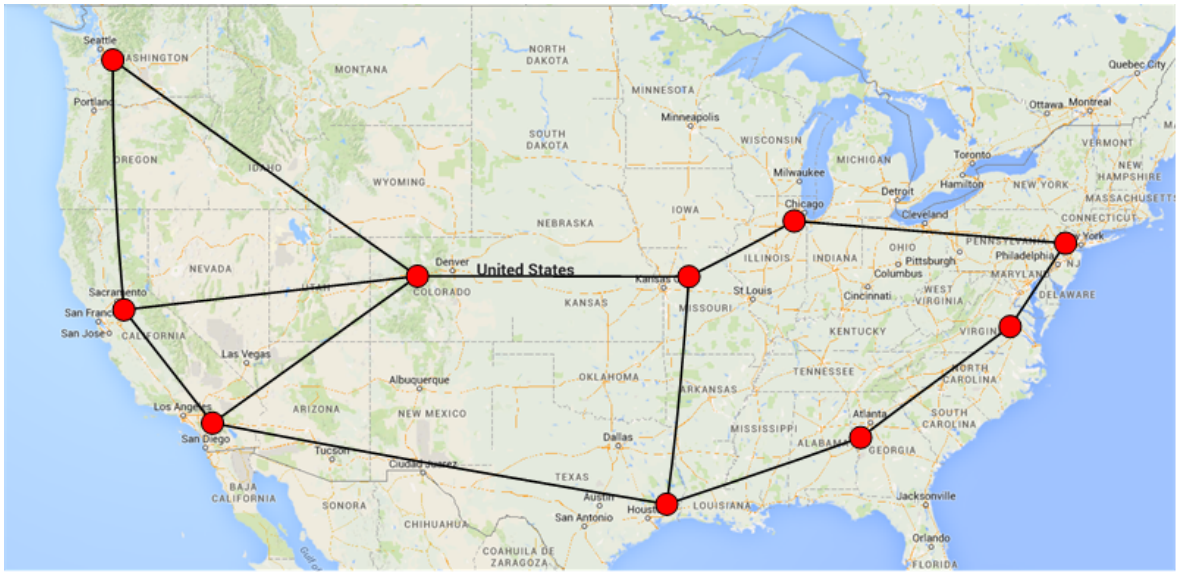

Figure 4.2. Modified Abilene topology used our work.

Table 4.1. System parameters

Parameter	Definition
N_n	number of nodes
N_h	number of hosts
$\mathbb{S}$	set of switches
$\mathbb{C}$	set of controllers
T_c	Controller period
T_s	Supercontroller period
T_o	Timeout period for flow entry in rule table
λ	host traffic generation rate (exponential)
w	weight of past B_ij at t=-1
z	weight of B_ij at t=0
α	coefficient for cost function
L	imbalance detection threshold
R_{total}	total number of flow-requests in the network
F_c	current number of flow-requests in the controller C to be processed

with the following formula

$$\widetilde{B}_{ij} = (1 - w - z)\widetilde{B}_{ij}^{-2} + w\widetilde{B}_{ij}^{-1} + zR_{ij} \tag{4.1}$$

where w and z are weighted coefficients to avoid sharp up and downs by taking into consideration past values (smoothing), and R_{ij} is the number of flow requests from switch i to switch j in a certain period. The super controller collects the calculated matrices from all controllers and tries to detect imbalance if average number of flow requests handled by any controller exceeds a relative threshold of the total flow requests rate, namely L, in the network. L parameter must be tunable according to the performance of the super controller, the number of controllers and the network environment. This operation is shown in Figure 4.3.

For reassignment and allocation, we use a greedy algorithm listed in Algorithm 4.4.

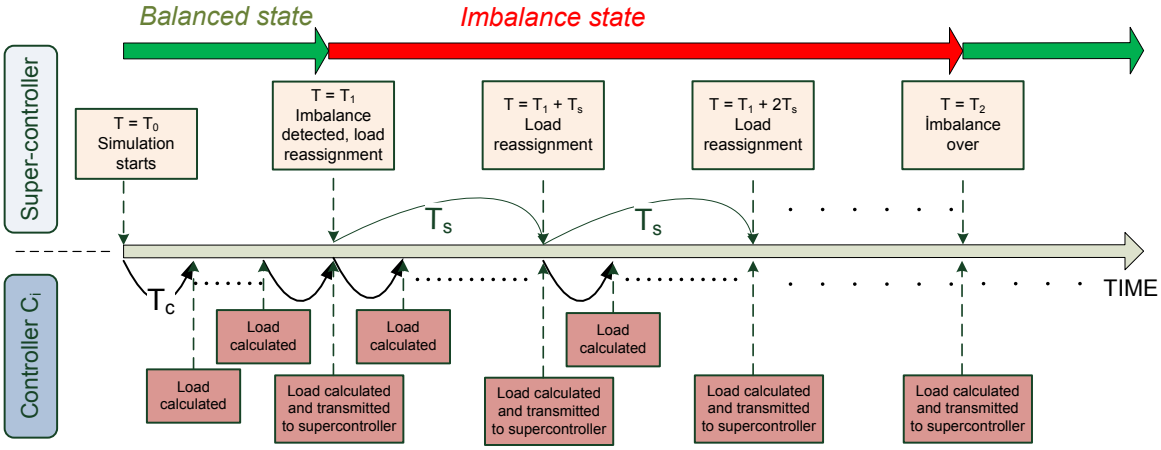

Figure 4.3. The operation of super-controller and controllers for COLBAS operation. The node-based calculations and decisions/actions are depicted for different epochs.

the objective is to keep the total number of flow requests handled by any controller below the given threshold. After the switch pairs are sorted in descending order of their $\widetilde{B}_{ij}$, the cost function is calculated:

$$C_s = \overline{Q_s} + P_{ij}^s + \alpha v_M D_{is} \tag{4.2}$$

where $\overline{Q_s}$ is the average number of flow requests already handled by controller s at last two T_s periods, P_{ij}^s is the flow requests that is about to be handled by that controller, and D_{is} is the delay from switch i to controller s.

$$\overline{Q_s} = \frac{Q_s^{-1} + Q_s^0}{2} \tag{4.3}$$

$$P_{ij}^s = \widetilde{B}_{ij} \frac{T_s}{T_c} \tag{4.4}$$

v_M is a variable that makes controllers' load and propagation delays comparable, and $v_M = R_{total}/D_{avg}$ where R_{total} is the total number of flow requests in the network, and D_{avg} is the minimum average node-to-controller and controller processing latency.

$$D_{avg} = D_{min}^t + D_{min}^p \tag{4.5}$$

Initialize:
Sort $\mathbb{S}$ in descending order of average number of flow-requests
Assign:
for **do**$P_{ij} \in$S
for c **do** $= 1 \rightarrow$ M ▷ Assign to a controller C_m
$b_c =$ cost of adding P_{ij} to controller C
end for
Allocate P_{ij} to the controller m = argmin$_b$ b_m
end for

Figure 4.4. Flow-request assignment algorithm in the super-controller.

α is the variable to adjust the weights between controllers' load and propagation latencies in the cost function. Too small α represents that the benefit of reducing latencies is ignored, we can adjust it too large, if we are not interested in load-balancing. The delay D_{is} consists of propagation or transmission delay D_s^t and processing delay D_s^p:

$$D_{is} = D_{is}^t + D_s^p \tag{4.6}$$

The processing delay for a flow packet in controller is calculated according to

$$D_s^p = \left\lceil \frac{F_c}{\mu_s} \right\rceil \gamma \tag{4.7}$$

where μ_s is the packet processing power of controllers and γ is a delay multiplier for representing the delay for packet batches.

The following process after the decision step is to actuate the new flow rules in the switches. They are represented as low-level rules and sent to the associated switches as in [20]. An example set of rules with normal and allocation rules in a switch are shown in 4.5. Subsequently, the switches operate under these new flow rules leading to a better distribution of load.

Type	Priority	Matching Field	Action	Timeout	Note
Normal Rules	High	Source: 1100 Destination: 0101	Forward: Port2	6 msec	Installed by Controller c1
	High	Source: 1010 Destination: 1101	Forward: Port10	6 msec	Installed by Controller c3
	…	…	…	…	…
Allocation Rules	Low	Source: 1001 Destination:0111	Forward: Controller c1	∞	Installed by SuperController
	Low	Source: 1101 Destination: 1111	Forward: Controller c3	∞	Installed by SuperController
	Low	…	…	…	…

Figure 4.5. Rules in a switch extended with allocation rules for load balancing scheme.

5. PERFORMANCE EVALUATION

We have developed a discrete-time event simulator based on SimPy [6]. The simPy infrastructure is summarized in Figure 5.1. SimPy is a process-based discrete-event simulation framework based on Python programming language. It can be used for asynchronous networking or to implement multi-agent systems (with both, simulated and real communication). The behavior of active components (like switches, controllers or hosts) is modeled with processes. All processes live in an environment. They interact with the environment and with each other via events. We have extended this environment with SDN related entities such as controllers and algorithmic components such as our load distribution mechanism.

We can call them process function or process method, depending on whether it is a normal function or method of a class. During their lifetime, they create events and yield them in order to wait for them to be triggered.

This section explains the experiments and measurements done using the system and devices explained in Chapter 4, and discusses the results of these experiments. The simulator related SimPy information is shown in Figure 5.1. The values for system and simulation parameters are shown in Table 5.1. The simulation duration for the system is 20 seconds and the supercontroller is activated after 4 seconds.

5.1. Experimental Results

In this section we present experimental results for COLBAS. In our experiments we first set simulation parameters to the values shown in Table 2.1 and the we re-assigned different values to see the effect of delay sensivity parameter α, the control period of super controller T_s, and the average matrix update period of each controller T_c.

All simulations are executed for a period which corresponds to 20 seconds in

major SimPy classes

- **Process**: simulates an entity which evolves in time, e.g. one customer who needs to be served by an ATM machine; we will refer to it as a thread, even though it is not a formal Python thread
- **Resource**: simulates something to be queued for, e.g. the machine

major SimPy operations/function calls

- **activate()**: used to mark a thread as runnable when it is first created
- **simulate()**: starts the simulation
- **yield hold**: used to indicate the passage of a certain amount of time within a thread; **yield** is a Python operator whose first operand is a function to be called, in this case a code for a function that performs the hold operation in the SimPy library
- **yield request**: used to cause a thread to join a queue for a given resource (and start using it immediately if no other jobs are waiting for the resource)
- **yield release**: used to indicate that the thread is done using the given resource, thus enabling the next thread in the queue, if any, to use the resource
- **yield passivate**: used to have a thread wait until "awakened" by some other thread
- **reactivate()**: does the "awakening" of a previously-passivated thread
- **cancel()**: cancels all the events associated with a previously-passivated thread

Figure 5.1. SimPy code-level components.

Table 5.1. Parameter values

Parameter	Value
N_n	10
N_h	10
$\| \mathbb{C} \|$	3
$\| \mathbb{S} \|$	10
T_c	10 msec
T_s	1000 msec
T_o	6 msec
λ	10
w	0.3
z	0.2
α	0.1
L	0.5

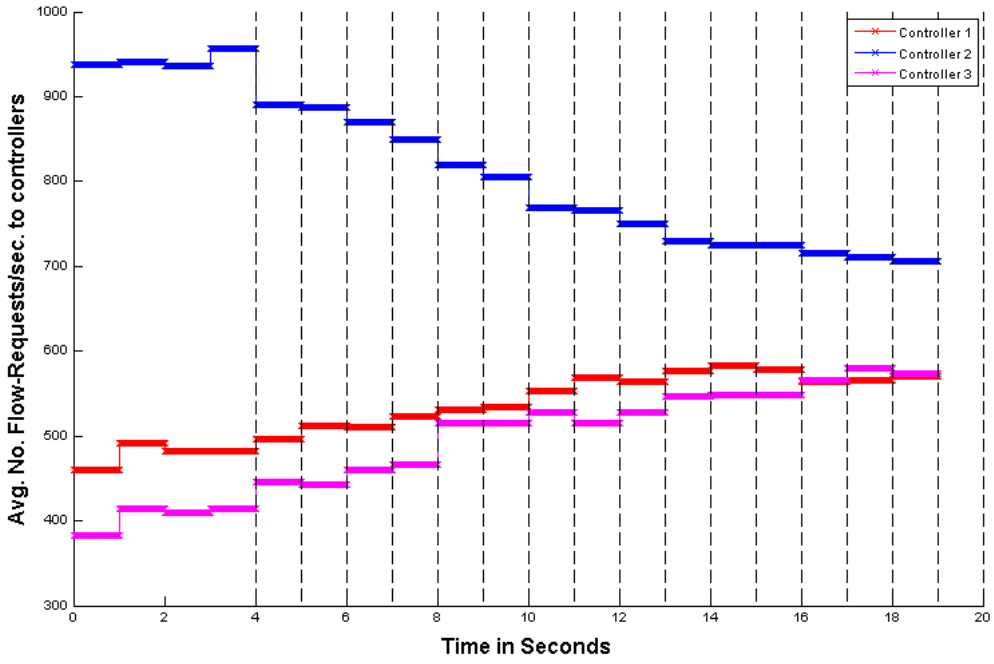

Figure 5.2. The avg. number of flow-requests to controllers with parameters given in Table 5.1.

real-life. To see how COLBAS detects load imbalance and takes its role to decrease this overload, we started supercontroller after 4^{th}sec and it checks controllers' load each second. As it is shown in Figure 5.2, if one of the controller's load exceeds the threshold it starts to share its burden until the excess decreases to second threshold (the half of the main threshold).

5.1.1. Impact of Delay Sensitivity (α)

We change the coefficient for delay-sensitive traffic. α adjusts the weights between controllers' load and propagation delays in the cost function. If we want to decrease the effect of latencies and just concentrate to balance the controllers' load, we must select too small α. If it is set too large, in the case of imbalance, any node will not change its current controller with a further one and load balancing becomes an insignificant concept. In Figure 5.3 and Figure 5.4, we set it to the half of its previous value and the impact of delay ignorance can be seen. Packets load more on different controller in different periods and don't take into consideration delay.

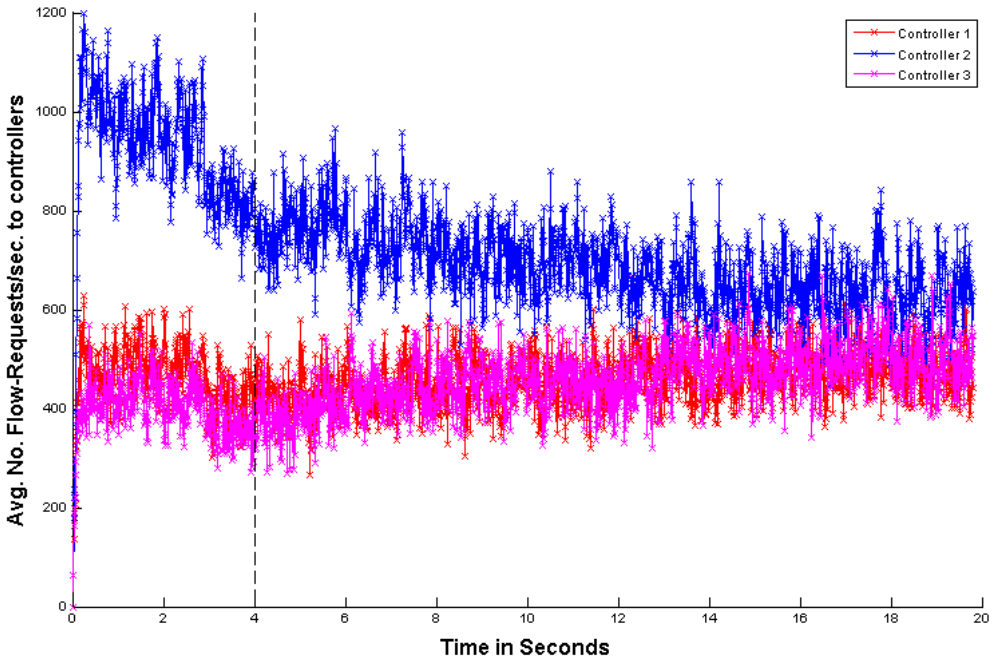

Figure 5.3. Too small α causes delay ignorance.

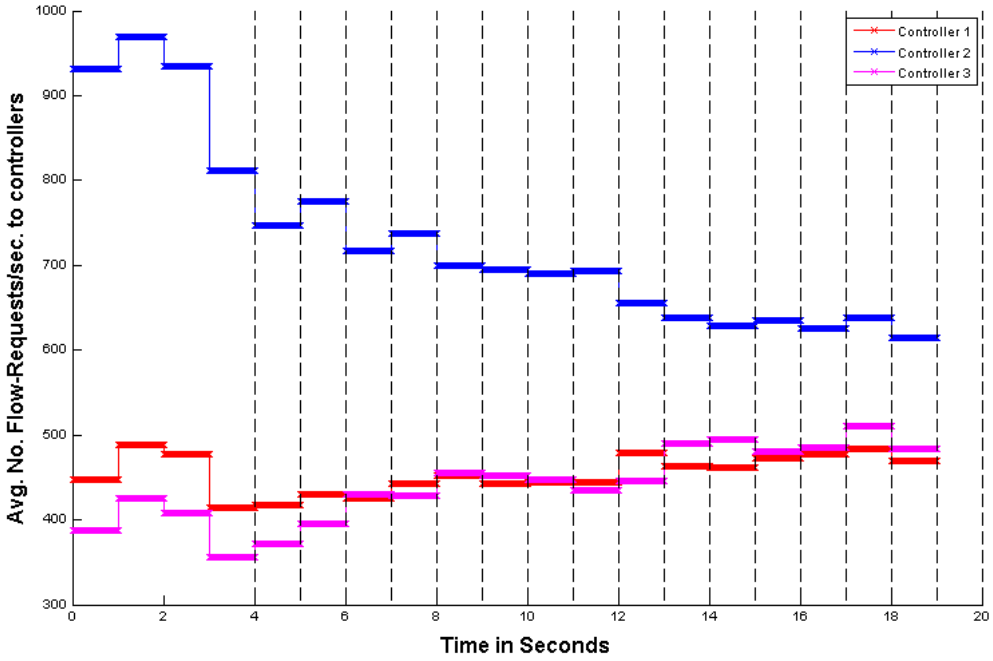

Figure 5.4. Too small α causes delay ignorance.

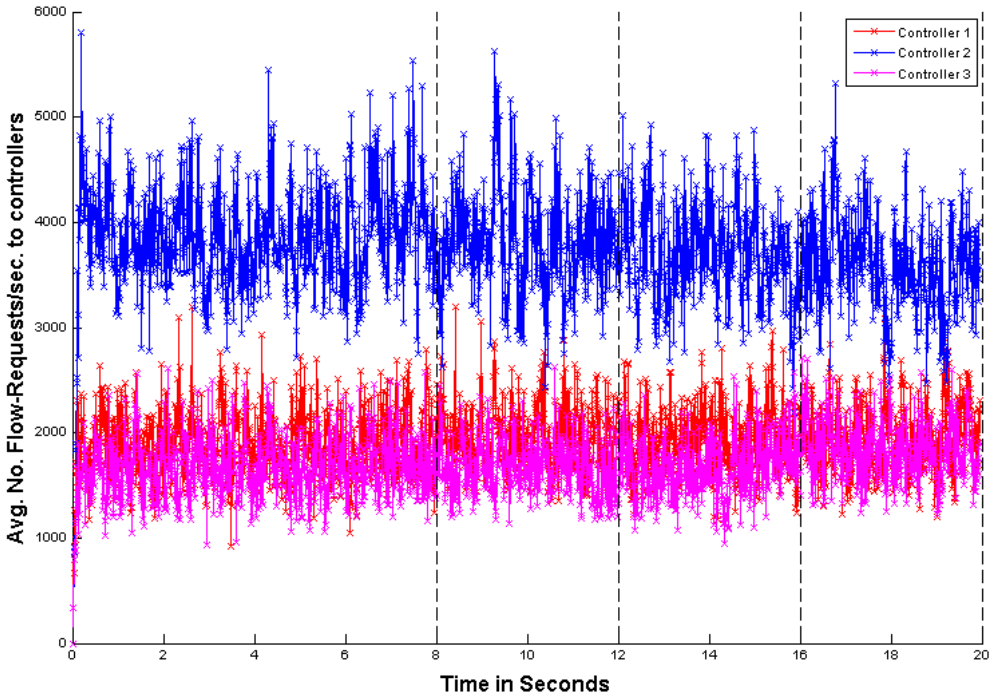

Figure 5.5. Effect of Super Controller's imbalance detection period on network load-balancing.

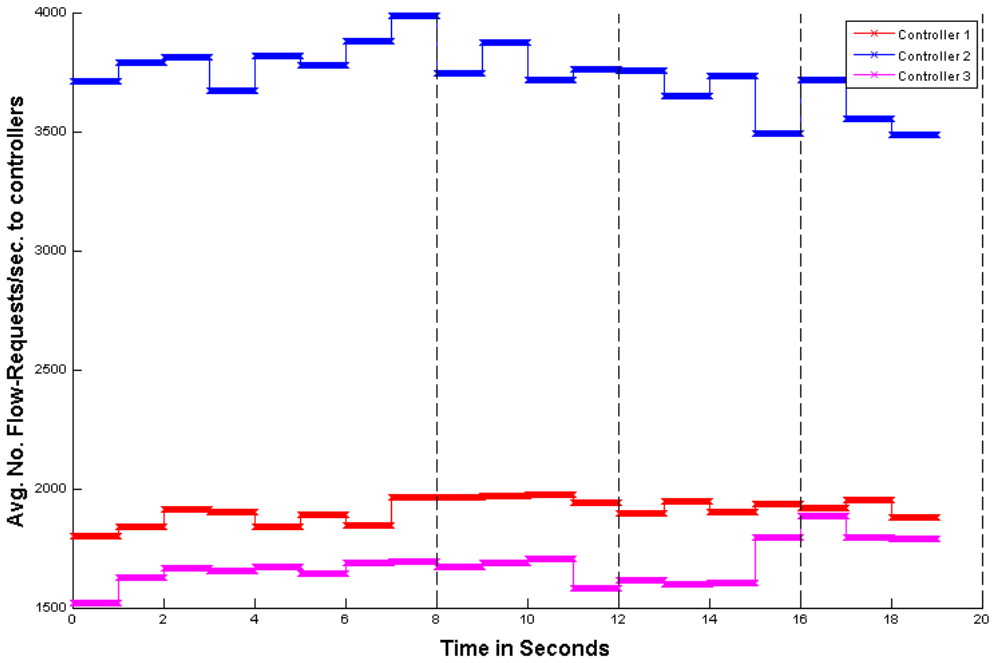

Figure 5.6. Effect of Super Controller's imbalance detection period on network load-balancing.

5.1.2. Impact of Decision Making Period of Super Controller (T_s)

We change the T_s period for decision on super controller. We expect the load balancing performance to degrade for dynamic traffic when T_s increases. Figure 5.5 and Figure 5.6 affirms our expectations, since we started to execute super controller after 8 seconds of simulation start and we increment its control period to 4 seconds. The main impact of this change can be seen when we look the $y - axis$ of the figures, in the previous figures, controller2's avg. number of flow request per second was about 1000s but in this figure the load reaches 4000 flow requests per second.

5.1.3. Impact of Average Load Calculation Period of Standard Controllers T_c

To see the effect of controllers' period on their loads, we change controllers' information exchange T_c period. We set 10 times bigger time interval for it. According to Figure 5.7 and Figure 5.8, the period before the super controller starts to execute does not provide a generous idea about the result, it acts as the previous test. Since our algorithm keeps history, we do not expect sharp zig-zags but this result is unexpected one. After the 8^{th} sec imbalance increases but controller3's load again tends to decrease at the end of the simulation times. This case requires further investigations and longer tests with different parameters.

5.1.4. Impact of Imbalance Detection Threshold L

Another parameter that we must test its effects on controllers' load is threshold. Since it plays a significant role in the imbalance detection, we must be attentive to determine its optimal value for our network structure. Especially, in the case that all controllers have different capacities, we might need to assign different loads on different controllers and the threshold value is one of the key parameters to adjust this kind of assignments. Figure 5.9 and Figure 5.10 shows us the smaller threshold the closer load rate between all controllers but if we want to assign more burden on a specific controller, we should assign this parameter in this manner.

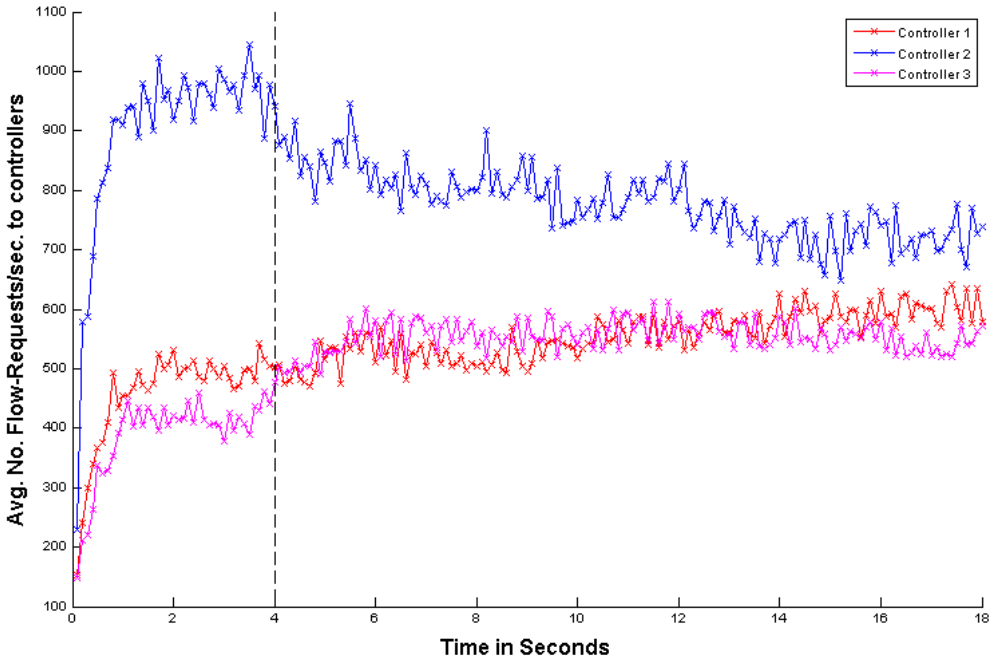

Figure 5.7. Effects of controllers' recalculation period on network load-balancing (T_c=10msec).

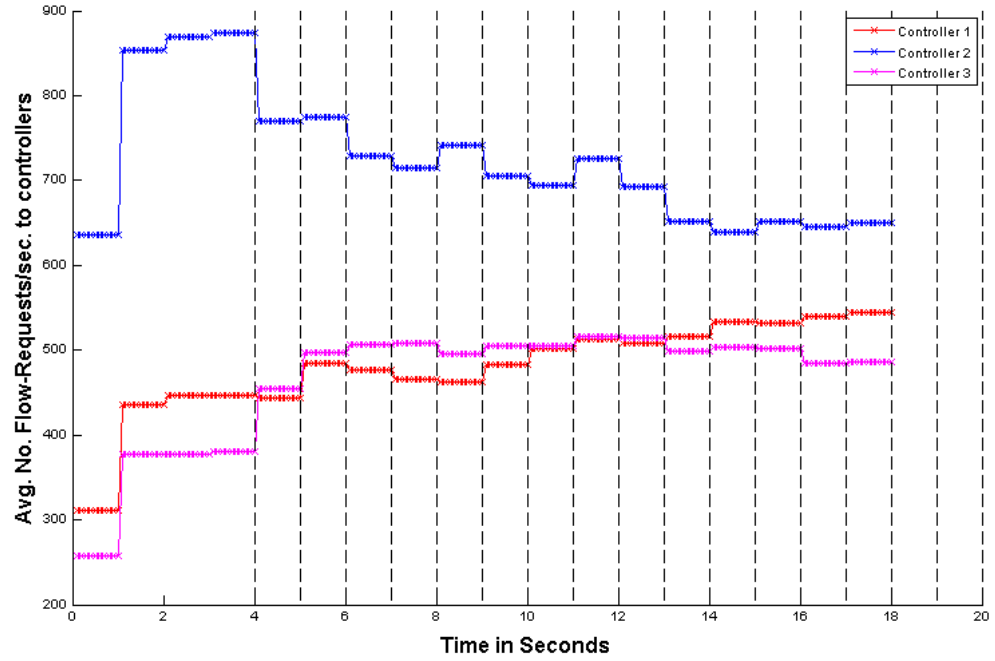

Figure 5.8. Effects of controllers' recalculation period on network load-balancing (T_c=100msec).

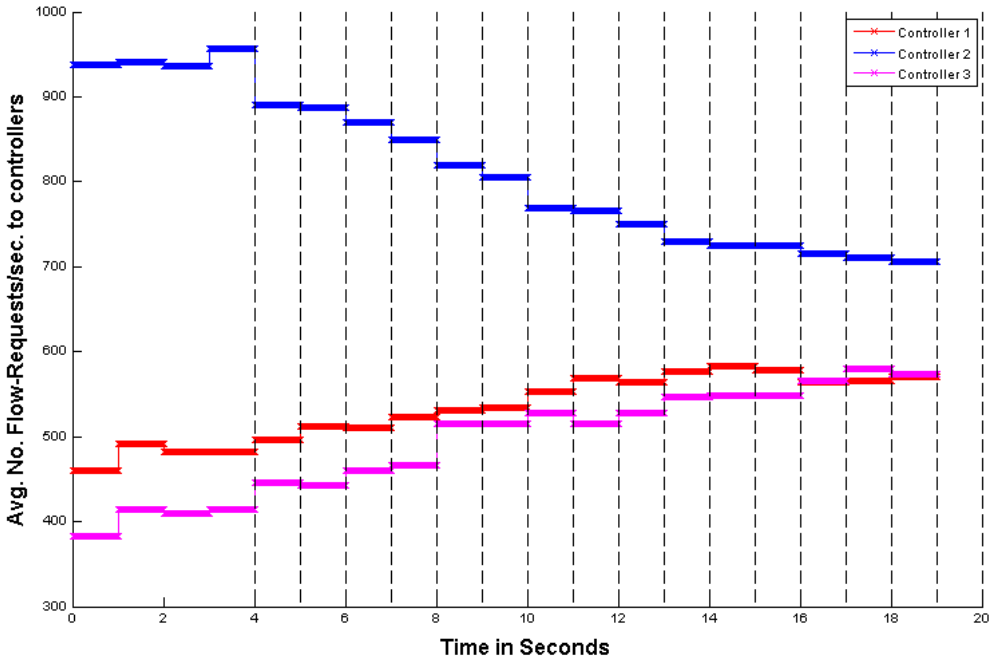

Figure 5.9. Effects of threshold value on controllers' loads (L=0.5).

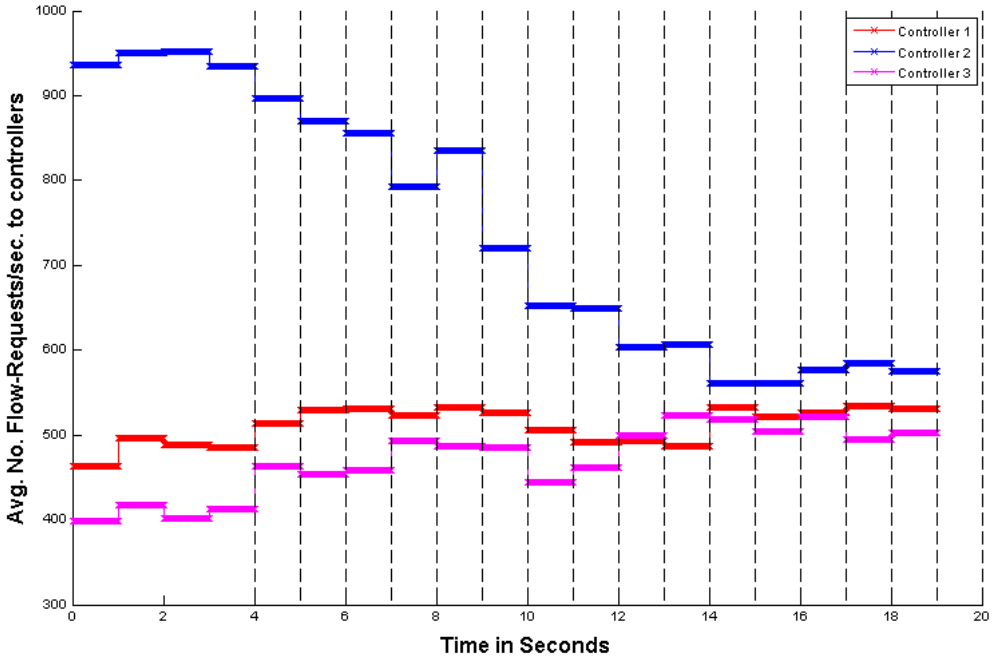

Figure 5.10. Effects of threshold value on controllers' loads (L=0.3).

5.1.5. System Overhead and Potential Drawbacks of COLBAS

In this section, we convey potential drawbacks and system overhead of COLBAS for a balanced discussion. The main drawbacks valid for centralized architectures are also applicable to COLBAS. These are resilience and fault-tolerance issues related to single-point-of-failure situations. However, the fall-back scenario of operating without a super-controller is always possible corresponding to the baseline case. Therefore, even if the super-controller fails, the system is operational. However, please note that these phenomena are also valid SDN paradigm and not limited to COLBAS. Therefore, those are not introduced into SDN system for the sole purpose of COLBAS operation but extended for load balancing.

Peculiar to COLBAS is the computational overhead due to load monitoring and calculation. For instance, each controller needs to maintain a matrix with elements updated every T_c period. Similarly, the super-controller performs computations every T_s period. Moreover, there is the signaling overhead which entails the communication of load matrices to the super-controller and the assignment of load to the controllers.

6. CONCLUSIONS

The controllers which are centralized network control entities are crucial for meeting the performance and efficiency requirements of software-defined networks. The placement of these network entities, switch assignment and their load-aware operation is a challenging research question. Although various load-balancing schemes are proposed in the literature, this research topic is yet to be explored adequately. In this thesis, we discuss load-balancing issues in SDN and propose COLBAS scheme for load balancing of control traffic. Moreover, we evaluate the performance of this scheme and investigate important trade-offs.

As future work, we plan to consider heterogeneous controllers which is a more realistic scenario. Different capabilities of controllers may include:

- Capacity for packet-handling rules (e.g., maximum number of entries)
- Range of matches and actions (e.g., which actions are supported)
- Multi-stage pipeline of packet processing

More advanced cost and assignment mechanism shall be required for this situation. For the cost calculation, additional dynamic factors can be considered such as

- Current size for packet-handling rules (we can adopt a more detailed delay/cost model for increasing ruleset size)
- Current set of matches and actions defined in the system
- Queue sizes in network entities

Moreover, we may consider diversity of flow requests: some may be easy to process while some are difficult. We may improve flow size calculations and integrate them into the cost formulation. On the communication side, a controller has intricacies considering delay and overhead attributes. For instance,

- Controller is much slower than the switch since it performs complicated lookup and matching computations.
- Processing packets leads to delay and overhead on the controller side.

In our current work, we assume a perfect channel between switch-controller ans supercontroller-controller. However, in actual networks, there may be transmission impairments and buffer overflows which cause communication challenges.

REFERENCES

1. Yazici, V., M. O. Sunay, and A. O. Ercan, "Controlling a Software-Defined Network via Distributed Controllers", *Computing Research Repository*, 2014.

2. Yu, M., J. Rexford, M. J. Freedman, and J. Wang, "Scalable Flow-based Networking with DIFANE", *Proceedings of the Association for Computing Machinery SIGCOMM 2010 Conference*, SIGCOMM '10, pp. 351–362, 2010.

3. Curtis, A. R., J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks", *Proceedings of the Association for Computing Machinery SIGCOMM 2011 Conference*, SIGCOMM '11, pp. 254–265, ACM, New York, NY, USA, 2011.

4. Nunes, B., M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks", *Communications Surveys Tutorials, IEEE*, Vol. 16, No. 3, pp. 1617–1634, Third 2014.

5. Open Networking Foundation (ONF), "OpenFlow Specifications", 2014, `https://www.opennetworking.org/sdn-resources/openflow/, January 2014`.

6. SimPy, "SimPy discrete-event simulation framework", 2015, `https://simpy.readthedocs.org/en/latest/, December 2014`.

7. Fernandez, M., "Comparing OpenFlow Controller Paradigms Scalability: Reactive and Proactive", *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pp. 1009–1016, 2013.

8. Limoncelli, T. A., "OpenFlow: A Radical New Idea in Networking", *Queue*, Vol. 10, No. 6, pp. 40–46, June 2012.

9. NEC, "Ten Things to Look for in an SDN Controller", 2014, `https://www.necam.com/Docs/?id=23865bd4-f10a-49f7-b6be-a17c61ad6fff`, `September, 2014`.

10. Bari, M., A. Roy, S. Chowdhury, Q. Zhang, M. Zhani, R. Ahmed, and R. Boutaba, "Dynamic Controller Provisioning in Software Defined Networks", *Network and Service Management (CNSM), 2013 9th International Conference on*, pp. 18–25, Oct 2013.

11. Tootoonchian, A., S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On Controller Performance in Software-defined Networks", *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pp. 10–10, USENIX Association, Berkeley, CA, USA, 2012.

12. Voellmy, A. and J. Wang, "Scalable Software Defined Network Controllers", *SIGCOMM Comput. Commun. Rev.*, Vol. 42, No. 4, pp. 289–290, August 2012.

13. Heller, B., R. Sherwood, and N. McKeown, "The Controller Placement Problem", *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pp. 7–12, ACM, New York, NY, USA, 2012.

14. Zhang, Y., N. Beheshti, and M. Tatipamula, "On Resilience of Split-Architecture Networks", *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pp. 1–6, Dec 2011.

15. Hock, D., M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, "Pareto-optimal resilient controller placement in SDN-based core networks", *Teletraffic Congress (ITC), 2013 25th International*, pp. 1–9, Sept 2013.

16. Benson, T., A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild", *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pp. 267–280, ACM, New York, NY, USA, 2010.

17. Ban, B., "Design and Implementation of a Reliable Group Communication Toolkit for Java", 1998, `http://www.jgroups.org/papers/Coots.ps.gz, November, 2014`.

18. Tootoonchian, A. and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow", *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pp. 3–3, USENIX Association, Berkeley, CA, USA, 2010.

19. Tavakoli, A., M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the Datacenter", *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

20. Hu, Y., W. Wang, X. Gong, X. Que, and S. Cheng, "BalanceFlow: Controller load balancing for OpenFlow networks", *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, Vol. 02, pp. 780–785, Oct 2012.

21. The Internet Topology Zoo, "Abilene Network Topology", 2015, `http://www.topology-zoo.org/dataset.html, January, 2014`.

22. Hassas Yeganeh, S. and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications", *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pp. 19–24, ACM, New York, NY, USA, 2012.

23. Alizadeh, M., A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)", *Proceedings of the Association for Computing Machinery SIGCOMM 2010 Conference*, SIGCOMM '10, pp. 63–74, ACM, New York, NY, USA, 2010.