UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DESIGN, IMPLEMENTATION, AND EVALUATION OF JOIN AND SPLIT STRATEGY FOR TRANSMISSION CONTROL PROTOCOL RUNNING ON SOFTWARE DEFINED NETWORKS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

WEI GUO

Norman, Oklahoma

2017

DESIGN, IMPLEMENTATION, AND EVALUATION OF JOIN AND SPLIT STRATEGY FOR TRANSMISSION CONTROL PROTOCOL RUNNING ON SOFTWARE DEFINED NETWORKS

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Sridhar Radhakrishnan, Chair

Dr. S. Lakshmivarahan

Dr. Qi Cheng

Dr. Mohammed Atiquzzaman

Dr. Charles D. Nicholson

To my family, Xuan Qi, Jade X.Guo

Acknowledgements

This dissertation could not have been finished without the help and support from a number of individuals. I would like to express my deepest appreciation to all of them.

First of all, I am deeply grateful to my advisor and committee chair Dr. Sridhar Radhakrishnan. He guided me through this wonderful journey and provided valuable feedback throughout my doctoral work.

Furthermore, I would like to thank Dr. Mahendran Veeramani for his understanding, patience and valuable feedback. My thanks also go out to Dr. Yuh-Rong Chen who spent countless hours helping me develop the Integer Programming models.

I further like to thank the professors with whom I had taken classes, which had laid a solid foundation for my research work.

My special gratitude goes to my friends and colleagues: Yiming Xu, Michael Nelson, and Jincheng Zhuang for stimulating discussions and their readiness to help.

My special thanks are extended to the wonderful staff members in the School of Computer Science at the University of Oklahoma, Jonathan Mullen, Virginie Perez Woods, Lindsay Rice, and James Cassidy for their timely help, encouragement, and advice since my first day in this program.

Last but not least, I would like to thank all my family members for their love, encouragement and support. Especially my parents, Chengfa Guo, Chenguang Huang who raised me and supported me in all my pursuits; and my wife Xuan Qi who is always gave me her best wishes and support; and my daughter Jade X.Guo who always cheers me up.

Table of Contents

List of Tables

List of Figures

Abstract

Software Defined Networks (SDN)-enabled switches of today can be empowered to intelligently forward as well as elastically steer the network traffic. In this work, we focus on developing a SDN-based framework to provide improved delivery performance (of applications) in the network.

This dissertation proposed a new TCP join and split proxy on SDN platform. The proposed framework allowed part of TCP (Transmission Control Protocol) optimization to migrate from the application server to the proxy. Therefore, with a control plane built between SDN controller and proxy, the SDN controller can further improve the TCP delivery performance. The proxy (join-proxy) joins all TCP flows at the beginning of the shared path into one long TCP flow. At the end of the shared path, the proxy (split-proxy) splits the long flow for each joined client with the same TCP session state. With the help of centralized controller of SDN and customized SDN switch, the new design simplifies the TCP session synchronization between proxies. Also, this dissertation developed Linked-ACK ((Acknowledgement) to maintain the end-to-end semantic and limit the buffer size in each proxy by coupling the ACK of three TCP flows separated by the join and split proxy. At the last, this dissertation shows that the proposed proxy can well integrate with wireless network and MPTCP (Multi-Path TCP) proxy [1]

The extensions of the proposed TCP Join and Split platform are applied to Smart Grid network for improving fairness, WiFi network for reducing gaming traffic delay, and Data Center network for addressing Virtual Machine (VM) live migration problem.

First, the proposed TCP Join and Split platform can be applied to Smart Grid

network to provide better fairness on the application layer. The latest research in Smart Grid communications has advocated the aggregation of multiple traffic flows in order to achieve an improved throughput. While aggregation improves the overall throughput, the individual flows still suffer from unfair throughput performance. As a result, the enablers for time sensitive Smart Grid services, such as load-shedding which requires a timely report of data, are mostly affected.

This dissertation proposed a novel SDN-based framework to provide fairness among smart-meters (SMs) through flow aggregation and scheduling. By exploring the SDN's flow-level manageability features, for the first time in this paper, we present an implementation-based architecture to perform effective aggregation-and-scheduling of traffic flows. The proposed framework ensures fairness (among the smart-meters) as well as improve the throughput performance. Our extensive experimental results validate the efficacy of our proposed framework.

Second, the proposed TCP Join and Split platform can be applied to WiFi network to reduce the gaming traffic delay. WiFi users typically expect different performance requirements for various types of applications. For instance, users expect 'better and consistent throughput' for Internet video consumption, and 'minimal delay' for local network gaming applications. The wireless access substrate (at the consumer-end), typically being the bottleneck in these networks, causes different users (in the same WiFi coverage) to experience unfair and fluctuating network performance. To combat such unfair situations, we need approaches to effectively control and steer the applications' traffic in the shared WiFi medium. However, a network that deals with a crowd or private end-users (such as gaming multiplayers or the Internet content distributors), encounters a major challenge in controlling the traffic without involvement or modification at the end-host application devices.

In this dissertation, we propose a SDN-based seamless traffic steering and control strategy in order to provide effective application-specific delivery services, such as

reduced delay (for gaming traffic) and improved throughput (for video consumption). Unlike simulation-based solutions, our approach is production-ready, as we have implemented our framework on a real network testbed environment. With extensive performance study and sufficient mathematical insight, we demonstrate the prowess of our proposed framework.

Last but not the least, the proposed TCP Join and Split platform can be applied to Data Center network to optimize the VM live migration. With the growth of data volumes and a variety of Internet applications, virtualization has become commonplace in modern data centers and an effective solution to provide better management flexibility, lower cost, scalability, better resources utilization, and energy efficiency. One of the powerful features provided by virtualization is Virtual Machine (VM) live migration, which facilitates moving workloads within the infrastructure with negligible downtime and minimal impact on workload. However, the performance of running applications is likely to be negatively affected during a live VM migration. The objective of this paper is to optimize the total performance degradation of concurrent VM live migration in the data center network by exploiting the SDN platform. The problem is modeled using mixed integer linear programming(MILP) for VM live migration with a fixed path and VM live migration with path selection. To provide a practical optimization, the greedy algorithm is proposed. Numerical study results show that a significant decrease occur in performance degradation in MILP model and greedy algorithm when the number of VMs increases. The proposed greedy algorithm cannot yield the optimum solution as the problem become harder, but it provides better solution than MILP model in terms of the time constrain exhibited in case of large problems.

Chapter 1

Introduction

There has been an enormous growth in the number of connected devices to the Internet which in turn has produced high-speed and high-capacity networks and routers that are capable of processing packets at rapid speed. For example, Juniper T-series routers can forward in up wards of 30 billion packets per second. Such extreme speeds are only possible with high-speed and multi-core router architectures. With at advent of such high-speed router architectures is it natural to determine if the routers can provide additional computational help other than providing the packet header lookup and forwarding service. In fact, many router manufacturers have provided additional services to run on the routers such as deep packet inspection to detect worms and viruses based on known signatures.

The route control protocols and configuration of the routing tables are accomplished by routers using its control plane. The data plane is responsible for determining which output port to send the packet based on the destination address. Routers also provide services such as monitoring and configuration services and this constitutes the management plane. Software Defined Network (SDN) consists of routers that run the SDN software which enhances the ability of the data, control, and management planes. With the availability of SDN routers, it becomes easy to control the network traffic that is based on user-defined rules and not the one size fits all approach of the current routers. As packets enter the SDN enabled router, the rules are applied, which may require deep packet inspection and the packet is forwarded to the user defined output port on the router.

1.1 Software Defined Network SDN

SDN is a new network architecture that decouples the control and data planes and allows the network control to be directly programmable [2]. Compared to, the network gains programmability, automation, and network control to build highly scalable, flexible, and adaptable networks.

A high-level view of SDN architecture is shown in Figure 1.1. The goal of SDN architecture is to provide a controlled connection and open interface for enabling the developer to inspect and modify the network traffic. SDN architecture can be divided into data plane, control plane, and application plane. The data plane contains the network elements implemented with SDN datapath which consists of Control-Data-Plane Interface (CDPI) agent to communicate with control plane, and forwarding engine and processing function which process packets following the management of CDPI. Both physical and virtual switches can be network elements. The application plane implements the business logic to the network through the Northbound Interfaces (NBIs) provide by the control plane which allows the control plane to communicate with a higher-level component that is application plane in the SDN architectures. Therefore, the control plane working between data plane and application plane translates the business logic to the low-level interface in the SDN datapath and information up to the application.

Control to Data-Plane Interface (CDPI) is an open source, vendor-neutral, and an interoperable interface between the control plane and data plane. The essential components of this interface are the flow table and external controller. When a first packet belonging to a flow (a flow is a sequence of packets going from a source to a destination) enters a SDN router for the first time, the packet is sent to a controller - an external device that is connected to the same network. The controller then inserts forwarding information including packet matching rules on all the routers in

the path from source to destination. The place when you insert this information is called the Flow-Table. The Flow-Table contains packet matching rules and actions that correspond to them. For example, an action would be discard the packet (as for example when a virus is detected).

One of the most well known CDPI is OpenFlow that provides an open protocol to program the flow table in different switches and routers [3]. The OpenFlow Switch and Controller communicate via a secure channel, which defines three message types, controller-to-switch, asynchronous, and symmetric. The controller initiates Controller-to-switch messages to manage and inspect the state of the switch, such as insert or remove flow tables. The switch initiates asynchronous messages to update the controller of network state of the switch, such as new packets come in, links lost connection. Symmetric messages are initiated by either the switch or the controller and mainly used for maintaining the connection between the switch and controller [4].

A packet received by OpenFlow switch is matched on each of the flow tables on each router shown as Figure 1.2. Each Flow-Table contains multiple flow entries. A flow entry in the OpenFlow Flow-Table consists of six fields: (1) Match Fields: match against packets (2) Priority: provide precedence for each flow entry (3) Counters: calculate matched packets (4) Instructions: modify the action set (5) Timeouts: is the expired time (6) Cookie: is used for the controller to analyze flow entries [4].

In the market, OpenFlow protocol is supported both on commercial and virtual switches. Mininet [5] is one of the most successful OpenFlow network simulator enabling to build an OpenFlow network on the local machine. Moreover, there are several controllers that implemented OpenFlow protocol, such as Floodlight [6] and Nox [7]. In this report, we use Floodlight as the controller to show an example of simulated OpenFlow testbed. Shown in Figure 1.3, the OpenFlow network is comprised of 4 OpenFlow (OF) switches which are simulated in Mininet simulation environment. The Controller Floodlight is logically linked to all the OFSwitches.

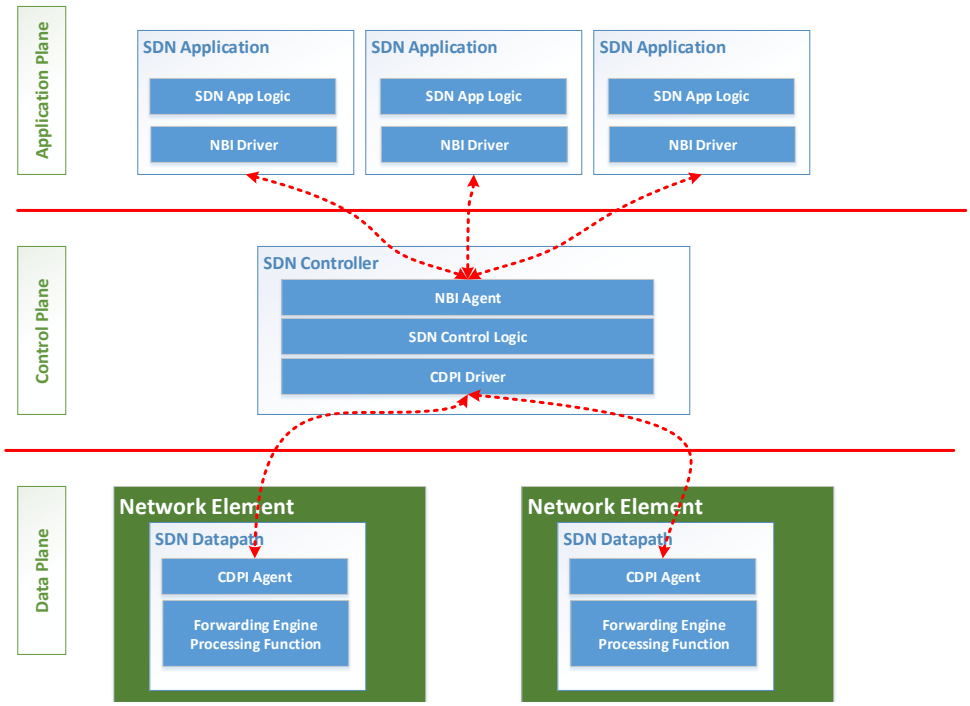

Figure 1.1: SDN Architecture

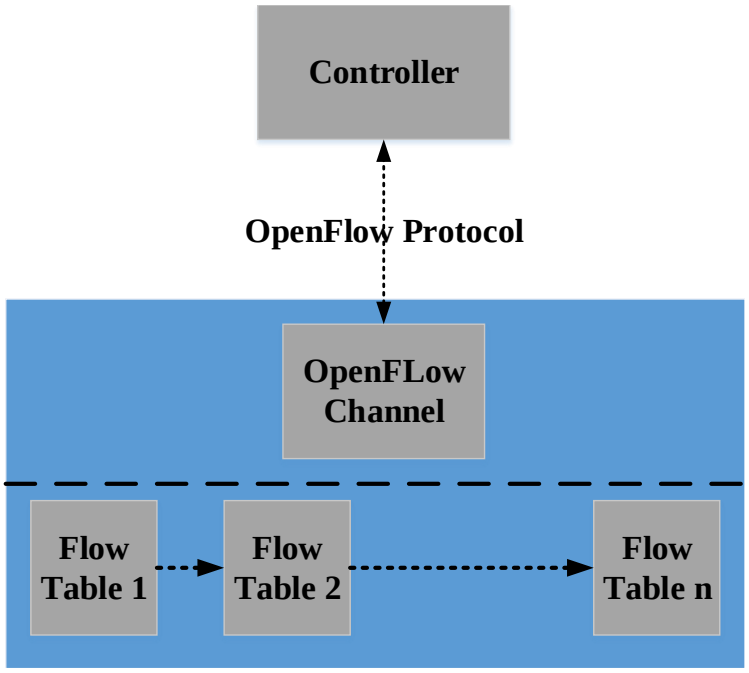

Figure 1.2: OpenFlow protocol

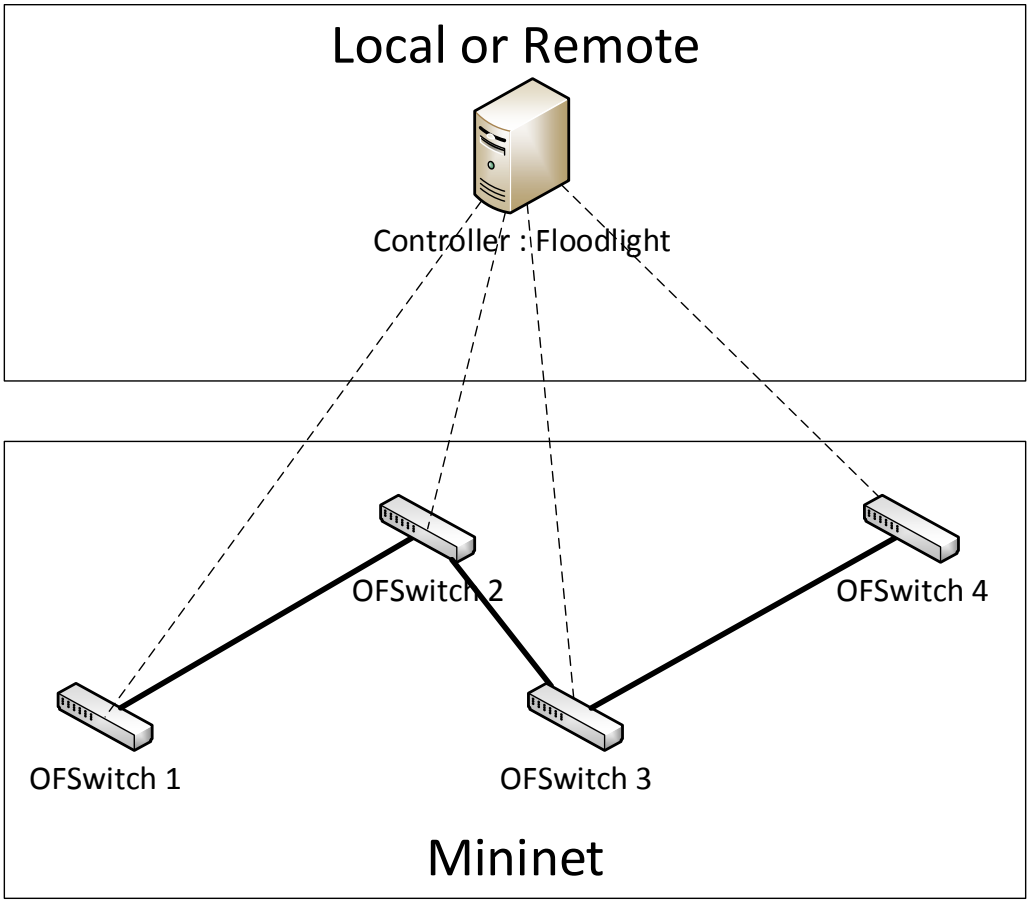

Figure 1.3: SDN Testbed

1.2 Transmission control protocol (TCP)

TCP has been proposed since 1981 [8] and is still one of the most popular protocols in the current internet. TCP is a reliable end-to-end, connection-oriented, byte-stream protocol of transport layer of OSI model [9]. TCP flows start with establishing a connection between two hosts to initialize and maintain the session for the data stream. Data can be delivered in-order of a stream of bytes in each direction on top of a less reliable network. TCP also includes flow control mechanism which re-transfers lost packets and adopt sending data speed according to the network status. Moreover, the TCP provides security mechanism and allows many simultaneous TCP flows within a single host.

1.2.1 TCP Connection Establishment and Termination

In the normal case, TCP uses a Three-Way Handshake to establish a connection shown as Figure 1.4. The purpose of Three-Way Handshake is that the Client and Server agree on the starting sequence numbers that the two sides want to use for their respective byte streams [10]. The following the process of Three-Way Handshake:

1. Both client and server start from CLOSE state. The Client performs an active open which creates a transmission control block (TCB) to store all the important information about the connection and sends out a SYN message to the server with a random sequence number X. Then the Client moves to SYN-SENT state to wait for an acknowledgment (ACK) to SYN. The Server creates a TCB and moves to LISTEN states to wait for contact from a client.

2. The server receives the SYN message and replies with am SYN-ACK message which the acknowledgment number is the received sequence number X plus 1, and the sequence number is random number Y generated on the Server. Then

the Server moves to SYN-RECEIVED state to wait for the ACK message.

3. Finally, the Clint receives the SYN+ACK that confirms the sequence number X is received on the Server. Then the client sends back an ACK which the sequence number is received $X + 1$, and acknowledgment number is $Y + 1$. Then the Client moves to ESTABLISHED state. The Server receives the ACK and also moves to the ESTABLISHED state. Since then, a full-duplex communication is established.

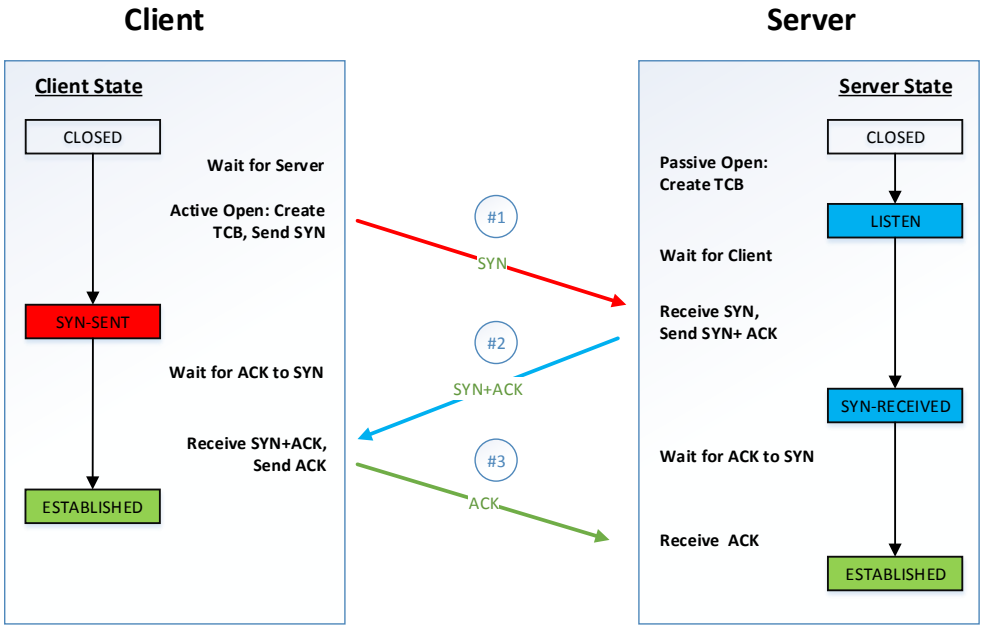

Figure 1.4: TCP Connection Establishment

In the typical case, the TCP uses Four-Way Handshake to close the connection [11] shown as Figure 1.5. Each side of the connection terminates the connection independently [12]. The process of Four-Way Handshake is similar to TCP Three-Way Handshake. When the Client wishes to terminate the connection, it sends a FIN message to the server and moves to FIN-WAIT-1 state to wait for ACK and FIN from the server. The server receives FIN, sends ACK, informs the application (APP) to stop, and moves to CLOSE-WAIT state. When the APP on Server is ready to close, the Server sends FIN to the Client and move to LAST-ACK state. Once the

Server receives the ACK to FIN, it closes the connection move to CLOSE state. The Client side is more complicated than the Server. The Client receives the ACK and moves to FIN-WAIT-2. Then the Client receives FIN, sends ACK to the server, and moves to TIME-WAIT. After 2 Maximum Segment Life(MSL), the Client closes the connection and moves to CLOSE state.

The TIME-WAIT state is designed for two purposes: First, it makes sure the ACK reliable transmit to the Server. Second, it provides a time gap to isolate the current connection with any subsequent ones. Otherwise, the TCP segments from different connections could be confusion. Hence, the Client in TIME-WAIT state is not available for establishing a new connection. The standard MSL is 120s. In modern networks, the operating system allows selecting a lower value if it will lead to a better performance.

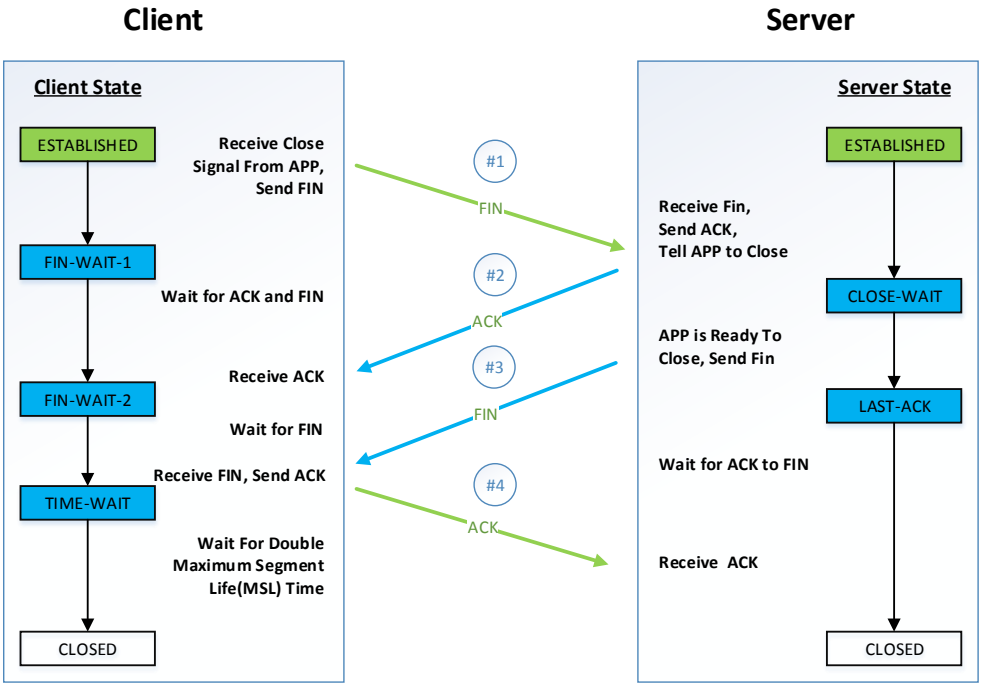

Figure 1.5: TCP Connection Termination

The TCP state diagram shown as Figure 1.6 is a summary of the TCP state transition for both TCP connection establish and terminate. In additional, it demonstrates

the edge cases for simultaneously establishing and terminating TCP session. More details can be found in [8]

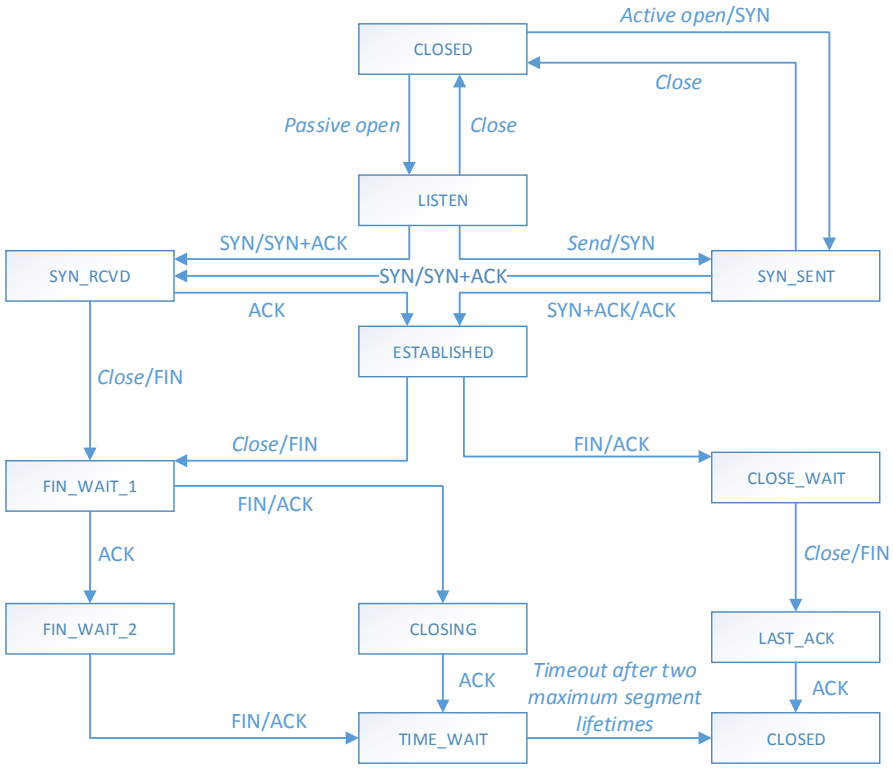

Figure 1.6: TCP state Diagram

1.2.2 TCP Fairness

The flow control mechanism also provides TCP fairness. If K TCP sessions share same bottleneck link of bandwidth R, each should have an average rate of R/K. This dissertation uses Jain's fairness index [13] to measure the TCP fairness. The equation is shown in Eq 2.1 where x_i represents the throughput of TCP flow-i, n is the total number of flows. If each TCP flow shares the bottleneck link equally, then $F = 1$ is the upper bound of Jain's fairness index and stands for the best fairness. The worst case is that one flow occupies the entire bandwidth of the bottleneck, then $F = \frac{1}{n}$ is

the lower bound of Jain's fairness index and stands for the worst fairness.

$$F(x_1, x_2..., x_n) = \frac{(\sum_{i=i}^{n} x_i)^2}{n \sum_{i=i}^{n} x_i^2} \tag{1.1}$$

1.2.3 TCP Proxy

A TCP proxy [14] consisting of multiple TCP servers and clients acts as an intermediary node between clients and remote servers. For each TCP flow from the client to the remote server, a TCP proxy maintains two TCP sessions which are from the client to TCP proxy server and TCP proxy client to the remote server. The TCP proxy server receives the layer 5-7 data from the client and forwards it to the remote server from the TCP proxy client. TCP proxy also can perform more than forwarding data, such as provides weighted fair queue that allows TCP proxy to define traffic classes and then assign different bandwidth to each class [15], initials TCP session with optimized TCP configuration other than the default of the client and remote server [16].

However, lacking the globe view of the network, the proxy doesn't have the capacity to make a global optimum decision. Extending the SDN controller to fully control the TCP proxy will solving this problem, shown as Figure 1.7. SDN controller not only keeps tracking the entire network but also can monitor and manipulate the packets of layer 2-4. TCP proxy is further expanded, such as can be transparent to both client and the remote server, cooperate with other TCP proxies, glue each TCP flow control status together.

This dissertation optimize the performance of TCP with SDN in different network scenario by exploiting TCP proxy. A few recent studies have focused on improving the TCP throughput performance through the technique of combining different flows in the network. In [17], the authors with the help of simulation experiments improve the smart grid meters' traffic through a flow-aggregation framework. Along similar lines,

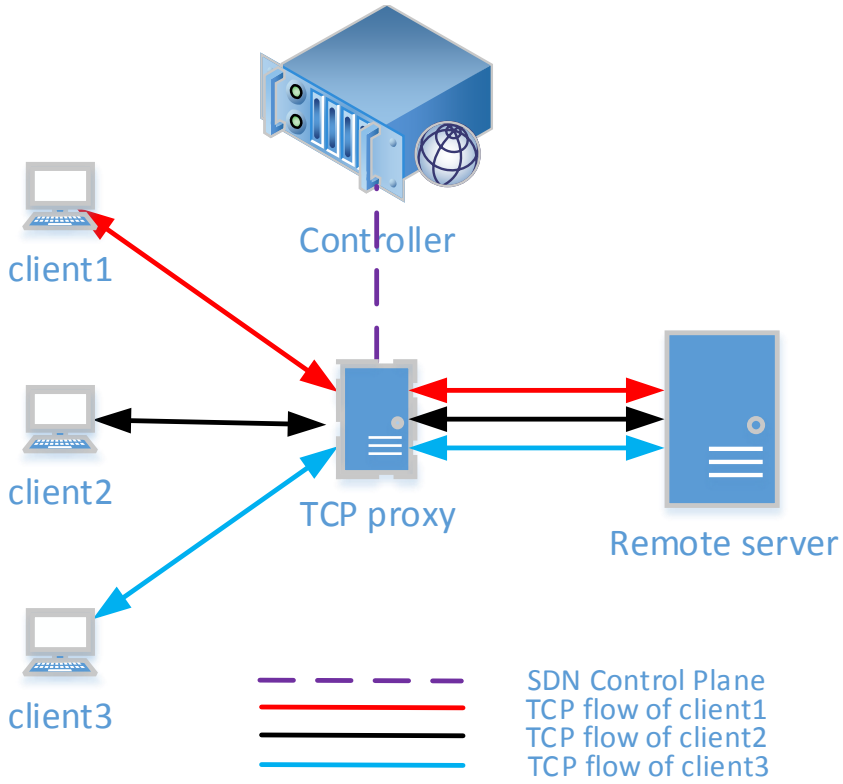

Figure 1.7: TCP proxy on SDN platform

in a LTE-based wireless smart grid scenario [18], we showed that with appropriate TCP aggregation and scheduling, fairness among the TCP flows can be achieved (in addition to obtaining improved throughput).

In a different work [19], we proposed an integration of IoT-based MQTT messages at the edge switches (also known as fog nodes) for achieving improved delivery performance. The aforementioned works in common support the logic of simple flow-aggregation frameworks that cannot be non-trivially extended and applied to generic network scenarios that involve flow joining and splitting of flows. In a different work [20], we proposed split-only framework to separate flows into a chain of two flows with first flow providing congestion-free wireless transport, and the second (part of the) wired-network flow providing regular congestion based TCP transport.

Unlike the existing works that independently address either aggregation-only [18] or split-only [20] frameworks of TCP flows, in this dissertation, we propose a unified

join-and-split framework of TCP flows that provides an effective information synchronization platform among the join and fork proxy points in the network. This feature enables aggregation and split functionality to work at different points of the same network.

Works such as [21] [22] [23] exploit the idea of preserving end-to-end semantics of flows by caching the ACK segments, with the help of proxy nodes. However, they do not focus on aggregating multiple TCP flows into a single flow. On the other hand, MPTCP flow-based proxy framework has been proposed in [1]. However, this work did not provide the details of preserving end-to-end semantics in the MPTCP proxy. We in this dissertation integrate the MPTCP proxy into our 'join-and-split' framework along with a concept of Linked-ACK to maintain the TCP end-to-end semantics.

1.3 Smart Grid

Smart Grid network is defined as the next generation power grid networks in which the electricity distribution and management are upgraded by incorporating advanced two-way communications and pervasive computing capabilities for improved control, efficiency, reliability, and safety [24]. There is still a lot of variation definition of Smart Grid. Typically, all of them consist of distributed intelligence, communication technologies, and automated control systems. [25]

Smart metering (SMs) is a device deployed at the distribution-end with bi-directional communication to collect data or receive feedback. The purpose of SMs is to enable continuous monitoring and better utilization of resources at the customer-end users (*i.e.*, the electricity consumers). Some of the benefits include automatic billing, load balancing, remote connect/disconnect [26]. The latest SMs are advanced with processing capabilities and are integrated with full network transport suites such as

TCP or UDP (User Datagram Protocol is an alternative communication protocol of TCP, and primarily for low-latency and loss tolerating connections).

Data communication is the key enabler in Smart Grid networks. The deployment of communication paradigm in the power domain yields benefits to all participants in the system such as utility companies, governments, and consumers. Typical Smart Grid network spans a vast geographical area connecting many devices such as SMs. In a city scenario, millions of smart meters are distributed in the whole city. To successfully collect, transfer, analyze, and store such massive data, move the server of the Smart Grid to the Cloud, which is such a useful technology for Smart Grid information management.

On a wired smart grid network, the authors in [27], [28] demonstrate an improved TCP performance for the SMs using aggregator nodes that combine multiple TCP connections. However, a natural extension of studying fairness among tandem aggregator nodes is not investigated. In our earlier work [29], we proposed SDN-based Fog computing nodes for Internet of Things (IoT) applications, and demonstrated an improved TCP performance, by migrating the remote server functionality to the edge switch for immediate response. A backup data transport is enabled by a single TCP from edge switch server to a remote end-host server. However, the concept of multiple edge-servers (aggregators) and their corresponding fairness is not explored. Unlike the aforementioned works, in this dissertation, we exclusively study the fairness among multiple (tandem) flows over wireless and wired networks in a smart grid scenario.

1.4 Gaming traffic

Online games become significant contributors to Internet traffic, [30]. Recent reports suggest that the game traffic is getting a dominant share of the internet traffic. It is

worth to note that the global volume of game traffic has 22% of Compounded Annual Growth Rate (CAGR) [31].

One of the most popular video game categories is the first-person shooter (FPS) [32] which is centered around guns and other weapon-based combat in a first-person perspective. Most of FPS games feature an online multiplayer model that let players compete or cooperate with other players in one virtual 3D environment in real time.

A famous FPS game is Counter-Strike in which teams of terrorists and counter-terrorists battle [33]. The Counter-Strike build the multiplayer game with a client-server network model shown as Figure1.8. It's a star network topology in which the central hub node hosts the game server, all leaf nodes are game clients. During the game, the server collects and synchronizes data to all clients and tries to enforce all game players sharing the same players status. The advantage of the client-server network model is that game clients can join or leave without disturbing the network. Counter-Strike is very fast paced, delay sensitive, and has the capacity to host tens of players in one virtual world. It relies on UDP protocol with high packet rate, short packet size and short packet inter-arrive times.

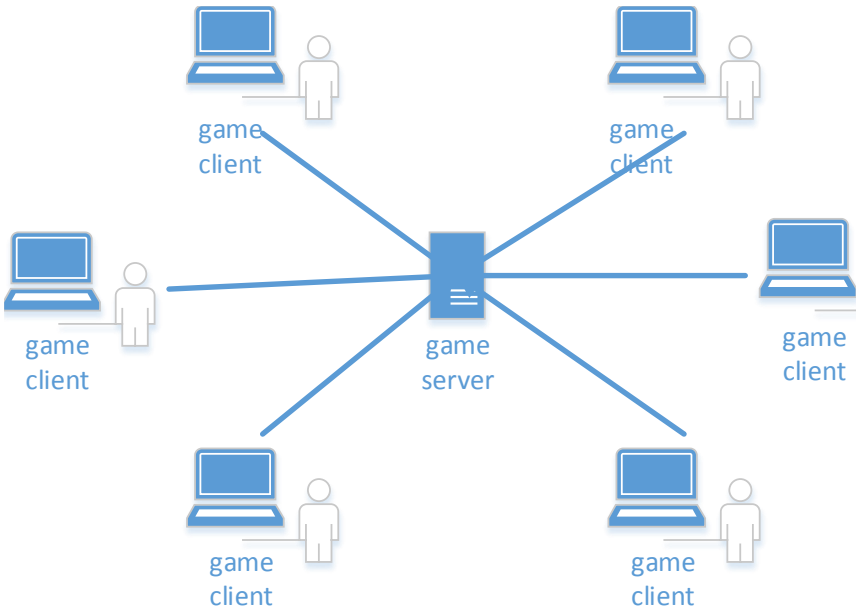

Figure 1.8: Client-server network model

This dissertation will address the gaming trafffic delay problem in WiFi network. The authors in [34] studied the throughput unfairness among homogeneous types of TCP uplink and TCP downlink traffic, and proposed a solution of modifying TCP ACK segments in order to control the throughput, and as a result achieved fairness among the flows. As this solution, stands on modifying TCP ACKs which is clocked by the WiFi AP at the speed of the propagation delay (which includes both wired and wireless part of the network). The throughput of TCP is effectively reduced by this approach. On the other hand, in this dissertation, we in the first place split the TCP between the wired and wireless part, therefore the throughput of TCP is a function wireless (MAC) propagation delay only. Subsequent to the we contain the congestion control of wireless TCP's counterpart. Intuitively our proposed approach effectively improves the TCP throughput.

The authors in [35], provided fairness among TCP uplink and TCP downlink traffic by dynamically adjusting the WiFi AP's buffer size in a simulation environment. On the other hand, our solution of ensuring fairness developed in SDN based framework is production-ready and presents a functional proof on a real testbed. In a different work, the authors in [36] address the TCP unfairness among TCP uplink and downlink traffic by appropriately modifying 802.11e WLANs EDCF configuration to ensure fairness. This solution is effective only on a specific standard of 802.11e WiFi, and recent studies [37] have shown that the ease of 802.11e's configurable parameters can also lead to creating selfish WiFi nodes that throttles the throughput of other nodes. On the other hand, our solution relies on higher network layer that works on wide versions of 802.11 protocols including the widely deployed relatively configuration robust 802.11b protocols.

1.5 Virtual machine live migration

Many enterprises, not only the large-market companies like Netflix and Snapchat, but also small-market technology startup companies rely in large part on the data center for computing infrastructure or business [38] [39]. These companies often impose multifarious resource demands (storage, compute power, bandwidth, latency) on the underlying infrastructure [40]. Moreover, the resource demands may change over time according to companies' requirement. To provide an effective, flexible, security, scalable, energy efficiency approach to manage the data center resources, the data center visualization technology is proposed and implemented in the current data centers like AWS from Amazon, Azure from Microsoft.

Data center virtualization uses software or firmware called Hypervisor that virtualizes hardware such as servers, switches, links to be virtual machines, virtual switches, and virtual links. For example, a physical machine (server) can be virtualized as multiple independent VMs with different hardware capacities and operating systems.

One of the powerful features provided by virtualization is Virtual Machine (VM) live migration, which facilitates moving workloads within the infrastructure to bring multiple benefits such as higher performance, improved manageability and fault tolerance. Moreover, live migration of VMs often allows workload movement with negligible downtime, minimal impact on workload, and no disruption of network connectivity [41] [42].

Clark et at [43] proposed a VM live migration system which transfers Memory, storage and application status (CPU state, registers,non-pageable memory) of the virtual machine from the original server to the destination. The system handles the live migration by two main techniques, Pre-copy memory migration, and Stop-and-copy memory migration shown as Figure 1.9.

In pre-copy memory migration, the Hypervisor typically copies all the memory

pages from source to destination while the VM is still running on the source. The updated memory pages during this process are re-copied until page dirtying rate is faster than the rate of copy. In Stop-and-copy memory migration, it transfers the remaining memory pages and application status to the destination, then stops the original VM and resume on the destination.

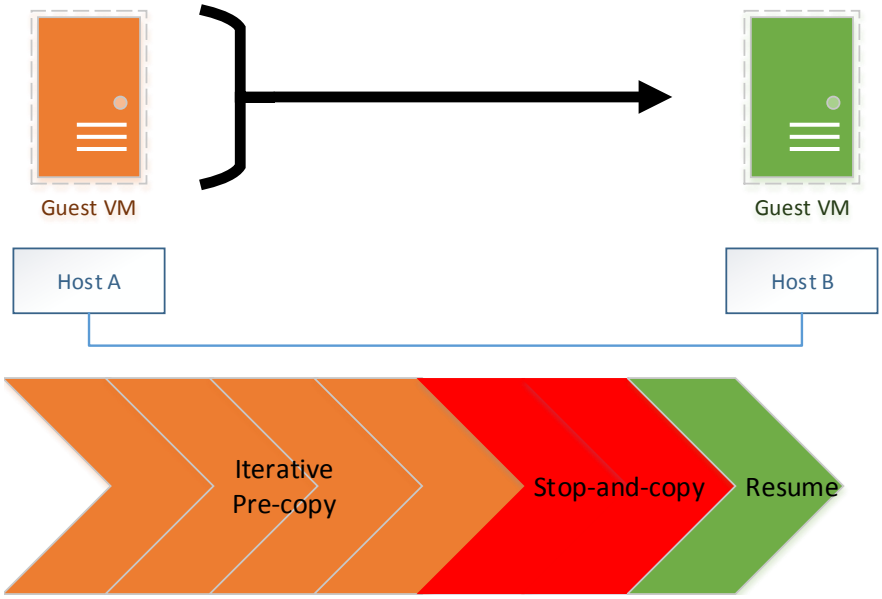

Figure 1.9: VM Live Migration Pre-copy and Stop-and-copy phase

This dissertation will optimize the scheduling of VM live migration problem. The cost of VM migration has been defined in different ways in terms of a combination of network, physical server and application performance. However, some of them did not consider the application performance degradation or some of them didn't consider the total performance degradation for all VMs. Liu at el. [44] defined total VM migration cost as a synthesized formula that integrates migration latency, total network traffic during migration, application downtime and energy cost. They evaluated their model with a real testbed for 8 VMs and did not consider the application performance degradation during the migration. Breitgand et al. [45] defined the cost as the portion of the requests to the VM that are not satisfied by their deadline. They found out that the more bandwidth the migration program uses, the faster the migration

will finish but more requests will miss the deadline. Fei at el. [46] proposed an interference-aware VM live migration strategy where the interference is defined as the performance degradation that the migrating VM imposes on other VMs hosted on the same source or destination server. The interference-aware strategy they proposed for making decisions on which VMs are to be migrated comes from an empirical study. Mann et al. [47] proposed a system named Remedy that uses a cost estimation model to minimize the cost of VM live migration, where cost is the network traffic that is generated due to migration. They assume that the destinations for the VMs are not known apriori and tries to relocate VMs in such as to reduce the network traffic.

1.6 Contributions of this dissertation

SDN is one of the most significant innovation in the networking field in recent times. This dissertation proposes a TCP join and split protocol on SDN platform to improve the performance of TCP. This protocol is extended to improve TCP fairness in Smart Grid Networks and to reduce delay in gaming traffic on WiFi networks. SDN is also maturing rapidly in Data Center networks which provide virtual infrastructure for the organization. This dissertation addresses the performance of virtual machines (VMs) live migration by proposing a Mixed Integer Linear Programming (MILP) model and a heuristic algorithm which exploits the TCP join and split protocol. The main contributions of this dissertation:

1. Developed and implemented a novel 'join and split' TCP framework based on SDN that seamlessly joins and splits TCP flows to achieve better performance. Linked-ACK concept is proposed to maintain the TCP end-to-end semantics, and effectively control the buffer usage of the proxy network points. Provide a platform to offload TCP fine tuning from clients and servers to 'join and split' proxy points, for better controllability [20] [48].

2. We propose a novel SDN-based aggregation-cum-scheduling framework to improve fairness and as well as maintain the improved TCP throughput performance found in traditional aggregation-only frameworks. Unlike conceptual idea on simulation, we present a white-box design of the proposed framework by highlighting the implementation functionalities equivalent to developing a working prototype. We extensively study the throughput performance and fairness with appropriate analytically model validating the experimental results [18] [49].

3. We provide an SDN-based solution of TCP-splitting along with partially-controlled wireless sending rate to ensure fair throughput. The fair use of TCP downlink resources allows the UDP game traffic to effectively utilize the residual shared resources such as AP buffer, which enables them to achieve reduced delay. We present an extensive performance study on the real testbeds to demonstrate the prowess of our approach. Our implementation is production ready, as it is tested on the off-the-shelf network components [50].

4. We improved the performance of virtual machines (VMs) live migration by proposing a Mixed integer linear programming (MILP) model and a heuristic algorithm which exploits the TCP join and split protocol [51].

Chapter 2

Join and spilt TCP for SDN network: design, implementation and evaluation

2.1 Introduction

Today's Internet is constantly growing with the addition of new sets of applications and devices. A network increasing in scale should also renew and reinvent its core functional needs and adapt to new design paradigms in order to provide improved delivery performance. Software-Defined Network (SDN) [2] is one candidate paradigm that provides effective network management and dynamic flow steering capabilities that enable engineers to build efficient network services.

While the traditional switches and routers perform simple packet routing and forwarding, the SDN-enabled switches of today, on the other hand, can intelligently forward as well as dynamically steer the network traffic. Appropriate steering and management of flows in the network can be helpful in improving the delivery performance of the associated flows. In this work, we exploit SDN technologies and develop efficient frameworks to provide improved delivery performance of the network flows. To this end, we present an SDN-based end-user agnostic 'join-and-split' framework for TCP flows, that effectively maintains end-to-end flow semantics. By end-to-end semantics, we mean that upon a successful packet delivery, an acknowledgment for a data segment comes from its original destination host, and not from any other intermediate system [52].

In the framework, all TCP flows that share a common path are aggregated at

the beginning of this shared path. In a typical network, this may happen on a node at the network edge. For instance, geographically deployed (cellular-based) smart grid meters that send their data to different servers that serve different purposes such as monitoring, and power load balancing [53]. Another strategic point near the server-side part of the access network can form the other end-point of this shared path. Between these two join and split points, a single long TCP flow can steadily transfer the aggregated network traffic by exploiting the common congestion control mechanism of this TCP. The significant task of this framework lies in effectively synchronizing necessary information between join and split network points. To this end, we provide a control plane functionality to the join and split network points for enabling synchronized information transfer. Moreover, this framework needs to function in a seamless manner without user interference. The aggregated flow must be routed between the join and the split points of the network in a user agnostic manner.

Our framework maintains synchronized TCP session states between split point-server part of the flows, and the corresponding client-join point of the flows. TCP connections carry state information in the form of TCP options that are negotiated during the connection setup time [54], such as enable selected ACK, enable timestamp, and enable MPTCP [55]. These options are lost when the TCP flows aggregate into a single long flow. This dissertation exploits SDN controller to perform Deep Packet Inspection (DPI) [56] on each SYN segment from client, and synchronize the parsed TCP options to the split TCP point (proxy) to establish the connection with same options as required by the client.

The two proxy points in the network split the TCP flows into three non-overlapping independent flows. These separated flows will have different throughput, and unless properly handled would break the end-to-end flow semantics [57]. In this work, we propose a new concept of Linked-ACK, wherein the ACKs of server side flows are sent

along the path from the server to the client. Therefore, the clients receive ACKs only when the packet is received at the server. The Linked-ACK mechanism can help in maintaining the end-to-end flow semantics. In addition, the Linked-ACK limits the total buffered data proportional to the maximum (sender) congestion window size of the corresponding flows. In this manner, the buffers of the join and split proxy nodes are protected from a potential overflow.

While a number of recent works have been attempted to improve TCP performance by tuning the server-side TCP parameters, our framework provide the designers more controls to fine-tune the TCP flow and achieve a better performance. Thanks to the SDN technologies, with the help of SDN controller we can provide global view and control of the network information which can be appropriately utilized to improve the network performance. For instance, to support Multi-Path TCP (MPTCP), end-users are required to upgrade to compatible kernel-code. On the other hand, our framework facilitates the 'join' proxy to be used as MPTCP proxy point and help the end-users to benefit from the advantages of MPTCP without modifying enduser side kernel codes. We believe this will provide a flexible and scalable solution for the legacy systems in the network.

In summary, the main contributions of this work are as follows:

1. Develop and implement a novel 'join and split' TCP framework based on SDN that seamlessly joins and splits TCP flows to achieve better performance.

2. Linked-ACK concept is proposed to maintain the TCP end-to-end semantics, and effectively control the buffer usage of the proxy network points

3. Provide a platform to offload TCP fine tuning from clients and servers to 'join and split' proxy points, for better controllability.

2.2 System Design and Implementation

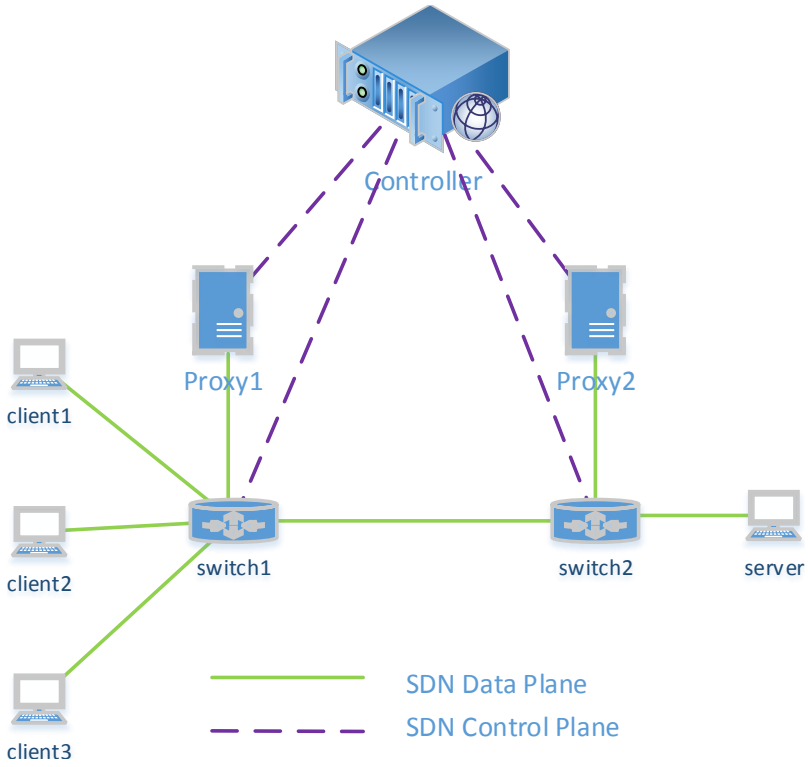

Figure 2.1: System Model. Clients, switches and proxies are wired connected. SDN controller connects to both switches and is expanded to connect to both switches

Figure 2.1 shows the system model used throughout the dissertation. Each switch is considered to support SDN's OpenFlow protocol. A proxy computing node is attached to each of the switches. An SDN controller is connected to these OpenFlow switches and proxy nodes. The controller interacts with the switches through OpenFlow protocol, and sends commands to the proxy nodes through a custom-designed application-layer protocol running over TCP network stack. Without loss of generality, three clients share the same path from Switch-1 to Switch-2. Each of the three clients are represented as C1, C2, and C3. The join proxy is represented as P1, and the split-proxy is represented as P2. The server are represented as 'S'.

2.3 SDN-based TCP Join and Split Framework

In our framework, the TCP flows from each of the clients are combined to form a long TCP flow along a shared path of the network. The SDN controller generates a Unique IDentification (UID) number, and associates it to each of the flows to be aggregated. The join-proxy attaches the UID to each received data, and sends it to the split-proxy node. The received data on split-proxy can split the flows properly with the help of UID. As shown in Fig. 1, proxy1 joins TCP flows originated from three different clients, namely C1, C2, and C3. Proxy2 splits the flow from each client, and sends them to the server.

Clients and the server need not necessarily learn any information from the network. To make the join-proxy and split-proxy transparent to both clients and the server, the SDN controller is programmed to setup flow tables to create fake the TCP connections between clients and join-proxy node, and subsequently between the split-proxy and the server. While the clients assume that they are transmitting data to the server, the data is actually sent to join-proxy node. A simple join-and-split framework has been presented in [20]. In this join-and-split framework, we need to ensure that the end-to-end semantics are maintained. Each TCP connection from every client to the server is divided into three individual TCP connections at the join and split proxy points. Each of these individual TCP connections adjusts the throughput in its own path, and maintains its own TCP states. Typically, as incoming flow rate is higher than the output rate at the proxy node, it creates an unstable queue. Therefore, it is necessary to maintain the end-to-end semantics, and also maintain queue stability. In this dissertation, we propose a "Linked-ACK" to link the ACKs of three individual TCP flows through custom-defined Open- Flow protocol in the switches.

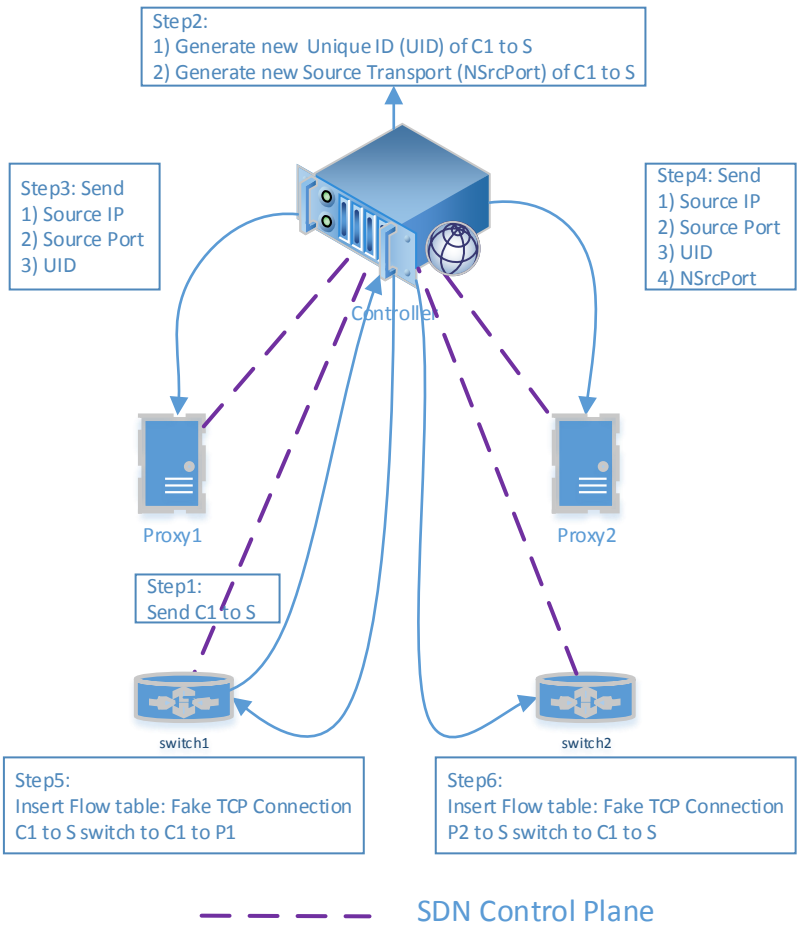

Figure 2.2: SDN join and split control plane. When a new SYN segments come to switch, the SDN control plane execute the program following the step 1 to 6

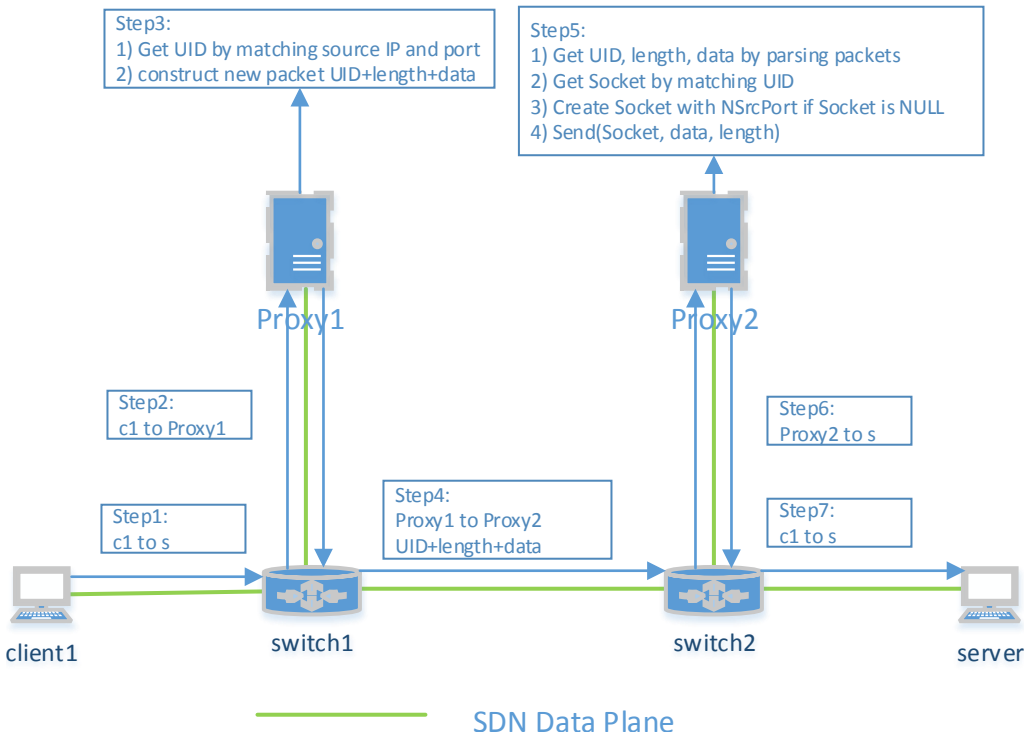

Figure 2.3: SDN join and split data flow initiation using TCP SYN segments

2.3.1 SDN-based TCP join and split

SDN supports both proactive and reactive ways of flow routing. Proactive way populates flow tables ahead of the traffic coming from the switch. On the other hand, the reactive way handles the flows 'on the fly' depending on the information provided by the incoming flows. In our work, we consider reactive way of routing the flows. Without loss of generality, we consider that the routing table entries directing flows from, and to the (join and split) proxy nodes are pre-installed. The first segment of the incoming TCP flow from each client is forwarded to the SDN controller for analysis. The sequence of steps in the flow table setup process is shown in Fig 2.2. Switch-1 forwards the first packet (TCP SYN packet) from C1 (destined towards S) to the SDN controller.

The SDN controller analyses the received segment, extracts the client's information, and distributes this information to the split and join proxy points. The SDN

controller then generates a UID, and creates a new TCP transport (namely, NSrc-Port) between the split-proxy point and the server, which is shown in step2 in the Fig 2.2. In a special case, when the new flow is one of the subflows of an MPTCP flow, they would share the same UID. The UID with associated flow information is the key component used in splitting and joining of flows. While a standard (out-of-the-box) SDN controller lacks any control of the proxy, we extend the SDN control plane functionality to allow the controller to manage the proxies attached to each OpenFlow switch with a custom application protocol developed over TCP.

After analyzing the SYN packet, the controller computes the routing path and creates a fake TCP connection between the client and join-proxy point. To this end, a flow table entry substitutes the server information with join-proxy point's information in the specific fields of data link layer, network layer, and transport layer of the incoming flow from the client. This modified TCP flow can be accepted by the TCP server of join-proxy and vice versa. In a similar way, another fake TCP connection is established between split-proxy and the server. This TCP connection fakes the TCP client information (which typically comes with an arbitrary source port number assigned by the client's OS). It is worthwhile to note that the TCP flow from split-proxy to the server with random source port can't be associated with the client information in the SDN controller. Therefore, split-proxy starts the TCP connection with a given source port NSrcPort for SDN controller to retrieve client information.

After the flow tables are configured, the join-proxy receives data from clients and retrieves the UID and the clien's information (i.e., source IP address, and port number). Then it constructs a new packet containing the following application layer information: "UID+length+data", where 'length' is the total length of the constructed packet). Split-proxy point retrieves the UID and data, and pushes the data to a pre-established socket, as shown as Fig 2.3.

We consider a 2-Byte UID thereby supporting a maximum of 65532 concurrent

TCP flows, and use 'source IP and port-number' information to identify these flows. The TCP flows come with different TCP options that needs to be handled properly when the split-proxy point establishes a new connection with the server. With the help of SDN technologies, we can perform deep-packet inspection on these packets, and forward the necessary TCP header configuration information to the split-proxy. The split-proxy point can then establish a TCP flow with same configuration as if it was an original client exchanging packets with the server (hence the name 'proxy' points). The SDN controller is capable to inspect the application-layer information over TCP. Therefore, by extending application layer functionality, our system (in future) can also support application-layer join and split frameworks.

2.3.2 Preserving End-to-End Flow Semantics with 'Linked-ACK'

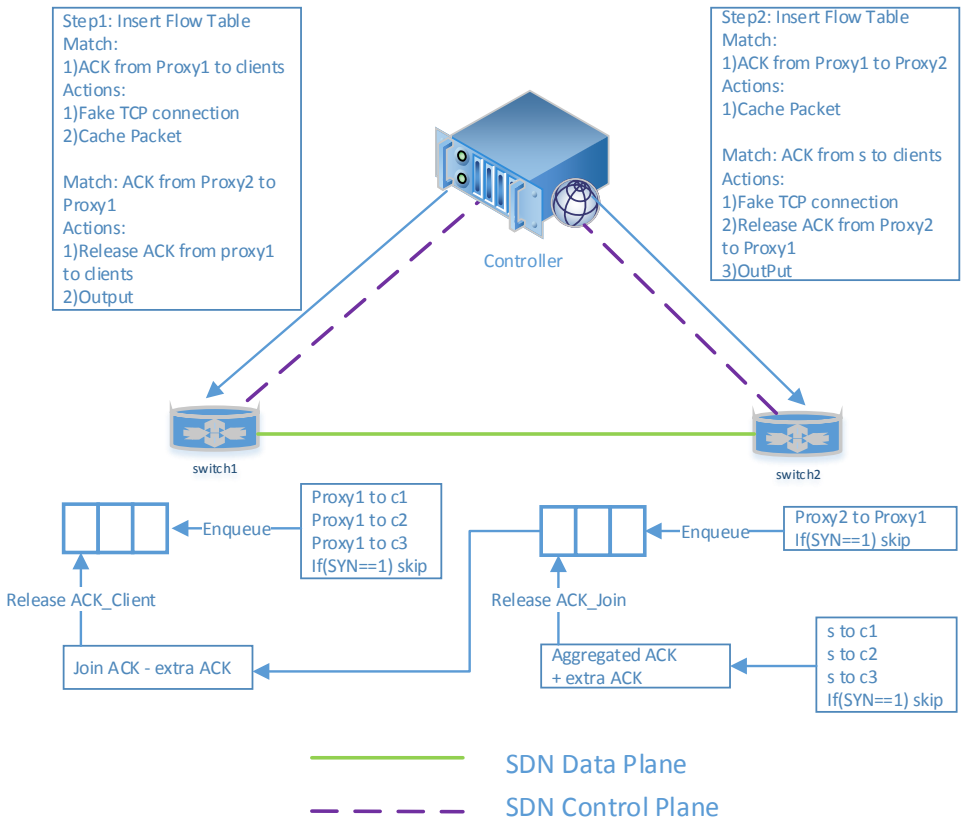

Figure 2.4: Linked-ACK. It shows the ACK collection, caching and distribution with relative installed flow tables

In this section, we describe the "Linked-ACK" framework that we have developed to maintain the end-to-end semantics of the flows. As the TCP flow is split into 3 independent TCP flows, each of the resultant split flows will have its own TCP congestion state, and throughput rate. Assuming the client is sending data at a constant rate, the join-proxy maintains a buffer to store the received data from the client. While a larger buffer size is expensive, a small buffer size on the other hand negatively impacts with a reduced TCP throughput.

Our proposed 'Linked-ACK' provides a better solution, and bounds the buffer to a finite size. For brevity, let us represent the different ACK messages along a server-client network path as follows: (i) The ACK message 'from server to split-proxy' be represented as ACK split, the ACK message from 'split-proxy to join proxy' be represented as ACK join, and the ACK message from 'join-proxy to client' be represented as ACK client. Our 'Linked-ACK' framework operates in a lock-step fashion wherein the ACK join is not released until its associated ACK split packet is released. In a similar way, an ACK client is received only upon the release of its associated ACK join. We customize the standard OpenFlow protocol, and add the following four actions: caching ACK join, caching ACK client, release ACK join, and release ACK client. The modified flow table entries for routing the ACK packets are shown in step1 and step2 in Fig 2.4. The ACK client is cached (instead of being released) after the fake TCP connection is established on the switch-1. This fake TCP connection directly connects to the join-proxy node. In a similar way, ACK join is cached on the switch-2.

The ACK join, and the ACK client are stored in a FIFO queue data structure, with an exception to the SYN+ACK segment which will be released immediately to complete the three-way handshake. To guarantee that no ACK segments are lost, the length of the FIFO queue is set to a size larger than the sender's total congestion window (CWND) size. Even when sender-side CWND is large, our approach works

due to the TCP's 'cumulative ACK' mechanism wherein one ACK message represents the summary of received in-order bytes.

The Algorithm-1 describes the release of ACK-join segment on split proxy node to the join proxy node. We maintain an aggregated ACK (namely, aggregateACK) for all TCP-split segments. This is done because the delayed ACK has to combine several ACK responses together into a single response. As TCP uses the cumulative ACKs, our 'Linked-ACK' releases ACKs based on the increased ACK value, not on the number of ACK segments. To this end, the IncreasedACKValue() function returns the increased ACK value by comparing ACK values with the previous ACKs. For a corner case, this function returns 0 if the ACK number is 0, which is caused by SYN or RST flag.

The extra information such as UID and length which is attached to the data message has to be taken into account in the ACK join. Therefore, a function called extraACK() adds an extra ACK value to the aggregated ACK. Subsequently, by using a loop this algorithm keeps checking for an increased ACK value in the ACK join message stored in a queue. If ACK join has smaller increased ACK value than the aggregated ACK segment, the ACKjoin is dequeued and sent to the join-proxy.

The release ACK Client function which runs on join-proxy nodes works in a similar fashion. This function releases the ACK-client messages to the clients. However, with the following difference: the added extra ACK has to be removed to tally extra (UID+length) data cached in the join-proxy node.

2.3.3 Linked-ACK Framework Based TCP State Machine

Fig 2.5 shows the extended TCP state diagram with Linked-ACK implemented on the proxy nodes. The traditional TCP state diagram [8] describes the different states of a TCP sender/receiver. On the other hand, our extended state diagram in Fig. 5 describes the TCP sender and TCP receiver inside the proxy aggregation node. In the

Algorithm 1: Release ACK function that runs on the split-proxy node

```
1: aggregateACK += IncreasedACKValue(ACK split)
2: aggregateACK += extraACK(ACK split)
3: while true do:
4:     if increasedACK join > aggregateACK then
5:         return
6:     end if
7:     aggregateACK -= increasedACK join
8:     pop and send ACK join
9: end while
```

proxy aggregation node, the TCP receiver (server) receives TCP segments from end host (clients). An application aggregates and buffers the received segments and ACK messages, appropriately. A TCP sender maintains a connection with the TCP split proxy node. Figure 5 shows the exchange of data and ACK segments between senders and receivers. In addition to the traditional states, the following extra states are used: i) PROXY RECEIVER ESTABLISHED, ii) PROXY SENDER ESTABLISHED, iii) DATA BUFFERED, iv) RECEIVER RCVD, v) SENDER SENT, AND vi) ACK BUFFERED.

In PROXY RECEIVER ESTABLISHED state, the TCP receiver (server) accepts TCP connections from end-host clients. In the RECEIVER RCVD state the received data from the clients are pushed into a buffer, and the TCP server transitions to DATA BUFFERED. Senders in DATA BUFFERED state read data from the buffer that are fed by the receivers. In PROXY RECEIVER ESTABLISHED state, when ACK split message is received from the split proxy node, the join proxy node's TCP client transitions to ACK BUFFERED state and releases the buffered ACK join messages to the appropriate receivers. To guarantee that no data segments are lost in the buffer, the minimum buffer size is set to the maximum receiver CWND size.

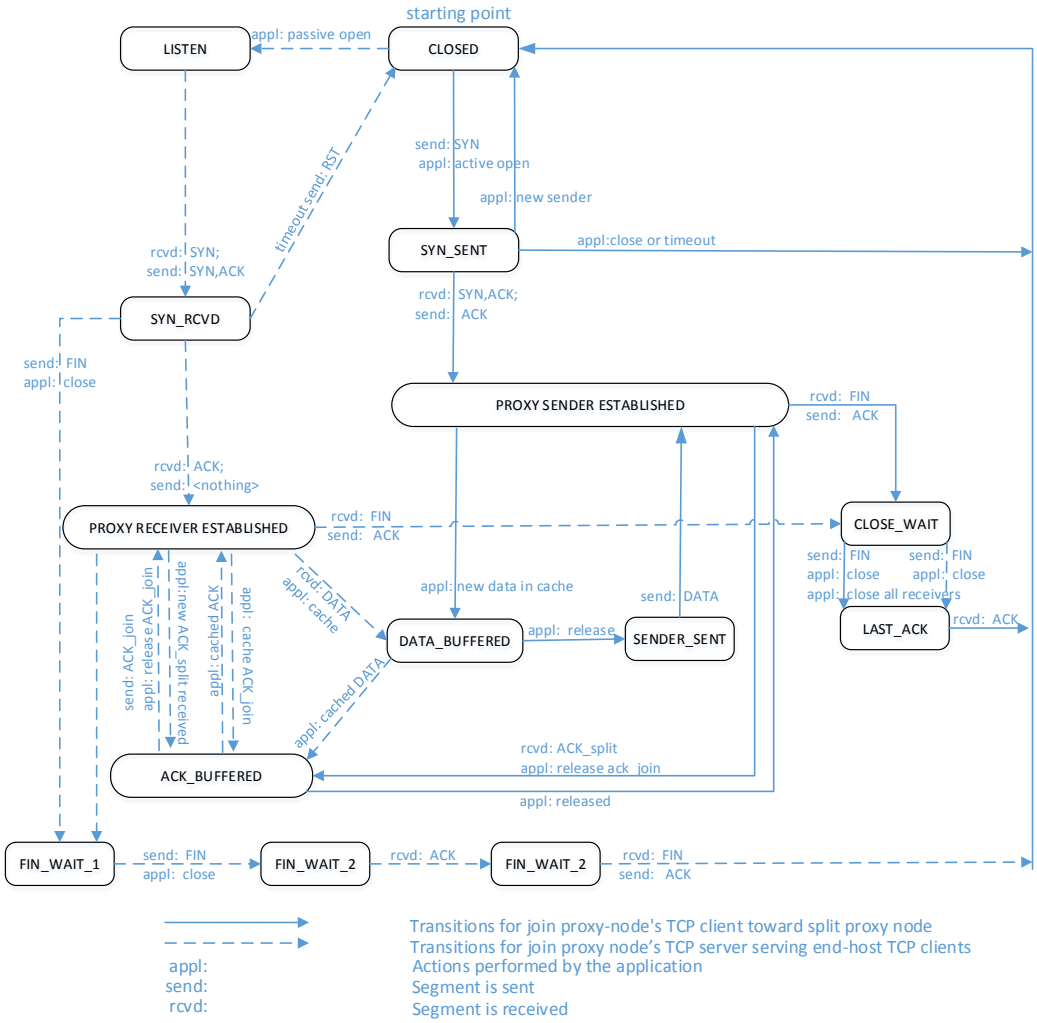

Figure 2.5: TCP state diagram. This diagram show hows how Linked-ACK couples the proxy sender and receiver TCP state

2.4 Performance Evaluation and Results

The network topology shown in Fig 2.1 is used for our performance study. The considered network is emulated in a Mininet network environment [5]. Hosts in the Mininet run in different network namespaces with their own set of network interfaces, IP and routing tables. Switches of Mininet support OpenFlow to enable SDN functionalities. Links in the Mininet emulate bandwidth, delay, and packet loss probability. The popular Floodlight [6] open-source SDN controller is used for managing flow tables in our experiments. All the TCP flows use the default Linux kernel configuration, but the MultiPath TCPs are installed from [58].

2.4.1 Aggregated TCP Goodput Performance

In this section, we show that the aggregated TCP flow can substantially improve the TCP goodput. Using the topology shown in Fig 2.1, we simulated up to a maximum of 800 concurrent TCP flows. The bottleneck link from switch-2 to the server is set to 1.5 Mbps, and all other links are configured to 1 Gbps. The switch to proxy links are considered to have unrestricted bandwidth. The average TCP goodput performance of aggregated TCP, and its non-aggregated TCP counterpart is shown in Fig 2.6. The goodput performance is shown as values normalized using the total link bandwidth. We used 95% confidence. Each test lasted for about 180 seconds. From Table 2.1, it is clear that our approach is potentially scalable as the goodput remained consistently higher with an increase in the number of flows. On the other hand, the regular TCP goodput suffered throughput degradation with an increase in the number of flows. Therefore, we can conclude that the aggregating TCP flows yields better TCP throughput even as the number of TCP flows increases.

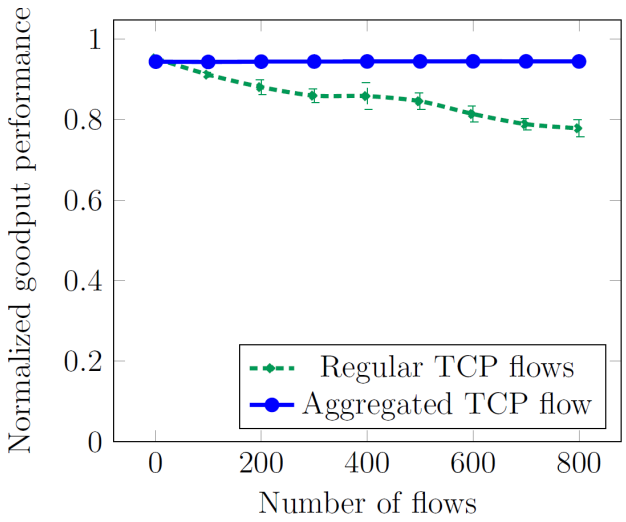

Figure 2.6: Goodput comparison of aggregated TCP flows vs equivalent regular TCP flows

Table 2.1: Aggregated TCP throughput and confidential interval

Flows	Regular TCP throughput			Aggregated TCP throughput		
	Mean	Upper	Lower	Mean	Upper	Lower
1	0.9517	0.9518	0.9516	0.9438	0.9442	0.9435
100	0.9116	0.9178	0.9054	0.9431	0.9437	0.9425
200	0.8806	0.8987	0.8626	0.9439	0.9444	0.9434
300	0.8594	0.8769	0.8418	0.9441	0.9446	0.9437
400	0.8582	0.8907	0.8257	0.9443	0.9445	0.9441
500	0.8455	0.8669	0.8242	0.9444	0.9446	0.9443
600	0.8138	0.8338	0.7938	0.9446	0.9447	0.9444
700	0.7886	0.8028	0.7938	0.9445	0.9446	0.9445
800	0.7783	0.799	0.7576	0.9443	0.9447	0.9440

Flows : Number of concurrent TCP long flows
Mean : Mean value of TCP goodput/total bandwidth.
Upper : Confidential interval upper bounder.
Lower : Confidential interval lower bounder.
The confidential level is 95%.

2.4.2 Linked-ACK Throughput Performance

With our Linked-ACK' framework implementation, the throughput performance in the sub-path between 'split-proxy and the server' path gets synchronized with the throughput along the sub-path between 'client and join-proxy'. Fig 2.7 shows the TCP throughput performance of a long TCP flow from a single client to the server with 'Linked-ACK' framework. It is clear that both the flows have about the same throughput performance. Fig 2.8 shows the total received bytes of the flows in the respective sub-paths of client to join-proxy, and split-proxy to server. It is clear that these flows have almost the same number of total received bytes, which indicates the fact that they are well synchronized.

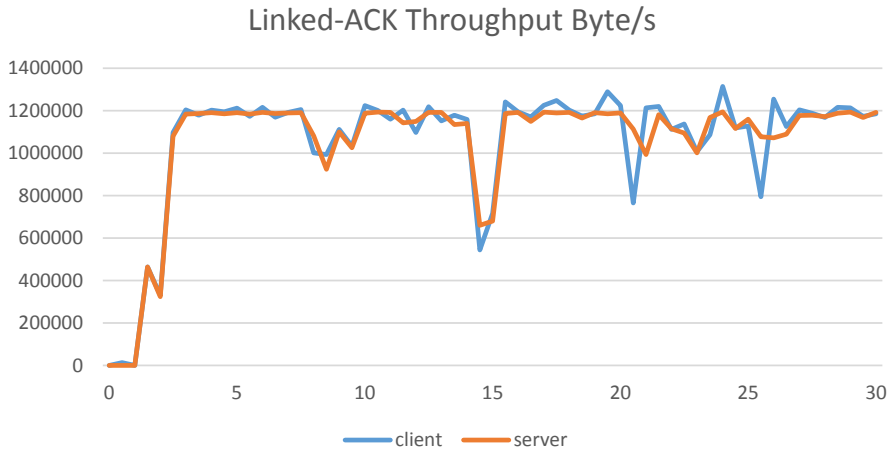

Figure 2.7: Linked-ACK throughput of the client to the join-proxy (show as client) the split-proxy to the server (show as server). These two flows synchronized by Linked-ACK.

2.4.3 Proxy Buffer Analysis

Fig 2.9 shows a time plot of a client's congestion window size and the buffer sizes of join-proxy and split-proxy nodes. The client congestion window size is usually higher than the buffer size of each proxies. By exploiting TCP's flow control mechanism, the proxy nodes can limit the maximum receiver window size to control the sender

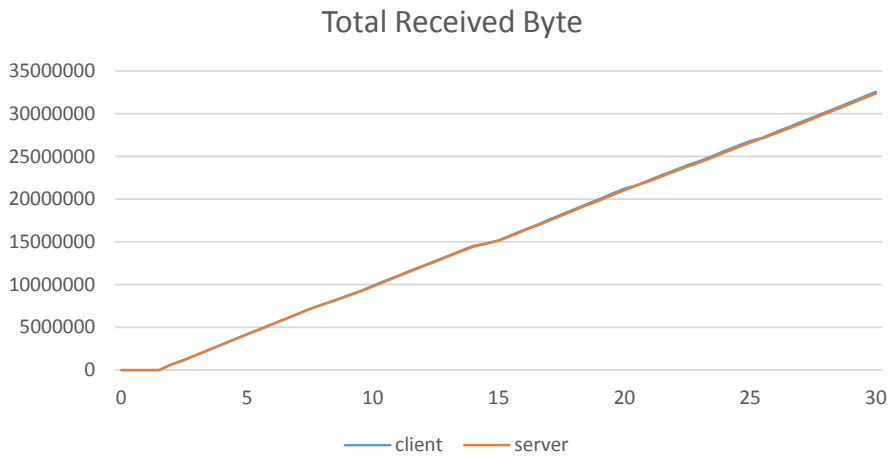

Figure 2.8: Linked-ACK Total received byte of the client to the join-proxy (show as client) the split-proxy to the server (show as server). Two lines are almost overlapped

client's congestion window size. Our framework guarantees zero packet loss on the proxy application layer, when the minimum buffer size of each TCP flow is set to the maximum receiver window size. While the buffer size in the proxy is mostly small for most of the time, it only shoots up when the TCP congestion window size reduces (due to packet loss). Also, the total buffer size of join-proxy and split-proxy is typically not larger than the TCP congestion window size.

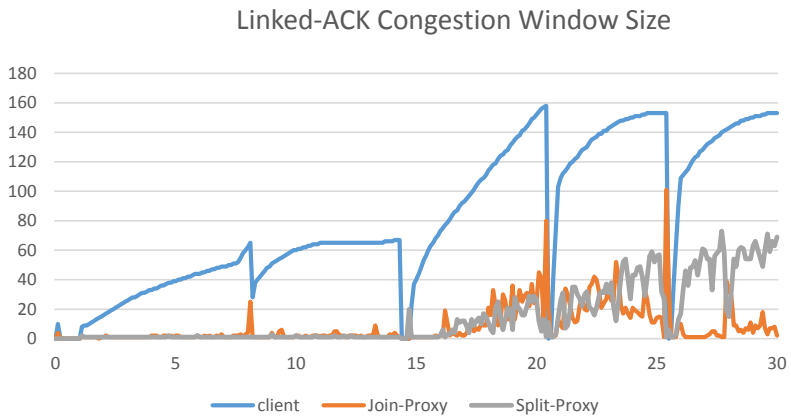

Figure 2.9: Linked-ACK client congestion window size and two proxies queue size. The queue size of each proxy is always smaller than the congestion window size

2.4.4 Fairness Application

In this section, with the help of a weighted round robin scheduler in our flow joining framework we show that a better TCP fairness among different flows can be achieved. However, this application in its native form doesn't preserve the TCP end-to-end semantics, and it also requires unbounded buffer size making the implementation less practical. We hence integrated this fairness framework with our linked-ACK to solve the unbounded buffer size problem and preserve end-to-end semantics. Fig 2.10 and Fig 2.11 respectively show the flow of data and ACK segments in our proposed aggregation framework with Weighted Fair Queuing (WFQ). Figure 12 compares the throughput of 3 TCP long flows in the absence of fairness framework. We consider same round trip time (RTT) for all of the flows. Figure 13 shows the throughput of 3 long flows with the integrated fairness application. We use Jain's fairness index to study the throughput fairness of flows in our experiments. Figure 2.14 shows the Jains fairness index value of 3 TCP flows of traditional setup in Fig 2.12, and 3 TCP flows with proposed fairness framework integrated setup in Fig 2.13. The Jains fairness index F is shown in Eq 2.1. where xi represents the throughput of flow-'i', n is the total number of flows, and F = 1 stands for complete fairness where each gets an equal share of the bandwidth. The average fairness index of Fig 12 and Fig 13 are 0.8914 and 0.9993. The integrated fairness application provides close to 1 fairness is 12.1% better than original TCP flows.

$$F(x_1, x_2..., x_n) = \frac{(\sum_{i=i}^{n} x_i)^2}{n \sum_{i=i}^{n} x_i^2} \tag{2.1}$$

2.4.5 Wireless Application

In this section, we study the performance of the network in the presence of wirlessly connected clients. In this network, each TCP flow experiences packet loss from both

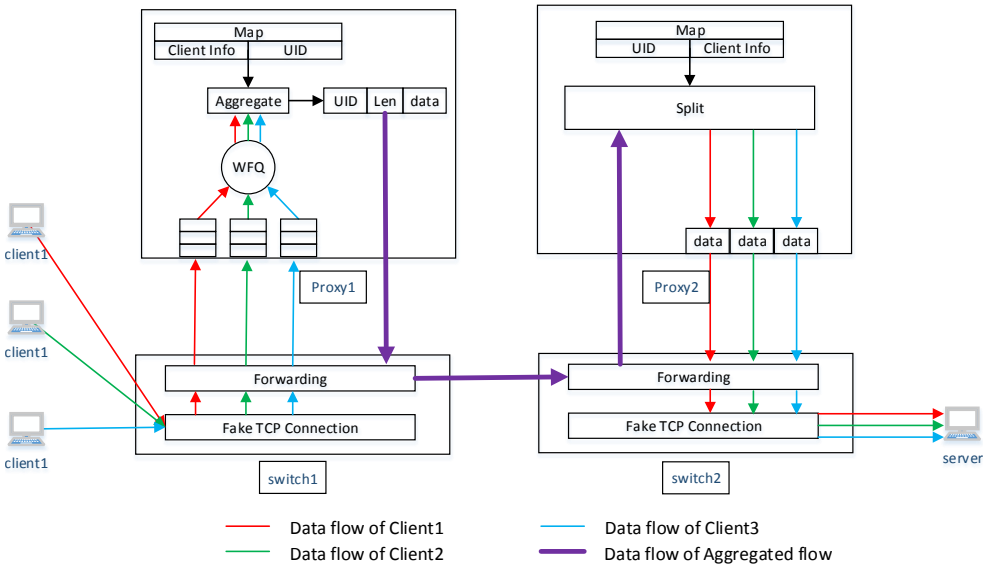

Figure 2.10: Schematic diagram showing the flow of data segments between clients and server in our Proposed network framework with flow aggregation using Weighted Fair Queue (WFQ)

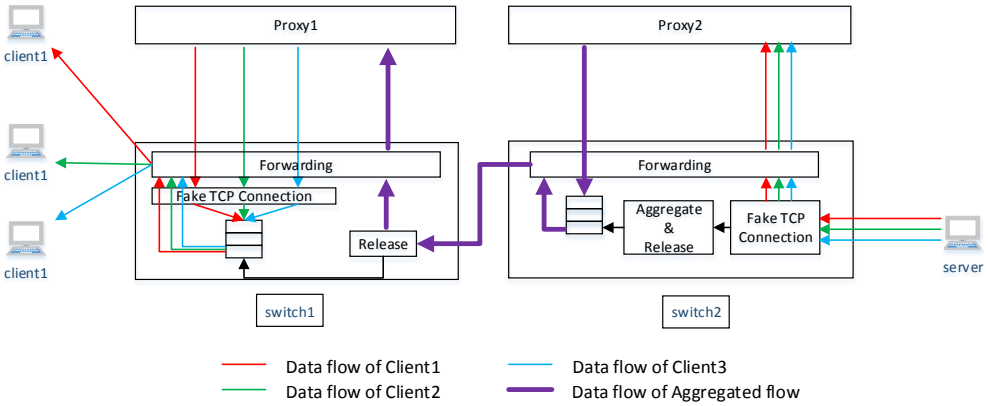

Figure 2.11: Schematic diagram showing the flow of ACK segments between clients and server in our Proposed network framework with flow aggregation using Weighted Fair Queue (WFQ)

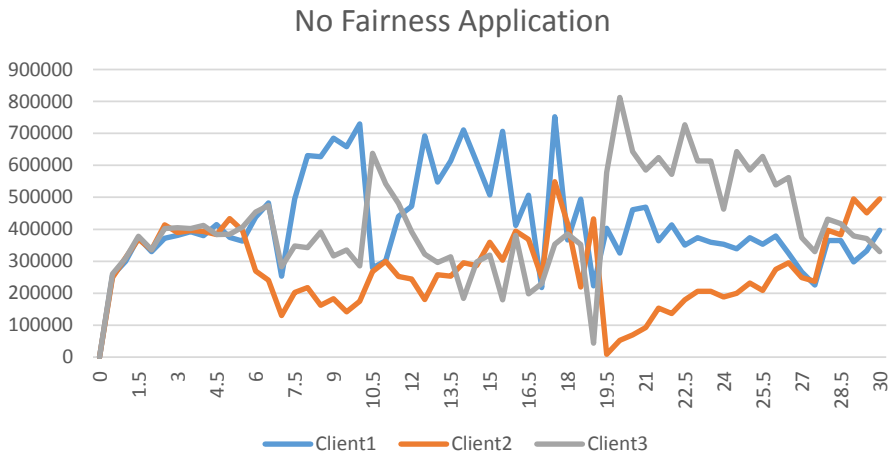

Figure 2.12: Three TCP flows unfair throughput. TCP receive throughput of three TCP flows with TCP CUBIC congestion control algorithm.

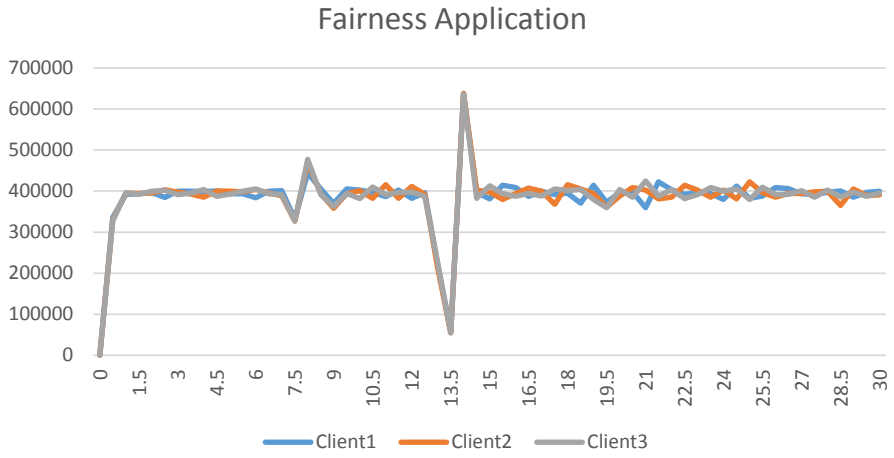

Figure 2.13: Three TCP flows fair throughput. TCP receive throughput three TCP flows with weighted round robin application on proxy

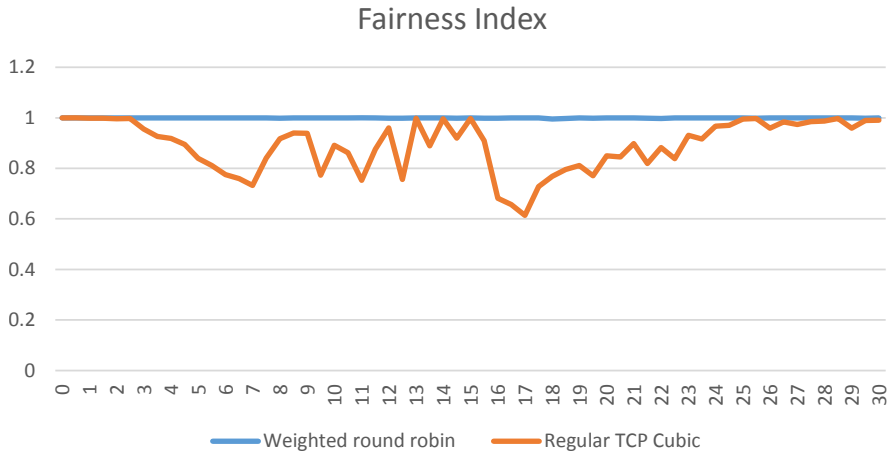

Figure 2.14: Three TCP flows fair throughput. TCP receive throughput three TCP flows with weighted round robin application on proxy

wireless and wired parts of the network. Join proxy node placed at the boundary of wireless and wired network protects the influential factors of the wireless TCP from not being carried over to the wired part of the network, vice versa. In our simulation, the wireless network link loss rate is set to $p1 = 1 cause a packet loss rate of p2 = 1$

The closed-form expression [59] of TCP throughput T is shown in Eq 2.2:

$$T = \frac{MSS \times C}{RTT \times \sqrt{p}} \tag{2.2}$$

where MSS is the Maximum Segment Size, C is a constant value, and p is the link loss rate. The queue loss is factored-out by setting the queue size to a large value. Without the proxy-based framework, the throughput T1 is given by Eq 2.3,

$$T_1 = \frac{MSS \times C}{(t_1 + t_2 + \Delta_1) \times \sqrt{p_1 + (1 - p_1) * p_2}} \approx \frac{MSS \times C}{(t_1 + t_2 + \Delta_1) \times \sqrt{p_1 + p_2}} \tag{2.3}$$

where p1 and p2 are the link loss rates on the respective links of client1-switch1 and switch2-server, t_1 and t_2 are the propagation delays on the respective links of client1-switch1 and switch2-server, and Δ_1 is the total queuing delay along the path from client1-switch1-switch2-server. In our simulation, unless otherwise specified the p_1 and p_2 values are set to 1%, respectively. The propagation delays t_1 and t_2 are set to 40 ms and 40 ms, respectively.

On the other hand, the TCP throughput with our proposed proxy-based framework is given by Eq 2.4

$$T_2 = \min\left(\frac{MSS \times C}{(t_1 + t_2 + \Delta_2) \times \sqrt{p_1}}, \frac{MSS \times C}{(t_2 + \Delta_3) \times \sqrt{p_2}}\right) \tag{2.4}$$

where Δ_2 is the total queuing delay along the path from client1-switch1- proxy1-switch1-switch2-proxy2-switch2-server, Δ_3 is the total queuing delay along the path from proxy2-switch2-server. As shown in Eq 2.4, T2 is computed as the minimum

throughput from the following two paths: between client and join-proxy node, and split-proxy node to the server. Though the proxy node separates the respective packet-losses from wired and wireless counterparts, the delay is influenced by the entire network. Therefore Eq 2.5,

$$t_1 + t_2 + \Delta_2 > t_2 + \Delta_3 \tag{2.5}$$

The ratio of throughput from the respective frameworks is given in Eq 2.6.

$$\frac{T_1}{T_2} = \frac{(t_1 + t_2 + \Delta_2) \times \sqrt{p_1}}{(t_1 + t_2 + \Delta_1) \times \sqrt{p_1 + p_2}} \tag{2.6}$$

Fig 2.15 shows respective throughput T1 and T2, and their average values were found to be 0.915 Mbps and 1.12 Mbps, respectively. Fig 2.16 shows the respective total delay values of $t1 + t2 + \delta_1$ and $t1 + t2 + \delta_2$. By substituting the values to the throughput ratio T1/T2, the simulation data $\frac{0.915}{1.12} = 0.8169$ matches with the right side is Eq 2.6 which is 0.8145.

Hence the matching TCP throughput model validates our proposed 'linked-ACK' based flow aggregation framework, and also proves that the TCP performance is improved. The extra delay caused by large queue size in proxy shown as Fig 2.16 can be reduced by assigning a smaller maximum receiver congestion window size value on the proxy-side.

2.4.6 MPTCP Application

MPTCP was proposed as an extension to TCP extension in order to enable multipath forwarding. It provides the ability to simultaneously use multiple paths between peers to improve robust data transport and throughput [55]. MPTCP can be feasible on the devices with two or more network interfaces, such as latest smart phones and tablets that come with WiFi and cellular radios. With MPTCP it is also possible that the

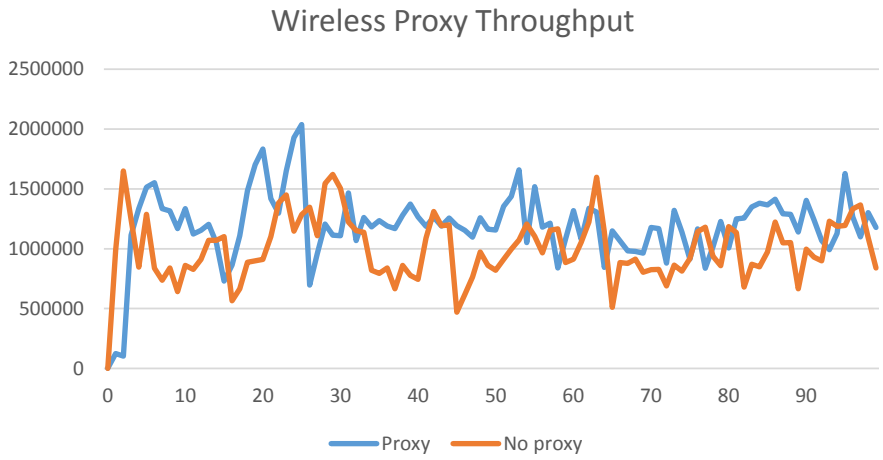

Figure 2.15: TCP throughput performance in a wireless network environment

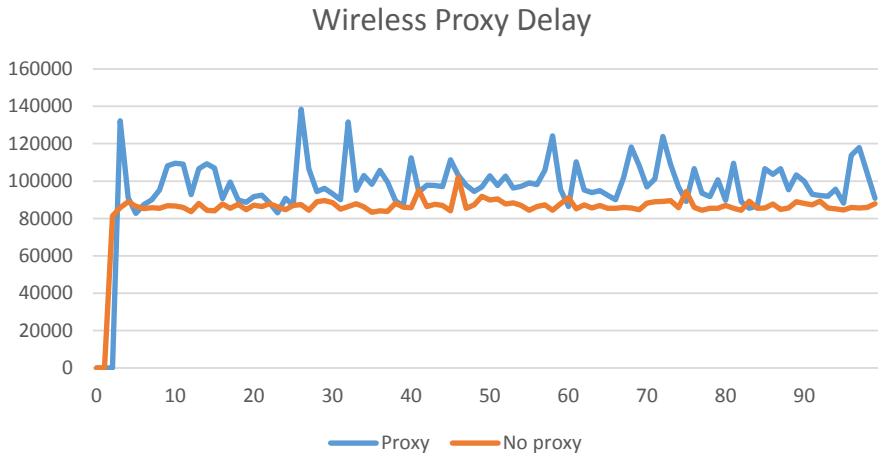

Figure 2.16: TCP delay performance in wireless network environment

different subflows from wireless network can be forwarded to a same path towards the server on the wire network. In our framework, the linked-ACK based join-proxy node is placed at the start of the overlapped path converts MPTCP to be regular TCP. In this setting, MPTCP has to be typically installed on both client and server. By using our 'linked-ACK' based proxy node framework in this setting, we can have multiple benefits. For instance, the server doesn't need to have MPTCP installed as the proxy node aggregates flows into one native TCP flow. In this manner the drawback of MPTCP sub-flows propagating on the overlapped path can be avoided. Moreover, 'linked-ACK' achieves better goodput by requiring less bytes as compared to having 12 byte TCP option in each TCP segment of MPTCP. In the simulation, the topology of MPTCP is slightly different from the one in Fig 2.1. Instead of having one path from client1 to switch1, client1 has two interfaces connecting two non-overlapped path to the switch1. Client1 establishes the MPTCP connection with the join-proxy node, and two subflows transfer data from two different path. Join-proxy only mark the source IP address of the first subflow to the socket. All data received from MPTCP socket are attached with the flow ID of the first subflow. Therefore, the split-proxy establishes one new TCP flow with server by using the first subflow's IP address. The flows between join-proxy and split-proxy, and split-proxy to server run on regular TCP with CUBIC congestion algorithm. Fig 2.17 shows that the throughput received on the server matches well with the total MPTCP throughput performance, and the throughput drop on regular TCP reflects well to all MPTCP subflows.

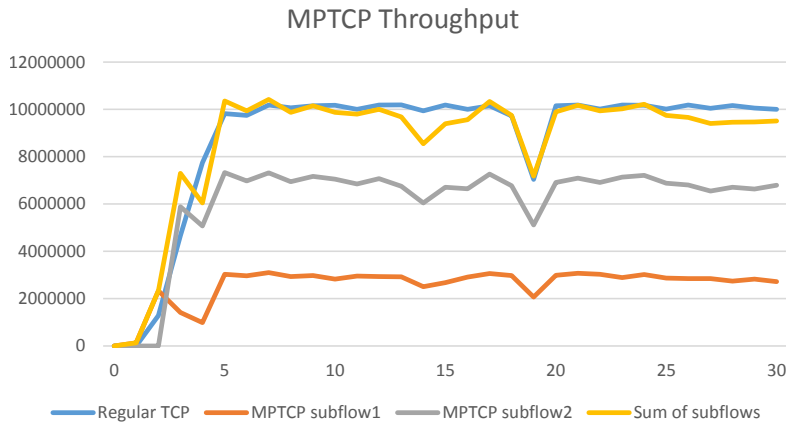

Figure 2.17: Multipath TCP throughput with proxy. Two subflows of MPTCP successfully convert to conventional TCP. The Linked-ACK preserved the end-to-end semantics and synchronized well with MPTCP subflows

2.5 Summary

In this work, we have proposed and implemented a generic join-and split SDN framework of aggregating and splitting TCP flows, with 'linked-ACK' mechanism to preserve end-to-end semantics. The framework developed is implemented in an user-agnostic manner so as to make it more practical. With extensive simulation experiments, we have demonstrated the efficacy of our proposed framework. We have showed the following benefits as achieved by our proposed framework: i) achieves an improved TCP goodput performance, ii) improved buffer usage at the respective split and join nodes, iii) provides fairness among different client flows, iv) improved wireless network throughput, and v) integrates MPTCP based proxy node which provides a hybrid implementation of supporting MPTCP nodes to traditional TCP flows. Despite that our framework also improves goodput performance in MPTCP environment. In future, we plan to extend this work to support application-specific optimization by exploiting our flow aggregation and splitting framework.

Chapter 3

Achieving Throughput Fairness in Smart Grid Using SDN-Based Flow Aggregation and Scheduling

3.1 Introduction

Data communication is the key enabler in smart grid networks. The deployment of communication paradigm in the power domain yields benefits to all participants in the system such as utility companies, governments, and consumers. Typical smart grid network spans a vast geographical area connecting many devices such as Smart Meters (SMs). The purpose of SMs is to enable continuous monitoring and better utilization of resources at the customer-end users (*i.e.*, the electricity consumers). Some of the benefits include automatic billing, load balancing, remote connect/disconnect [26]. The latest SMs are advanced with processing capabilities, and are integrated with full network transport suite such as TCP or UDP.

An essential requirement of SM-based communication includes the reporting of meter-reading information to the end-server (for performing fault-detection, and load shedding), in a *timely* manner. From the networking context, this requires the communication infrastructure to provide a homogeneous delivery rate to all SMs. There are several factors a communication infrastructure needs to consider in order to enable all SMs to deliver data with a fair throughput performance. One factor includes heterogeneous propagation delay. Typical in smart grid environments, the data from SMs in different geographical region would experience different propagation delays before reaching the end-server.

Latest research encourages the communication of SMs over wireless networks such as cellular Long-Term Evolution (LTE) [26], [60]. The LTE-based communication has its own benefits such as geographically wide coverage range (in the order of few kms), which is an essential design requirement for the ubiquitously deployed SMs. In addition to large coverage range, LTE also supports high data rates. From the base-station, the SMs' data is transported over wired network to the end-server. Much of the existing studies on throughput frameworks in smart grid communications considers only the wired infrastructure [28]. In order to be future-proof, it is vital to consider both LTE and wired communication substrates in the infrastructure providing the smart grid services.

Even in the wired smart grid networks, the conventional way of enabling end-to-end transport between each SMs and the end-server causes throughput degradation [27], [61]. Because a large number of short end-to-end TCP flows form a chaotic performance in the network causing more retransmissions and connection failures. As the individual TCP flows do not get sufficient time to sense the present congestion condition in the network, they fail to tune their sending rates, thereby severely degrading the delivery throughput performance. To combat such scenario, recent works [28], [29] have advocated the use of aggregate-points in the network.

The aggregator nodes (in the network) function in the form of an application layer on top of the regular TCP transport protocol suite. This application-layer combines several short TCP connections (mice flows) into a single long TCP flow (also known as 'Elephant flow'). The TCP sender of the long flow can effectively sense the congestion in the network, and thereby adapts the sending rate appropriately. In this manner, the associated clients such as SMs can benefit from an improved throughput performance. While such aggregation point based approaches only help in improving the throughput performance, we argue in this dissertation that such frameworks would not provide fairness to individual SMs, which is an important requirement in smart

grid communications.

For illustration, let us consider a typical urban scenario, with end-server located at the city center that is connected to SMs distributed across a vast geographical area covering the outskirts of the city. A group of (spatially close) SMs are connected wirelessly over LTE base stations, called eNodeBs (eNBs). Each region has multiple eNBs deployed, and the number of eNBs are based on the number SMs to be covered in an area (for example, city center sees a denser SMs (so more eNBs) than in the outskirts). Thanks to the advancement in LTE allowing cloudlet-based computing frameworks such as SMORE [62], we can perform aggregation of SM flows right at the respective eNBs. Subsequently, aggregation can be done at the joining points of multiple long flows in the wired network part (as observed in [28]).

The resulting communication framework will have a number of aggregation points in tandem, along the path from SMs to the end-server. The individual long TCP flows between aggregation points sense the congestion at the frequency of received ACK messages, and subsequently control the sending rates at the clock of Round-Trip Times (RTTs), i.e., the time between a data sent and its ack received. The RTTs are comprised of two factors of delay, namely link propagation delays, and link queuing delays. To avoid bufferbloat scenarios, the network engineers recommend the use of small queue size at the network routers. Therefore it is practical to assume negligible queueing delay in the network with small queues. In the case of smart grid networks, with the SMs typically distributed in a geographical distance (in the orders of tens to hundreds of kilometers), the propagation delay becomes a predominant factor that causes heterogeneous RTTs, for different aggregation points. This becomes the root cause for the unfairness among SMs across different regions.

To the best of our knowledge, we are the first to propose a framework to enable fairness to SMs (insensitive to their geographical distance). Unlike existing approaches our model comprehensively captures both the wireless and wired scenarios of a typical

smart grid. The efficacy of our proposed framework is demonstrated by the fact that fairness is achieved without much loss in throughput performance; typically obtained from an aggregation-only framework.

Fortunately, with the advent of Software-Defined Networks (SDNs) the network becomes more accessible, manageable, and programmable than ever. Using SDN, we address fairness among multiple flows with our novel implementation-based 'Aggregation-and-Scheduling' framework. Unlike existing works such as [28] that proposed the conceptual use of aggregation idea in a simulation environments and demonstrated improved performance; we on the other hand take a practical approach of an implementation-based aggregation-and-scheduling framework, with a white-box design that describes intricacies involved in a real working prototype. In summary, our contributions in this work are as follows:

- We propose a novel SDN-based aggregation-cum-scheduling framework to improve fairness and as well as maintain the improved TCP throughput performance found in traditional aggregation-only frameworks.
- Unlike conceptual idea on simulation, we present a white-box design of the proposed framework by highlighting the implementation functionalities equivalent to developing a working prototype.
- We extensively study the throughput performance and fairness with appropriate analytical model validating the experimental results.

3.2 Smart Grid Network Model

We consider a typical urban scenario smart grid environment, wherein the SMs are connected through LTE User Equipments (UEs) to the wireless LTE eNBs, and the several eNBs are connected to the end-server (located at the city center) via wired

network. We considered the UDP transport from SMs to eNBs, and TCP transport from eNBs to the end-server. The reason for UDP is to fully utilize the wireless bandwidth, and also to have fairness among the first-mile LTE wireless uplinks. In our ns-3 wireless experiments, we have obtained a success rate of receiving 99% packets at wireless links for a wide range of input traffic rates (between 50 packets per second and 500 packets per second), with each packet of 1500 bytes in length. In future, for higher traffic loads, without loss of generality, a reliable UDP protocol can be considered to further improve the success rates. The fairness at the wireless uplinks between multiple UEs and eNBs comes from the following factors: (i) in a smart grid all nodes are static *i.e.*, the UEs, SMs, and eNBs. Static nodes create fixed and same channel conditions across all UEs [63], and (ii) the LTEs default proportional fair MAC scheduler allows a round-robin type of switching among all the backlogged UEs. Our experiments showed accurate fairness among all UEs connected to the same eNBs. The results are not show due to page constraints.

The schematic diagram of the described network is shown in Fig. 3.1. We have numbered different eNBs from 1 to N from left to right (*i.e.*, from outskirts region towards city center to the end-server). We consider multiple SMs close in a colony are connected over wired Ethernet links to a LTE UE, which is in turn connected wirelessly to an eNB. To capture a scenario of denser SMs in city-center, and relatively sparse SMs in the outskirts, we use one UE at the edge of the network and increase the number of UEs towards the server. This naturally captures denser SMs towards the path to the server node. The UDP packets received (from multiple SMs) at eNB are sent via a single TCP sender in the wired-part of the network. As the wireless LTE ensures fairness (through appropriate MAC scheduling and static nodes), we need to provide fairness in the wired-part of the smart grid.

We consider TCP Reno version throughout this dissertation. The TCP reno version is a widely used variant of TCP in the Internet. As we focus on long TCPs,

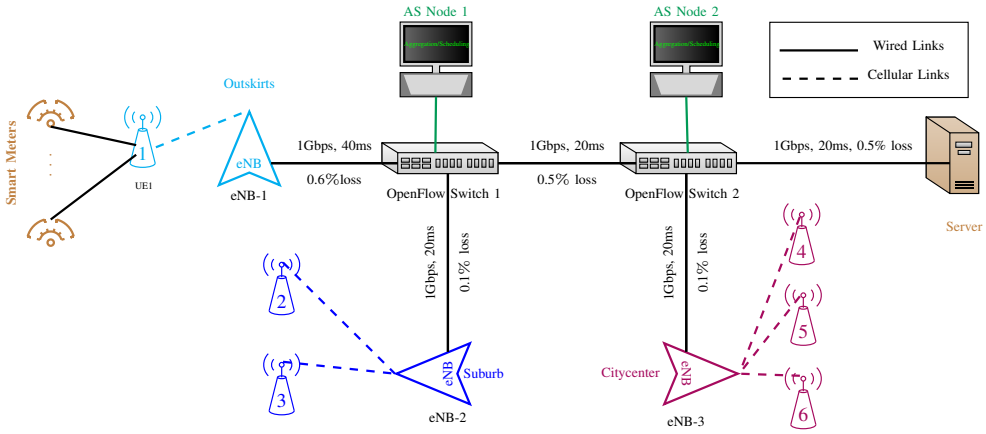

Figure 3.1: Schematic diagram of OpenFlow network with Aggregator-cum-Scheduling (AS) nodes. *For brevity, SDN controller is not shown. Also the smart-meters in suburb, and citycenter regions are not shown.*

we consider the performance effects comprising the congestion avoidance phase of TCP. The effects of TCP receiver window and network link bandwidth is considered to be sufficient enough to not affect the throughput performance. Without loss of generality, we assume packet loss as the indication of congestion, and we consider a simple identical and independent probability distribution for the packet loss.

As shown in Fig. 3.1, we have two eNBs connected to first switch, where these two long TCP flows are aggregated into one composite long TCP flow through the Aggregation/Scheduler (AS) node 1. eNB-3 joins switch 2 wherein it is mixed with the incoming aggregated flow at AS node 2. This represents a typical scenario of many flows getting aggregated towards the proximity of the server, eventually making the server's first hop as the bottleneck link. In the next section, we describe the proposed aggregation/scheduling framework and its implementation.

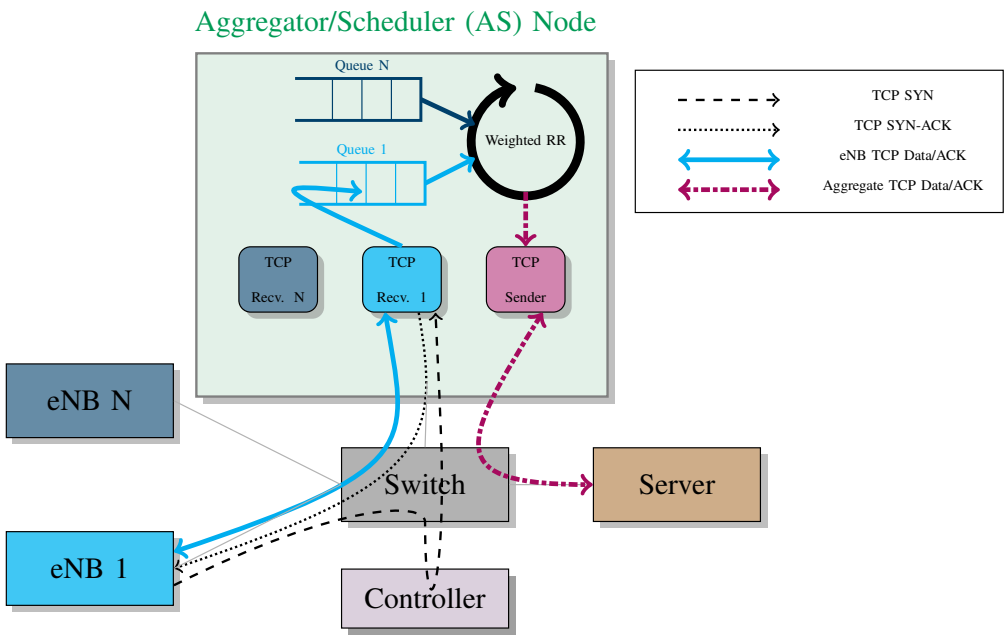

Figure 3.2: Internal functionality of the proposed framework with single aggregator/scheduler node. *For brevity, only the connection-sequence related to eNB-1 is shown.*

3.3 Proposed SDN-Based Aggregator/Scheduler Framework

Figure 3.2 depicts the whitebox functional description of the Aggregator/Scheduler (AS) node and the associated SDN framework . For brevity, we depict the model with one AS node. The AS node implementation design is conceived with following goal:

- **Transparent Aggregation and Scheduling:** To enable the framework to be market-ready, the aggregation and scheduling of flows need to be performed in a seamless manner, without explicit modifications at the end-hosts, namely the UEs and the servers.

To achieve aforementioned design goal, we believe the SDN [2] would be an ideal choice based on the offered features such as: *(i)* centralized controller logic, and decoupled data and control planes for better manageability, and *(ii)* flow-based programmable control logic for enabling flexible network services. As seen in the Fig 3.2,

the (SDN) controller primarily connects with all the SDN-based OpenFlow switches in the network. The controller manages the control plane of the network to administer flow-level management by writing appropriate flow-entries at the switches.

A switch upon receiving the first packet of the TCP flow (either from the eNB or from the preceeding AS node) will forward it to the controller, for appropriate decision making. In our scenario, the first TCP packet TCP-SYN is forwarded by the switch to the controller. The controller with the available topology information, will compute an appropriate flow entry to direct this flow to the near-by AS node (an end-host). Upon writing this flow-entry, all the future packets of this flow will be diverted (by the switch) to the AS node. To enable the TCP receiver 1 inside the AS node to further receive and process the incoming segments, an additional flow-entry is programmed at the switch (by the controller). This additional entry enables the switch to modify/rewrite the destination-field of the TCP header for the eNB-1's flow with AS node information. A similar entry, is made for return traffic carrying ACK packets from TCP receiver in AS node. Therefore, the eNB is abstracted from this rerouting logic to the AS node.

Each TCP receiver is associated with an independent queue for buffering the incoming packets. Without loss of generality, we consider this buffer to be sufficient enough to store incoming packets. To enforce fairness, the scheduler should know the information necessary for distinguishing the incoming composite flows. Though the independent queues serve the purpose of distinguishing different flows (at the link-level); in a tandem aggregator scenario each of these flows could be a composite flow aggregated at the preceding AS nodes. Therefore, the scheduler should be made aware of the appropriate scheduling information necessary for ensuring fairness. Thanks to the SDN's centralized controller logic that receives first packet of all the participating flows in the network. The controller therefore maintains the global knowledge of each flows, and their corresponding first aggregation points (*i.e.*, AS nodes).

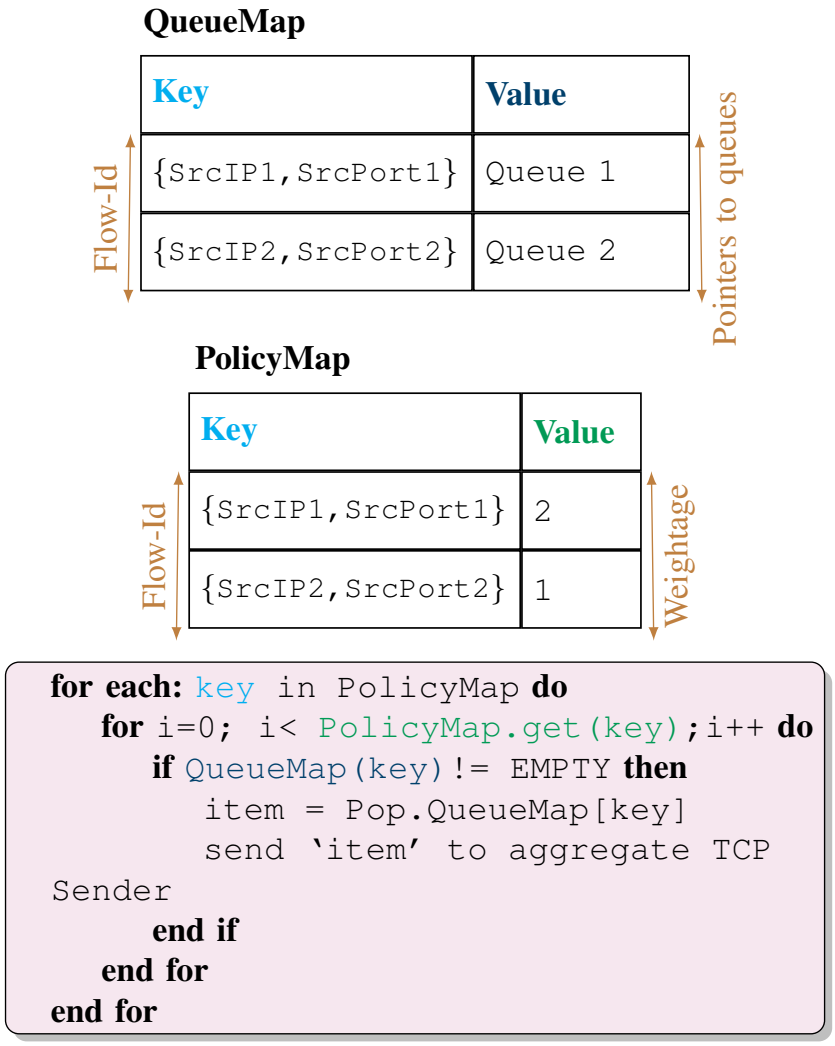

Figure 3.3: Weighted round robin scheduling and aggregation implementation at the AS node

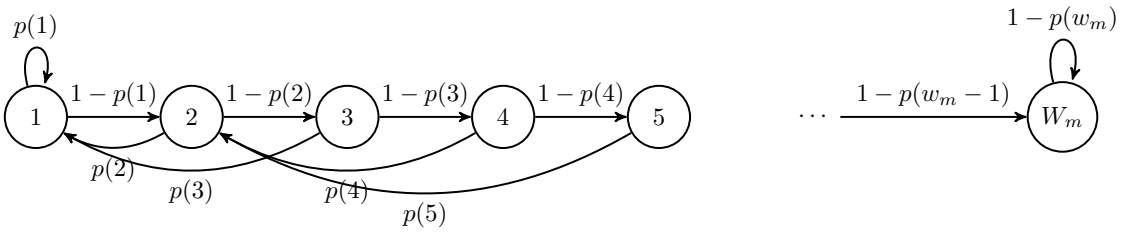

Figure 3.4: Markov Chain Model Long TCP Congestion Window Size Evolution in CA phase. Where $p(i)$ is the loss probability with congestion window size of 'i' packets.

Each AS node maintains two 'key-value' lookup data-structures namely, QueueMap and PolicyMap as shown in Fig. 3.3. The QueueMap is used by the scheduler to locate the respective queues of the different flows. The PolicyMap is the important data-structure that stores the weightage for each (aggregated) flows indicating the number of actual flows it is composed of or carrying with it. As explained earlier, the controller provides the weightage information for the AS nodes to maintain in their PolicyMaps. The underlying logic for enabling fairness is made possible with a simple Weighted Round Robin Scheduling at the AS nodes, and with the global information support from SDN. The AS nodes aggregate traffic received from each participating eNBs (or preceeding AS nodes), and the TCP senders of the respective AS nodes react only to the associated TCP receiver. Each TCP sender therefore adjusts its sending rate which is homogeneous to all the packets, thereby solving the different RTT scenario observed in a typical non-aggregated network scenario. Therefore the simple yet effective implementation of aggregation-cum-scheduling can provide almost perfect fairness for the participating TCP flows. The underlying aggregation and weighted round robin scheduling logic of presented as 'pseudo-code' in Fig. 3.3.

We now study the throughput performance through analytical investigation and measure the fairness by using the popular Jain's Fairness Index [13]. All our experiments are performed in a Mininet network emulator environment [64]; which is a

realistic virtual network running real kernel and switch functionalities. Therefore, with minimal changes our frameworks can be deployable on a real working prototype. The TCP Reno's congestion window evolution for the long TCP flow is analytically modeled by authors in [65], and studied the statistical performance of the evolution around mean. Fig. 3.4 shows the TCP Reno congestion window evolution in the Congestion Avoidance (CA) phase. Let the congestion window size W be represented in packets with values ranging from 1 to w_m. The value of W evolves at each RTT and is controlled by the loss of packets in the network. The TCP behavior in CA phase is captured in Fig. 3.4. Upon a packet loss event the congestion window is halved, and on the other hand, a no loss event increases the congestion window size by 1 packet. For analytical tractability, without loss of generality, we assume a Bernoulli link loss model wherein the loss of packets is independent and identically distributed with a probability p. Therefore the loss probability $p(w)$ can be computed as follows: $p(w) = (1 - (1 - p)^w)$.

As observed in Fig. 3.4 the Markov chain is approximately aperiodic and irreducible for practical values of maximum congestion window size; therefore, the chain has a unique steady-state distribution, say π. By Birkhoff's ergodic theorem, the sample mean of congestion window with scale (or sample size N), represented as $\overline{W}^N$ almost-surely converges to mean of the steady-state distribution π, when N grows to ∞ (N is time in our context). Mathematically expressed as follows [65]:

$$\overline{W}^N = \frac{1}{N}\sum_{i=1}^{N} W_i \xrightarrow[N\to\infty]{a.s} \overline{W}^\infty = \sum_{w=1}^{w_m} \pi_w \times w = E[W]. \tag{3.1}$$

Where $E[W]$ is the average congestion window size. In other words, from Eq. 3.1 the scaled mean throughput $\overline{W}^N$ and its difference in value with $\overline{W}^\infty$ tends to 0 as N grows to infinity; and this behavior of difference converging to zero is termed as 'rare event' which can be characterized by large deviations theory.

From the classical results of the large deviations theory, an irreducible and aperiodic Markov chain, with finite state space, holds a large deviations spectrum as given below [65]:

$$\lim_{\epsilon \to 0} \lim_{N \to \infty} \frac{1}{N} \log \mathbf{Pr}(\overline{W}^{(N)} \in [\alpha - \epsilon, \alpha + \epsilon]) = f(\alpha) \tag{3.2}$$

where, $f(\alpha)$ is called the large deviations spectrum. In our context, the large-deviations spectrum $f(\alpha)$ can be computed [66] as the Legendre Fenchel transform of spectral radius'(ρ) logarithm of a matrix $(\mathbf{R}(q))_{ij} = \exp(qj)\mathbf{T}_{ij}$. Where $\mathbf{T}_{ij}$ is the transition matrix underlying the Markov chain of states of TCP congestion window as shown in Fig. 3.4.

$$f(\alpha) = \inf_{q \in \mathbb{R}} (\log \rho(\mathbf{R}(q)) - \alpha q) \tag{3.3}$$

An essential property of large-deviations spectrum is that it is concave and satisfies the following property that if $\alpha = \overline{W}^{\infty}$ then $f(\alpha) = 0$, and for all other values $f(\alpha) < 0$. Therefore, the mean congestion window size E[W] can be computed as the value of α at $f(\alpha) = 0$. Subsequently, the average throughput E[T] in (bps) is computed as follows:

$$E[T] = \frac{E[W]}{RTT} \times MSS. \tag{3.4}$$

where RTT is the round-trip time (in seconds), and MSS is the Maximum TCP Segment Size (in bits). It is worthwhile to note that for the proposed aggregation-cum-scheduled framework, the measured RTTs were almost steady to a fixed value in our experiments, which also a significant factor towards ensuring fairness.

3.4 Performance Evaluation

The LTE-based network is studied in ns-3 [67] simulation environment, and wired-part of the network is studied in Mininet network environment. We integrated ns-3 simulation to Mininet environment in order to get a unified model of the smart grid

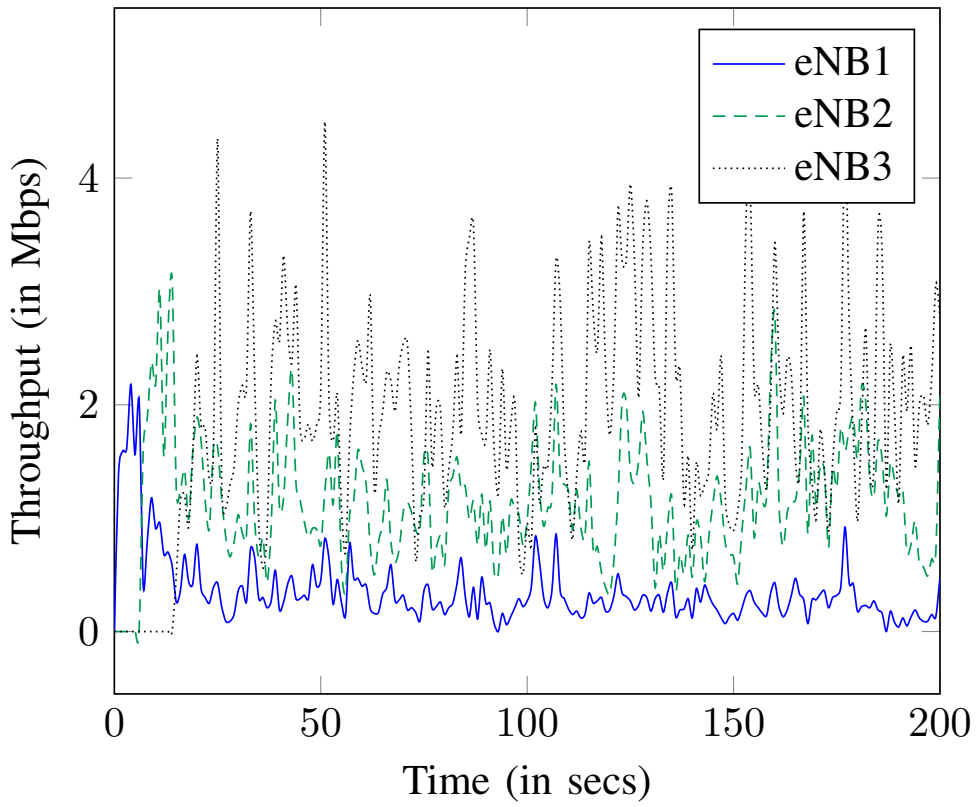

Figure 3.5: eNBs TCP throughput with aggregation and without scheduling.

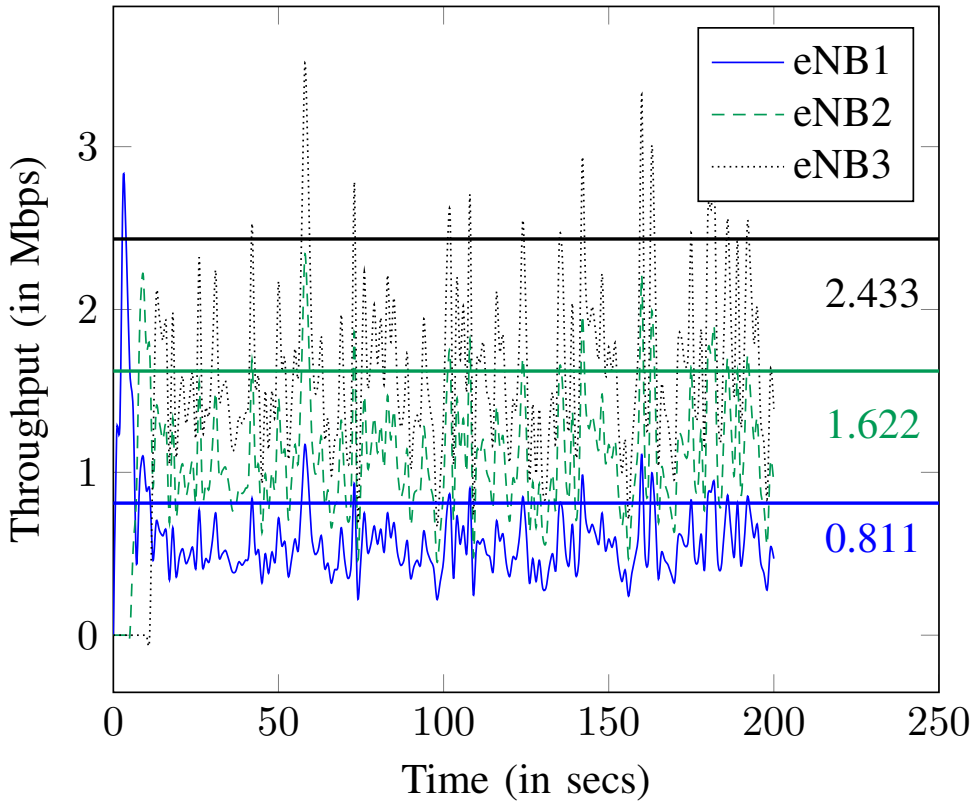

Figure 3.6: eNB TCP throughput in Proposed Aggregation and Scheduling Framework. The *horizontal lines and the respective values show the analytical average throughput, as computed from Eq. 3.4.*

network.

Figure 3.5 shows the individual eNB's throughput performance in the aggregation-only setting. In other words, weighted round robin is not considered. The unfairness among flows are prevalent. This is due to the fact, that total throughput is determined by the 'last ASnode2 and the server' pair, as this forms the bottleneck link. Due to the high throughput in preceding links, the packets get buffered in the queues and are always available to the last AS node to consume. However, different flows from the preceding links experience different RTT the way they feed the queue is heterogeneous thereby propagating the unfairness at the server. It is also noted that different flows complete their transfer at different points in time due to substantial difference in throughput.

Figure 3.6 shows the improved fairness conditions in the system utilizing our proposed aggregation-and-scheduling framework, which is evident from the eNBs throughput being proportional to the number of UEs connected. To understand the accuracy of fairness, we have used Jain's Fairness Index to evaluate the fairness of the three respective network settings, namely no-aggregation, aggregation-only, and aggregation/scheduling. The Jain's Fairness Index [13] is computed as follows:

$$\texttt{Fairness index} = \frac{(\sum_{i=1}^{n} E[T_i])^2}{n \times \sum_{i=1}^{n} E[T_i]^2} \tag{3.5}$$

where $E[T_i]$ throughput of TCP flow i and n is the total number of flows. Figure 4.6 shows the fairness of the system for three different network settings. It is more evident that our proposed framework outperformed both no-aggregation, and aggregation-only setups by exhibiting perfect fairness for all the three heterogeneous RTT eNBs in the network. With the practical operating scenario including heterogeneous RTT flows, and composite multi-level aggregated flows due to tandem aggregators, our proposed framework handled fairness more effectively than other traditional systems.

Figure 3.8 and Fig. 3.9 show the received throughput of independent UEs clearly demonstrating the fairness of the proposed framework. To understand the throughput deviations of all three types of frameworks, Fig. 3.10 shows the total throughput distribution for the three different policies. In addition to achieving fairness, the throughput of proposed aggregation with scheduling framework is still close to the aggregation-only framework.

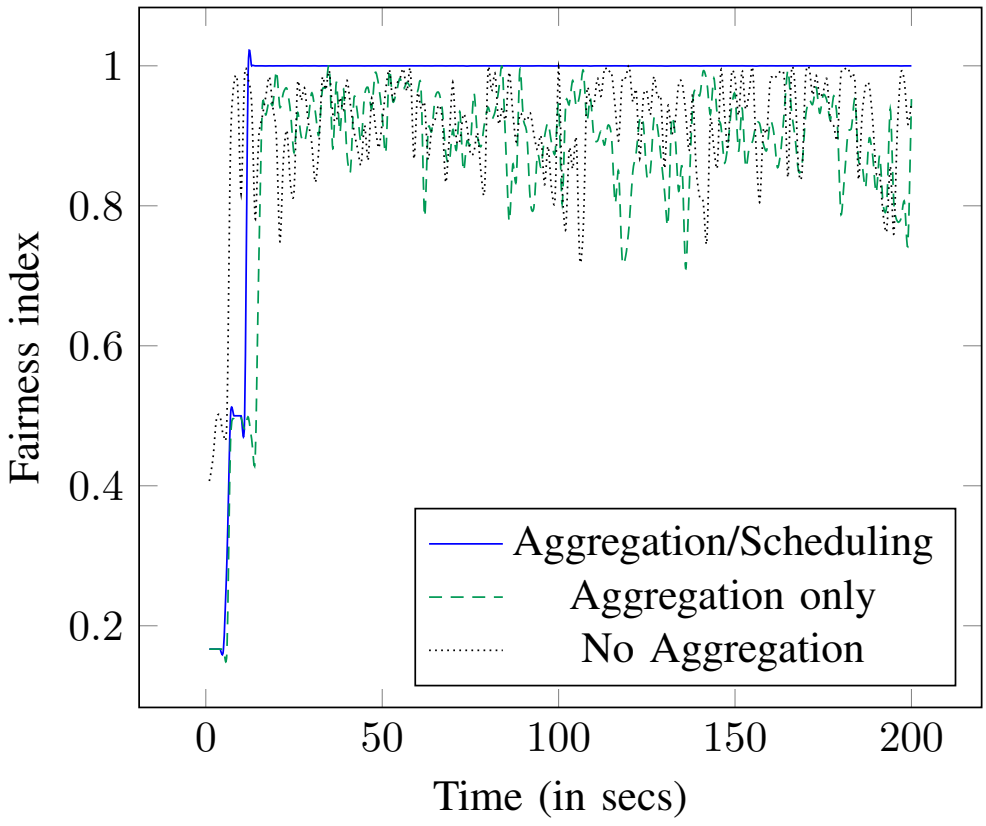

Figure 3.7: Throughput fairness index of 3 different frameworks, computed using Eq. 4.6.

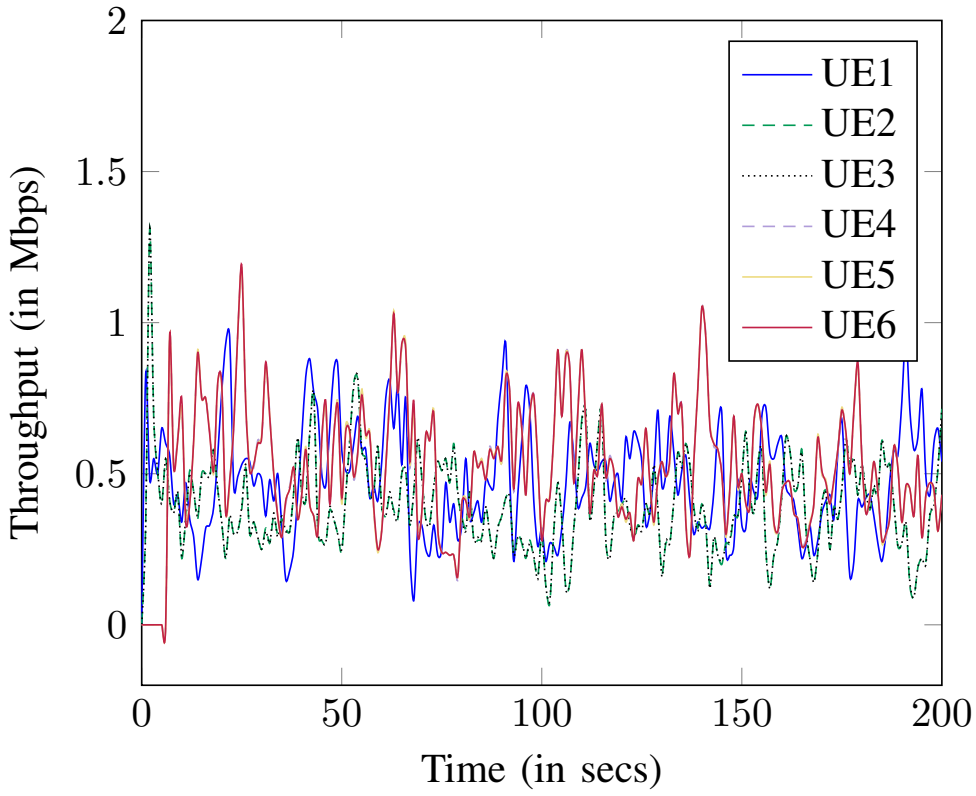

Figure 3.8: Individual UEs throughput in without aggregation and scheduling.

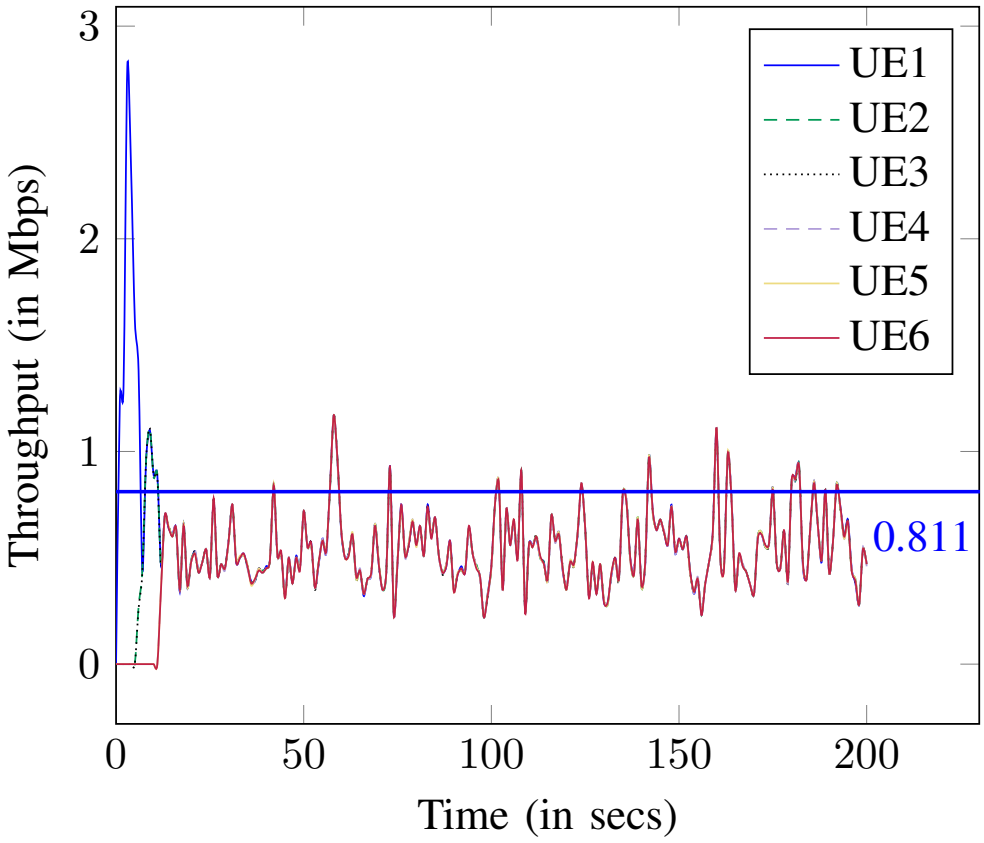

Figure 3.9: Individual UEs throughput in with proposed aggregation and scheduling. *The horizontal line shows the analytical throughput average value.*

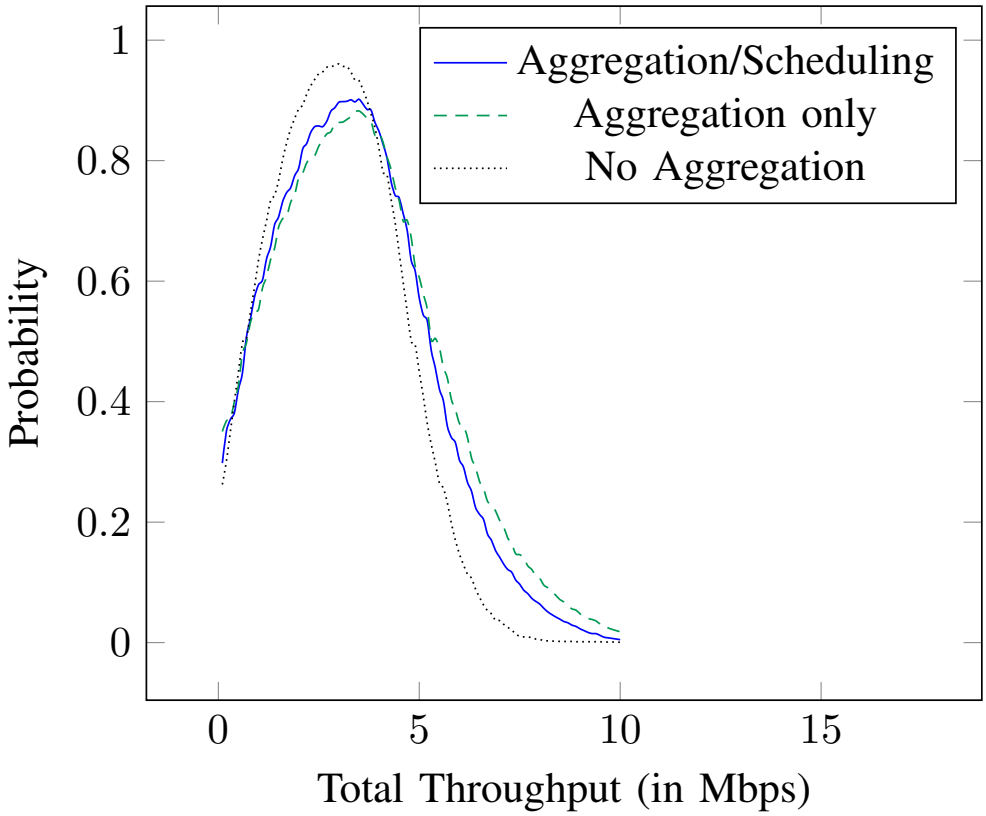

Figure 3.10: Throughput probability distribution for three policies, no aggregation, aggregation without policy, and proposed aggregation with policy

3.5 Summary

We have proposed a novel SDN-based TCP Aggregation/scheduling smart grid framework that achieved a better throughput performance and fairness. We also have proposed the white-box model of implementation with a detailed functional design. The throughput performance is extensively studied and compared with the respective traditional no-aggregation, aggregation with no scheduling, and our proposed aggregation and scheduling frameworks. Our proposed framework demonstrated fairness to significant level of accuracy through out the course of the experiment, while maintaining a high throughput.

In future, we plan to study the WiFi-based SM communications with TCP over wireless links, to achieve throughput fairness.

Chapter 4

Improved Video Throughput and Reduced Gaming Delay in WLAN Through Seamless SDN-Based Traffic Steering

4.1 Introduction

Last decade has witnessed pervasive deployment of WLANs at residential apartments, and public hotspots such as cafeterias. Typical users in these environments use the network for entertainment services such as watching movies, listening songs, and playing games. Recent reports suggest that the dominant data traffic in the Internet is associated with video streaming and gaming applications. The most popular online video content services such as Netflix and YouTube account for 37.1% and 17.9% of Internet traffic in North America [68]. Of late, the game traffic is also getting dominant share of the network traffic. It is worth to note that the global volume of game traffic has 22% of Compounded Annual Growth Rate (CAGR) [31].

The dominant data traffic in such networks requires stringent delivery conditions for better quality of experience. For instance, online video watching requires better as well as consistent streaming throughput for constant playback. The gaming applications on the other hand require low delay. On a local wireless network multiplayer scenario, a small increase in Round-Trip Time (RTT) (of one user) can significantly affect the other users of the game. Because the gaming states of the network multiplayers are strictly synchronized (by a local server), so that every users movements/actions will be constantly updated to all other users. Therefore, a larger RTT experienced by even one gaming-user would negatively impact the playing experience

of other gamers in the network.

It is well-known that the wireless part of the network becomes the performance bottleneck that primarily affects these application-services. Moreover, the composite network-ecosystem comprised of local (game) traffic and exogenous (video) traffic sharing the same WLAN network substrate further exacerbates the bottleneck throughput, as well as increases application delays. Different types of applications share the single AP resources (such as buffer) in a non-uniform manner, which also leads to a larger delay, and reduced throughput. It is therefore, essential to ensure fair use of network resources by these application.

In WLAN networks, unfairness among same-type applications stems from the underlying transport layers used by them. For instance, within same-class applications the fairness is heavily influenced by the underlying transport protocol. For example, the inter-play of 802.11 MAC behavior, and TCP flow/congestion control mechanisms leads to unfairness among multiple TCP uplink flows [36]. In a different work [34], the unfairness among TCP uplink and TCP downlink traffic is observed over WLAN access network. The work observed reduced fairness to TCP downlink traffic over TCP uplink traffic.

In a practical scenario, the WiFi access network sees a mix of UDP and TCP based applications. It is worth to note that major multiplayer (client-server) WLAN gaming applications (such as CounterStrike) support the use of UDP transport. The popular Internet video content providers such as YouTube and Netflix use TCP transport. The UDP traffic alongside the TCPs negatively impacting congestion mechanisms on wireless scenario, affect the performance of both the applications. For instance, video users experience unfair and low TCP downlink throughput, and the gaming users experience high latency.

In this dissertation, we present an end-host agnostic Software-Defined Network (SDN) [2] based traffic steering and control approach to provide effective network

performance. We consider a composite practical network comprised of applications using heterogeneous transport protocols, namely TCP downlink (for video), local UDP uplink, and local UDP downlink traffic (for WLAN gaming). To ensure the fairness to the penalized downlink TCP traffic in the considered network, we logically suppress the TCP's congestion behavior *only* in the wireless part of the network, and subsequently provide normal TCP behavior for the same flow in the wired part of the network. By logically suppressing TCPs reactive behavior over WiFi part, the inbuilt fairness feature of the 802.11's DCF will play its role. In such a system, every downlink application will benefit from the station-level fairness as offered by the WiFi MAC protocol.

Thanks to the programmatic features offered by SDN, the congestion-independent wireless TCP and congestion-based wired TCP for a single traffic flow can be made possible by realizing a split TCP approach. Through the SDN's powerful features of flexible flow management, and deep-packet inspection, we have implemented the TCP splitting near the WiFi AP, and also provided a congestion-free wireless TCP downlink to the users. Moreover, such flow-management and split-TCP features are agnostic to the end-users; which is an important design requirement for the considered application scenario. Our contributions in this dissertation are as follows:

- We provide an SDN-based solution of TCP-splitting along with partially-controlled wireless sending rate to ensure fair throughput. The fair use of TCP downlink resources allow the UDP game traffic to effectively utilize the residual shared resources such as AP buffer, which enables them to achieve reduced delay.

- We present an extensive performance study on the real testbeds to demonstrate the prowess of our approach. Our implementation is production ready, as it is tested on the off-the-shelf network components.

4.2 System Model

The WiFi AP is hosted on a OpenFlow switch that serves the wireless clients. An SDN controller is wire connected to the AP. A video server (end-host) is also wire connected to the AP, which is used by each of the wireless clients through TCP transport protocol. A local WLAN multiplayer game session is created among different wireless clients. The multiplayer game uses UDP transport, and one of the clients acts as game server, that serves gaming clients running on the other wireless hosts. Reflecting the practical network scenario, the wired links are set with higher bandwidth so that the wireless network forms the bottleneck. The video server sends continuous traffic with sufficient data to saturate the wireless bandwidth in the network.

In the aforementioned system, we consider three wireless clients, namely C1, C2, and C3. The client C1 functions as the gaming server. It is worth to note that the UDP server client C1 sees more UDP traffic than other clients. The network system model is shown in Fig. 4.1.

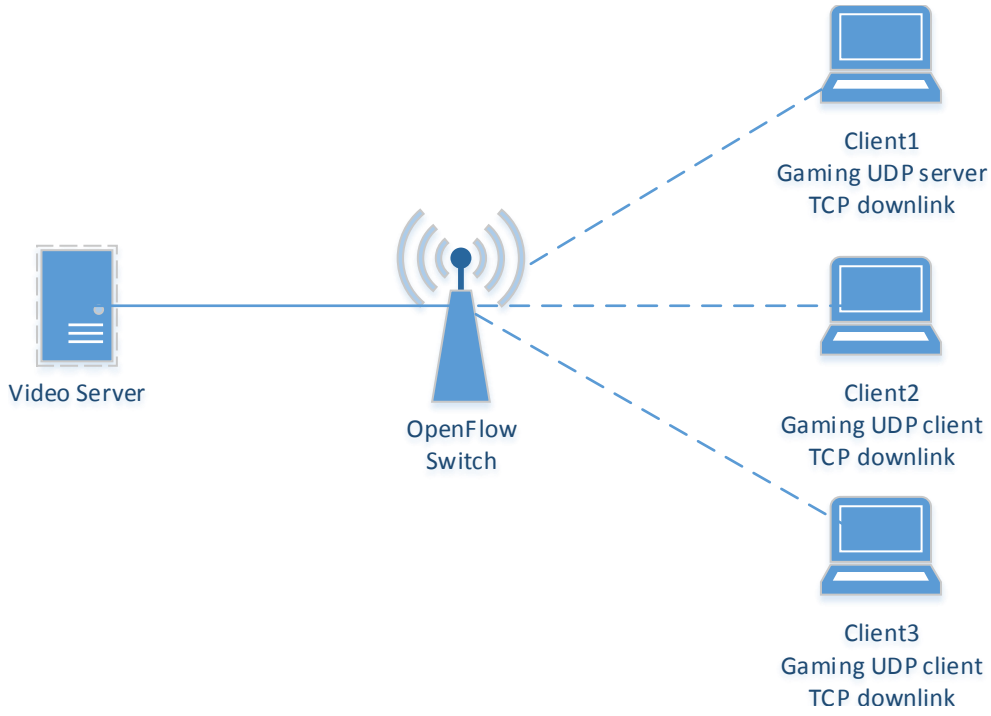

Figure 4.1: WLAN System Network Model. *Solid line indicates wired connection, and dotted lines indicate WiFi wireless connection.*

4.3 Proposed SDN Fairness Model

A new end-host is wire connected to the WiFi AP, which acts as a proxy node that is programmed to enable split TCP to the video downlink flow. The proxy in addition caches the received data from (the relatively high-speed) wired TCP connection. Splitting the TCP helps in enhancing the throughput by separating wireless and wired counterparts. As the TCP throughput is inversely proportional to the Round Trip Time (RTT) which is the function of propagation delay, unlike the traditional end-to-end delay non-split TCP, our proposed framework's split TCP throughput is determined only by the wireless propagation delay, thereby yields higher throughput.

As a second part of the proposed framework, the wireless TCP split connection has to be made insensitive to its congestion control behavior. To ensure that, we set the TCP senders at the proxy node (not at the application end-hosts) to limit their

TCP congestion window size set to 2 segments. With small congestion window size the TCP will send roughly one or two segments per wireless RTT. Therefore, the lock-step fashion of DATA-ACK TCP traffic, enables uniform packet distribution at each of the clients and AP queues. Subsequently, the LAN gaming's client-server configuration, uses the lock-step fashion of Server UDP datagram-Client UDP datagram alternating patterns. The lock-step fashion of each applications traffic along with the WiFi MAC's Distributed Coordination Function's (DCF) station-level fairness ensures fairness among all packets in the buffer (invariable to TCP, UDP packets). In this manner, we ensure fairness among all TCP applications and local UDP applications.

To achieve the aforementioned solution in an application end-user agnostic manner, we utilize flow-management and packet inspection features of the SDN framework. The AP deployed on an SDN based OpenFlow switch provides the functionality of inspecting flows, and routing them dynamically to the proxy machine (a host machine) that provides the split-TCP functionality. With SDN's flow rerouting and packet inspection features we *seamlessly* route the incoming TCP traffic to the proxy machine, which splits the TCP of the wireless part from the wired part. The split TCP segments are appropriately header-modified at the AP openflow switch, so that both end-to-end application hosts receive the segments as if they have received it from original end-user (without proxy). The gaming UDP traffic is routed as usual in a traditional manner (through AP).

SDN provides support for proactive and reactive ways of routing flows. Proactive flow populates flow tables ahead of the traffic coming to the switch. On the other hand, the reactive flows handle the incoming flows on the fly. The first packet of each flow is forwarded to the SDN controller, which inspects the packet and inserts appropriate flow-entry into the AP OpenFlow switch. As the video traffic can come from any end-user, we use reactive flows to configure (or) route the flows. During this

process, we reroute to the proxy host, split the TCP and send back from proxy to the AP, from which it routed to the application end-hosts. This provides transparency to the system. Consider the dynamic routing features, the UDP gaming application traffic is also routed using the reactive flow configuration method.

4.3.1 Network Flow in the Proposed Framework

Figure 4.2 shows the architecture of the proposed framework, as discussed in the previous section. The flowing sequence of network flows happen in the system as follows:

1. The wireless client starts the TCP session, and sends out TCP SYN packet. The first packet of each new flow is sent to the SDN controller. The controller upon inspecting the received packet will insert flow tables for wireless to proxy, and proxy to wireless client (for return traffic). These flow tables hide the proxy node by appropriately manipulating the Layer2 and Layer3 header fields.

2. The proxy starts a new TCP connection to the wired video server. The first packet in Step 1 triggers the current step's action, as the proxy does not know which video server to connect. The controller sets up flow tables so that the video server date can reach the wireless client.

3. When gaming server starts a WLAN multiplayer session, the controller routes the flows to the UDP gaming server, which is a wireless client.

4. The controller also routes the gaming server traffic to the appropriate gaming wireless clients.

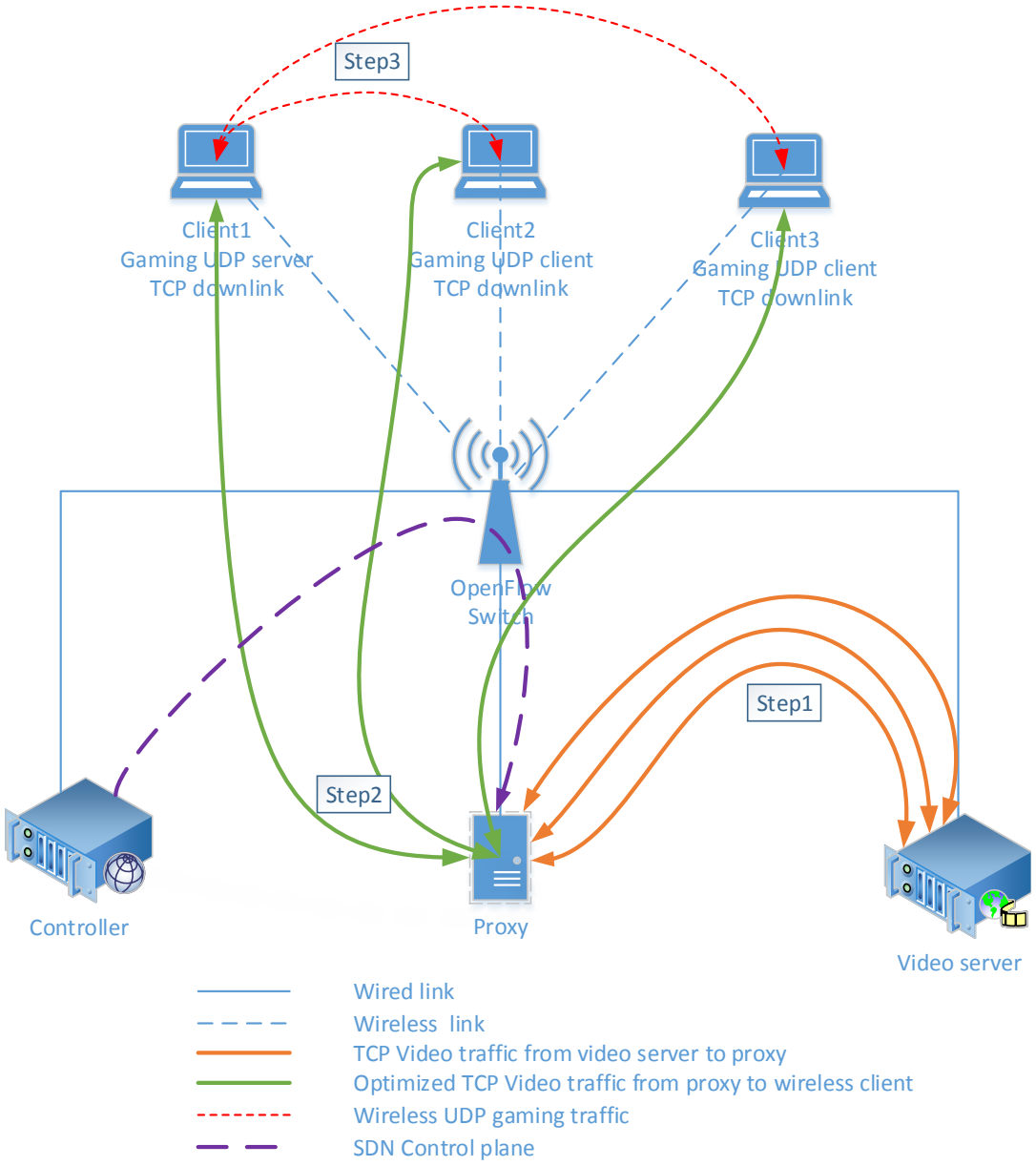

Figure 4.2: Proposed SDN based WLAN Network Architecture.

4.4 Delay and Throughput Analysis of the Proposed Framework

The background TCP video and UDP gaming traffic share the same queue. The UDP traffic suffers more delay caused by TCP video traffic occupying the shared AP buffer. Let us assume the AP buffer size can hold 'B' packets, and the AP buffer service rate as S, in Mbps. Let $T_S = 1500$ be the TCP segment size in bytes, and $T_a = 40$ be the TCP ACK packet size in bytes. By considering, the TCP sender and receiver window size to be large enough with immediate ACK, the TCP flow will always try to fill in the buffer and eventually leads to packet loss. Let us assume that the total number of TCP flows in the network as N. The maximum congestion window size of each TCP flow be CW. For each of the game clients, let U_p be the number of the (game) UDP packets cached in the queue, and U_b be the total UDP cached traffic in bits.

The following condition needs to be satisfied, to have fair TCP throughput among the flows, which is achieved through the MAC layers DCF functionality:

$$2 \times N \times CW + U_p \times (N - 1) < B \tag{4.1}$$

It is worth to note that the fairness also ensures low delay for each TCP traffic. The available share of buffer for TCP flows can be computed as given below:

$$B - U_p \times (N - 1) \tag{4.2}$$

To achieve fairness without buffer overflow, the following condition needs to be satisfied, with subject to a small congestion window (namely C) for each of the flows:

$$B - U_p \times (N - 1) - (2 \times N \times C) > 0 \tag{4.3}$$

The delay of each flow can be approximated by only considering the queuing delay of the AP. All other factors causing delay are small enough to be ignored in our study. According to the Little's law, the TCP Round Trip Time (RTT) can be computed as follows:

$$RTT = \frac{(N-1) \times U_b + 8N \times C \times T_a + 8N \times C \times T_s}{S} \tag{4.4}$$

To satisfy the constraint in Eq. 4.3, and also to achieve a minimum delay as in Eq. 4.4, the value of C has to be as small as possible. We therefore choose C to be 2. Therefore, the at most $2C$ packets are cached in the buffer. Under this condition, there is no buffer overflow, and the TCP clients achieve fairness and low delay.

The throughput of TCP is computed as follows:

$$\frac{C \times MSS}{RTT} = \frac{C \times MSS \times S}{(N-1) \times U_b + 8N \times C \times T_a + 8N \times C \times T_s}, \tag{4.5}$$

where MSS represents the Maximum TCP Segment Size.

4.5 Performance Evaluation Study

Figure 4.3 shows the experiment testbed used for our performance study. The OpenFlow Switch is a MikroTik wireless router with 802.11b radio, is running on OpenWrt. The Open vSwitch is installed to support OpenFlow protocol implementation. Floodlight is the open-source SDN controller that is used in our experiments for the OpenFlow protocol functioning. Initially, the switch will broadcast the ARP packets to locate the controller. Once the controller is found, the controller is connected through the TCP connection. The CounterStrike game is used to perform WLAN multiplayer gaming session. The video traffic starts first, and each gamers join the system one by one. The TCP Reno version is used in the experiments.

Figure 4.4 shows the improvement in gaming delay traffic for the respective SDN-

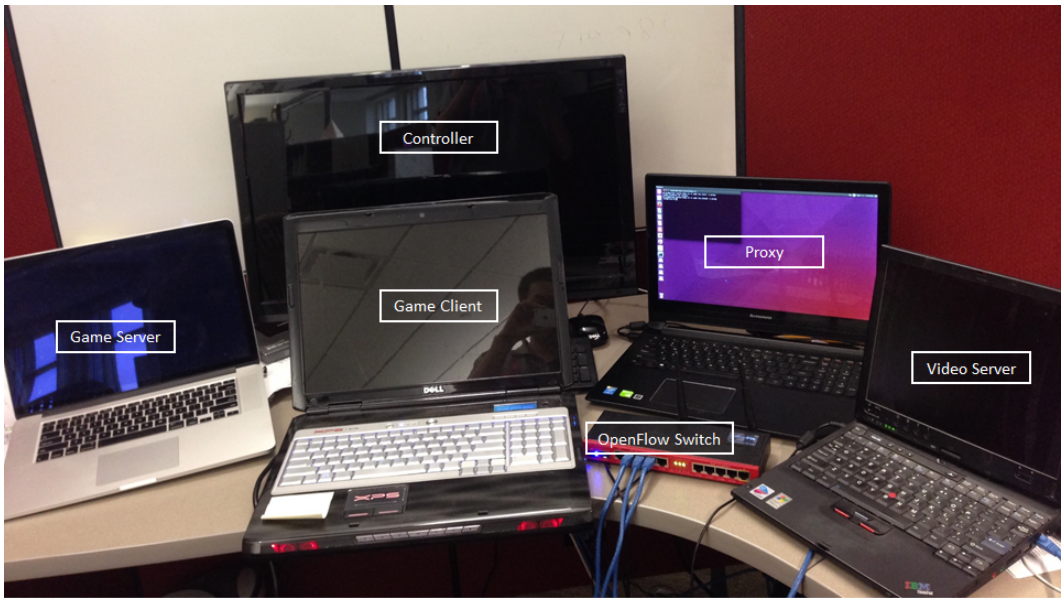

Figure 4.3: Real testbed used for the performance study.

Table 4.1: Average TCP throughput of each clients (in kbps) for the proposed and traditional experiment setup.

	2 Client		3 Clients			4 Clients			
	Client 1	Client 2	Client 1	Client 2	Client 3	Client 1	Client 2	Client 3	Client 4
Proposed	2487.77	2319.96	1458.23	1364.81	1464.58	916.88	912.95	932.82	952.2
Traditional	2167.77	2453.48	618.33	1388.03	1651.00	80.83	1074.03	1263.97	1273.78

based proposed framework, and the traditional setup. The increase in delay is due to the TCP downlink traffic predominantly occupying the AP buffer. With our proposed framework, the TCP traffic is split and wireless counterpart is contained with limited congestion window size with value $C = 2$, therefore, the AP buffer is uniformly occupied by all traffic types, this phenomenon significantly decreases the gaming delay.

Figure 4.5 shows the video traffic delay of the proposed SDN framework with TCP only traffic, and with TCP+UDP traffic. It is evident that the analysis presented in previous section matches well with the experiment results. Though the analysis does not capture gaming UDP traffic, the delay is not substantially different from the TCP-only counter part.

Table 4.1 shows the TCP video throughput for the respective proposed frame-

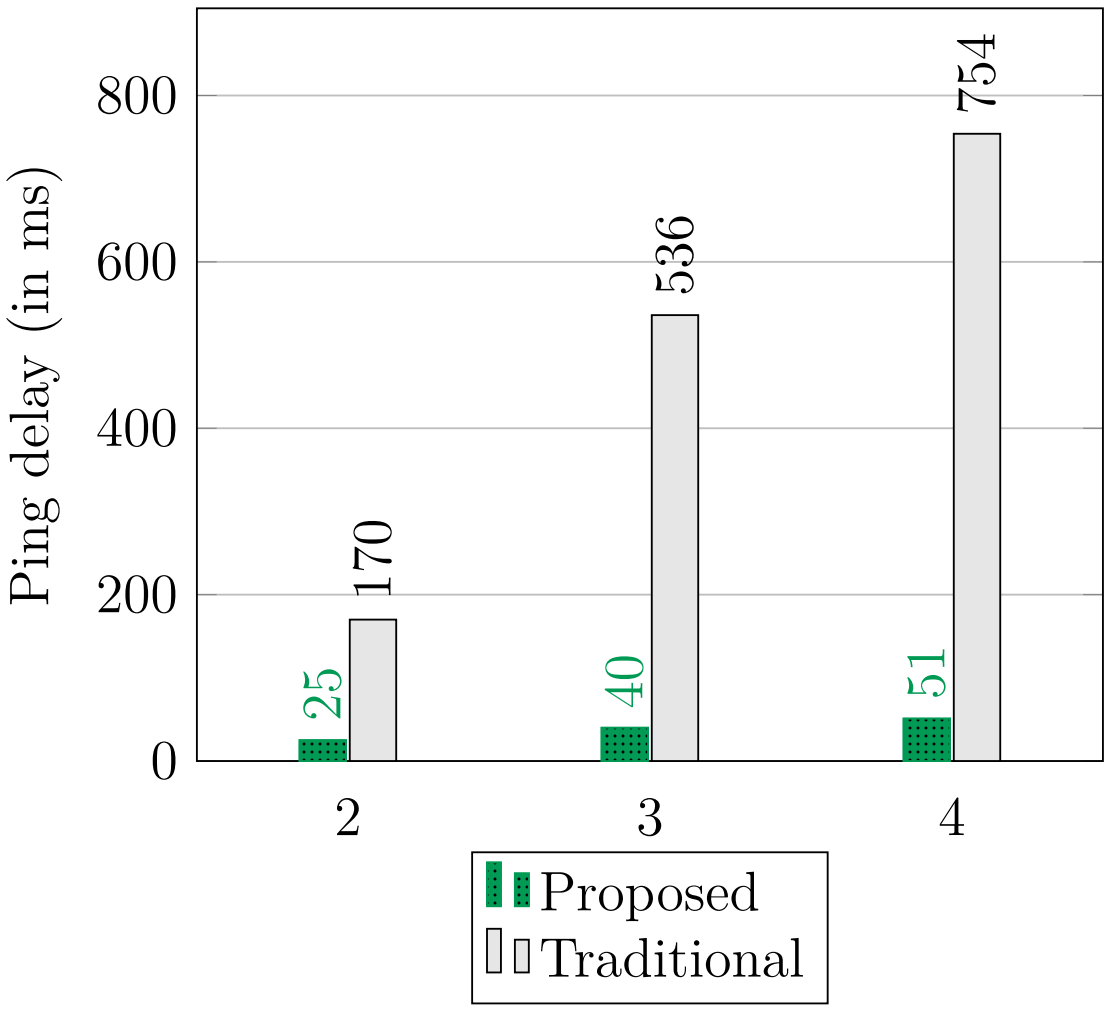

Figure 4.4: Gaming traffic Ping delay (in ms) for the respective proposed and traditional setup. *X-axis represents the total number of gaming clients.*

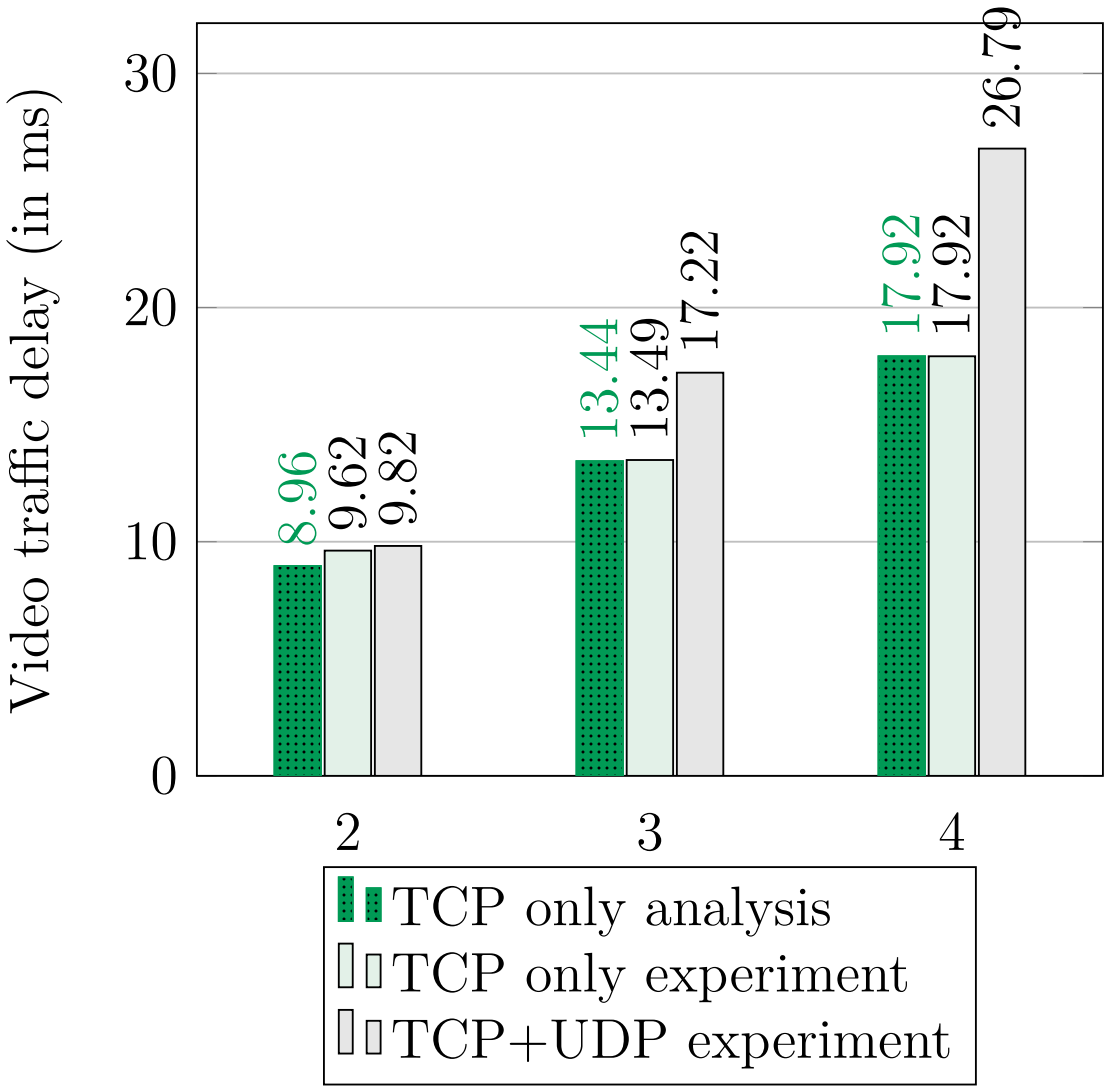

Figure 4.5: Video traffic delay (in ms) for TCP-only analysis and experiment, and TCP+UDP experiment results. *X-axis represents the total number of gaming clients.*

Table 4.2: Average UDP throughput (in kbps) for two clients setup, for the proposed and traditional experiment setup.

	Client 1	Client 2
Proposed	37.84	39.13
Traditional	40.65	47.7

work, and traditional framework. The proposed framework significantly enhances the throughput and also maintains fairness among different clients. This therefore demonstrate the prowess of the proposed SDN based split TCP framework. Table 4.2 shows the gaming traffic throughput which is minimal as the gaming clients exchange data in a sporadic manner.

The Jain's fairness index [13] plots in Fig. 4.6 and Fig. 4.7 show the accuracy of throughput fairness throughout the sessions for an experiment with three clients and four clients, respectively. Figure 4.8 shows the throughput fairness comparison of the gaming UDP applications for the respective traditional and proposed frameworks. The traditional fairness is maintained in the proposed framework (with a minimal added fairness as reflected by an *almost straight-line in the figure*). This shows the fact that the reduced delay is achieved without loss of fairness in the network. The Jain's fairness index is computed as follows:

$$\texttt{Fairness index} = \frac{(\sum_{i=1}^{n} E[T_i])^2}{n \times \sum_{i=1}^{n} E[T_i]^2} \tag{4.6}$$

where $E[T_i]$ throughput of TCP flow i and n is the total number of flows.

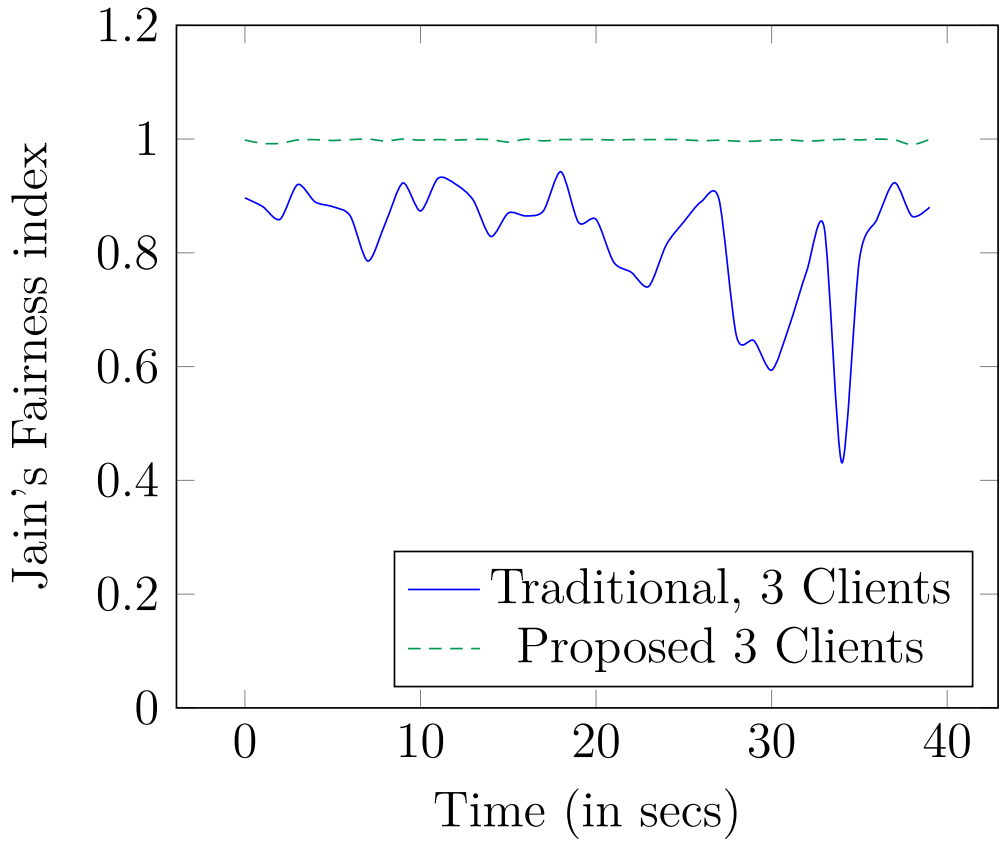

Figure 4.6: TCP Throughput Fairness comparison among traditional and proposed frameworks, for 3 Clients set up.

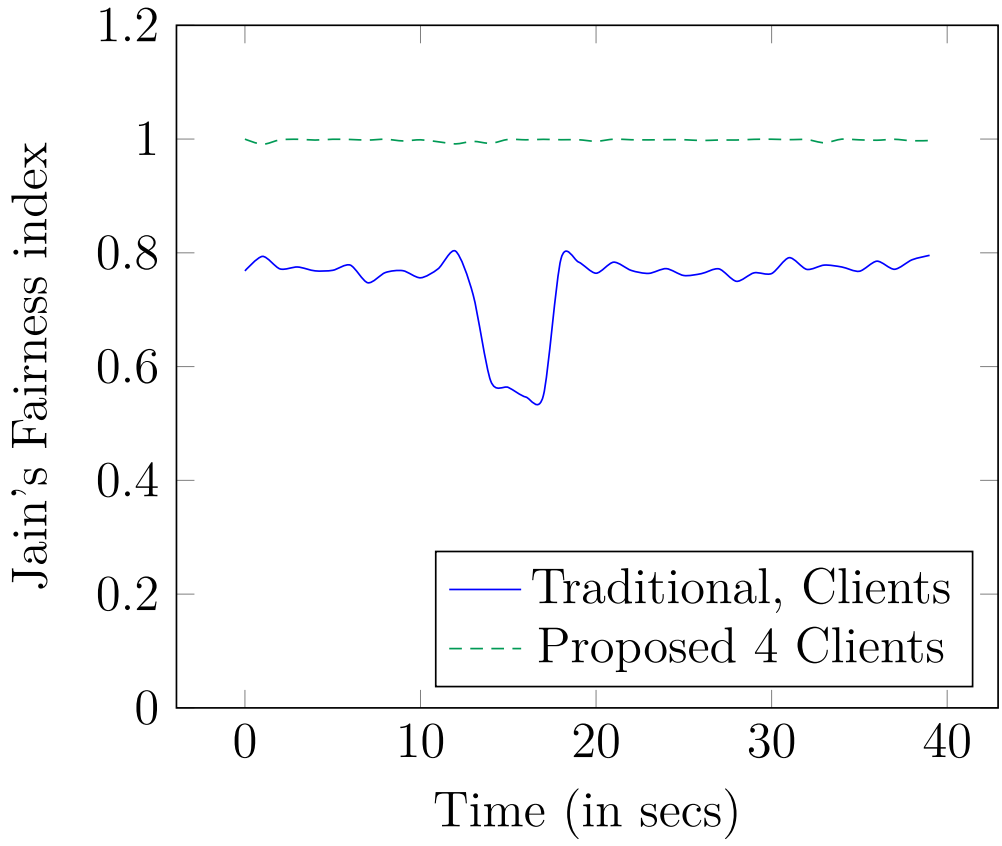

Figure 4.7: TCP Throughput Fairness comparison among proposed and traditional frameworks, for 4 Clients set up.

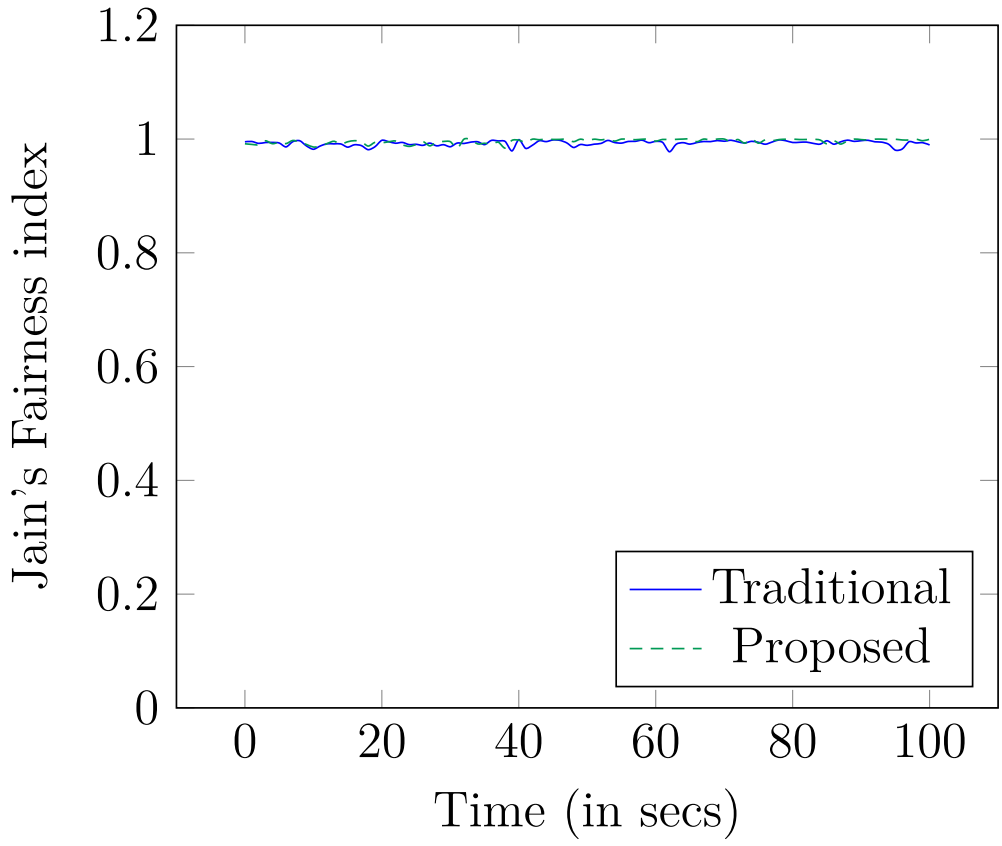

Figure 4.8: UDP Throughput Fairness comparison among proposed and traditional frameworks, for 1 game server and 2 game clients set up.

4.6 Summary

In this dissertation, we have proposed a novel SDN-based framework to enable split TCP, and limit the wireless TCP-side congestion window. The system therefore utilized the WiFi MAC's DCF fairness functionality effectively. As a result, the video traffic throughput were significantly increased fairly among all the clients, and at the same time the delay of the gaming traffic has been significantly reduced. The proposed approach is practical in a way that it works in a transparent manner, agnostic to the application end-users.

In future, we plan to study the analytical performance of the network by including both TCP and UDP traffic.

Chapter 5

On Scheduling Multiple Simultaneous Live Virtual Machine Migrations

5.1 Introduction

Many enterprises, not only large-market company like Netflix and Snapchat, but also small-market technology startups rely on cloud platforms for computing infrastructure or business [38] [39]. A cloud platform, such as Amazon Web Services (AWS) or Windows Azure, enables on-demand provisioning of computational resources in the form of Virtual Machines (VMs) deployed in the cloud provider's data center. These services meet peak or fluctuating service demands, serve multiple users in a secure, flexible and efficient way [69].

With the growth of data volumes and a variety of Internet applications, virtualization has become commonplace in modern data centers and an effective solution to provide better management flexibility, lower cost, scalability, better resources utilization, and energy efficiency [40] [70]. One of the powerful features provided by virtualization technology is VM live migration, which facilitates moving workloads within the infrastructure to bring multiple benefits such as higher performance, improved manageability and fault tolerance. Moreover, VM live migration of often allows workload movement with negligible downtime, minimal impact on workload, and no disruption of network connectivity [41] [42].

Clark et al. [43] proposed a VM live migration system which transfers memory, storage and application status (CPU state, registers, non-pageable memory) of the

virtual machine from the original server to the destination. The system handles the live migration by two main techniques, namely pre-copy memory migration, and stop-and-copy memory migration. In the pre-copy memory migration, the *hypervisor* typically copies all the memory pages from the source to the destination while the VM is still running on the source. The updated memory pages during this process are then re-copied until page dirtying rate is faster than the rate of copy. In stop-and-copy memory migration, the system transfers the remaining memory pages and application status to the destination. Then the original VM is stopped and resumed at the destination.

However, the performance of running applications is likely to be negatively affected during this process. Clark et al. [43] also showed that around 12% performance degradation is observed during the migration of a VM running a web server. One of the most common reasons to justify VM migration is to move VMs out of overloaded servers. To execute the live migration on an overloaded VM, the VM has to assign part of its computing resources to the migration program and could result in lower system performance.

An experimental study by Voorsluys et al. [41] have found that performance degradation depends on the application's behavior, for example, the number of related memory pages need to be transferred and the migration program CPU utilization. Beloglazov and Buyya [71] proposed a nice model for measuring the performance degradation of a VM during the live migration taking into account the pre-copy phase and is given below:

$$f = t_0 + \frac{m(0) + \sum_{i=1}^{t} m(i)}{T} \tag{5.1}$$

$$D = d \int_{i=0}^{f} u(i)\ di \tag{5.2}$$

where D is the total performance degradation of the VM, t is the completion time for the migration in worst-cast and can be computed by assuming the worst-case size of the VM to be migrated, d is a constant value and varies with the type of applications running on the VM, t_0 is the start time of the migration process, f is the migration completion time, $m(0)$ is the memory size at the time when the migration was initiated (initial size), and $m(i)$ is the increase or decrease of the $m(0)$ at time step i. In the stop-and-copy phase, the migration has to re-transfer the dirty pages and hence, $m(i)$ captures the changes in memory size $m(0)$ during each time period i. T is the average throughput of the migration flow, $u(i)$ is the CPU utilization of the program at time period i that is used to perform the migration.

In this dissertation, we have developed a Mixed Integer Linear Programming (MILP) model that will provide a schedule (preemptive) for the transfer of multiple VMs from source to destination. This MILP model assumes that $m(i)$ and $u(i)$ be available for each time-period i. In addition to the MILP model, we have also provide a heuristic that is based on path selection (for example, shortest widest path [72]) and priority assignment for flows corresponding to each VM migration. Our heuristics uses the changes in memory size ($m(i)$) and CPU utilization ($u(i)$) at each time step i to determine the migration schedule. We have made several performance comparisons including total performance degradation (TPD) and makespan (MS) which is the total time taken to complete all the migrations. Additionally, we have developed a simple MILP model for the problem, wherein we assume constant memory size for each VM and constant CPU utilization.

We will assume that we will use VM transfer program that is based on Transmission Control Program (TCP). When two or more flows corresponding to VM migrations share a link, the TCP program will allocate the bandwidth in a fair manner [73]. We show in this dissertation that this fair allocation will not result in optimal values for TPD and MS.

To enforce the computed schedules for the VM transfers, *each node should be provided with a schedule that provides for each time step i the order in which each packet from each flow be be forwarded*. We would like to point out that our MILP models and the heuristic provide such an output.

To realize the application of the proposed ideas, we have developed a novel join-and-split TCP protocol. In this protocol, several competing TCP flows are joined at the front end of the shared link and the flows are then split at the back end of the same shared link. During this join process, we can assign the priorities dictated by the MILP model or by the proposed heuristic at nodes to forward packets from different flows in the order in which the scheduling algorithms (MILP or Heuristic) have dictated. This join-and-split protocol is built on the network with routers that are programmable (software defined network (SDN) [3]).

The remainder of this dissertation is organized as follows.

In Section 5.2, we provide MILP models for variations of VM live migration problems Section 5.4 presents our greedy algorithms, and compare them with the MILP formulations. Experimental results are presented in Section 5.5. Conclusions are drawn in Section 5.6.

5.2 Problem Formulation

Consider the Virtual Machine (VM) live migration problem. Let $G = (V, E)$ be a network where V is the set of nodes, $E = \{e_1, e_2, \cdots e_m\}$ is the set of links and B_j is the bandwidth of link e_j, $1 \leq j \leq m$.

Consider the set of n VMs, $H = \{h_1, h_2, \cdots h_n\}$, to be migrated. Each VM h_i is being migrating from node $s_i \in V$ to node $d_i \in V$. Let $M_i(0)$ be the initial memory size of h_i, which is the initial size of the content to be transferred to the new location, AG_{it}, be the increase in memory size of $M_i(0)$ at time interval t for h_i, and C_{it} be

the non-negative migration performance degradation constant of CPU utilization of h_i on each time interval t.

Apart from the above, we provide our MILP model, for each (s_i, d_i) the set of possible paths (Q_i) from s_i to d_i. Please note that we do understand while there may be infinitely many paths from s_i to d_i in an arbitrary network, we restrict the size of Q_i. In special networks such as the Fat-Tree [74] common in data center networks, the values Q_i is bounded (and fixed).

We will use $\hat{P}_{i,k},\ h_i \in H, j \in \{1, 2 \cdots, Q_i\}$ to denote each path. Each possible path is $\hat{P}_{i,k}$ represented using a binary array $[P_{i,k,0}\ P_{i,k,1}\ \cdots]$ defined as follows:

$$P_{i,k,j} = \begin{cases} 1, & e_j \text{ is used in } \hat{P}_{i,k} \\ 0, & \text{otherwise} \end{cases}$$

We assume that only one path is selected and used for each h_i during the migrating process. For simplicity, we also assume that time is discretized as intervals $T = \{t_1, t_2, \cdots t_p\}$, where t_p is the last time interval ($p >= f$, in equation (1)).

The following decision variables are introduced to model the problem:

1. $f_i \in T$ (integer), the migration end (finish) time of h_i
2. $p_{i,k} \in \{0, 1\}$ (binary), path $\hat{P}_{i,k}$ is chosen
3. $x_{i,t}$ (continuous), allocated bandwidth of h_i during time $t \in T$
4. $u_{i,k,j,t}$ (continuous), bandwidth usage of h_i on path k and link e_j during time $t \in T$, which is equal to $P_{i,k,j} \cdot p_{ik} \cdot x_{it}$
5. $r_{i,t} \in \{0, 1\}$ (binary), if h_i migrating during time $t \in T$
6. $z_{it} \in \{0, 1\}$ (binary), if the t ofz_{it} lager than f_i, $z_{it} = 0$ else $z_{it} = 1$
7. $y_{1,i,t}$, $y_{2,i,t}$ (binary), help to linearize z_{it}

To help us model the problem, we also define $R = \{R_1, R_2, \cdots R_p\}$ as a one-dimensional array such that its length is the same as T (which is p) and $R_i = i$.

We adopt the definition of VM live migration performance degradation from Equation 5.2 where $u(i) = C_{it} \cdot z_{it}$ and assume the time interval is 1. Therefore, $Z = \sum_{i=1}^{n} \sum_{t=1}^{p} C_{it} \cdot z_{it}$ and our goal is to minimize the total VM live migration performance degradation Z. The problem then can be formulated as the following Mixed Integer Linear Programing (MILP).

Model VMLM

minimize

$$Z = \sum_{i=1}^{n} \sum_{t=1}^{p} C_{it} \cdot z_{it} \tag{5.3}$$

subject to:

$$\sum^{h_i \in H, \hat{P}_{i,k} \in Q_i} u_{ikjt} \leq B_k \qquad \forall \ e_j \in E, \ t \in T \tag{5.4}$$

$$u_{ikjt} \leq P_{ikj} \cdot K_3 \cdot p_{ik} \qquad \forall \ h_i \in H, \ e_j \in E, \ 1 \leq j \leq |Q_i|, \ t \in T \tag{5.5}$$

$$u_{ikjt} \leq P_{ikj} \cdot x_{it} \qquad \forall \ h_i \in H, \ e_j \in E, \ 1 \leq j \leq |Q_i|, \ t \in T \tag{5.6}$$

$$u_{ikjt} \geq P_{ikj} \cdot x_{it} - P_{ikj} \cdot K_3 \cdot (1 - p_{ik}) \qquad \forall \ h_i \in H, \ e_j \in E, \ 1 \leq j \leq |Q_i|, \ t \in T \tag{5.7}$$

$$\sum_{j=1}^{|Q_i|} p_{ik} = 1 \qquad \forall \ h_i \in H \tag{5.8}$$

$$0 \leq (z_{it} - 1) + K_5 \cdot y_{1it} \tag{5.9}$$

$$(f_i - R_t + 1) \leq K_6 \cdot (1 - y_{1it}) \tag{5.10}$$

$$0 \leq (0 - z_{it}) + K_8 \cdot y_{2it} \tag{5.11}$$

$$(R_t - f_i) \leq K_9 \cdot (1 - y_{2it}) \tag{5.12}$$

$$\sum_{t=1}^{p} x_{it} = M_i + \sum_{t=1}^{p} AG_{it} \cdot z_{it} \qquad \forall \ h_i \in H \tag{5.13}$$

$$K_1 \cdot r_{it} \geq x_{it} \qquad \forall \ h_i \in H, \ t \in T \tag{5.14}$$

$$K_2 \cdot x_{it} \geq r_{it} \qquad \forall \ h_i \in H, \ t \in T \tag{5.15}$$

$$f_i \geq r_{it} \cdot R_t \qquad \forall \ h_i \in H, \ t \in T \tag{5.16}$$

Objective function (5.3) captures the total VM live migration performance degradation. Constraint (5.4) enforces the bandwidth limitation of each link $e_j \in E$ at each time slot t. Constraint (5.5 – 5.7) is used to linearize $u_{ikjt} = P_{ikj} \cdot p_{ik} \cdot x_{it}$. K_3 is a constant value, which is larger than the maximum bandwidth in the network $K_3 \geq max(B_k)$. p_{ik} is a binary variable which only can be 0 or 1. In our case, if $p_{ik} = 0$, then $u_{ikjt} = 0$ else $u_{ikjt} = P_{ikj} \cdot x_{it}$. Thus, u_{ikjt} is a linear equation on both cases of p_{ik}. Constraint (5.5 – 5.7) shows how to describe the If Then logic in MILP model. Put $p_{ik} = 0$ into Constraint (5.5 – 5.7) and this model assumes that all the variables are larger or equals than 0. Then Constraint (5.5 – 5.7) make sure that $u_{ikjt} = 0$. When $p_{ik} = 1$, then Constraint (5.6) $u_{ikjt} \leq P_{ikj} \cdot x_{it}$ and Constraint (5.7) $u_{ikjt} \geq P_{ikj} \cdot x_{it}$. Thus, u_{ikjt} has to equal to $P_{ikj} \cdot x_{it}$. Constraint (5.8) ensures that only one possible path is selected during the whole process. Constraint (5.9 – 5.12) is used to linearize $z_{it} = 0$ when the index t larger than f_i, else $z_{it} = 1$, K_4 to K_9 are constant value that large enough to help for the linearization. Constraint (5.13) measures the total data transfer of each VM migration, this must match the size of each VM. Constraints (5.14) and (5.15) are used to couple $x_{i,t}$ and $r_{i,t}$. If $x_{i,t}$ is non-zero, $r_{i,t}$ is 1; other wise, $r_{i,t}$ is 0. K_1 and K_2 are constants chosen as follows for this purpose: K_1 is a constant value, which is larger than the maximum bandwidth in the network $K_1 \geq max(B_k)$. K_2 is a constant value such that, for any $x_{it} > 0$, $K_2 \cdot x_{it} >= 1$. Constraint (5.16) is used to capture the end time of each VM migration.

5.2.1 Simplified Formulation

In this section we provide a simplified version of the formulation of Model VMLM. In this formulation we provided the exact path for each (s_i, d_i) pair, and assume constant values for CPU utilization and VM size (that is, it does not change during the course

of the migration). Let the the paths between each pair of s_i and d_i are given as:

$$\bar{P}_{i,j} = \begin{cases} 1, & h_i \text{ uses } e_j \text{ during the migration} \\ 0, & \text{otherwise} \end{cases}$$

Then the CPU utilization C_{it} is a constant value, the formulation can be simplified as C_i. The changes in memory is set to $AG_{it} = 0$ and hence is removed from the formulation. The simplified problem can be formulated as the following model.

Model VMLM-Simplified

minimize

$$Z = \sum_{i=1}^{n} C_i \cdot f_i \tag{5.17}$$

subject to:

Constraints (5.14)−(5.16)

$$\sum_{t=1}^{p} x_{it} = M_i \qquad \forall \ h_i \in H \tag{5.18}$$

$$\sum_{i=1}^{n} \bar{P}_{ij} \cdot x_{it} \leq B_j \qquad \forall \ h_i \in H, \ e_j \in E, \ t \in T \tag{5.19}$$

Because C_i is a constant value, so Objective function (5.17) can simplified to $Z = \sum_{i=1}^{n} C_i \cdot f_i$ Constraints (5.14)−(5.16) are same as in **Model VMLM**. Constraint (5.19) enforces the bandwidth limitation of each link $e_j \in E$. Constraint (5.18) removes the unused AG_{it} from Constraint (5.13)and make sure that each VM send out the initial memory size of data.

5.2.2 Examples

In [74], a k-ary fat-tree topology is introduced. There are k pods, each containing two layers of $\frac{k}{2}$ switches, namely edge layer and aggregation layer. Each k-port switch

in the edge layer is directly connected to $\frac{k}{2}$ hosts. Each of the remaining $\frac{k}{2}$ ports is connected to $\frac{k}{2}$ of the k ports switch in the aggregation layer of the hierarchy.

There are $(\frac{k}{2})^2$ k-port core switches. Each core switch only has one port connected to each of the k pods. The i^{th} port of any core switch is connected to i^{th} pod. Assume Core $S = \{S_1, S_2, \cdots, S_x\}$ is the set of core switches, $A = \{A_1, A_2, \cdots, A_y\}$ is the set of aggregation switches of same pod. Each aggregation switch A_i connects to each of core switches from $S_{\frac{k}{2}\cdot(i-1)}$ to $S_{\frac{k}{2}\cdot i}$. In general, a fat-tree network built with k-port switches supports $\frac{k^3}{4}$ hosts.

The advantage of the fat-tree topology is that all switching elements are identical, enabling us to leverage cheap commodity parts for all of the switches in the communication architecture.

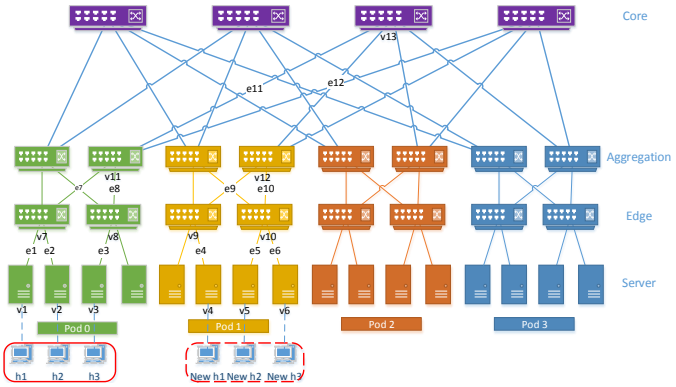

Figure 5.1: Fat-Tree Topology

Fig. 5.1 shows an example of 3 VM migrations on a Fat-Tree topology. VM h_1 on node v_1 migrates to $New\ h_1$ on node v_4 along the path $\{e_1, e_7, e_{11}, e_{12}, e_9, e_4\}$, similarly for VM s_2 and s_3. We consider the following 4 cases in different examples. Let $C = \{c_1, c_2, c_3\}$ be the migration program CPU utilizations of VM 1 to 3, $M = \{m_1, m_2, m_3\}$ be the memory sizes of VM 1 to 3 and $B = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12}\}$ be the bandwidth of link e_1 through e_{12}.

- Case 1. Link bandwidths (set at 10), VM memory sizes are the same (set at 10), CPU utilizations are different. Let $C = \{10, 8, 4\}$. The schedule corresponding

to the optimal solution is shown in Table 5.1, which indicates that migration process with higher CPU utilization should be scheduled first. The optimal total performance degradation value $Z = 10 \times 4 + 8 \times 8 + 4 \times 12 = 152$. Note that the finish times for the three VMs are $4, 8$, and 12.

- Case 2. Link bandwidth (set at 10) and CPU utilization (set at 10) are the same, VM memory sizes are different Let $M = \{30, 40, 50\}$. The schedule corresponding to the optimal solution is shown in Table 5.2, which indicates that VM with smaller memory size should be scheduled first. The optimal total performance degradation value $Z = 10 \times 3 + 10 \times 7 + 10 \times 12 = 220$. Note that the finish times for the three VMs are $3, 7$, and 12.

- Case 3. Link bandwidth are different, VM memory sizes (set at 40) and CPU utilizations (set at 10) are the same. Let $B = \{10, 8, 4, 10, 10, 10, 10, 10, 10, 10, 10, 10\}$. This problem can be converted to find the minimum total finish time and result shown in Table 5.3 The optimal total performance degradation value $Z = 10 \times 12 + 10 \times 6 + 10 \times 11 = 290$. Note that the finish times for the three VMs are $12, 6$, and 11.

- Case 4. Link bandwidths, VM memory sizes and CPU utilization are all different. Let $C = \{10, 8, 6\}$, $M = \{40, 50, 30\}$ and $B = \{6, 8, 7, 10, 10, 10, 10, 10, 10, 10, 10, 10\}$. This is a mixed case of Case 1 to 3. The schedule corresponding to the optimal solution is shown in Table 5.4. The optimal total performance degradation value $Z = 10 \times 8 + 8 \times 13 + 4 \times 6 = 208$. Note that the finish times for the three VMs are $8, 13$, and 6.

Table 5.1: bandwidth allocation and Scheduling for Case 1

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
h_1	10	10	10	10	0	0	0	0	0	0	0	0
h_2	0	0	0	0	10	10	10	10	0	0	0	0
h_3	0	0	0	0	0	0	0	0	10	10	10	10

Table 5.2: bandwidth allocation and Scheduling for Case 2

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
h_1	10	10	10	0	0	0	0	0	0	0	0	0
h_2	0	0	0	10	10	10	10	0	0	0	0	0
h_3	0	0	0	0	0	0	0	10	10	10	10	10

Table 5.3: bandwidth allocation and Scheduling for Case 3

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
h_1	0	0	0	0	0	0	6	6	6	6	6	10
h_2	6	6	8	6	8	6	0	0	0	0	0	0
h_3	4	4	2	4	2	4	4	4	4	4	4	0

Table 5.4: bandwidth allocation and Scheduling for Case 4

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
h_1	6	6	6	4	3	3	6	6	0	0	0	0	0
h_2	0	0	0	0	0	2	4	4	8	8	8	8	8
h_3	4	4	4	6	7	5	0	0	0	0	0	0	0

5.3 Complexity

Model VMLM-Simplified is a special case of Model VMLM and we show the computational complexity of this model. According to the Proposition 1 in paper [75], if $M_i/C_i = 1 \; for \; i = 1, ..., m$, the interchange of two adjacent migration tasks does not alter the result of total performance degradation. Therefore, this special case equals to makespan problem $minimize \; f_{max}$. The optimal solution of makespan problem is also the optimal solution for the aforementioned special case of total performance degradation. Since minimizing makespan problem is NP-hard, then minimizing the total performance degradation is NP-hard [76].

Consider a network that each host only has one network adapter. So, the maximum number of simultaneous file transfers that the given host can engage in is 1.

Forwarding is not allowed; each segment is transferred directly between the source host to destination host. This special case has proven to be NP-hard in [77]. If assume that each VM only has 1 segment to migrate and each segment has same length, then our problem will be equivalent to the file transfer problem in [77]. Therefore, the Model VMLM-Simplified and Model VMLM are NP-hard problems.

5.4 Greedy Algorithms

We designed a greedy algorithm to solve the TPD of VM live migration which are NP-hard problems. Then we proposed an implementation framework to deploy the greedy algorithm on the Date Center network.

5.4.1 Greedy Algorithms for Model VMLM and VMLM-Simplified

For the VM live migration problem (Model VMLM), we designed a greedy Algorithm 1. Our algorithms assigns priorities to the VMs based on the CPU utilization and its memory size. Consider the results shown in Table 5.1 and 5.2. Clearly, in one case when we schedule the VM with larger CPU utilization first yields an optimal total performance degradation value while in the other case scheduling VMs with smaller memory size yields a better result. Using this, we define *G-Ratio* as the ratio of a VMs CPU utilization over its memory size. Please note as indicated earlier, both CPU utilization and memory size changes over time and hence

$$G - Ratio(VM_i) = \frac{C_{it}}{M_i + \sum_{t=1}^{t_c} AG_{it} - \sum_{t=1}^{t_c} BW_{it}} \tag{5.20}$$

where BW_{it} is allocated bandwidth to VM_i on each time interval.

The Algorithm 1 works as follows to produce the schedule. We first sort the VMs in the descending order of the current G-Ratio values (stored in H'). Once this is done, we will determine the path P_i for VM_i. While there are many possible paths,

the choice we have made is to compute the shortest-widest path [72]. Once path is determined for a particular VM, we have chosen not to change it (of course, one can design other heuristics that allows for changes in path). The rationale for this to aid our implementation on the SDN networks, wherein once the flow table is fixed it may be expensive to change it.

As we sequentially choose each VM_i from the sorted list H', we will consume bandwidth on the links (done by reducing the bottleneck bandwidth for the path P_i). If a VM_i cannot be routed because the residual bandwidth is 0 on one or more links in the path P_i, then this VM is scheduled later (actually after a VM completes its migration). After a feasible set H'' (as set of VMs routed along the chosen paths) of VMs is determined, we assume that they start the migration process. During the migration process, note that all VMs experience changes in memory size and CPU utilization, thus impacting their G-Ratio value. As soon as a single VM completes the migration, we reset, and once again assign priorities based on their current G-Ratio values and start the whole process again. Note that in line 18 of Algorithm 1 we say start migrating VMs. By this we mean that we will now determine the schedule for each VMs at each of the nodes in the network (at least for the nodes along the path P_i for each VM_i in H''.

The time-complexity of our algorithm is as follows. The total time for step 8 will be $O(n \times (|E| + |V| \log |V|))$, $|E|$ is the number of links and $|V|$ is the number of nodes in the network G. Each time step 5 is executed it take a worst case time of $O(n \log n)$ and it is done n times and hence has a total time complexity of $O(n^2 \log n)$. Each path P_i can be of size $|V| - 1$ and hence step 10 and step 14 will take $O(n \times |V| - 1)$ each time through loop giving raise to a complexity of $O(n^2 \times (|V| - 1))$. Step 18 has a total time complexity of $O(n^2 \times (|V| - 1))$. The other steps do not increase the already determine time complexity and hence Algorithm 1 has a time-complexity of $O(n \times (|E| + |V| \log |V|)) + n^2 \times (|V| - 1))$.

Algorithm 2: VM Migration Scheduler (Network G, VMs H)

```
1  H' = H;
2  G' = G;
3  Selected (P_i)= False for all 1 ≤ i ≤ n; //n number of VMs;
4  while H' is not empty do
5  |  Sort VMs in H' in descending order of G-Ratio; Let H'' be this sorted list;
6  |  foreach VM_i in H'' do
7  |  |  if not Selected(P_i) then
8  |  |  |  Find path P_i in G' (e.g. shortest-widest) for each VM_i (s_i, d_i) in H''
   |  |  |    (go through the sorted list);
9  |  |  end
10 |  |  Let B(P_i) be the bottle next bandwidth on P_i;
11 |  |  if (B(P_i) == 0) then
12 |  |  |  Remove VM_i from H''
13 |  |  else
14 |  |  |  Reduce B(P_i) from the links in the path P_i in G';
15 |  |  |  Selected(P_i) = True;
16 |  |  end
17 |  end
18 |  Start migrating VMs in H'' and let VM_k be the first one to finish in H'';
19 |  Update v.c and v.m values for VMs in H';
20 |  G' = G;
21 |  Remove VM_k from H'.;
22 end
```

5.4.2 Implementation

When the Algorithm 1 completes its execution, it produces a schedule for each node for each time step. The schedule will be something like node 6 will forward packets (TCP since we want reliable transfer of VMs) belonging to flow 1 (which may correspond to the migration of VM 1), followed by packets from flow 4, and so on. The number of packets to be forwarded for each flow by a single node at each time step is determined by Algorithm 1 (note that the number of packets is determined by the bottleneck bandwidth reserved for the particular migration). In a future time step, the same node might be forwarding packets from flow 8 which is followed by flow 4, and so on.

The current routers are not equipped to manage the different flows differently without significant changes to existing router protocols. Our proposed heuristic can be implemented on a network that supports Software Defined Network (SDN) infrastructure. The central controller has the entire view of the network and hence the algorithm for path finding can be locally executed. Once the paths and schedules are determined the flow tables can be sent to each of the routers. Our next challenge would be to adjust the order and rate of packet forwarding at each node. The problem is further confounded because of the underlying TCP protocol.

To address this issue, we have provided a simple join-and-split protocol. This protocol runs on the SDN router (Network Function Virtualization in the SDN lingo). The idea is when there are multiple flows (TCP flows), we will join them and send one output flow. At the split location, we will separate the flows. The source and destination nodes are oblivious to this process. Since we are able to join the TCP flows, we can then use a Weighted Fair Queuing mechanism to regulate the flow as dictated by the scheduling algorithm.

Figure 5.2 shows an example with 3 tcp flows for the join and split mechanism.

First, each SDN enabled switch attaches a TCP proxy that is under the control of SDN controller. The TCP proxy can run as a NFV (network function vitalization) on remote server or an application inside the switch. Flows are aggregated to be a long TCP flow at the switch where their paths start to overlap. A unique ID (UID) and packet length (Len) are attached to the data before sending to the aggregate flow. A WFQ (weighted fair queue) adjusts the weight following the instruction from the SDN controller and controls the packet sending rate to the aggregate flow. The aggregated flow splits at the switch where the path start to divide or is the last switch by striping UID+Len from the data.

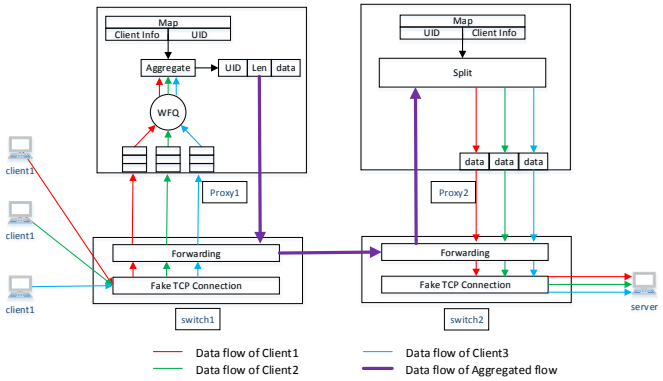

Figure 5.2: end-user agnostic TCP join and split framework in SDN

5.5 Performance Evaluation and Result Analysis

In this section, we show the results of the execution our optimization models and heuristic under different scenarios and show the values for Total Performance Degradation and Makespan. Our input network is a Fat-Tree topology with up to 8 pods (network with 208 nodes). We choose leaf nodes as the source and destination nodes for the migration. We varied the number of VMs to be migrated from 4 to 20. Our optimization models were coded in Gurobi optimizer and all algorithms ran on on a server with Intel(R) Xeon(R) CPU E5-1620 v3 running 3.50GHz, and uses 32 GB of memory. Using random seeds we changed the values of initial CPU utilization, initial

VM size, changes in values in CPU utilization and memory over time, and values of bandwidth on the links. All our experiments where repeated and obtained a 95% confidence interval on the values.

Other experiment configurations are defined as following:

1. The bandwidth of each link is randomly selected from 1 to 1000 Mbps by assuming that Data Center network reserves bandwidth for other different applications.

2. The initial memory size of each VM ranges is randomly selected from 1 to 8000 MB

3. The changes in memory size during each time period is randomly selected from 0% to 5% of initial memory size.

4. The migration process CPU utilization is is randomly selected from 1 to 20 for each time period.

We used a combination of flow control and path selection mechanisms. For example, one flow control we used was the traditional TCP (that is, each VM is migrated to the destination using TCP). Note that TCP uses fair sharing of link bandwidths and all VMs are scheduled at the same time. Since we are using TCP it is also noted that the scheduling here is non-preemptive. Other flow control mechanisms are dictated by the schedule given by the MILP models and Algorithm 1 (referred as **Greedy** in the Figures). The path selection impacts the TPD and Makespan. We incorporate two path selection criteria, a) Random: paths are chosen at random (among the set of paths), and b) SW: shortest widest path is chosen. TCP-Random for example would mean that we have chosen TCP as the main flow control mechanism on the paths for each migration that has been chosen at random from a set of paths.

Figure 3 shows the TPD of the scenario in Model VMLM where the number of VMs ranges from 1 to 8. Please note that we ran the Gurobi modeler for 30 minutes to get the results we have in Figure 3. We did not run the modeler for VMs above 8 as it took a very long time compute the final results. Both TCP and Greedy algorithms ran in few milliseconds for all ranges of VMs from 1-20. Comparing the result of Greedy-SW with MILP, Figure 3 shows that the our proposed greedy algorithm get better result than MILP model when the number of VM is 8. When we ran the MILP model for a long time (like 6 hours) we found that the MILP model outperformed the Greedy algorithm (see Figure 5.7). The Greedy algorithm outperforms the TCP based approach in both the metrics: TPD and Makespan.

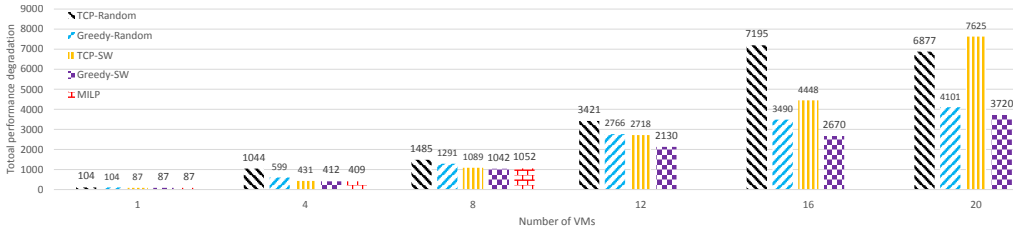

Figure 5.3: Average total performance degradation of Model VMLM

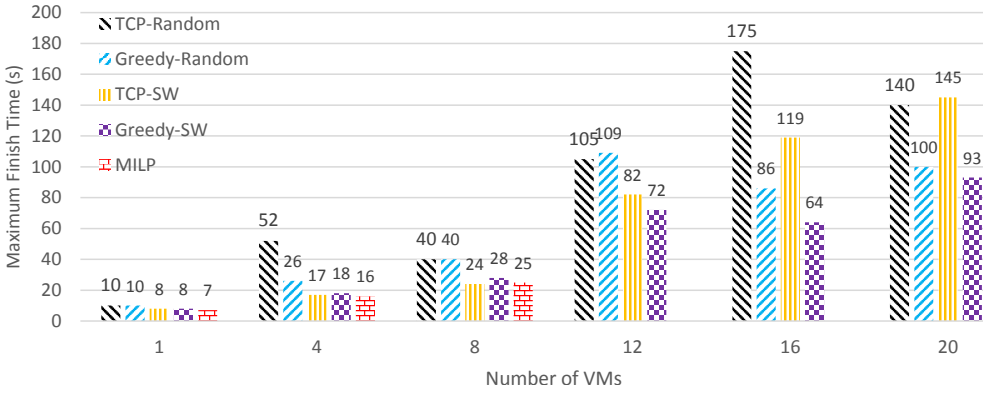

Figure 5.4: Average makespan of Model VMLM

We also studied the scenario of Model VMLM-Simplified to analyze how much TPD the Model VMLM-Simplified, greedy algorithm can reduce by comparing the result of TCP, Greedy, MILP for both Random and SW path shown as Figure 5.5. The result shows that the Greedy and MILP reduce more TPD when the number of

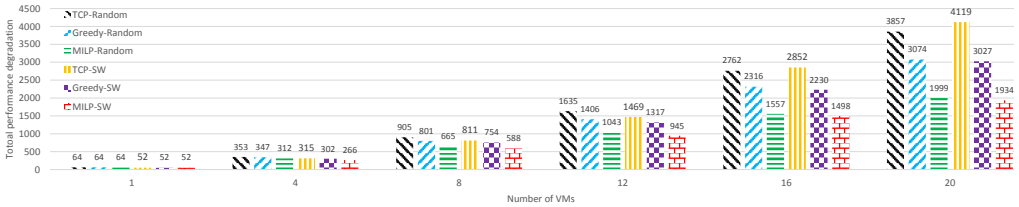

Figure 5.5: Average total performance degradation of Model VMLM-Simplified

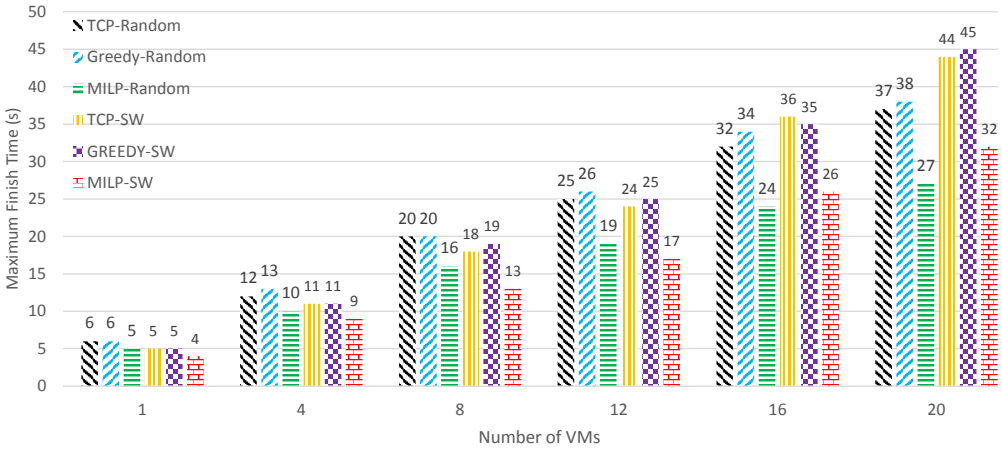

Figure 5.6: Average makespan of Model VMLM-Simplified

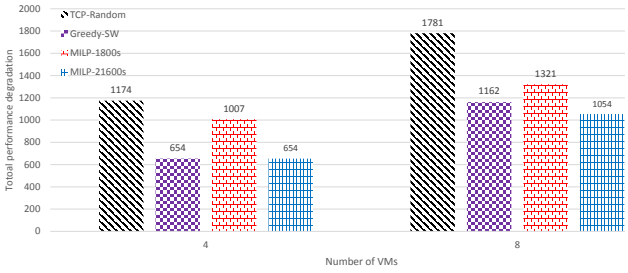

Figure 5.7: Average TPD values for model execution for over 5 hours

VMs increasing. The Greedy reduces 20.3% and MILP reduces 48.2% of TPD when the number of VMs is 20 and the path is randomly selected. We also notice that the SW get better result when the number of VM decreasing. When the number of VM is 20, Figure 5.5 shows that $TCP - SW/TCP - Random = 106.8\%, Greedy - SW/Greedy - Random = 98.5\%, MILP - SW/MILP - Random = 96.7\%$.

Once can notice that the MILP models reduce the makespan and thereby minimizing total performance degradation. In general, it is not possible to have a smaller makespan and smaller total performance degradation. For example, in Figure 3, when the number of VMs is 12, the TPD is TCP-random and Greedy-random is 3421 and 2766, respectively, while the make span values are 105 and 109, respectively. This indicates that a higher makespan could have a lower TPD value.

We compare VMLM and VMLM-Simplified and our results show in Tables 5.5 and 5.6 that VMLM has a Makespan of 8 while VMLM-simplified has a value of 10. The total performance degradation values for VMLM and VMLM-Simplified are 225, and 232, respectively.

Table 5.5: bandwidth allocation and Scheduling for Case 2

	t1	t2	t3	t4	t5	t6	t7	t8
h_1	806	806	806	526	806	0	0	0
h_2	390	0	0	0	0	0	0	0
h_3	633	633	633	633	409	633	633	633
h_4	191	191	76	471	191	997	997	997

Table 5.6: bandwidth allocation and Scheduling for Case 2

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
h_1	960	870	960	960	0	0	0	0	0	0
h_2	390	0	0	0	0	0	0	0	0	0
h_3	88	528	528	528	528	528	528	528	528	528
h_4	609	609	609	609	609	609	457	0	0	0

5.6 Summary

In this dissertation, we model the total performance degradation for concurrence VM live migration by MILP and prove this problem is NP-hard. Then we provide an efficient greedy algorithm for this problem. The simulation result show that our greedy algorithm works better as compared with straight TCP, when either the number of VM increasing or considering the variant CPU utilization and memory size. We provided a practical implementation mechanism that works on SDN networks using the concept of Join-and-Split TCP. When a time limit is imposed to the MILP mode, the proposed greedy algorithm can get better solution than MILP model when the problem is very complex.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this dissertation, we mainly addressed the TCP optimization problem on SDN platform in Smart Grid network, WiFi network and Data Center network.

First, we have proposed and implemented a generic join-and split SDN framework of aggregating and splitting TCP flows, with 'linked-ACK' mechanism to preserve end-to-end semantics. The framework developed is implemented in an user-agnostic manner so as to make it more practical. With extensive simulation experiments, we have demonstrated the efficacy of our proposed framework. We have showed the following benefits as achieved by our proposed framework: i) achieves an improved TCP goodput performance, ii) improved buffer usage at the respective split and join nodes, iii) provides fairness among different client flows, iv) improved wireless network throughput, and v) integrates MPTCP based proxy node which provides a hybrid implementation of supporting MPTCP nodes to traditional TCP flows. Despite that our framework also improves goodput performance in MPTCP environment.

The join-and-split framework is the cornerstone of this dissertation. The following research are all based on this platform. Different extensions of the join-and-split framework are applied to a Smart Grid network, WiFi network and Data Center network as follows:

In the Smart Grid network, we have proposed a novel SDN-based TCP Aggregation/scheduling Smart Grid framework that achieved a better throughput perfor-

mance and fairness. We also have proposed the white-box model of implementation with a detailed functional design. The throughput performance is extensively studied and compared with the respective traditional no-aggregation, aggregation with no scheduling, and our proposed aggregation and scheduling frameworks. Our proposed framework demonstrated fairness to a significant level of accuracy throughout the course of the experiment, while maintaining a high throughput.

In the WiFi network, we have proposed a novel SDN-based framework to enable split TCP, and limit the wireless TCP-side congestion window. The system, therefore, utilized the WiFi MAC's DCF fairness functionality effectively. As a result, the video traffic throughput was significantly increased fairly among all the clients, and at the same time the delay of the gaming traffic has been significantly reduced. The proposed approach is practical in a way that it works in a transparent manner, agnostic to the application end-users.

In Data Center network, we have optimized the total performance degradation for live migration in Data Center network. It studied two scenarios that VM migration with a fixed path and VM migration with path selection. With the help of SDN platform, this paper proposed a preemption scheduler that modeled by MILP for both scenarios. Moreover, this dissertation demonstrated the greedy algorithm for each MILP model. A fat-tree topology with pod size 8 is used to test a maximum number of 20 VM live migrations. The result shows a significant performance degradation decreasing both in MILP model and greedy algorithm when the number of VMs increases. The greedy algorithm can't give the optimum solution as the problem becoming harder, but it could provide a better solution than MILP model in terms of the time constrain exhibited in case of large problems.

6.2 Future Work

In the future, we plan to extend the TCP join-and-split framework with more applications on global TCP optimization and study the performance in the presence of background traffic. Moreover, current join-proxy or split-proxy running on the standard Linux system which part of TCP parameters are global variable for all TCP sessions and part of TCP parameters are not accessible. If we can convert global TCP parameters to private parameters for each TCP session and provide access to each TCP parameter, the TCP join-and split framework will be more powerful and flexible. To widely deploy this framework, we can encapsulate this framework a network function of Network functions virtualization (NFV) [78] which can quickly scale up and reducing the time to deploy new network function.

In the Smart Grid Network, 5G will replace LTE with much higher throughput and lower delay. We would like to know how the smart grid network cooperates with 5G.

In the WiFi gaming network, we plan to study the analytical performance of the network by including both TCP and UDP traffic. Moreover, mobile gaming gains popularity over PCs, such as Pokemon Go which suffers serious network issue at the beginning. We would like to know how the mobile gaming traffic work in the lasted WiFi standard.

In the Data Center Network, the greedy algorithm could be improved by shorting the VM migration maximum completion time. The proposed greedy algorithm can be implemented with the TCP join-and-split framework.

Bibliography

[1] T. Klein and G. Hampel, "Mptcp proxies and anchors," 2012.

[2] Wikipedia, "Sdn architecture." https://www.opennetworking.org, 2017. [Online; accessed 13-June-2017].

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

[4] O. N. Foundation, "OpenFlow Switch Specification." https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, 2017. [Online; accessed 13-June-2017].

[5] Mininet, "Mininet." http://mininet.org/, 2017. [Online; accessed 13-June-2017].

[6] Floodlight, "Floodlight." http://www.projectfloodlight.org/floodlight/, 2017. [Online; accessed 13-June-2017].

[7] Wikipedia, "Nox (platform) — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Nox%20(platform)&oldid=682469514, 2017. [Online; accessed 28-June-2017].

[8] J. Postel *et al.*, "Transmission control protocol rfc 793," 1981.

[9] Wikipedia, "OSI model — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=OSI%20model&oldid=785832226, 2017. [Online; accessed 22-June-2017].

[10] L. L. Peterson and B. S. Davie, *Computer networks: a systems approach*. Elsevier, 2007.

[11] Wikipedia, "Transmission Control Protocol — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Transmission%20Control%20Protocol&oldid=789321854, 2017. [Online; accessed 06-July-2017].

[12] C. M. Kozierok, *The TCP/IP guide: a comprehensive, illustrated Internet protocols reference*. No Starch Press, 2005.

[13] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation, 1984.

[14] L. Han, "Method to allocate buffer for tcp proxy session based on dynamic network conditions," July 15 2014. US Patent 8,782,221.

[15] F. Lu, G. M. Voelker, and A. C. Snoeren, "Weighted fair queuing with differential dropping," in *INFOCOM, 2012 Proceedings IEEE*, pp. 2981–2985, IEEE, 2012.

[16] R. Craven, R. Beverly, and M. Allman, "A middlebox-cooperative tcp for a non end-to-end internet," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 151–162, 2015.

[17] T. Khalifa, A. Abdrabou, K. Naik, M. Alsabaan, A. Nayak, and N. Goel, "Split- and aggregated-transmission control protocol (sa-tcp) for smart power grid," *IEEE Transactions on Smart Grid*, vol. 5, no. 1, pp. 381–391, 2014.

[18] W. Guo, V. Mahendran, and S. Radhakrishnan, "Achieving throughput fairness in smart grid using sdn-based flow aggregation and scheduling," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2016 IEEE 12th International Conference on*, pp. 1–7, IEEE, 2016.

[19] Y. Xu, V. Mahendran, and S. Radhakrishnan, "Towards sdn-based fog computing: Mqtt broker virtualization for effective and reliable delivery," in *Communication Systems and Networks (COMSNETS), 2016 8th International Conference on*, pp. 1–6, IEEE, 2016.

[20] W. Guo, V. Mahendran, and S. Radhakrishnan, "End-user agnostic join and fork framework for tcp flows in sdn," in *Consumer Communications & Networking Conference (CCNC), 2017 14th IEEE Annual*, IEEE, 2017.

[21] I. Moiseenko and D. Oran, "Tcp/icn: Carrying tcp over content centric and named data networks," in *Proceedings of the 2016 conference on 3rd ACM Conference on Information-Centric Networking*, pp. 112–121, ACM, 2016.

[22] G. Hasegawa, M. Nakata, and H. Nakano, "Receiver-based ack splitting mechanism for tcp over wired/wireless heterogeneous networks," *IEICE transactions on communications*, vol. 90, no. 5, pp. 1132–1141, 2007.

[23] J. Navarro-Ortiz, P. Ameigeiras, J. J. Ramos-Munoz, and J. M. Lopez-Soler, "Removing redundant tcp functionalities in wired-cum-wireless networks with ieee 802.11 e hcca support," *International Journal of Communication Systems*, vol. 27, no. 11, pp. 3352–3367, 2014.

[24] Y. Yan, Y. Qian, H. Sharif, and D. Tipper, "A survey on smart grid communication infrastructures: Motivations, requirements and challenges," *IEEE communications surveys & tutorials*, vol. 15, no. 1, pp. 5–20, 2013.

[25] M. Hashmi, S. Hänninen, and K. Mäki, "Survey of smart grid concepts, architectures, and technological demonstrations worldwide," in *Innovative Smart Grid Technologies (ISGT Latin America), 2011 IEEE PES Conference on*, pp. 1–7, IEEE, 2011.

[26] T. Khalifa, K. Naik, and A. Nayak, "A survey of communication protocols for automatic meter reading applications," *IEEE Communications Surveys Tutorials*, vol. 13, no. 2, pp. 168–182, 2011.

[27] T. Khalifa, A. Abdrabou, K. Naik, M. Alsabaan, A. Nayak, and N. Goel, "Design and analysis of split- and aggregated-transport control protocol (SA-TCP) for smart metering infrastructure," in *SmartGridComm '12: Proceedings of the International Conference on Smart Grid Communications*, pp. 139–144, 2012.

[28] T. Khalifa, A. Abdrabou, K. Naik, M. Alsabaan, A. Nayak, and N. Goel, "Split- and aggregated-transmission control protocol (SA-TCP) for smart power grid," *IEEE Transactions on Smart Grid*, vol. 5, no. 1, pp. 381–391, 2014.

[29] Y. Xu, V. Mahendran, and S. Radhakrishnan, "Towards SDN-based fog computing: MQTT broker virtualization for effective and reliable delivery," in *COMSNETS WACI' 16: IEEE COMSNETS Workshop on Wild and Crazy Ideas on the Interplay Between IoT and Big Data*, 2016.

[30] S. Ratti, B. Hariri, and S. Shirmohammadi, "A survey of first-person shooter gaming traffic on the internet," *IEEE Internet Computing*, vol. 14, no. 5, pp. 60–69, 2010.

[31] "Tutorial: Traffic of online games. Last accessed: 3 November, 2016." `https://www.ietf.org/proceedings/87/slides/slides-87-tsvarea-1.pdf`.

[32] Wikipedia, "First-person shooter — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=First-person%20shooter&oldid=787563551`, 2017. [Online; accessed 29-June-2017].

[33] Wikipedia, "Counter-Strike — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Counter-Strike&oldid=779967225`, 2017. [Online; accessed 29-June-2017].

[34] S. Pilosof, R. Ramjee, D. Raz, Y. Shavitt, and P. Sinha, "Understanding TCP fairness over wireless LAN," in *INFOCOM '03: Proceedings of the IEEE Conference on Computer and Communications.*, vol. 2, pp. 863–872, 2003.

[35] S. R. Pokhrel, M. Panda, H. L. Vu, and M. Mandjes, "TCP performance over Wi-Fi: Joint impact of buffer and channel losses," *IEEE Transactions on Mobile Computing*, vol. 15, no. 5, pp. 1279–1291, 2016.

[36] D. J. Leith, P. Clifford, D. Malone, and A. Ng, "TCP fairness in. 802.11e WLANs," *IEEE Communications Letters*, vol. 9, no. 11, pp. 964–966, 2005.

[37] X. Cao, L. Liu, W. Shen, J. Tang, and Y. Cheng, "Real-time misbehavior detection in IEEE 802.11e based WLANs," in *GLOBECOM '14: Proceedings of the IEEE Global Communications Conference*, pp. 631–636, 2014.

[38] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy, "Managing risk in a derivative iaas cloud," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[39] Amazon, "Netflix Case Study." `https://aws.amazon.com/solutions/case-studies/netflix/`, 2016. [Online; accessed 13-June-2017].

[40] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[41] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *IEEE International Conference on Cloud Computing*, pp. 254–265, Springer, 2009.

[42] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty, "Xvmotion: Unified virtual machine migration over long distance.," in *USENIX Annual Technical Conference*, pp. 97–108, 2014.

[43] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[44] H. Liu, H. Jin, C.-Z. Xu, and X. Liao, "Performance and energy modeling for live migration of virtual machines," *Cluster computing*, vol. 16, no. 2, pp. 249–264, 2013.

[45] D. Breitgand, G. Kutiel, and D. Raz, "Cost-aware live migration of services in the cloud.," in *SYSTOR*, 2010.

[46] F. Xu, F. Liu, L. Liu, H. Jin, B. Li, and B. Li, "iaware: Making live migration of virtual machines interference-aware in the cloud," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 3012–3025, 2014.

[47] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer, "Remedy: Network-aware steady state vm management for data centers," *NETWORKING 2012*, pp. 190–204, 2012.

[48] W. Guo, V. Mahendran, and S. Radhakrishnan, "Join and split tcp for sdn networks: Architecture, implementation, and evaluation." submitted to Computer Networks Elsevier.

[49] W. Guo, V. Mahendran, and S. Radhakrishnan, "Tcp fairness optimization in smart grid network." submitted to IEEE Transactions on Smart Grid.

[50] W. Guo, V. Mahendran, and S. Radhakrishnan, "Improved video throughput and reduced gaming delay in wlan through seamless sdn-based traffic steering," in *Consumer Communications & Networking Conference (CCNC), 2017 14th IEEE Annual*, IEEE, 2017.

[51] W. Guo, Y. Chen, S. Radhakrishnan, and V. Mahendran, "On scheduling multiple simultaneous live virtual machine migrations." submitted to INFOCOM.

[52] K. Ratnam and I. Matta, "Wtcp: an efficient mechanism for improving wireless access to tcp services," *International journal of communication systems*, vol. 16, no. 1, pp. 47–62, 2003.

[53] P. Siano, "Demand response and smart grids-a survey," *Renewable and Sustainable Energy Reviews*, vol. 30, pp. 461–478, 2014.

[54] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural guidelines for multipath tcp development," tech. rep., 2011.

[55] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath tcp.," in *NSDI*, vol. 11, pp. 8–8, 2011.

[56] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, "Sdn-based application-aware networking on the example of youtube video streaming," in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pp. 87–92, IEEE, 2013.

[57] G. Hampel, A. Rana, and T. Klein, "Seamless tcp mobility using lightweight mptcp proxy," in *Proceedings of the 11th ACM international symposium on Mobility management and wireless access*, pp. 139–146, ACM, 2013.

[58] icteam, "Multipath TCP." `https://www.multipath-tcp.org/`, 2017. [Online; accessed 13-June-2017].

[59] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling tcp throughput: A simple model and its empirical validation," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 303–314, 1998.

[60] T. H. Chuang, M. H. Tsai, and C. Y. Chuang, "Group-based uplink scheduling for machine-type communications in lte-advanced networks," in *WAINA '15: Proceedings of the Advanced Information Networking and Applications Workshops*, pp. 652–657, 2015.

[61] T. Khalifa, K. Naik, M. Alsabaan, and A. Nayak, "A transport control protocol suite for smart metering infrastructure," in *ICEDSA '11: Proceedings of the International Conference on Electronic Devices, Systems and Applications*, pp. 5–10, 2011.

[62] J. Cho, B. Nguyen, A. Banerjee, R. Ricci, J. Van der Merwe, and K. Webb, "SMORE: Software-defined networking mobile offloading architecture," in *AllThingsCellular '14: Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges*, pp. 21–26, 2014.

[63] U. Parthavi Moravapalle, S. Sanadhya, A. Parate, and K.-H. Kim, "Pulsar: Improving throughput estimation in enterprise LTE small cells," in *CoNEXT '15: Proceedings of the ACM International on Conference on Emerging Networking Experiments and Technologies*, (New York, NY, USA), ACM, 2015.

[64] "Mininet." `mininet.org`.

[65] P. Loiseau, P. GonÃ§alves, J. Barral, and P. V.-B. Primet, "Modeling TCP throughput: An elaborated large-deviations-based model and its empirical validation," *Performance Evaluation*, vol. 67, no. 11, pp. 1030–1043, 2010.

[66] A. Dembo and O. Zeitouni, *Large Deviations Techniques and Applications*. Jones and Bartlett, Boston, 1993.

[67] "ns-3 network simulator." `https://www.nsnam.org`.

[68] "Sandvine global Internet phenomena report 07 Dec 2015 press release. Last accessed: 3 November, 2016." `https://www.sandvine.com/pr/2015/12/7`.

[69] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet computing*, vol. 13, no. 5, 2009.

[70] M. F. Bari, M. F. Zhani, Q. Zhang, R. Ahmed, and R. Boutaba, "Cqncr: Optimal vm migration planning in cloud data centers," in *Networking Conference, 2014 IFIP*, pp. 1–9, IEEE, 2014.

[71] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012.

[72] T. Korkmaz, M. Krunz, and S. Tragoudas, "An efficient algorithm for finding a path subject to two additive constraints," *Computer Communications*, vol. 25, no. 3, pp. 225–238, 2002.

[73] S. Molnár, B. Sonkoly, and T. A. Trinh, "A comprehensive tcp fairness analysis in high speed networks," *Computer Communications*, vol. 32, no. 13, pp. 1460–1484, 2009.

[74] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 63–74, ACM, 2008.

[75] J. Bruno, E. G. Coffman Jr, and R. Sethi, "Scheduling independent tasks to reduce mean finishing time," *Communications of the ACM*, vol. 17, no. 7, pp. 382–387, 1974.

[76] R. G. Michael and S. J. David, "Computers and intractability: a guide to the theory of np-completeness," *WH Free. Co., San Fr*, pp. 90–91, 1979.

[77] E. G. Coffman, Jr, M. R. Garey, D. S. Johnson, and A. S. LaPaugh, "Scheduling file transfers," *SIAM Journal on computing*, vol. 14, no. 3, pp. 744–780, 1985.

[78] Wikipedia, "Network function virtualization — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Network%20function%20virtualization&oldid=788990066`, 2017. [Online; accessed 14-July-2017].