Data Movement Challenges and Solutions with Software Defined Networking

Ryan Izard
Clemson University, rizard@g.clemson.edu

Data Movement Challenges and Solutions with Software Defined Networking

---

A Dissertation

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

Computer Engineering

---

by

Ryan Izard

May 2017

Abstract

With the recent rise in cloud computing, applications are routinely accessing and interacting with data on remote resources. Interaction with such remote resources for the operation of media-rich applications in mobile environments is also on the rise. As a result, the performance of the underlying network infrastructure can have a significant impact on the quality of service experienced by the user. Despite receiving significant attention from both academia and industry, computer networks still face a number of challenges. Users oftentimes report and complain about poor experiences with their devices and applications, which can oftentimes be attributed to network performance when downloading or uploading application data. This dissertation investigates problems that arise with data movement across computer networks and proposes novel solutions to address these issues through software defined networking (SDN).

SDN is lauded to be the paradigm of choice for next generation networks. While academia explores use cases in various contexts, industry has focused on data center and wide area networks. There is a significant range of complex and application-specific network services that can potentially benefit from SDN, but introduction and adoption of such solutions remains slow in production networks. One impeding factor is the lack of a simple yet expressive enough framework applicable to all SDN services across production network domains. Without a uniform framework, SDN developers create disjoint solutions, resulting in untenable management and maintenance overhead. The SDN-based solutions developed in this dissertation make use of a common agent-based approach. The architecture facilitates application-oriented SDN design with an abstraction composed of software agents on top of the underlying network.

There are three key components modern and future networks require to deliver exceptional data transfer performance to the end user: (1) user and application mobility, (2) high throughput data transfer, and (3) efficient and scalable content distribution. Meeting these key components will not only ensure the network can provide robust and reliable end-to-end connectivity, but also that network resources will be used efficiently.

First, mobility support is critical for user applications to maintain connectivity to remote, cloud-based resources. Today's network users are frequently accessing such resources while on the go, transitioning from network to network with the expectation that their applications will continue to operate seamlessly. As users perform handovers between heterogeneous networks or between networks across administrative domains, the application becomes responsible for maintaining or establishing new connections to remote resources. Although application developers often account for such handovers, the result is oftentimes visible to the user through diminished quality of service (e.g. rebuffering in video streaming applications). Many intra-domain handover solutions exist for handovers in WiFi and cellular networks, such as mobile IP, but they are architecturally complex and have not been integrated to form a scalable, inter-domain solution. A scalable framework is proposed that leverages SDN features to implement both horizontal and vertical handovers for heterogeneous wireless networks within and across administrative domains. User devices can select an appropriate network using an on-board virtual SDN implementation that manages available network interfaces. An SDN-based counterpart operates in the network core and edge to handle user migrations as they transition from one edge attachment point to another. The framework was developed and deployed as an extension to the Global Environment for Network Innovations (GENI) testbed; however, the framework can be deployed on any OpenFlow enabled network. Evaluation revealed users can maintain existing application connections without breaking the sockets and requiring the application to recover.

Second, high throughput data transfer is essential for user applications to acquire large remote data sets. As data sizes become increasingly large, often combined with their locations being far from the applications, the well known impact of lower Transmission Control Protocol (TCP) throughput over large delay-bandwidth product paths becomes more significant to these applications. While myriads of solutions exist to alleviate the problem, they require specialized software and/or network stacks at both the application host and the remote data server, making it hard to scale up to a large range of applications and execution environments. This results in high throughput data transfer that is available to only a select subset of network users who have access to such specialized software. An SDN based solution called Steroid OpenFlow Service (SOS) has been proposed as a network service that transparently increases the throughput of TCP-based data transfers across large networks. SOS shifts the complexity of high performance data transfer from the end user to the network; users do not need to configure anything on the client and server machines participating in the data transfer. The SOS architecture supports seamless high performance data transfer at scale for multiple users and for high bandwidth connections. Emphasis is placed on the use of SOS as a part of a larger, richer data transfer ecosystem, complementing and compounding the efforts of existing data transfer solutions. Non-TCP-based solutions, such as Aspera, can operate seamlessly alongside an SOS deployment, while those based on TCP, such as wget, curl, and GridFTP, can leverage SOS for throughput improvement beyond what a single TCP connection can provide. Through extensive evaluation in real-world environments, the SOS architecture is proven to be flexibly deployable on a variety of network architectures, from cloud-based, to production networks, to scaled up, high performance data center environments. Evaluation showed that the SOS architecture scales linearly through the addition of SOS "agents" to the SOS deployment, providing data transfer performance improvement to multiple users simultaneously. An individual data transfer enhanced by SOS was shown to have increased throughput nearly forty times the same data transfer without SOS assistance.

Third, efficient and scalable video content distribution is imperative as the demand for multimedia content over the Internet increases. Current state of the art solutions consist of vast content distribution networks (CDNs) where content is oftentimes hosted in duplicate at various geographically distributed locations. Although CDNs are useful for the dissemination of static content, they do not provide a clear and scalable model for the on demand production and distribution of live, streaming content. IP multicast is a popular solution to scalable video content distribution; however, it is seldom used due to deployment and operational complexity. Inspired from the distributed design of todays CDNs and the distribution trees used by IP multicast, a SDN based framework called GENI Cinema (GC) is proposed to allow for the distribution of live video content at scale. GC allows for the efficient management and distribution of live video content at scale without the added architectural complexity and inefficiencies inherent to contemporary solutions such as IP multicast. GC has been deployed as an experimental, nation-wide live video distribution service using the GENI network, broadcasting live and prerecorded video streams from conferences for remote attendees, from the classroom for distance education, and for live sporting events. GC clients can easily and efficiently switch back and forth between video streams with improved switching latency latency over cable, satellite, and other live video providers.

The real world deployments and evaluation of the proposed solutions show how SDN can be used as a novel way to solve current data transfer problems across computer networks. In addition, this dissertation is expected to provide guidance for designing, deploying, and debugging SDN-based applications across a variety of network topologies.

Dedication

For their unconditional love, support, and encouragement, I dedicate this dissertation to my wife Anna Maria Izard and daughter Bowman Claire Izard.

Acknowledgements

I would like to thank my advisor, Dr. Kuang-Ching Wang, for his guidance, support, and patience throughout both my undergraduate and Ph.D. studies at Clemson. He has been instrumental in my growth as a researcher and engineer. Dr. Wang demonstrated ceaseless commitment to my success during my time at Clemson for which I am forever grateful.

I would also like to thank my committee members, Dr. Harlan Russell, Dr. Richard Brooks, and Dr. Jim Martin for their support and encouragement of my work over the years, including taking the time to review and provide direction on my dissertation.

In addition, I would like to thank all my friends and colleagues from our research group at Clemson, including Dr. Ke Xu, Dr. Fan Yang, Dr. Juan Deng, Qing Wang, Geddings Barrineau, Junaid Zulfiqar, Benton Kribbs, and Joe Porter. I would not have been able to persevere without their constant enthusiasm and support of my work.

I would also like to thank all who I worked with over the years at the Global Environment for Network Innovations (GENI) project office and in the GENI community, including Dr. Niky Riga, Dr. Vic Thomas, Dr. Ali Sydney, Heidi Picher Dempsey, Manu Gosain, Sarah Briggs, Tim Upthegrove, Luisa Nevers, Marshall Brinn, Tom Mitchell, Kirk Webb, and Joe Breen. Through my work on various GENI projects that contributed to my dissertation, these individuals have helped tremendously in the setup of various experiments, the efficient collection of data, and debugging my GENI experiments. Without their help, support, and encouragement, I would not have been able to complete my work in a timely manner.

Lastly, I would like to thank my parents, Richard and Linda Izard, for surrounding me with their unconditional love, support, and encouragement as I strived to earn my degree.

Contents

List of Tables

List of Algorithms

List of Figures

Chapter 1

Introduction

Computing technology has become ever-present in the day-to-day lives of people all around the world. It is constantly shrinking in form factor, increasing in efficiency, and decreasing in price, making it more accessible and appealing to users. A major trend in recent years has been cloud computing. Rather than users relying on and accessing device-local resources, they are becoming increasingly reliant on remote cloud-based resources to access media-rich content while on the move. As a result, the performance of the underlying network infrastructure can have a significant impact on the quality of service experienced by users.

Despite receiving significant attention from both academia and industry, computer networks still face a number of challenges. Users oftentimes report and complain about poor experiences with their devices and applications. Examples include trouble displaying high quality video streams and accessing remote content in a timely manner. These service degradations and failures are sometimes due to the design of the device or of the application itself; however, the cause of such problems can also be attributed to poor performance when transferring data over the network. This dissertation investigates problems that arise with data movement across computer networks and proposes novel solutions to address these issues through software defined networking (SDN). These solutions are used to form a framework for a SDN-based network architecture that improves the quality of service and network experience provided to end users, network operators, and application developers. There are three key components modern and future networks require to deliver an exceptional quality of service when moving data between network endpoints: (1) user and application mobility, (2) high throughput data transfer, and (3) efficient and scalable content distribution.

1.1 User and Application Mobility

Users frequently require access to remote resources while on the go. Horizontal handovers are well studied [1, 2, 3] and have been implemented in networks today [4] within single administrative domains. Vertical handovers, on the other hand, are well-studied but are still evolving. They are often reliant on the application itself to recover broken connections after a handover occurs. The broken connections are due to post-handover loss of end-to-end IP connectivity. Mobile IPv4 (RFC 5944) [5] provides a mechanism by which a device can retain the use of an IP address even after it has associated with a foreign network. However, it results in a triangle-routing problem after handovers [6]. The introduction of IPv6 features to Mobile IP (MIPv6) [7] running on an IPv6 network can alleviate the triangle through the use of what is known as route optimization. A limitation of these MIPv6 schemes is that they require custom software on the client to enable mobility and have yet to be fully developed and integrated into many network infrastructures. Furthermore, a broad consensus has yet to be achieved for mobility across administrative domains.

This dissertation explores the use of SDN as a network layer agnostic solution to achieving mobility across both heterogeneous networks and across administrative domains. SDN has been proposed as a possible solution to achieve device mobility across wireless networks. The authors in [2, 3] have shown its feasibility in homogeneous networks, such as WiFi, LTE, and Ethernet, respectively. Furthermore, others [8] have explored OpenFlow combined with the IEEE 802.21 Media Independent Handover specification [9] to handover between heterogeneous wireless networks; however, they have only considered the use of SDN [10] on the mobile device and not in the network core. Additional work [11] has focused on algorithms for network selection between handovers, but did not focus on the development of an architecture to support such handovers. In this dissertation, a SDN-based framework to support handovers between heterogeneous networks and across administrative domains is proposed.

1.2 High Throughput Data Transfer

Users naturally desire to transfer data as quickly as possible. This requires a network capable of facilitating high throughput data transfers. Between endpoints on local area networks, data transfers can occur rapidly and reliably without assistance using Transmission Control Protocol (TCP) [12]. However, over high-performance wide area networks, link latency poses a problem. Common reliable transport protocols, such as TCP, have difficulty making full use of the available bandwidth due in part to protocol windowing mechanisms that ensure reliability [13]. Other transport protocols, have been proposed that can achieve a higher throughput over the network. Parallel TCP, such as MPTCP [14] is particularly successful due to its use of multiple TCP connections spanning multiple network paths. Such solutions can utilize the aggregate link bandwidth to achieve higher throughput [13]. However, to achieve this increased level of performance, such solutions require modification to end user machines. Modification required by the end user entails the installation of additional kernel modules and special configuration of the standard network stack to support these high-performance, specialized transport protocols. This is oftentimes difficult for average network users who either do not have permission, the required software, or the expertise to perform such modifications.

This dissertation proposes the use of SDN to achieve transparent, reliable, high throughput data transfer as a network service. The proposed solution is scalable to any number of users as long as there is sufficient bandwidth available in the network. Such a solution can be realized underneath or in supplement to existing data transfer tools to further increase performance. It can also be used in a cloud-based approach to enhance data transfer speeds to and from remote locations where a high bandwidth network connection is unavailable.

1.3 Efficient and Scalable Content Distribution

A recent study has shown that a significant percentage of traffic on the Internet consists of live, streaming multimedia content [15]. This number is only expected to rise in the coming years [16]. The use of content delivery networks (CDNs) is a common practice to enable mass-delivery of static content to users distributed geographically [17]. Real-world deployment of such techniques requires negotiation with regional Internet service providers (ISPs), who serve as content access points for regional end users [17]. Furthermore, although CDNs have are widely deployed for the distribution of static content, they were not designed for the distribution of live, streaming content.

Other methods of content distribution are based on IP multicast. This requires integration and support within backbone and regional ISPs [18]. Furthermore, IP multicast uses a subscription-based model that is reliant on timers in order to detect when a router no longer requires a particular content stream for downstream distribution [19]. This delay can result in unnecessary use of network bandwidth when there is zero demand for content. And lastly, IP multicast has not achieved widespread adoption, due in part to its complex architecture and configuration requirements.

This dissertation explores the use of SDN as a demand-driven mechanism to efficiently distribute video content to users at scale. The solution proposed combines the effectiveness of a distributed architecture proposed by CDNs with novel use of SDN in place of IP multicast. The result is an architecture that makes efficient use of available network bandwidth for the delivery of content to the user. It can be deployed in the cloud to ride over top of backbone networks and ISPs, eliminating the need for integration into physical forwarding devices. An overlay is advantageous, since many modern forwarding devices have limited or insufficient SDN capabilities for the deployment of the proposed solution. Like a CDN, for integration into an ISP, the proposed solution imposes the requirement of a single server connected to the video distribution network, in contrast to IP multicast solutions that require complex per-forwarding-device configuration.

1.4 Problem and Objectives

A fundamental problem limiting the three requirements of user and application mobility, high throughput data transfer, and efficient and scalable content distribution is a lack of centralized control over network forwarding devices. These three requirements have traditionally been tackled around the existing, rigid network infrastructure. In the recent years, SDN has taken a great leap forward with the advent of OpenFlow [10] – a SDN protocol. OpenFlow allows software developers to extend their algorithms into the network infrastructure by providing an interface over which network forwarding elements, such as switches and routers, can be programmed from a centralized control framework. This adds flexibility and extends programmatic control into the network data plane, which until recently was closed and difficult to manipulate in a uniform manner and on a large scale.

OpenFlow shows great promise as a SDN protocol [20] due to its simplicity and wide and increasing adoption by incumbent and startup networking vendors [21, 22]. It has even been adopted within the enterprise networks of largely application-focused companies [22].

Researchers have proposed a number of use cases for OpenFlow, which augment network infrastructure and architecture. To help facilitate these endeavors, Internet2 [23] at one time used OpenFlow to run their Advanced Layer 2 Service (AL2S) [24] and Advanced Layer 3 Service (AL3S) [25] networks, which provide high-speed network access to research institutions across the United States. Furthermore, the Global Environment for Network Innovations (GENI) [26] was founded as a future Internet testbed with SDN as a core technology. GENI is operated by OpenFlow and provides a service for researchers conducting experiments with SDN protocols such as OpenFlow.

The objective of this dissertation is to develop methods for a network infrastructure that allow end users to access their content and transfer data while on the go, provide a way for high performance data transfers to make better use of the available network bandwidth over large links, and allow live, streaming application data to be disseminated to large number of users, across geographically large networks at scale. The work in this dissertation achieves these goals and demonstrates their viability through the use of real-world deployments and analytical analysis of the network architecture.

Chapter 2

Background

2.1 User and Application Mobility

Vertical handovers, or handovers between different radio access technologies, cause interruptions in connectivity for mobile users during the process of obtaining a different IP address and during the radio access technology "switch" [27]. This results in the temporary loss of IP connectivity and in the inability for the network to deliver existing connections to the device's new attachment point.

Many handover decision algorithms have been developed to alleviate the resource allocation problem presented by heterogeneous wireless networks. Most of these algorithms require the ability to make a handover decision based on the current conditions of all radio access technologies. 802.21 [9] is a framework that was developed to support media independent handovers. The goal of this framework is to provide a standardized interface for every radio access technology that handover decision algorithms can utilize to create a global view of the available network states.

In commercial cellular networks, handovers are conducted between base stations using carefully crafted and specific routes in the network. Base station adjacencies are known, where connections are proactively routed to neighboring base stations based on an expected client migration pattern [28]. Any deviation from this pattern results in a dropped connection as the network works to recover the connections and reroute them to the unexpected new client attachment point.

As shown in Figure (2.1.1), Mobile IPv4 (RFC 5944) [5] provides a mechanism by which a mobile device can retain the use of an IP address even after it has associated with a foreign network. Upon migration, the mobile device reports its new IP address to the home network, and a tunnel is formed

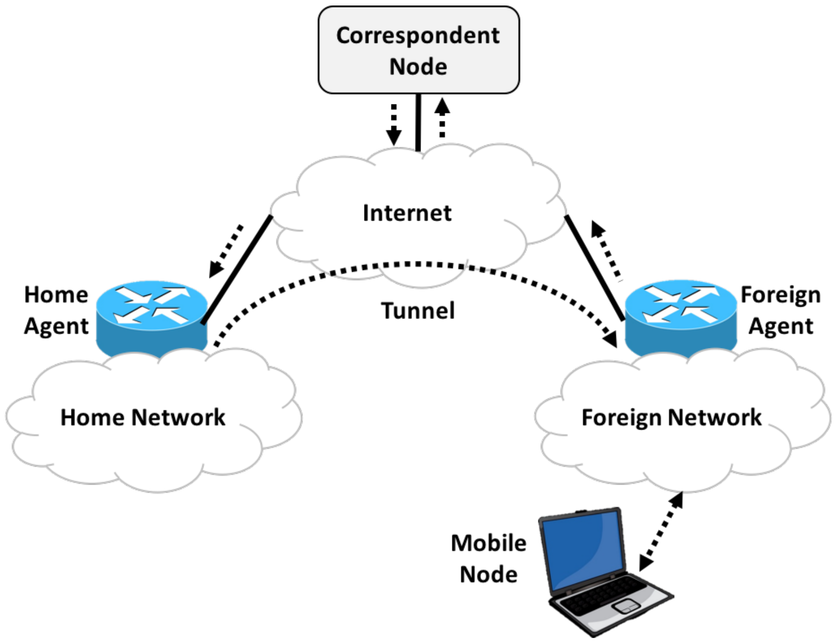


Figure 2.1.1: Mobile IPv4

between the mobile device and a home agent at the mobile device's home network. Egress traffic from the mobile device is routed normally, while ingress traffic is routed back to the home network and through the tunnel to the mobile device. The use of a tunnel creates what is known as the Mobile IP triangle routing problem, which adds delay, increases overhead due to encapsulation, and consumes extra network resources. The introduction of IPv6 features to Mobile IP (MIPv6) [7] running on an IPv6 network can alleviate the triangle routing problem through the use of what is known as route optimization. Mobile IPv6 route optimization utilizes binding messages exchanged between the mobile host and the correspondent host. The mobile host notifies the correspondent host of its new IP address using a binding message that includes this new IP in the form of a "care of" address. After processing this binding message, the correspondent host then relays all packets to the host at the "care of" address instead. A limitation of these MIPv6 schemes is that they require custom software on the client to enable mobility and have yet to be fully developed and integrated into many network infrastructures for support on forwarding devices and on the server-side hosts.

The WiRover [29] project at Wisconsin-Madison is a system that utilizes multiple radios to increase the bandwidth and continuity of wireless network access for buses. WiRover uses pre-collected signal data along bus routes to allow their system to proactively make an intelligent handover decision. Although the WiRover project itself is does not facilitate vertical handovers, the researchers' previous efforts in [30] include using the aggregation and simultaneous use of 3G and WiFi networks.

There are several existing heterogeneous wireless network testbeds that provide vertical handover capabilities. For example, [31] uses MIPv4 to achieve IP mobility, enabling the researchers to evaluate novel handover decision algorithms. Other testbeds, such as [32], make use of MIPv6 and its route optimization features, as described above; however, they do not allow for mobility in an IPv4 environment.

The work in [33] started the discussion of how handover can be implemented as a network service. The researchers describe how the client-level component proposed in this dissertation could participate in a Handover as a Service scheme, using a central database to make the handover decision for a mobile host based on the mobile host's current location and historical network information for that location.

2.2 High Throughput Data Transfer

It is common for data transfers that can tolerate periodic data loss, such as video streaming, to use User Datagram Protocol (UDP) for data delivery. One lost UDP packet, translating to one lost video frame, for example, is not typically perceptible to the end user. However, a lost packet during a file download can result in a corrupt or incomplete file. As such, data transfers that require the complete, ordered delivery of data to the destination must use a transport protocol, such as TCP, that provides such guarantees [12].

Although TCP provides a guarantee of intact and ordered data delivery to the application, TCP's windowing algorithm restricts the flow of data and reduces the data transfer rate upon a loss through its additive increase multiplicative decrease congestion control policy [34]. Solutions such as selective acknowledgement [35] and additive increase smooth decrease [36] have been proposed to help reduce the effects of packet loss over a TCP connection by allowing for selective retransmission of data and utilizing a larger TCP window as compared to the traditional algorithm during loss situations, respectively. TFRC (TCP-friendly rate control) [37], WARC (window averaging rate control) [38], HERC (high speed equation based rate control) [39] as well as CUBIC [40], BIC-TCP (binary increase congestion control) [41], and H-TCP (high speed TCP) [42], to name a few, are additional TCP variants designed to alleviate the common "sawtooth" pattern exhibited by TCP. They overlook periodic TCP losses and attempt to maintain a stable TCP window size. Other solutions such as TCP quick timeout (TCP-QT) [43] aim to improve the time required for TCP to detect a data loss and initiate a retransmission. However, despite such improvements and even on reliable networks, TCP still struggles to keep up with the pace its unreliable data transfer counterpart (i.e. UDP) can achieve.

Although network hardware has increased in speed, sophistication, and reliability, such as slow dial up and DSL being phased out in favor of fiber to the home, an ever-present problem in computer networks is latency. The decrease in throughput due to latency is becoming an increasing problem with the increased requirement to transfer science datasets such as telescope imagery and physics and DNA data sets exceeding exabytes [44]. Such large data sets are unable to be effectively transferred over existing networks [45], and oftentimes the physical transportation of hard disks is seen as a more appealing solution [44].

The use of multiple, simultaneous TCP connections has been proposed and is proven as a viable solution to increasing the throughput of such large data transfers over high latency networks [46]. Many parallel TCP solutions are discussed in the literature, such as Globus GridFTP [47], which also supports file striping from parallel servers in addition to the use of multiple TCP connections. Furthermore, although it is not TCP-based, Aspera [48] has developed a rapid and reliable file transfer technology that operates using UDP for data transfer along with TCP feedback connections that ensure reliability. Aspera is a well known and successful technology; however, it is proprietary and requires custom software installed on both the client and the server conducting the data transfer. Like Aspera, other UDP-based technologies exist, such as UDT [49] and FDT [50], but they suffer from the same portability problem, requiring application layer solutions on the client and server devices. pTCP is a parallel TCP solution that utilizes different network interfaces to transmit multiple TCP connections across multi-homed devices, thus making use of the aggregate bandwidth available [51]. There are many MP-TCP (multi-path TCP) implementations [52, 53] that take advantage of multiple links and in-network paths in order to increase aggregate bandwidth. These can be and are often used on top of network layer load balancing solutions such as ECMP [54].

SDN has been used for data transfer applications. In particular, studies have been conducted on the use of parallel TCP and the minimization of its effect on background network traffic [55]. OpenFlow, due to its comprehensive network knowledge, has been proposed as a solution to help identify and alleviate bottlenecks introduced by the use of parallel and multipath TCP [56]. However, OpenFlow, or SDN in general has not been proposed as a mechanism to seamlessly implement parallel TCP to improve TCP throughput.

2.3 Efficient and Scalable Content Distribution

As the demand for multimedia content over the Internet increases, researchers have been exploring options for the efficient distribution of content at scale and with high quality of service. Current state of the art solutions consist of vast CDNs, where content is oftentimes hosted in duplicate at various geographically distributed locations. This serves as both a load balancing solution, as well as a means to reduce access latency [17]. Users in a given geographic area are served by the local content node or cluster of nodes. Such a solution often requires collaboration with ISPs in order to push the content closer to the edge [17]. Although CDNs are a popular content distribution solution that work well for static content, they are not designed for the delivery of live, streaming multimedia content.

IP multicast has been considered for the distribution of live video content. IPTV has been proposed to use IP multicast to provide television service to users around the world [57]. Multicast Backbone (MBone) [58] is another well-known IP multicast deployment used primarily for experimentation. IP multicast's distributed publish-subscribe algorithm is a natural fit for the distribution of content. Hosts utilize IGMP to advertise and request multicast content groups on demand [19]. Routers then utilize one or more protocols to fetch the requested content for users. A multicast tree is constructed between the participating routers, and reverse path filtering prevents the formation of loops.

In practice though, IP multicast has not gained wide support, for which there are many reasons. First, network routers must be configured to support IP multicast, which is often a challenging task [18]. Unlike IP unicast, IP multicast does not have a clearly defined service model. While IGMP is the dominant protocol used for host-to-router IP multicast, there are a vast array of implementations for intra-domain IP multicast, such as PIM-SM [59], PIM-DM [60], MOSPF [61], and DVMRP [62], to name a few. Inter-domain IP multicast also has many protocols, for example MSDP [63] and BGMP [64]. Since a well-defined IP multicast solution does not exist, many Internet service providers do not support IP multicast at all or support it in a limited fashion as per the IP multicast implementation chosen [65]. Inter-domain IP multicast use is further limited in practice, due to security and content management concerns [18].

Although IP multicast has shown promise as a video delivery solution, it is complex and there are inefficiencies that can result in excessive network bandwidth usage. Anyone can join a multicast group and anyone can send to a multicast group [18]. It is the job of the subscriber to filter the desired content, which means undesired content must traverse the network prior to being discarded. Furthermore, although studies have been done on IP multicast congestion control [66], a solution has yet to be widely accepted. Lastly, when leaving an IP multicast group, timers in hosts and routers are used to trigger stream termination. This allows IP multicast streams to persist for a period of time and utilize link bandwidth when there is no demand by users from a downstream router [19].

Chapter 3

An Agent-Based Framework

SDN enables users and operators with great freedom to customize their networks to support a wide range of applications and services. SDN's decoupled control plane in the form of a centralized controller makes it convenient to programmatically achieve application-specific, hard-to-do-today traffic forwarding methods [67]. For example, SDN has been envisioned to be useful for network operators to perform fine-grained traffic engineering, for applications to incur very high throughput data transfers, and for users to easily and consciously stay connected to the network on the move. Network Function Virtualization (NFV) is yet another new paradigm powered by SDN to implement scalable network services. However, despite them being feasible, production offerings of such services remain slow to enter. Among all factors, the lack of a simple yet unifying framework for deploying, managing, and maintaining the wide range of such services has critically contributed to the slow progress. While protocols such as OpenFlow offer the lowest, and the most crucial, level of abstraction that enables SDN, there remains the need of an overarching framework to provide clarity of the development, deployment, and life-cycle management process for all entities that constitute the large SDN ecosystem.

Without such a framework, "motivating demos" of such services [68, 69, 70, 71] have often been derived using a variety of methods, which do not have a standard at this point, such as (1) do-it-all in the controller, including making per-packet decisions for all packets via OpenFlow packet-ins, sometimes due to certain packet mutation or header match operations not implemented on the SDN switch, (2) design for a single domain without discussion of its coexistence with today's non-SDN networks, especially the Internet, (3) overlooking scale up paths needed in large production offer-

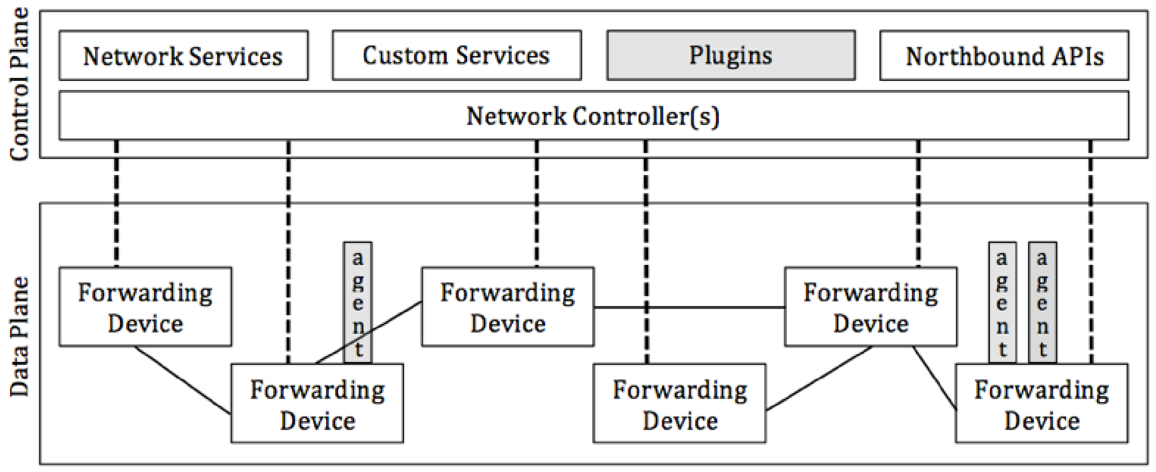


Figure 3.0.1: The agent-based framework for a SDN

ings, (4) bypassing discussion of the means of authentication, authorization, and accounting for such services (who, how, where), and (5) the use of vendor/experimenter extensions to add features to the OpenFlow protocol requiring non-standard support at the switch and the controller. These limitations result in control plane bottlenecks through excessive control plane processing of data plane packets and are oftentimes customized to a particular deployment. Furthermore, emphasis is seldom placed on the manageability of such use cases in production environments across network domains with diverse SDN capabilities, such as their installation, upgradability, scalability, elasticity, security, and fault tolerance. Through the agent-based solution proposed in this dissertation, these shortcomings can however be systematically addressed with a common software agent-based architecture. This common software agent-based architecture is discussed in Sections 3.1, 3.2, and 3.3.

Early research on SDN has largely focused on the southbound protocol architecture, i.e. controller-to-switch communication protocols such as OpenFlow, and the abstraction it exposes. The agent-based framework shown in Figure 3.0.1 is defined to supplement such architecture in order to address the common need of the integration of custom applications within the data plane independent of the southbound protocol. Functionally, the agent complements any SDN forwarding device to enable application-specific customization. The SDN controller can interact with the application-specific agents to perform packet manipulation using defined application program interfaces (APIs). To arrive at the agent-based solution, diverse SDN applications were studied and the core requirements of the various stakeholders were identified. These stakeholders can be largely categorized as: (1)

SDN operators, (2) application providers, and (3) application users, each of which imposes distinct requirements on a production SDN. SDN operators require a solution to manage a diverse set of custom applications and enhance performance. Application providers require a solution that facilitates uniform application design across various SDN deployments, provides scalability, elasticity, as well as one that provides persistent network connectivity. Application users require enhanced quality of service.

The framework has the following advantages: (1) it addresses the manageability of custom SDN applications by providing a modular framework for design, deployment, and resource allocation; (2) it removes controller bottlenecks caused by per-packet operations; (3) it addresses scalability and elasticity by allowing a variable number of agents to be deployed proportional to the capacity and scope of the application. The framework also has the flexibility of being deployed with software switches running on servers, which can also address hardware-specific limitations of a particular environment.

It is useful to note a parallel effort that also serves to address complementary needs of the ecosystem. The Open Networking Foundation (ONF) defines an agent-based architecture [72] where agents are communication interfaces in the controller and network elements to abstract away the southbound protocol from the controller and switch implementations. At the same time, the Open Networking Lab develops has been developing the Open Networking Operating System (ONOS) [73], which provides the foundation for building highly available SDN controllers that can operate across wide geographical areas controlling large numbers of OpenFlow forwarding devices. The agent-based framework proposed in this dissertation focuses on the structural division of responsibilities among customizable software agents and SDN controllers, whereas the agents proposed by the ONF serve a different purpose to abstract away southbound protocol implementations. Figure 3.0.2 shows southbound agents, such as those proposed by the ONF in comparison to the agents proposed in this dissertation. Throughout this dissertation, the Floodlight open source SDN controller [74] is used to demonstrate the interaction of the agent framework with a controller in the SDN architecture.

In the following sections the three stakeholder requirements are addressed -- of the SDN operators, the application providers, and the application users -- and how they define and validate the agent-based SDN framework. The technical components of the agent-based framework follow.

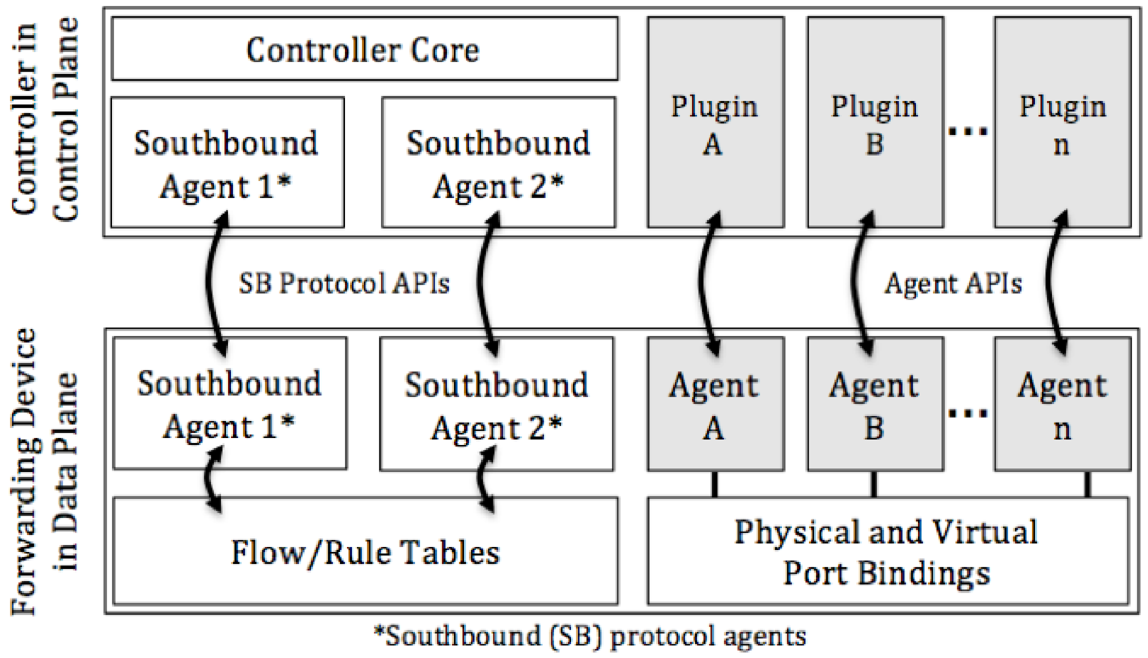


Figure 3.0.2: A comparison between southbound protocol agents and the proposed framework's agents

3.1 SDN Stakeholder Requirements

The agent-based framework proposed is derived from the needs of three comprehensive and unique perspectives -- (1) the SDN operators, (2) the application providers, and (3) the application users. The following discuss the requirements of each stakeholder and how the agent-based architecture addresses these requirements.

3.1.1 SDN Operators

SDN operators require a solution to manage a diverse set of custom applications and enhance performance. However, in order to allow applications and services to utilize the network, SDN operators need a production-quality method to manage these applications. Without this, routine procedures such as installing, updating, running, and otherwise maintaining applications could jeopardize the stability of the SDN. The agent-based framework proposed allows SDN operators to maintain the network and its applications in a modular way. Similar to how applications can be installed and maintained on general-purpose servers, the agent-based solution allows applications to be manipulated independent from one another such that the maintenance of one does not impact others.

SDN operators also have an interest in maintaining a reliable and efficient network that meets the needs of the applications that run on top of it. This includes orchestrating traffic engineering and big data transfers, such as efficient packet handling and optimal bandwidth utilization across large networks. Some of these items are addressed by applications implemented over present SDN protocols, such as OpenFlow. However, with the agent-based framework, these SDN operator requirements can all be implemented. Agents are deployed within the SDN to accomplish a range of network engineering tasks from simple load balancing to more complex tasks such as high-throughput data transfer. SDN operators can take advantage of the programmability of the agent-based framework in order to deploy custom applications of their own as underlying services to enhance the performance of higher-level consumer services.

3.1.2 Application Providers

Application providers require a solution that facilitates uniform application design across various SDN deployments, provides scalability, and provides elasticity. When application providers launch new SDN-based applications, they need these assurances in order to provide a consistent and reliable experience to their customers. An agent-based SDN framework provides a simple and universal way of integrating applications into the SDN. Application providers are able to concentrate more on the quality of their services and less on accounting for the various flavors of SDNs and their peculiarities. The agent-based framework also affords the deployment of multiple instances of agents throughout the SDN and across multiple SDNs. Furthermore, it is not rigid and allows agents to be allocated and decommissioned dynamically as application demand fluctuates. As such, the agent-based approach provides a scalable and elastic solution sought by application providers.

Application providers also demand the underlying network provide persistent network connectivity. For example, they need to ensure their services can operate over the network in mobile environments. The advent of laptops, tablets, smartphones, connected vehicles and other mobile devices has created an unprecedented demand for mobility support within the network. Multiplayer games, messaging applications, social networking applications, and video streaming applications, just to name a few, all require a persistent and reliable connection to the network in order to operate as designed. Solutions to connection persistence across networks do exist; however, an accepted solution does not exist to address migration across networks of different technologies. The ability to handover between heterogeneous networks increases the device's (and thus the application's) connection alternatives and allows an application connection to persist across various access technologies, as they are available. These requirements can be solved using the agent-based SDN framework proposed.

3.1.3 Application Users

Application users require enhanced quality of service. When they connect to the network, they desire fast access to content and services. This requires the underlying network be able to handle the demands of individual users on a large scale without negatively impacting other users on the network. Among many techniques, content delivery networks (CDNs) have been used to make this desire a reality and are deployed and used in production networks and services today. The CDN concept can be realized within the agent-based framework, where content can be processed by agents and delivered to the application users, resulting in potentially less load on edge servers and less network bandwidth usage. The client or customer needs only to request the content, which the agents can then react to and deliver in the most optimal manner. As such, the agent-based framework can be used to satisfy the quality of service requirements of application users.

3.2 Framework Components

As shown in Figure 3.0.1, the agent-based framework augments the existing SDN architecture with an agent component in the data plane and a plugin component in the control plane. These agents and plugins complement each other in a customizable relationship i.e. one-to-one, one-to-many, many-to-one, many-to-many, etc. to form the agent-based framework. Here, the design is broken down, with a discussion on the details of each component and how they come together to form the agent-based framework. Figure 3.2.1 shows the relationship between the logical components of the agent-based framework. Note that in subsequent figures, plugin-to-agent API links are omitted for brevity.

3.2.1 The Agent

The agent is functionally a software component associated with a forwarding device. The forwarding device can direct data to and from one or more agents, where the agents perform advanced packet processing. These agents can be installed as a part of the forwarding device itself, or they could be

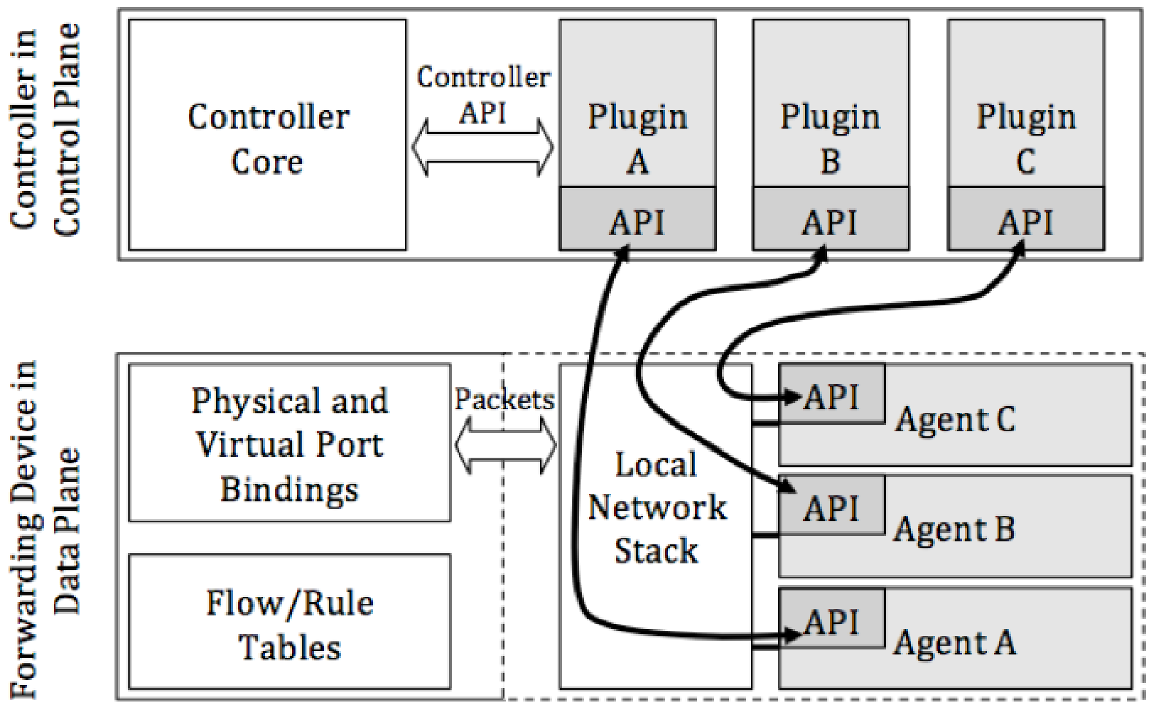


Figure 3.2.1: The logical components of the agent-based framework. The dashed lines indicate the agents can be within a forwarding device or on a separate machine attached to the forwarding device.

attached to the forwarding device in the data plane. The hardware on which the agent is configured is not rigidly defined and can be anything from a switch to a general-purpose server. On this hardware, multiple agents supporting one or more applications can be installed and managed independently.

The intent of any agent is to operate on packets within the network. As such, the agents tap into the underlying SDN using standard socket programming techniques and the local network stack of the agent's host. The forwarding device sends packets to and from the agent's host using rules installed in the data plane by the controller plugin. To allow packets to pass between the agent's host and the agent's network sockets, packets must be able to be delivered to the agent processes onboard the host. This involves configuring the agent's host operating system (OS) to route applicable packets to and from the data plane network interface(s) and the agent(s).

The controller plugin is responsible for communicating all relevant information to its agent counterpart(s) in the data plane so that they can operate on packets in the prescribed manner. An API is required between both the plugin and the agent to relay such information. A representational state transfer (REST) API is readily available in many OpenFlow controllers, such as the Floodlight controller. REST is recommended due to its ease of implementation and management, however any

API can be used.

The agent's custom software is not constrained in any way at the application layer. Agent software design is only limited by the host OS and the hardware capabilities of the machine on which the agent is hosted. As such, an easy-to-manage/customize OS is recommended, Linux for example, and hardware such as central processing units (CPUs), random access memory (RAM), network cards that can support the processing of packets at the rate required by the installed agent application is also recommended for optimum performance.

3.2.2 The Controller Plugin

In order to orchestrate the routing of packets to and from the agent software in the data plane, the control plane must also be augmented with a plugin. Figure 3.2.1 also shows a network controller, where the controller has been extended to include plugins. These plugins work in tandem with the agents to form the agent-based framework. Due to the modular nature of many network controllers, such as the Floodlight controller, plugins can be readily configured in the control plane without modifying or interfering with other controller functions. However, plugins need not be installed on the controller directly. Controller APIs may be leveraged to implement the control plane agent as a northbound application. The high availability concept can be implemented in the control plane to install, update, and otherwise modify plugins in a production SDN.

The controller plugins can be either reactive or proactive, depending on the use case. In the reactive case, OpenFlow packet-in messages can be received by the plugin and analyzed, for example. Rules can then be inserted to direct applicable packets to the corresponding agent installed in the data plane. In the latter case, the plugin proactively inserts rules to direct the packets of interest to the agents. In either case, if the agents require information from plugins in order to operate on the packets, an API can be used, such as REST, to relay the necessary information. In order for the agent's host OS to receive the packets directed to the agent application, the data plane rules inserted by the controller plugin may require the use of destination MAC and/or IP address rewrite actions.

3.2.3 Inclusion of a Software Forwarding Device

The inclusion of a software forwarding device in the agent-based framework is not a necessity; however, it can greatly increase the usefulness of the agent deployment. Present day hardware forwarding devices sometimes hinder SDN deployments by supporting limited feature sets of southbound protocols. If such missing features are required by the agent application, the use of a software forwarding device can overcome this limitation. The modular design of the agent-based framework allows for a software forwarding device to be used, and in the future, replaced with a hardware forwarding device as missing features required by the agent gain hardware support. The use of a software forwarding device also has the advantage of being able to be updated as new protocols and features are implemented. With respect to OpenFlow, a software forwarding device such as Open vSwitch [75] (OVS) supports all OpenFlow matches and actions and is readily extended with new OpenFlow features as subsequent protocol versions are released.

To use a software forwarding device in the agent-based framework, a general-purpose server can be used as a host for both the agent(s) and the software forwarding device. Data plane network interface(s) are attached to the host and wired to the software forwarding device installation. An OpenFlow software forwarding device such as OVS contains a LOCAL port, which is attached to the local network stack of the host of the software forwarding device installation. Packets can be relayed to and from the software forwarding device's LOCAL port and agent network sockets using the host machine's loopback interface. This requires the host machine's route table to be configured to pass these packets.

3.3 Deployment with a SDN

In order for the agent-based framework to operate, the use of a SDN infrastructure is a necessity; however, it is not required that the entire network be a SDN. Non-SDNs can be interconnected with SDNs in a larger network infrastructure equipped with an agent-based framework deployment. The agent-based framework assumes there is a controller plugin to insert rules that intercept and route packets to and from the agents within the data plane, rewriting packet headers if necessary. At the very least, the point of packet interception in the network must be under the control of a SDN.

When deploying the framework with a SDN, there are many items to consider. The recommended "recipe" for deploying agents within a SDN is to consider the purpose of the application, the data it will handle, and its requirements. Applications requiring low-latency will dictate where the agents are physically installed. The amount of data to be processed by the agent can also influence where the agent is installed, on what type of hardware, and the number of agent instances. If the goal of the agent(s) is to process traffic into and out of a particular sub-network, the agent(s) might need to be deployed at each ingress/egress point of the sub-network. On the other hand, if the goal of the agent(s) is to process packets from devices attached to the network, perhaps one agent or a collection of agents needs to be at each geographic edge. Similarly, the volume and rate of data the agent and the plugin are required to process can also impact the choice of hardware such as network interface cards and quantity of agents. Lastly, depending on the method and frequency of communication between the agent(s) and the plugin, more than one plugin might be necessary to balance the load and to scale.

This dissertation leverages a common agent-based framework in order to address data movement for user and application mobility, high throughput data transfer, and scalable content distribution. Application providers require persistent connectivity and mobility support over heterogeneous networks. The agent-based framework can be used to address these requirements by providing handover as a service to connected devices [33]. Likewise, high throughput data transfer is important for SDN operators to transparently and reliably enhance data transfers that traverse the network. Lastly, content dissemination at scale is a requirement of application users. The agent-based framework is used as the foundation for an SDN-based content distribution service. In all three of these data movement studies, the agent-based framework is used as a common platform in their SDN implementations. The agent, the controller plugin, and the inclusion of a software forwarding device are common themes and are present in each solution discussed in the following chapters.

Chapter 4

User and Application Mobility

Users desire persistent connectivity while on the go. As they migrate from one network to the next, they expect their applications to continue to operate seamlessly. Although applications oftentimes attempt to overcome this problem, some applications do not or do so unnecessarily. Through the use of SDN, a network architecture can be implemented such that user devices can continue to transmit and receive data as they migrate from one access location to the next or from one access technology to another, all while maintaining application sockets. An SDN-based solution is proposed to address the following issues:

- Handover architecture complexity. A single network controller can be used to manage all clients.
- Scalability. The architecture proposed can support clients at scale.
- Handover in vehicular networks. Data from the device can be leveraged to assist the network in proactively configuring data flows prior the handover.

The work so far has been published in [76]. In this chapter, Section 4.1 discusses the mobility problem, Section 4.2 introduces an SDN-based solution to the mobility problem, and Section 4.3 discusses scalability of the mobility solution and evaluation in a real-world deployment.

All source code for the handover implementation can be found at `http://www.github.com/rizard/GENIVerticalHandover`. It is broken down into client level and network level components.

4.1 The Mobility Problem

Present heterogeneous wireless networks provide vast connectivity options for users. In a smart phone context, users can choose to connect to their provider's cellular network or to connect to various WiFi access points. Oftentimes, users migrate from network to network while using an application onboard their smart device. As they migrate in and out of range of networks, the device must switch to provide the application with a network connection. However, due the combined effect of routing within the network and the fact that these access technologies are likely under the control of different administrative domains assigning different IP addresses in different subnets, the user's device will not be able to maintain its application connections after a network switch occurs. This break in existing application connections is oftentimes masked from the user by the application being "smart" enough to reestablish broken sockets with the correspondent node.

In an ideal scenario, adhering to the layered nature of the network stack, the application should not have to be responsible for recovering from handovers. The network should be able to provide the application with a guarantee of continuous connectivity regardless of any changes to network connectivity that occur during the life of the application's connections.

Another issue present in existing mobile network architectures is the use of carefully crafted handovers specific to deployments. Network providers use sophisticated yet complex solutions to make an educated guess on where network users are going to move next. If the user does not follow one of the predicted patterns, dropped connections will occur as a result. SDN, can be used to reduce the complexity of the handover algorithm and allow for user-tailored handovers as opposed to fitting the user into a set of predefined handovers. This can be especially problematic in vehicular networks where there are a potentially large number of clients moving at high speed.

4.2 An SDN-Based Mobility Solution

The SDN-based mobility solution utilizes OpenFlow to achieve IP mobility and application transparent handovers. It is designed to be easily deployed on top of networks that are OpenFlow-enabled or that can provide a subnet or VLAN for handover experimentation. The framework was developed and implemented at Clemson University as a part of the GENI WiMAX project. Figure 4.2.1 provides a general overview of how the framework is constructed and how it integrates with the

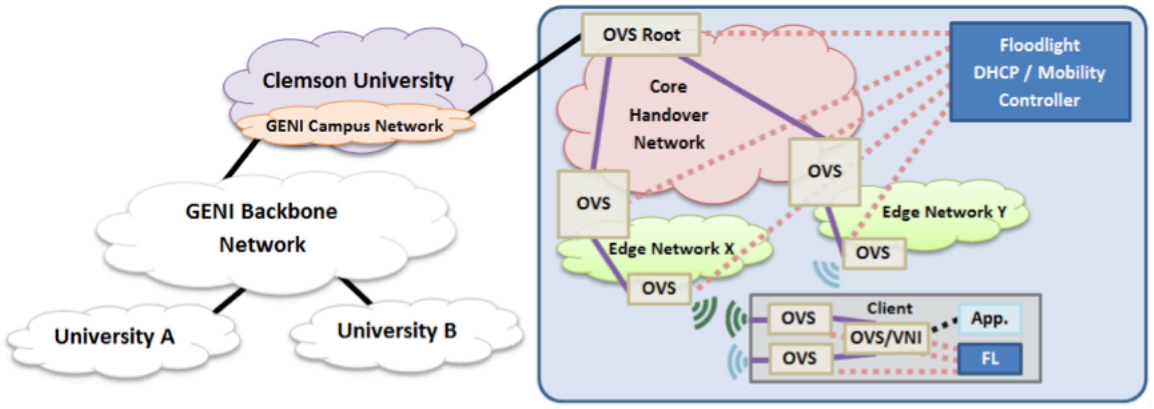


Figure 4.2.1: Handover system architecture as deployed at Clemson University. The blue-shaded right half of the figure indicates the SDN-based solution as a part of the larger campus network.

large-scale GENI testbed. Figure 4.2.2 provides a more general vantage point of the system architecture. At the architectural level, the framework allows OpenFlow-enabled mobile devices to roam across any OpenFlow-enabled wireless network. It requires one or more root OpenFlow switches at the testbed ingress/egress, as well as OpenFlow switches at each edge network. The challenge in designing a deployment based on the framework is the required integration with existing network infrastructure where the deployment is to occur. The framework allows a mobile device to roam from wireless Network X to Network Y (as illustrated in Figure 4.2.1), preserving end-to-end socket connections that the device has with other hosts located either on the campus network or in the Internet. With these minimal assumptions, the framework can be as simple or as complicated as desired. The design of the framework can be divided into two major components: the network level and the client level. In Figure 4.2.1, the SDN-based solution is shown shaded in blue, as a part of a larger network.

The network level component is required to manage and maintain client IP addresses, as well as the routing of client packets within the framework. Both of these tasks are performed with a Floodlight OpenFlow controller [74], which maintains a global IP address pool and handles migration events within the testbed. To maintain the global IP address pool, a custom DHCP server module is integrated into Floodlight. The Floodlight controller is designed to be extendable to support other use cases; for example, in a handover as a service framework, the Floodlight controller could also make the handover decisions for the mobile devices in the network [33]. A key component of the network level is one of more OpenFlow-enabled switches or OVSs [75] located at the root, such that

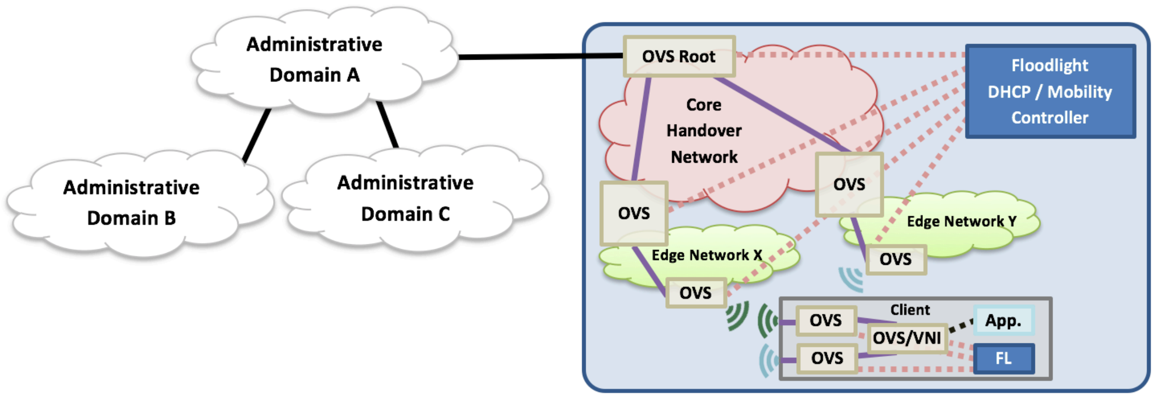


Figure 4.2.2: General handover system architecture. The blue-shaded right half of the figure indicates the SDN-based solution as a part of a larger network. For brevity and clarity, it is only depicted in Administrative Domain A; however, it is also present in each administrative domain.

all IP mobility-enabled networks on the edge are descendants of this root. As descendants of the root switch, other OpenFlow switches or OVSs are deployed in the network level in order to both forward client packets and detect migration events. From a network operations point of view, benefits of this tree-like design include (1) a single point of integration with the network infrastructure and (2) the requirement of no specialized hardware in the case where OVS is used in preference over physical OpenFlow switches.

The client level component of the testbed exists entirely on-board the client and is responsible for both switching the active physical interface and maintaining all client sockets during such a handover event. To maintain active sockets, a default virtual network interface (VNI) is installed on the client. All applications send and receive packets through this VNI, and by nature of a virtual interface, it is not impacted by physical interface states. The client is also equipped with an OVS and its own Floodlight OpenFlow controller. This controller is responsible for forwarding packets from the VNI's OVS to the physical interface of choice as determined by a handover decision. Further discussion on the importance of the VNI is presented in Section 4.2.2.

4.2.1 Network Level

On the network level of the testbed, all WiFi access points (APs) are configured with Debian Linux 5.1.10, and all WiMAX gateways are configured with Debian Linux 6.0.7. On the client level, testbed components have been verified on both Debian and Ubuntu Linux. All Linux distributions are using

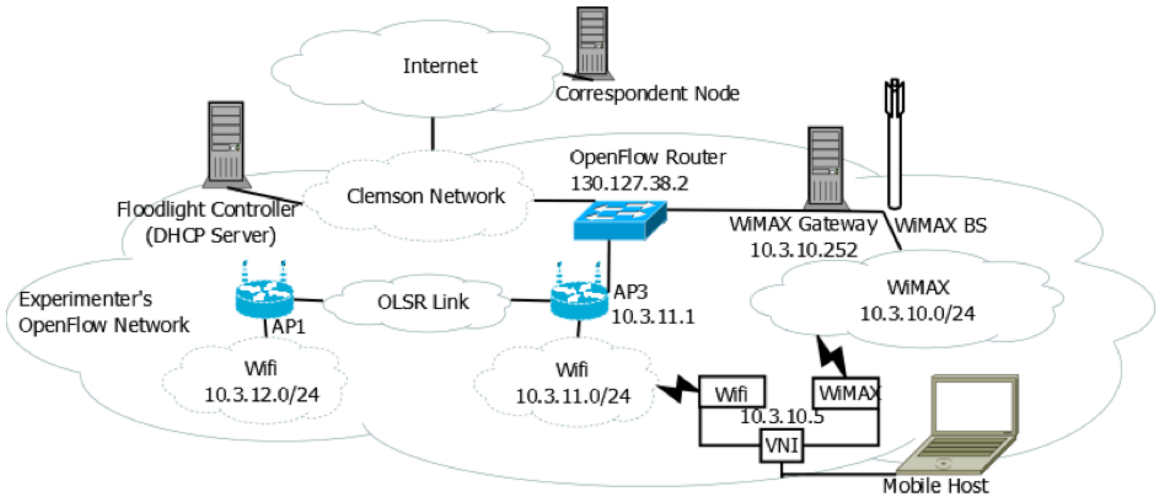


Figure 4.2.3: Network level architecture as deployed at Clemson University

kernel 2.6.32. The Floodlight OpenFlow controllers used on both the client and the network levels are sourced from Floodlight v0.90. Each controller has been extended with custom modules to enable the handover solution. Also common to both the network and the client levels are several OVS 1.9.0 software network bridges (OVSBs). A high-level diagram of the network level is shown in Figure 4.2.3, as deployed at Clemson University.

Within the framework, the network component has the responsibility of maintaining the IP address pool for every mobility-enabled network. The network level Floodlight controller acts as a DHCP server, using DHCP requests as a trigger for migration. In the event of a migration, this Floodlight controller is also responsible for efficiently and quickly updating the client's location and thus the flow of its application packets. The detection of a client connection and migration within the framework is achieved through the use of OVSBs and OpenFlow flows. These flows detect, encapsulate, and redirect client DHCP packets (on UDP destination port 67) to the network level Floodlight controller. This controller contains an integrated DHCP server module, which unlike traditional DHCP servers, associates an IP address lease with multiple MAC addresses. Each of these MAC addresses corresponds to a participating network interface card (NIC) on the client. When processing a DHCP packet, the controller cross-references the MAC address of the DHCP packet with all available MAC address lists. Upon a successful match within a MAC address list, the controller assigns the client who initiated the request the corresponding IP. Then, upon a mobile host's initial connection or migration to a foreign network, flows are inserted in OVSBs starting at the framework root and along every hop to the client's current location. These flows direct packets to and from the mobile client within the framework. When a client migrates away from this network, any existing flows associated with the client are removed and replaced with flows along the path from the root to the newly-migrated foreign network. The use of a root switch and tree hierarchy allows the network level controller to avoid undesirable triangle-routing in the event of a migration.

The mobility framework includes many OVSBs within the network. As discussed previously, the network level OVSBs connect to the network level Floodlight controller / DHCP server. These OVSBs are used in the detection of client migrations and the routing of client packets into and out of the framework. Specifically, the OVSBs on the framework edge detect client migration by intercepting DHCP request packets, while the OVSBs in the core direct the flow of client packets from the framework root to the client on the framework edge.

Each network level node with OVSBs also uses OVS patch ports (OVSPPs). To ensure proper routing of packets destined for an IP not routable by a foreign network, OVSPPs are used to connect the external and internal facing OVSBs installed on the gateways/APs. This allows independent subnets to operate within the framework. The OVSPPs, combined with flows that utilize these OVSPPs, force client packets to bypass Linux routing on each hop, thus supporting cross-subnet compatibility upon migration from the home network.

4.2.2 Client Level

Any mobile device should be able to connect to a network in the handover framework and maintain an IP through a vertical or horizontal handover. However, if the handover is to be truly seamless to an application, there needs to be a persistent VNI for the application to use. The VNI abstracts the handover from the application and provides the application with an interface that is persistent for the duration the client is active. In addition to the VNI, the client should also be able to switch between interfaces in a manner that is simple and straightforward to the client device using the framework. Similar to the network level design, OVSBs are also utilized in the client to achieve a seamless handover. These client level OVSBs are used in conjunction with a client level Floodlight OpenFlow controller and are installed for each mobility NIC on the client, as shown in Figure 4.2.4. The local Floodlight controller inserts flows in each OVSB via the integrated Static Flow Pusher. These flows route application packets from the client VNI to the NIC of choice. When a decision is

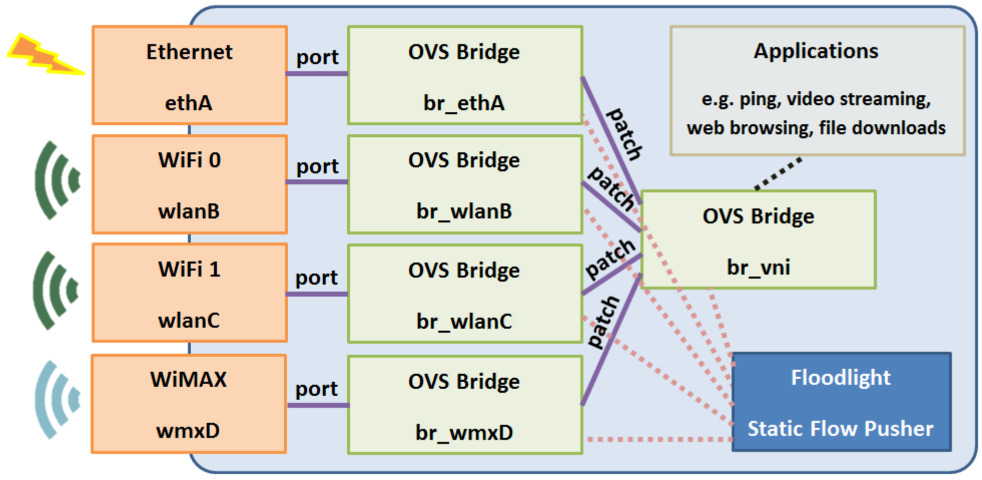


Figure 4.2.4: Client level architecture

made to switch NICs, the client will issue a DHCP request egress the new interface, which will then trigger the aforementioned events in the network level. As a result, these OVSBs with Floodlight-inserted flows allow the client to seamlessly switch from one network to another. All client level tasks are encapsulated in shell scripts to provide framework client devices with a simple and single command to execute a handover. Each client level OVSB also contains OVSPPs. To ensure proper routing of packets from the VNI to the NIC of choice, OVSPPs are used to connect the VNI OVSB with the OVSB of each participating NIC installed on the client, as shown in Figure 4.2.4. The OVSPPs, combined with flows that utilize these OVSPPs, serve to link the VNI to each NIC's OVSB. These flows define the path (and thus the NIC) used by application packets.

The use of a VNI introduces a problem when associating with networks and routing packets to the client from the network level. The MAC address of the VNI must be the same as that of the NIC, otherwise WiFi APs and other access mediums will not accept packets from or know how to route packets back to the client's VNI at the link layer. It is not reasonable to require the modification or "spoofing" of each NIC's MAC address to that of the VNI. The client level solution to this problem is to perform MAC-rewrite within the client OVSBs. When an application generates packets, they are routed out of the client via flows on each OVSB. These flows contain actions to rewrite the source MAC address of all egress packets from that of the VNI to that of the NIC. The flows also contain actions to rewrite the destination MAC address of all ingress packets from that of the NIC to that of the VNI. This rewrite process allows the client to send and receive packets from its VNI

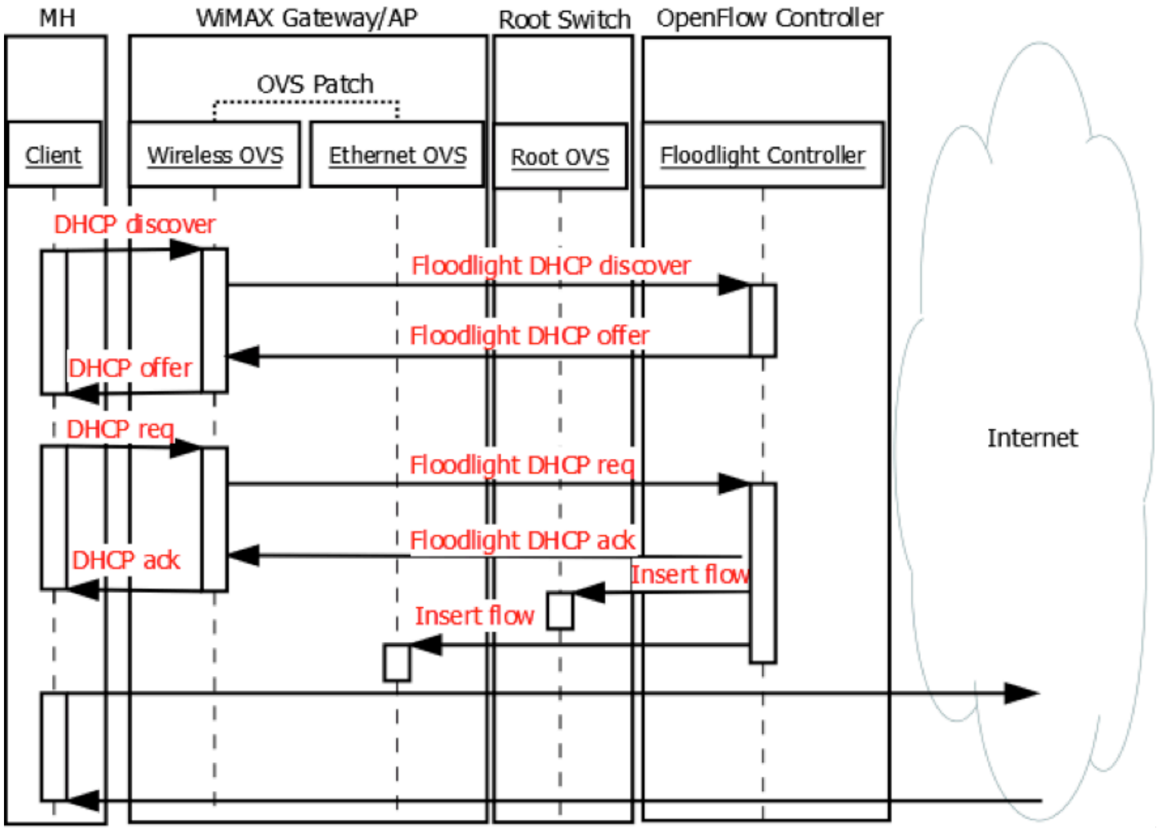


Figure 4.2.5: DHCP Signaling between the Client Level and Network Level Floodlight OpenFlow Controllers

with any associated network on the link layer. Due to a limitation of OpenFlow 1.0, ARP packets cannot be rewritten with flows; they must be processed instead by a controller. Thus, the client level Floodlight controller contains a custom module to rewrite all ARP packet MAC addresses within the controller itself. Although out-of-band processing of packets is inefficient as compared to in-band, ARP packets are not frequent, so an occasional rewrite within the controller is a compromise made in the client level implementation. With the advent of OVS OpenFlow 1.3 support, ARP rewrite can be performed using flows. This can reduce the client level complexity and eliminate ARP packet processing in the controller.

4.2.3 Network and Client DHCP Signaling

The sequence of events that result in a reactive handover as a client migrates to a foreign network is shown in Figure 4.2.5. To summarize the interaction between the two components, the events are as follows:

1. The client establishes layer 2 connection with the network and issues a DHCP discover
2. The network level Floodlight controller intercepts the DHCP discover packet, allocates an IP, and responds with an offer
3. The client responds to the offer by sending a DHCP request
4. The network level Floodlight controller intercepts the request, triggers a migration event, and sends a DHCP ACK to the client
5. The client receives the DHCP ACK, establishing layer 3 connectivity. Meanwhile, the network level Floodlight controller inserts flows at the root and gateway OVSBs and removes any existing flows belonging to the client.

After this process completes, the client will have established full network connectivity on the new attachment point through the root node.

4.2.4 Handover Architecture Complexity

The SDN-based handover solution discussed is architecturally simple and is based on the agent-based framework. It consists of a single control entity in the network and a single controller on the client. These controllers interact with switches to control the flow of traffic to and from the client. With such a simple architecture, only flows must be modified when a handover takes place. Figure 4.2.6 depicts the handover architecture in the context of the agent-based framework.

The framework also provides a convenient way to detect client initiated handovers via DHCP. Network initiated handovers can also be conducted with minimal signaling between the client and network controllers.

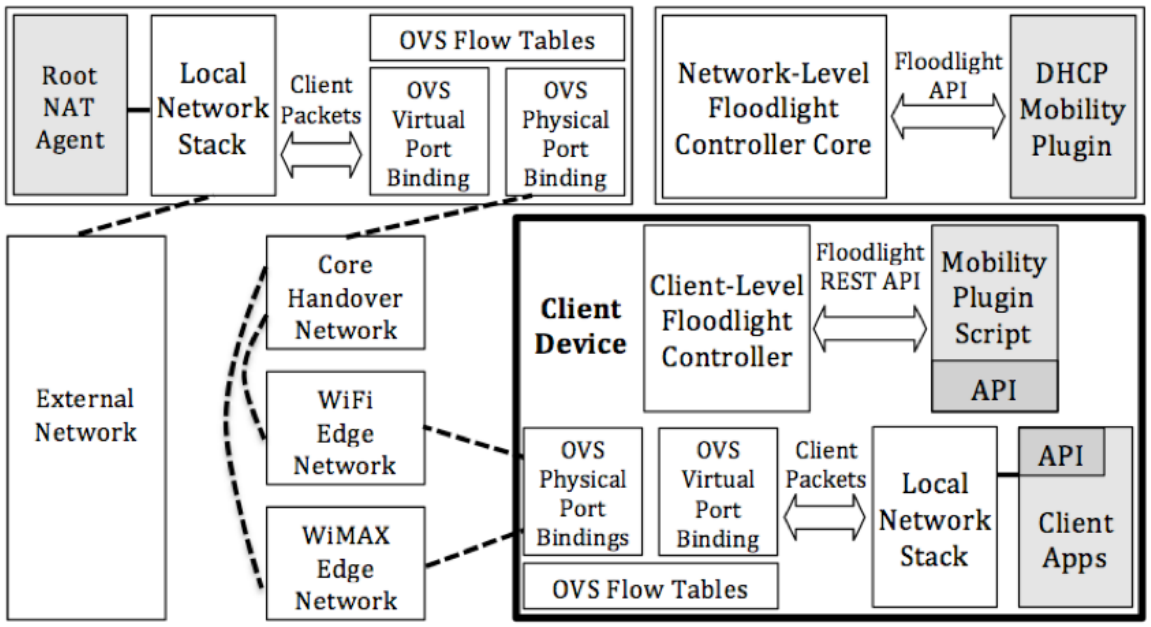


Figure 4.2.6: Influence of the agent-based architecture on the handover solution

Vertical and horizontal handovers are seamlessly supported. The underlying access technology is abstracted away in the SDN-based implementation.

4.3 Scalability and Evaluation

4.3.1 Scalability through Inter-Domain Handover

As clients migrate from network to network, connectivity must be maintained regardless of the network domain. In the real world, control needs to be maintained from within administrative domains. Service providers are unlikely to support a model where they cede control of their network to some centralized network level controller that governs the entire network. Such an implementation would require one or more parties relinquish control over some or all their domain forwarding devices. Furthermore, such an implementation cannot scale with a single centralized controller. The OpenFlow-based handover implementation can be used as a distributed solution across multiple administrative domains without the relinquishment of intra-domain control from any party. The problem and solution can be dissected into (1) inter-domain signaling and (2) handover delay between domains. These two problems are addressed first, followed by an example illustrating the inter-administrative domain handover process.

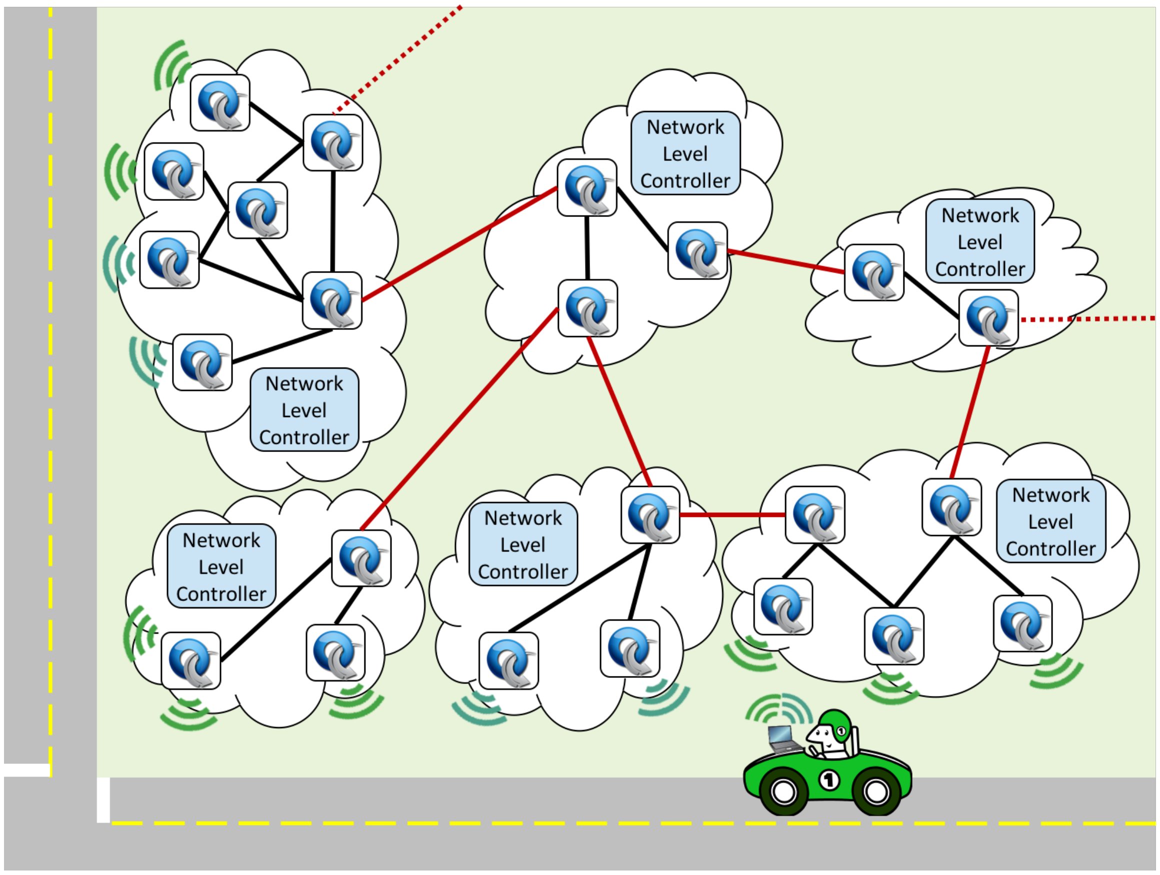


Figure 4.3.1: Inter-domain handover architecture enables handovers between administrative domains

4.3.1.1 Inter-Domain Signaling

The inter-domain handover problem only exists for clients that have migrated from one domain to another – say domain A to domain B. The client connections need to be rerouted from domain A to domain B, where the network level controller in domain A only has domain A visibility and the network level controller in domain B only has domain B visibility.

The solution lies within the design of the network level. As shown in Figure 4.3.1, each administrative domain can have one or more roots to its network under the control of the network level controller. Traffic that is egress the root(s) leaves the control of the administrative domain's network level controller, while traffic that is ingress the root(s) enters the control of the administrative domain. These network roots serve as bridging points between providers. Flows from one root can direct traffic into the root of another provider via root-to-root network links.

As a simple proactive example, when a migration between administrative domains occurs, a client must issue the DHCP request in order to start the network level handover process. This DHCP request will enter an access point of the adjacent administrative domain and will be forwarded to the adjacent network level controller. This controller needs to "know" who this client is in order to allocate the same IP address and insert the appropriate flows in the network. As such, the network controllers need to be capable of sharing client information with each other.

Various algorithms can used to accomplish such client information sharing. One such implementation is reactive in nature. As a client hands over from administrative domain A to administrative domain B, it sends a DHCP request to domain B. When the DHCP request is received from the unknown client, the network level controller in domain B issues a query to the geographically neighboring network level controller(s) for information on the client. When a neighboring controller from domain A who is presently serving the client receives such a query, the controller can respond with the IP address of the client, such that the new network level controller in domain B can assign the same IP and allow the client to perform the transparent handover. In such an approach, the domain A's network level controller reacts to the inter-domain migration and redirects client packets from its root to the root of the new, neighboring administrative domain B. The domain A controller is also be responsible for removing any local, intra-domain flows present for the client, as the client is now under the control over the adjacent administrative domain's network level controller in domain B.

The problem with the aforementioned algorithm is that it is reactive, which could introduce undesirable switching delay. A proactive alternative can be used to alleviate such delay. Periodic, location-triggered client information sharing can be performed between neighboring administrative domains. An administrative domain is aware of the network connections of its own clients that are adjacent to other administrative domains or on its geographic edge of control. As an example, consider a heterogeneous mesh network. The administrative domain that owns such a network knows which connections exists in the "middle" or "core" of the mesh versus at the "edge." It can then track the client's physical location within its domain using access point or base station association and/or GPS and inform known neighbor network level controllers when a client either is located at an edge or has a trajectory towards an edge. When the new, neighboring administrative domain receives the client's DHCP request packet, it will already be aware of the client as provided by its old, neighboring administrative domain. As such, a more expedient handover can be achieved.

4.3.1.2 Handover Delay Between Domains

To reduce the latency between a handover, the client's connections can be proactively duplicated and routed to the new anticipated attachment point before the client arrives. This process is simple for an intra-domain handover, since the network level controller has authority to directly add all necessary flows. In a single administrative domain, the network level controller can simply insert duplicate flows along the anticipated route of the client's new attachment point.

For an inter-domain handover between administrative domains, the local network level controller needs to inform the neighboring administrative domain network level controller of an incoming client connection. The client's local administrative domain's network level controller can duplicate and route the client's ongoing connections from it's root to the root of the neighboring administrative domain, which can then route the connections to the client's anticipated location. Information such as GPS location can be included to more precisely determine exactly where the client will attach to the new administrative domain. If the handover is network initiated, the two network level controllers can negotiate a new client attachment point and inform the client. Such a scenario could further reduce delay be increasing handover precision. More on the specific approach to proactively routing client traffic prior to conducting a handover is discussed in Section 4.3.2 with particular attention to vehicular networks.

4.3.1.3 Example Inter-Administrative Domain Handover

In this section, an example handover between administrative domains is performed. The purpose is to illustrate the steps taken and signaling performed by the client's origin and adjacent administrative domains. Each of the following five subsections indicates a logical step in an inter-administrative domain handover.

Client is Connected to Origin Administrative Domain (t=0)

First, assume the client is connected to an origin administrative domain and is communicating with a correspondent node. This state is illustrated in Figure 4.3.2.

Administrative Domain Notifies Adjacent Administrative Domain of Client's Upcoming Handover (t=1)

Since the client is on the edge of and leaving the origin administrative domain, the origin administrative domain notifies the appropriate neighboring administrative domain to prepare for a handover. As shown in Figure 4.3.3, the notification message is relayed from the network-level controller of the origin administrative domain to the adjacent administrative domain, where it is received as a packet-in event by its network level controller. This notification message includes information such as the client's list of MAC addresses, its assigned IP address, and any correspondent nodes with which it is communicating.

Adjacent Administrative Domain Fetches Client's Connections (t=2)

The adjacent administrative domain network level controller proactively installs flows to receive the client's connections to any correspondent nodes, as depicted in Figure 4.3.4. To avoid triangle routing, it does this by forming shortest path trees between the adjacent administrative domain and the IP address(es) of any correspondent nodes. These trees are computed based on the routing entries of each hop. When a hop is reached that has the client's connection(s) already, a flow operation to split and duplicate the flow(s) is performed, forming temporary tree(s) of the traffic between the client and the correspondent node(s). Note that the proactive branch in these trees only receives traffic from the correspondent node(s). The client is still attached to the origin administrative domain at this point.

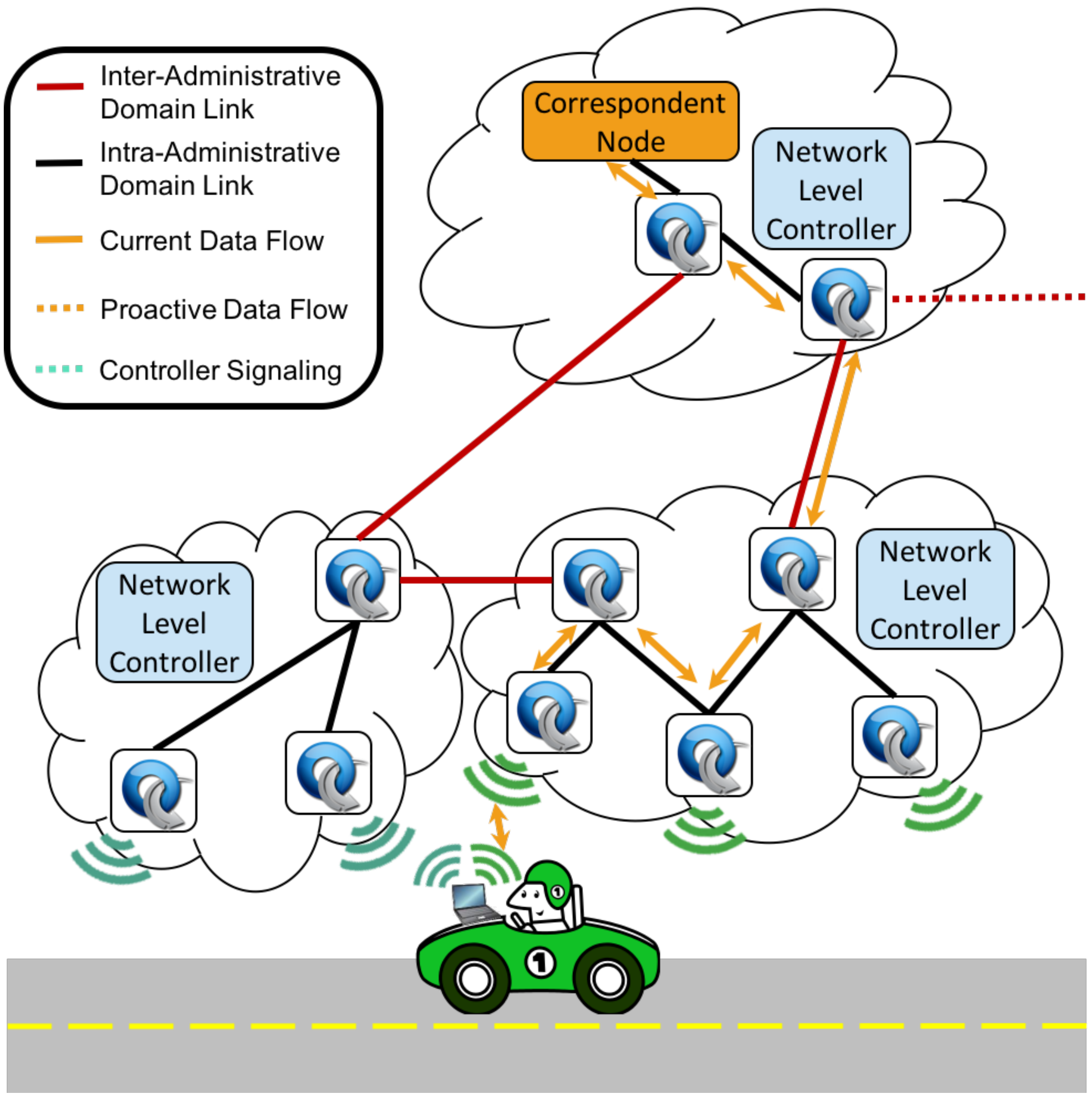


Figure 4.3.2: The client is connected to an administrative domain and the system in in a steady state.

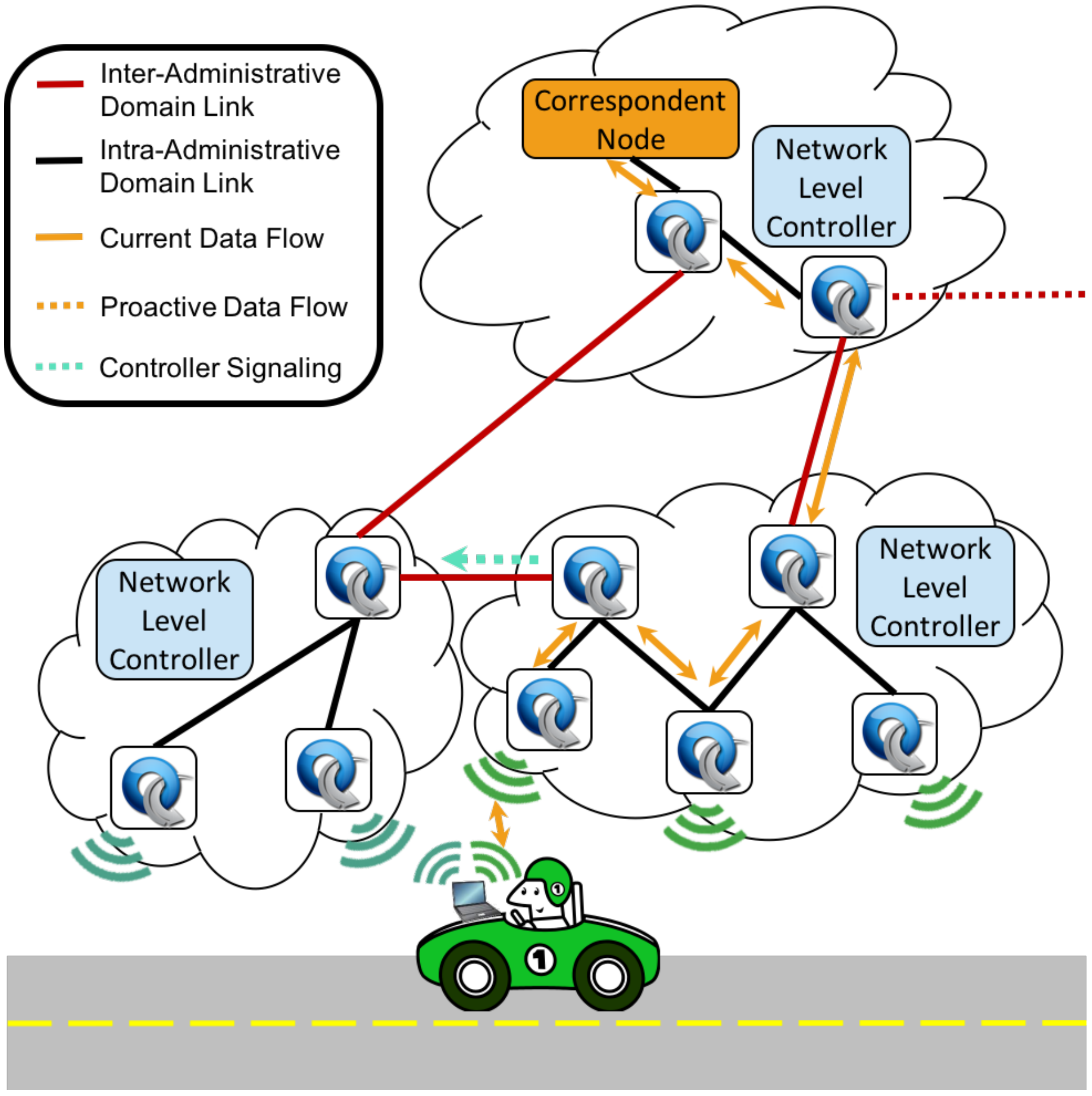


Figure 4.3.3: The client's current administrative domain notifies the neighboring administrative domain that the client is going to perform a handover.

Note that the client will be probing the adjacent administrative domain with DHCP discover messages to establish the new connection. The network-level controller in this adjacent administrative domain ignores these messages until it is ready to accept the client (at a later, upcoming stage).

Adjacent Administrative Domain Performs DHCP Handshake with Client (t=3)

When the adjacent administrative domain has completed fetching the client's connections to any correspondent node(s), the network level controller begins to process the DHCP discover messages from the client and completes the DHCP handshake, assigning the same IP address as advertised by the origin administrative domain's network level controller. Once this DHCP handshake is complete, the client level completes its handover and logically disconnects from the origin administrative domain. All communications between the client and any correspondent nodes occur through the adjacent administrative domain, as shown in Figure 4.3.5.

Adjacent Administrative Domain Notifies Origin Administrative Domain the Client Handover is Complete (t=4)

Next, as shown in Figure 4.3.6, the adjacent administrative domain relays a handover complete message to the origin administrative domain.

Origin Administrative Domain Tears Down Old Client Flows (t=5)

In the last step of the inter-administrative domain handover process, the origin administrative domain's network level controller tears down the client's old flows. It notifies each upstream administrative domain of the client's correspondent node(s) that the client's handover is complete so that it can also remove flows. Once this operation is complete, the network is once again in a steady state. Figure 4.3.7 represents this new steady state where the client's new origin administrative domain is now the former adjacent administrative domain. Logically, this is the same state as Figure 4.3.2.

4.3.2 Handover in Vehicular Networks

The handover solution proposed can be used to facilitate both horizontal and vertical handovers in vehicular networks, where factors such as GPS location, speed, velocity, and final destination can be used to assist in how the handover is conducted. Each component – both client level and network

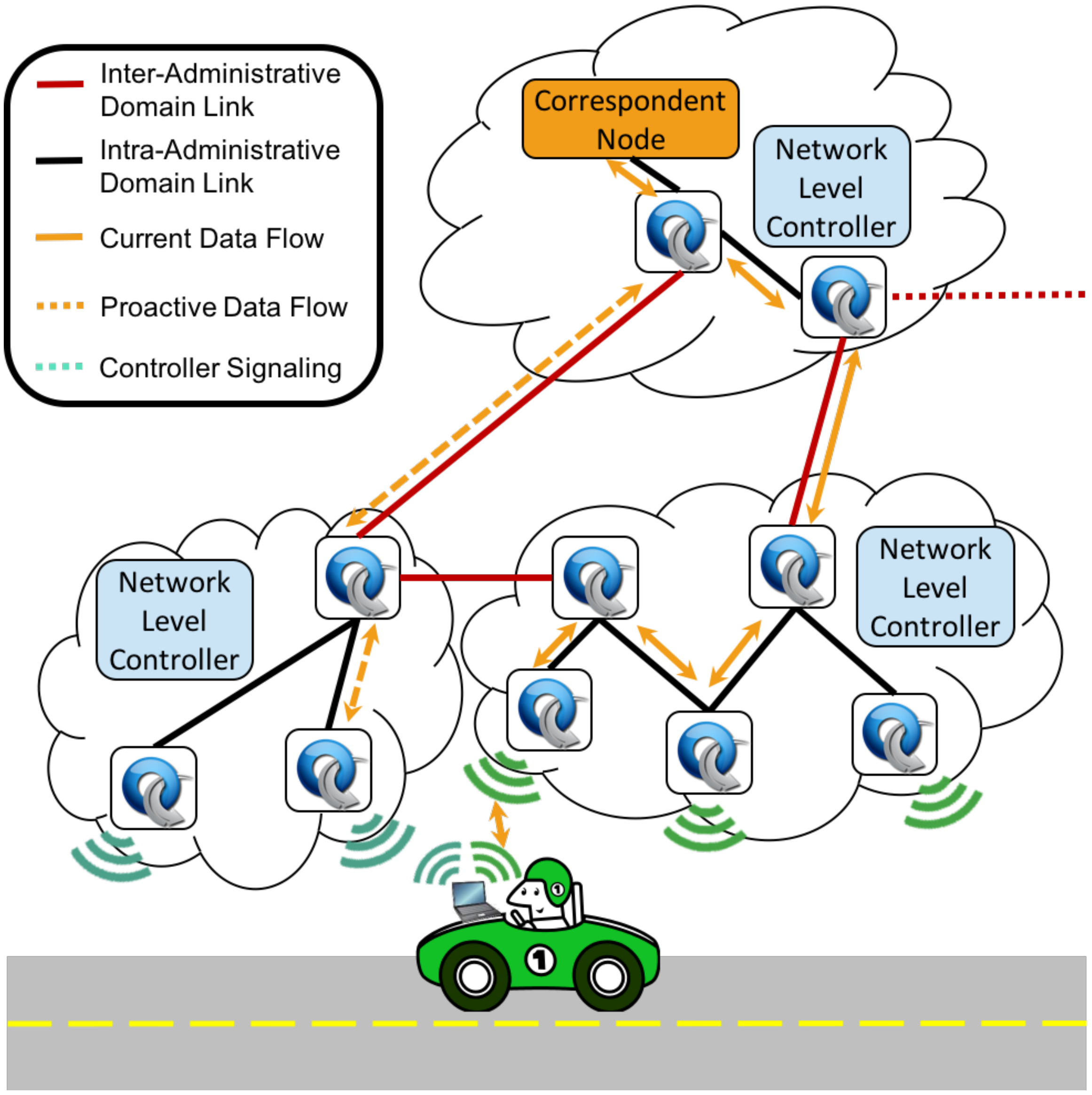


Figure 4.3.4: The adjacent administrative domain proactively fetches the client's connections to any correspondent node(s).

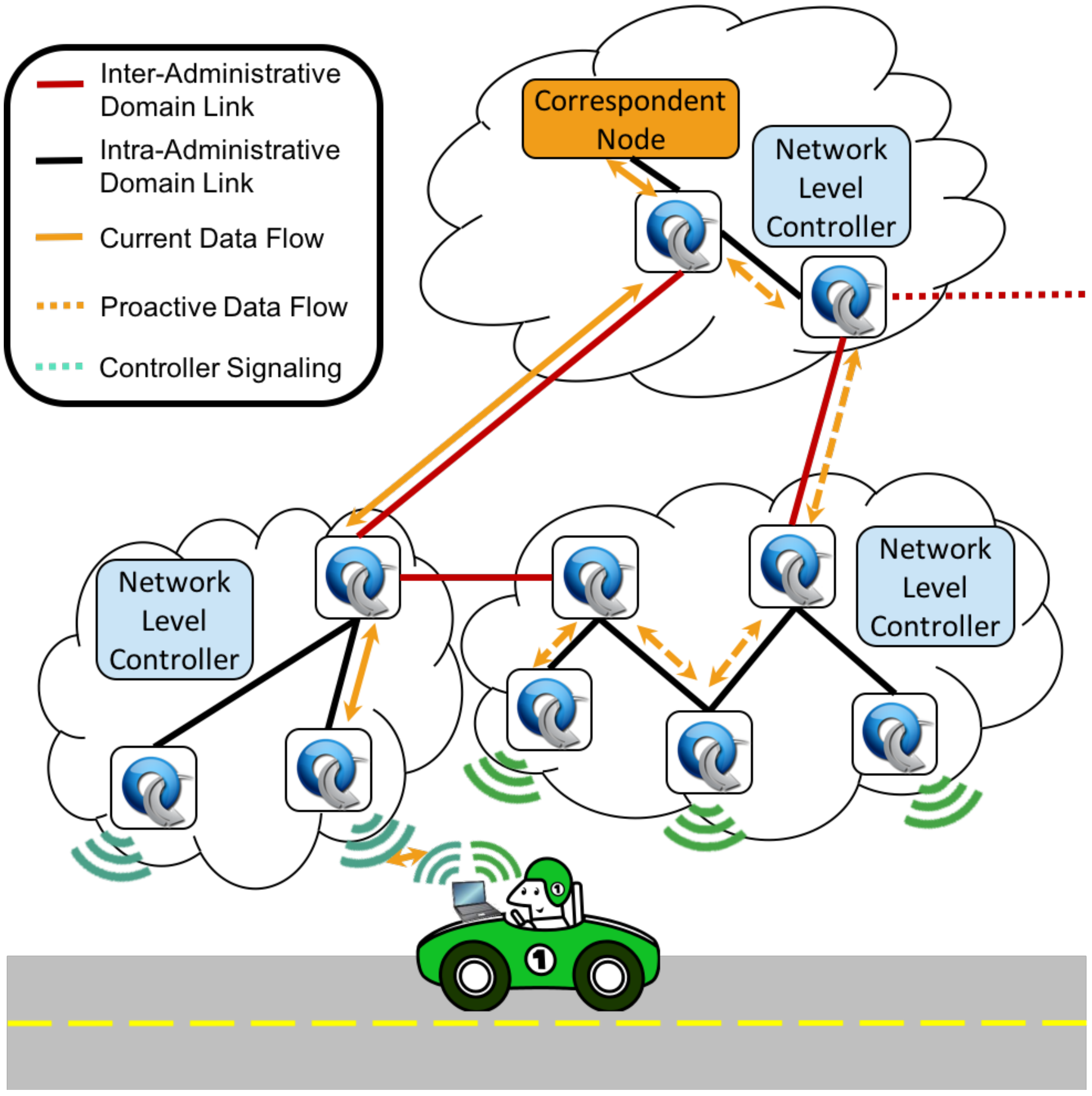


Figure 4.3.5: The adjacent administrative domain accepts the client's DHCP-initiated connection, resulting in the client-level handover to occur.

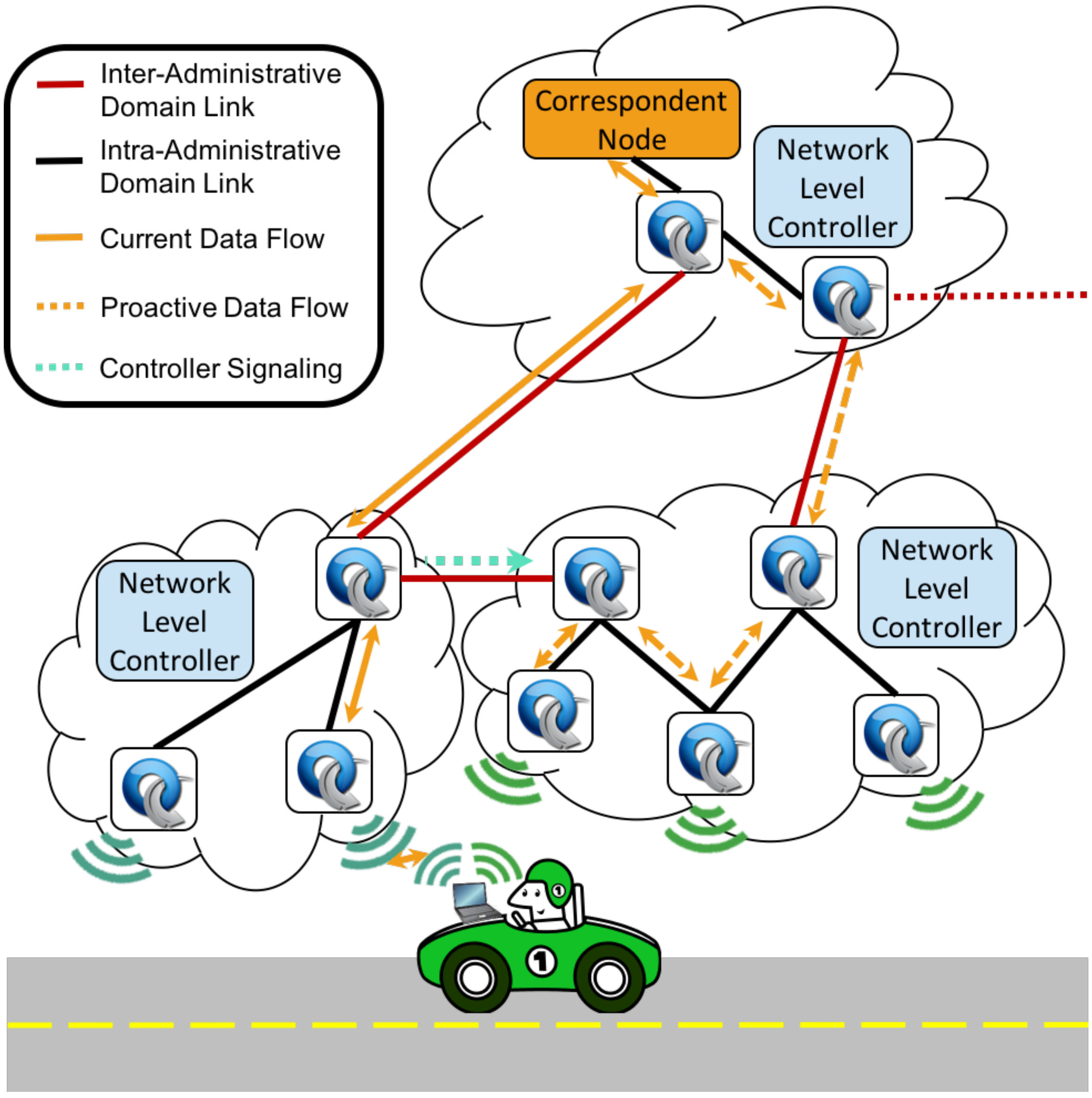


Figure 4.3.6: The adjacent administrative domain notifies the origin administrative domain that the client has completed the handover.

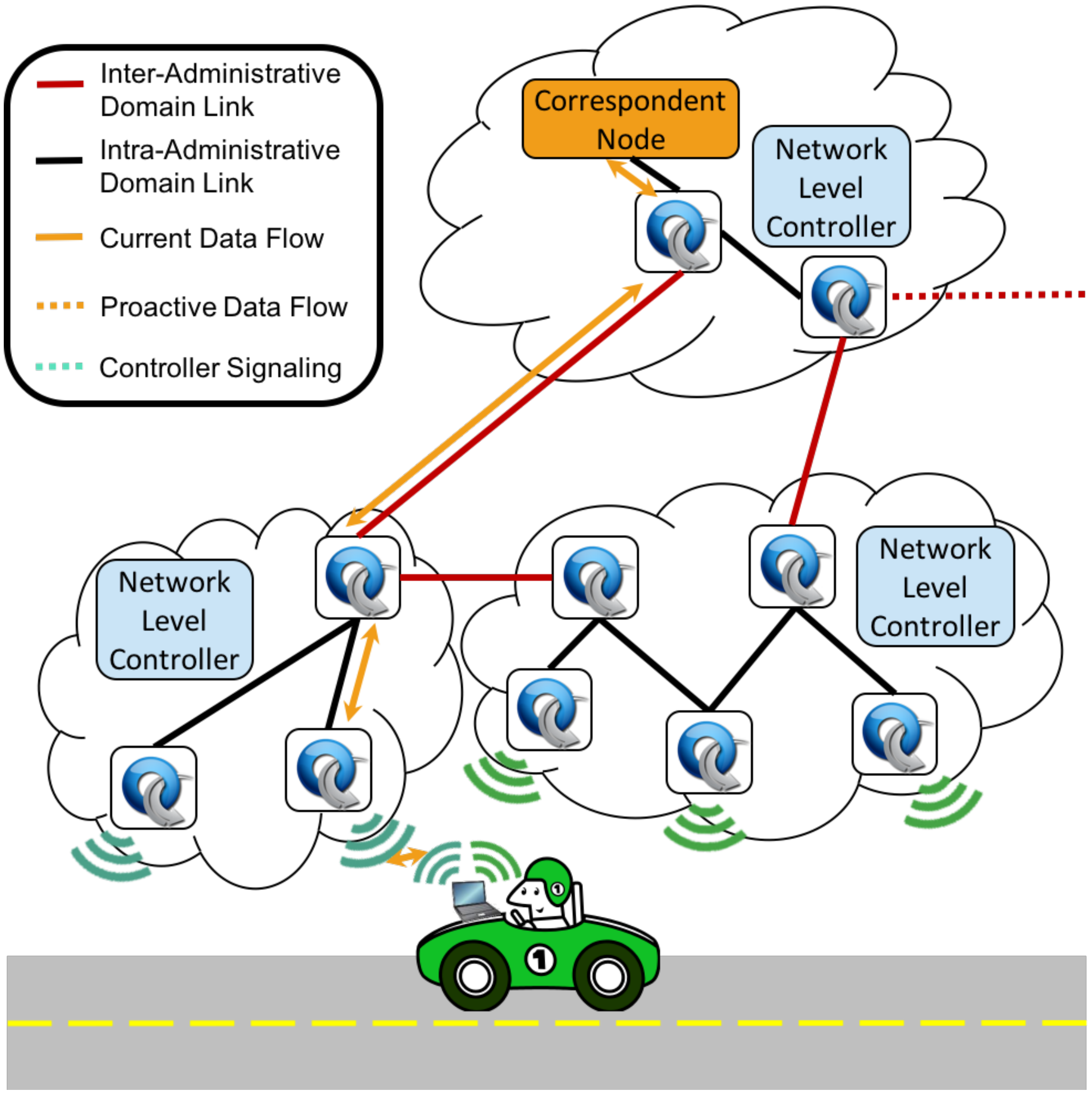


Figure 4.3.7: The origin administrative domain tears down the client's old connections, notifying any upstream administrative domains of the handover event.

level – is critical to its successful and seamless handover execution. An assumption is made that physical and link layer connections will be made prior to the handover. Information on available connections can be located in either a cloud-based database and pushed to the client from a source such as CyberTiger [77]. Alternatively, network coverage information could be preconfigured on the client.

As the client migrates through the network, there are a number of algorithms that can be chosen to conduct the handover. This dissertation is not focused on the handover decision algorithm but instead strives to show how the generic and flexible SDN-based architecture proposed can be used to provide a basis for any network selection algorithm.

A vehicular network is considered to be the worst-case environment in which to perform a handover. Unlike handovers in fixed locations, factors such as the vehicle's trajectory, speed, and GPS location need to be considered in order to minimize the delay incurred during the handover and to handover to the appropriate attachment point. To minimize client-perceived handover delay, such information can be used to proactively route the client's network traffic to the appropriate nearby network edge where it will attach. As the client moves faster, it must have its proactive routes installed earlier, assuming a fixed time to deploy the specific proactive route and for the client's traffic to propagate down to the edge. There are many factors that influence how and when proactive flows should be inserted for a vehicle at speed in order to result in a seamless handover using the SDN-based framework. These factors can be logically broken into those that impact or are related to the client level and those that affect the network level. For the client level, factors related to how to conduct a handover with minimal delay must be considered. This includes any physical switching delay inherent to the client device. For the network level, the primary factors to consider when conducting a handover include the flow installation time, use of bandwidth, and propagation delay of the client's traffic to its new attachment point.

4.3.2.1 Client Level

Operating under the assumption the connections are ready at the network edge to which a client is about to handover, a make-before-break handover has no delay, since the network interface switch on the client is assumed instantaneous on a modern processor and machine. Furthermore, on the client level OVS where the switch takes place, all matching, arriving packets are halted for the flow modification operation and are then allowed to resume traversal of the network. (The flow modification operation is atomic.) This means no packets will be dropped with appropriately sized VNI and NIC queues. As such, note that such a make-before-break handover is an ideal case. There will be no client-perceived delay due to handover under such a scenario.

However, there is an inherent delay that will always be present when switching from one network to another when using the same physical interface both before and after the handover. This delay is a consequence of the time it takes to disconnect from the first network and associate, authenticate, and establish a link layer connection with the new network. This delay is well studied in the literature and is assumed to be the minimal delay for such break-before-make handovers in the architecture proposed. As such, the goal for break-before-make handovers is to show the minimization of delay to the point where switching delay is as close as possible this minimum. The handover architecture can facilitate a break-before-make handover introducing no additional delay beyond the aforementioned inherent delay if flows have been proactively installed to forward the client's traffic to and from its new attachment point. This is shown in a general case in Section 4.3.3 through a discussion of each component of the handover architecture, the flows required to proactively forward packets, and the actual mechanism that performs the switch.

4.3.2.2 Network Level

In order to conduct a handover with minimal delay, the network level setup time for proactive flows must be taken into account. This involves tapping into existing flows to duplicate the client's traffic and send the traffic not only to the client's current, active attachment point, but also to the upcoming new attachment point. The location of this "join" flow should ideally be the lowest cost path from the existing client attachment point flows to get to the upcoming attachment point. The definition of "cost" is beyond the scope of this dissertation, but it would likely include available link bandwidth, latency, and hop count.

The time it takes for the controller to modify this flow or set of flows to include an additional output port must be taken into account. If more than one switch must be involved in this process, the controller can issue the flow add and modify requests to the switches at the same point in time such that they can be executed in parallel on the switches. The flow requests are asynchronous, meaning the controller does not need to wait for a reply in order to proceed with the next operation.

In addition to the flow that serves to duplicate the client's traffic, flows must also be inserted to forward the duplicated packets to the edge of the network where the client is going to attach in the near future. These flow insertion operations are also asynchronous and can be done in parallel for each switch involved and also in parallel with the aforementioned flow modify operation. As such the time it takes to perform the proactive flow insertion can be modeled as the time it takes to add or modify a single flow. The add/modify time includes the propagation delay of the request from the controller to the switch and the time required to perform the operation on the switch itself. The propagation delay is readily know to the OpenFlow controller, as it tracks the latencies of all switches connected to it. The difference in time to add versus modify a switch flow table is assumed negligible in the OpenFlow switches. Since the flow modification operations are done in parallel the time to perform the proactive flow insertion/modification is the longest propagation delay from the controller to all switches involved in the operations.

In order for the client to "see" its packets when it performs the handover in the client leave, the propagation delay of the packets from the point in the network level topology to the edge where the client will attach after it performs the handover must also be included in the proactive flow setup time. This time is defined as the time it takes for a packet to traverse the links from the "join" location to the network edge of the future attachment point. It can be approximated based on the sum of the latencies between each switch in the new path from the "join" to the edge. These latencies are readily available to the OpenFlow controller and are based on the propagation delay of LLDP packets between the OpenFlow switches collected during link discovery and monitoring. This sum is an approximation, since as will be shown in Section 4.3.3, the actual propagation latency is more complex. It is closely related to the control plane latencies due to the fact that the flows are installed in parallel from the "join" to the edge switch at the new attachment point.

After the proactive flows are inserted and the client's traffic has propagated down to the edge, the client level of the handover solution can then perform its network interface switch. The delay of the flow modification on the client-level OVS is the time to issue the switch command to the client level OpenFlow controller, the time for the controller to issue the flow modification messages to the switch, and the time for the switch to perform the flow table modification. Because these operations are performed on the client itself, their impact on the switching delay is assumed to be negligible. However, the inherent delay to establish a connection to the new network cannot be neglected for a break-before-make handover.

4.3.3 Handover Analysis

In this section, an analysis of the handover architecture is performed. Analysis is done through two lenses. The first is connection establishment time. The time it takes for client to connect to the architecture and use the network is important, since it is directly related to the client's perceived initial quality of service. The second is handover execution, where both handover time and bandwidth are considered. From a client's perspective, the handover time should be as quick as possible. However, as will be shown, having the fastest handover temporarily increases network bandwidth consumption. As such, handover execution time is a tradeoff between the client's perceived delay and the network bandwidth consumed.

4.3.3.1 Connection Establishment Time

Let T_c denote the time to establish a client connection from the client device to the root of the handover network. $T_c = T_a + T_d + T_f + T_e$, where T_a is the time for the client to associate with the edge access point or base station, T_d is the time for DHCP to assign an IP address to the client, T_f is the time for the OpenFlow controller to install flows in the network from the root to the edge where the client is attached, and T_e is the latency between the client and the edge switch.

T_a depends entirely on the access technology and is beyond the scope of this dissertation.

T_d can be broken into each component of the DHCP messaging process, $T_d = t_{disc} + t_{offer} + t_{req} + t_{ack}$. More precisely, it is the time to relay the DHCP packets to or from the controller and the edge switch and the client, $T_d = (T_e + c_e) + (T_e + c_e) + (T_e + c_e) + (T + c_e)$. This simplifies to two RTTs between the client and the OpenFlow controller or $T_d = 4(T_e + c_e)$.

T_f can be computed from the control plane latencies between the OpenFlow controller and OpenFlow switches involved in the handover. The controller installs flows concurrently, so the time to install the flows is simply the switch with the longest control plane latency to the controller, $T_f = max(c_i)$, where c_i is the latency between switch i and the controller. Note that T_f can vary though depending on whether or not the handover type includes delivery of data to the client. This is discussed where relevant in the following sections.

4.3.3.2 Handover Execution

The time to simply perform a handover from one location in an administrative domain to another is simply T_c. However, the relationship becomes more complex if the client has ongoing connections that must be maintained with a remote machine. The handover time when taking this into account depends on a choice of either bandwidth conservation of handover seamlessness.

When performing a handover, there are two extremes to be considered with respect to handover optimization. The first is minimization of network-level bandwidth utilization, were the network-level is optimized. The second is minimization of client-level content disruption, where the client-level quality of service is optimized. In practice, a blend of these two approaches might be acceptable depending on the service contract between the network provider and its clients.

The following analysis and computation is useful to determine handover latency from the perspective of a client device. It can be used to influence the design of the network core topology in order to reduce this handover latency.

Network-Level Bandwidth Minimization If bandwidth utilization is to be kept to a minimum and the client applications can tolerate small data loss or is equipped to retransmit lost data (i.e. is using a protocol like TCP), then flows need not be proactively pushed. Instead, flow adds and deletes can be performed simultaneously to switch the client's packets to its new attachment point. In this case, the handover time to where the client can receive its data from remote connections is dependent on the switch control plane latencies and link latencies in the data plane. This impacts the value of T_f.

Let L_i be the latency for a client connection to arrive to switch i in the tree from the root of the handover network, where the root is switch $i = 0$. Let c_i be the control plane latency between switch i and the controller. And, let l_i be the switch link latency between switch i and switch $i + 1$.

Consider the first case for L_0. The time for the connection to get to switch $i = 0$ is simply the time for the controller to program that switch, c_0. The second case is of course more complex. To aid in understanding, let's consider switch $i = 1$, which is the switch attached to switch $i = 0$ over link l_0. The delay for the connection to reach this switch can be derived from the maximum of either (1) the time for the connection to reach switch $i = 0$ and the latency of the data plane link between switches $i = 0$ and $i = 1$, namely $L_0 + l_0$, or (2) the latency of the control plane link of switch $i = 1$, c_1. Once flow mods are installed in switch $i = 0$, the connection packets will start to traverse the link between the two switches. At the same time the flow mod is installed to switch $i = 0$, the controller concurrently sends similar flows to switch $i = 1$ to forward the connection's packets.

If $c_0 = c_1$, then the control plane links are equal in latency and latency for the connection's packets to reach switch $i = 1$ is simply $c_0 + l_0$ or $c_1 + l_0$ given that the control plane latencies are equal. In general terms, this can be expressed between two adjacent switches i and $i - 1$ as $L_i = L_{i-1} + l_{i-1}$.

If $c_0 > c_1$, then the connection's packets will not leave switch $i = 0$ until c_0 has elapsed. The flows for switch $i = 1$ will be installed before the connection's packets leave switch $i = 0$, making control plane latency c_1 irrelevant. Thus, the latency for the connection's packets to reach switch $i = 1$ is simply $c_0 + l_0$. In general terms, this can be expressed between two adjacent switches i and $i - 1$ as $L_i = L_{i-1} + l_{i-1}$

If $c_0 < c_1$, then the connection's packets will leave switch $i = 0$ before the flows are programmed into switch $i = 1$. If $c_1 > L_0$ or the control plane latency of switch $i = 1$, c_1, is larger than the time for the connection's packets to get to switch $i = 0$, L_0, then at time L_0, there is still $c_1 - L_0$ time remaining for the flows to be installed in switch $i = 1$. This time competes with the latency of the link from switch $i = 0$ to switch $i = 1$, l_0.

If $c_1 - L_0 > l_0$, then the connection's packets will arrive to switch $i = 1$ before the flows are programmed and will be dropped until $c_1 - L_0 - l_0$ additional time has elapsed. The total time for switch $i = 1$ to to receive the connection's packets is thus the time for the connection to reach the previous switch, L_0, plus the residual control plane latency to program the flows on switch $i = 1$, $c_1 - L_0$. This can be summarized as $L_0 + c_1 - L_0$ or more simply just c_0. In general, between two adjacent switches i and $i - 1$, this can be written as $L_i = c_i$.

If $c_1 - L_0 \leq l_0$, meaning the link latency between switch $i = 0$ and $i = 1$, l_0, is greater than or equal to the residual residual control plane latency to program the flows on switch $i = 1$, $c_1 - L_0$, the flows will be programmed at switch $i = 1$ before the connection's packets arrive and will thus be received by switch $i = 1$ after the connection's packets finish traversing l_0 at time $L_0 + l_0$. More generally put, this can be written as $L_i = L_{i-1} + l_{i-1}$between any two adjacent switches i and $i - 1$.

Putting it all together, L_i can be expressed as a constant c_i for the base case where $i = 0$ given that this switch is the root of the tree. Between any two adjacent switches in the tree i and $i + 1$, L_i is at least L_{i-1} plus the slowest of either the residual control plane latency for switch i, $c_i - L_{i-1}$ or the link latency between switches i and $i - 1$. This allows us to conclude with a formulation for L_i:

$$L_i = \begin{cases} c_i & i = 0 \\ max\,(L_{i-1} + l_{i-1},\ c_i) & i > 0 \end{cases}$$

Thus, for a multi-homed client that switches from one interface to another in order to perform the handover, the time the client must go without receiving data from any remote correspondent nodes during the handover is $T_f = L_i$ from the root to the edge, where i is the edge switch. For a non-multi-homed client performing a handover with the same physical interface, the time the client must go without receiving data from any remote correspondent node during the handover is much greater at T_c.

When computing the flows and path of the new path after the handover, it must be joined to the existing path used by the client. The closer this "join" is to the edge network, the lower L_i can become. In a deployment where the join must occur at the network root, L_i will be directly dependent on this root's proximity to the edge switches for each handover. Instead, if multiple paths exist in the network such that the join can be accomplished closer to the edges where the client is currently and will be attached after the handover, a lower value of L_i can be achieved.

For the special case of inter-domain handovers under the assumption that proactive signaling is used between the domain controllers to share client information, the handover latency is the same as the intra-domain handover plus the latency between the root of the old and new domains. This additional latency must be factored in, since the packets might still traverse the root of the old domain en route to the new domain. This implies a multi-homed client will experience a disruption of $T_f = L_i$ plus an inter-domain latency of l_d, or $L_i + l_d$. Non-multi-homed clients will experience a disruption of $T_c + l_d$.

Client-Level Content Disruption Minimization If instead connection continuity is desired during the handover (i.e. for real-time applications like voice and video conferencing), then data loss and/or delay should be avoided. If the client device is not multi-homed for the handover, then it will experience a service disruption of T_c, which is not ideal. Instead, for the minimization of client-level content disruption, it is assumed the client device is multi-homed for each handover.

Assuming a multi-homed device, minimization of content disruption can be accomplished through proactive flow installation at the client's new attachment point prior to relinquishing the connection at the client's old/current attachment point. Thus, the time for a client to receive packets after the handover is 0, since the client will continue to send and receive packets through one interface until T_a, T_d, T_f, and T_e have elapsed. At this point, the client will switch to the new interface on the client-level and terminate its connection on the old interface.

In the network-level, flows are installed to the client's new attachment point by computing the shortest path from the root to the edge switch where the client will reside. This path is compared to the path the client is currently using at its current/old attachment point. The intersection of theses paths, or the "join," indicates where flows should be pushed to send packets to and from the client's new attachment point. This intersection switch incurs a flow modify, outputting packets towards the client's old attachment point and the new attachment point. Other switches in the new path from the edge to the intersection point receive flow add operations in order to forward packets to and from the client and the intersection switch. After the controller finishes pushing all flows for the client, it will modify the flow at the intersection switch by removing the output port leading to the client's old attachment point. The controller will simultaneously remove all flows in the path from the intersection switch and the client's old attachment point.

For the special case of inter-domain handovers, 0 seconds of disruption can be achieved in the same manner as discussed above for intra-domain handovers. In this case, the intersection point of the path from the old and new attachment points is the root of the old domain or some intermediate administrative domain. Here, flows are installed to send the packets simultaneously into the old domain and the new domain from the old domain's root. The new domain's controller will install the remaining flows in its domain from its root to the client at the edge. Once this operation is complete, the client will detect packets on its new attachment point interface and subsequently terminate its old connection to the old domain.

Chapter 5

High Throughput Data Transfer

When users desire to rapidly and reliably exchange large data sets across wide area networks, the windowing problem common to existing reliable transport protocols such as TCP results in a small percentage of the available network bandwidth being used, when in fact larger transfer rates could be achieved. This problem is well studied and has produced parallel TCP solutions and those that modify TCP congestion algorithms, usually in the form of user installed proprietary software solutions.

To address these shortcomings, the use of SDN is proposed. SDN provides the level of network control necessary to deploy a solution that is transparent to the end user, does not require the installation of any software on any machine involved in the data transfer, and can scale elastically. SDN also gives the ability to control network traffic in unconventional ways.

The high throughput data transfer aspect of this dissertation proposes an SDN-based solution to the windowing problem discussed and that is easy to use for all network users. It exploits the known performance increase achievable with multiple TCP connections in parallel. Rather than requiring the client and the server "participating" in the transfer handle the transfer in parallel, SDN is leveraged to perform the parallel transfer within the network, such that it is accomplished transparent to the client and server.

In this chapter, the architecture and design of the overall solution is discussed, where it is shown that the end users are not required to install or setup anything in order to benefit from high performance data transfer. Section 5.1 dissects the data transfer problem, revealing the fundamental requirements of any solution. These requirements serve as the cornerstone for the SDN-based solution. Section 5.2 introduces the SDN-based solution known as Steroid OpenFlow Service (SOS). Detailed discussion is provided on the design of the SOS architecture, its use of SDN, and its scalability. Section 5.3 evaluates the data transfer performance improvement SOS can provide over the commodity Internet as well as over a high-performance links and in data center environments.

The work so far has been published in [44], [78], and [79].

5.1 The Data Transfer Problem

High performance data transfer can be characterized by three fundamental components: (1) retrieval of relevant data from disk or storage to memory, (2) data transfer across the network, and (3) end-to-end data transport from the application conducting the transfer. Depicted in Figure 5.1.1, these three components are analyzed individually in order to identify and individually address potential bottlenecks to the data transfer as a whole. These components are then used to formulate the requirements and characteristics of a transparent, high performance data transfer solution.

5.1.1 Components of High Performance Data Transfer

5.1.1.1 Storage to Memory

In order for data transfer to occur, the data must be sourced from and saved to some location. In some applications, it might be generated and consumed entirely within the applications on the client and server. In others though, it must be retrieved from some storage location. The manner in which data is accessed must be carefully examined in order to avoid unintended bottlenecks. Components to consider include:

- storage device bandwidth
- storage controller bandwidth
- Non-Uniform Memory Access (NUMA)
- bus and input/output (I/O) bandwidth

Data storage has notoriously been one of the slowest components of computers. Modern architectures can be equipped with a variety of high speed storage controllers such as serial advanced technology attachment (SATA), serial attached small computer system interface (SAS), and redundant array of independent disks (RAID). When used with slower spinning disks, the disks are the I/O bottleneck,

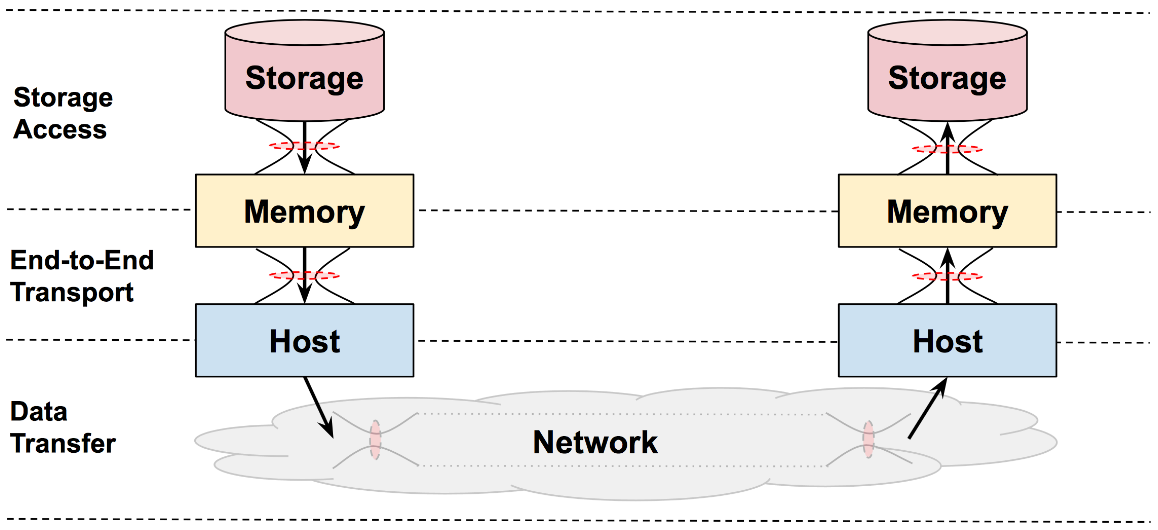


Figure 5.1.1: The components of a data transfer. When moving data between components, bottlenecks must be considered.

sustaining average data transfer rates of around 500Mbps, as compared to the ability of their controller. SATA III for example is capable of 6Gbps per port and an aggregate controller bandwidth of approximately 10-11Gbps depending on the controller implementation. When coupled with modern high-speed solid state drives, the bottleneck shifts from the disks themselves to the disk controller. With high performance SSDs capable of read and write speeds in excess of 4Gbps, just a few SSDs is plenty to saturate the disk controller.

The use of multiple disk controllers is the next logical step to eliminate the bottleneck of a single controller, but even in modern architectures, many cannot support multiple disk controllers on a single CPU socket due to available peripheral component interconnect (PCI) lanes or constraints in how the motherboard has wired PCI lanes. With NUMA architectures, multiple disk controllers can be installed on different CPU sockets to further increase throughput reading from and writing to disk. However, this imposes design constraints on the data transfer application, forcing it to be NUMA-aware and access data from the local disks/controller(s) only else incur a local, inter-CPU bus usage penalty – for Intel CPUs referred to as direct media interface (DMI) or quick path interconnect (QPI); HyperTransport for AMD. Furthermore, given this data is either from or destined to the network, the NIC it is from or is en route to should also be on the same CPU socket as the corresponding disk controller, otherwise, the local bus must be used to transfer the data to the remote CPU, imposing another potential bottleneck.

Many state of the art commercial solutions exist to alleviate this storage headache for consumers. Such solutions consist of multiple storage nodes that run in parallel, storing and rotating data through a hierarchy of storage technologies – solid state disks (SSDs) for offload at speed parity from the network, then cheaper and higher-capacity spinning disks for long-term storage. Multiple storage nodes are used in parallel to eliminate bottlenecks present in a single-node design described above and to scale to larger storage demands as data sizes increase and network speeds increase.

The design of storage access for a particular data transfer will of course depend on the data transfer rates desired. A single machine could be used with a single disk controller and SSDs for sub-10Gbps data transfers, multiple disk controllers and a NUMA-aware application could be used for 10Gbps+ data transfers, or a more advanced storage solution could be designed for a general purpose application. Such solutions could make use of state-of-the-art commercial techniques described above or a distributed parallel file system like OrangeFS [80]. Parallel file systems like OrangeFS present a network file system abstractly as a mount point on the local file system of a client. The client simply reads from and writes to this mount point through its filesystem. The OrangeFS daemon directs I/O through this mount point to remote nodes where the data is physically stored. To be effective, the additional network I/O used by OrangeFS must not contend with that used by the client's network card to receive or transmit the data. A second NIC might be necessary depending on the specific deployment scenario.

Despite being a complex and important part of the data transfer problem, the storage access problem presented here is in fact orthogonal to the network data transfer problem, and must be addressed independently for any complete end-to-end data transfer. Without adequate consideration, it can easily become the bottleneck, thwarting any efforts made in the network to further increase performance.

5.1.1.2 Memory to Network

Achieving high throughput data transfer from memory to the network is the goal of the memory to network component. Potential bottlenecks to consider include:

- sockets and kernel
- network interface card
- bus and I/O bandwidth

From memory, the application needs to write the data to a socket in order to have the kernel facilitate the data transfer to the network. To reduce latency, the application should minimize data copies from memory to memory. A memory-mapped buffer can be used so that the write system call does not need to perform any copies from user-space to kernel-space. However, this has the drawback of added buffer management complexity, since there is not a way to know for sure when the TCP state machine has consumed and successfully relayed the entire buffer to the remote host. Without memory-mapped buffers, when a write is performed to the socket, the kernel will copy the written data into buffers of the TCP state machine. From there, the data will be relayed by TCP from the kernel to the NIC via direct memory access (DMA) if possible or via CPU if DMA from memory to the NIC is not supported.

Data moving from memory to the NIC must traverse the PCI bus on which the NIC is connected. To reduce latency and avoid bottlenecks, the NIC should be connected to PCI lanes wired to the same CPU on which the application is running. Otherwise, a penalty due to CPU local bus usage will occur. This is the same penalty as previously mentioned for storage to memory considerations. This means that the NIC and storage controller should be wired to the same CPU socket and that the application should be pinned to a CPU core on the same socket in order to reduce latency and avoid bottlenecks.

5.1.1.3 End-to-End Data Transport

Facilitating the transfer of data from one user application to another is the goal of end-to-end data transport. To accomplish end-to-end data transport, the user has a choice as to which technology or application to use. Choices include edge based data transfer tools such as Aspera or GridFTP, parallel TCP, or even homegrown solutions. End user intervention to facilitate such edge initiated solutions introduces deployment, management, and usage hurdles, hindering their widespread adoption [44]. As such, many users choose to use regular TCP to achieve end-to-end data transport.

Tools such as Aspera and GridFTP accomplish high performance data transfer through thoughtful use of transport protocols. Due to the TCP windowing problem over large delay-bandwidth-product networks, GridFTP makes use of multiple TCP connections in parallel. This has a couple of advantages. First, the impact of network congestion and resulting packet loss is lessened. Only the TCP connection(s) that sustained loss will suffer decreased window sizes. The other TCP connections will be unaware and will continue at their current window size as if packet loss did not occur. Another advantage is that it allows GridFTP to work around the OS's likely low, default TCP buffer limits, which if data loss did not occur on a single TCP connection, would impose a data transfer bottleneck in the form of an insufficiently large TCP window.

GridFTP also has a UDP operating mode (via UDT), similar to Aspera. Both use UDP combined with slim TCP control connections to reliably transfer data. UDP is not a windowed transport protocol, so it does not suffer from the same limitations as TCP. It will naturally transfer data as rapidly as possible to the event the application can produce and read data and that the host operating system can send and receive the data on the network.

In common to each of these techniques is the attempt to fill the pipe with data. TCP-based techniques address the pitfalls present in the TCP protocol to make better use of available bandwidth and keep sustained throughput high, even with data loss. UDP-based techniques naturally keep the pipe full and have been augmented to be reliable and ordered, necessary for general purpose data transfer.

5.1.2 Transparency for High Performance Data Transfer

The goal of transparent high performance data transfer is to provide all the benefits of high performance data transfer to those users wish to use non-performance-oriented data transfer tools. Network based solutions require no intervention by the end user in order for them to be effective, thus they are said to be transparent to the end user. A network based solution must be used if the goal is to transparently improve the TCP data transfers of users who use non-optimized end-to-end data transport applications. Otherwise, non-transparent modification of user machines and/or applications must be performed.

A network based solution must meet the fundamental storage, data transfer, and end-to-end transport requirements inherent to any data transfer solution, along with the additional requirements that it must cause no harm to other edge based data transfer tools and it must be flexibly and scalably deployable.

5.1.2.1 Addressing Fundamental Requirements

Storage Access

In a network based solution, storage access is left to the user. The user must independently consider their data transfer needs and the storage requirements previously discussed. This is beyond the scope of a network based solution, which has no control over how or where the user sources and saves data.

Data Transfer

In order for a network based solution to have merit, it must make effective use of the available bandwidth over the long distance link. Any technique and requirements previously discussed for general data transfer must be implemented and enforced for network based data transfer. Data transfer must be performed on a capable device or on a general purpose compute resource. The machine must be equipped with sufficient compute resources in order to handle the marshaling and unmarshalling of data between the user's regular TCP connection and the data transfer implementation of choice at line rate or at the desired throughput. It must also have sufficient memory to buffer the packets from the data transfer implementation in order to ensure data is presented to the regular TCP connection in order.

End-to-End Data Transport

In a transparent network based solution, the user can use an arbitrary end-to-end data transport application. This "bring your own application" approach provides an application layer agnostic data transfer solution, supporting tools as simple as wget and web browser download managers to more complex applications like GridFTP. To be transparent, the network needs to detect this end-to-end data transport and usher it to the location where data transfer is implemented in the network. Network switches can be programmed to rewrite and deliver the user's regular TCP packets to a data transfer node and from a data transfer node to the server at the other end of the regular TCP connection.

5.1.2.2 Additional Requirements

Cause No Harm

A solution must be carefully crafted in order to cause no harm to existing rapid data transfer techniques. Existing techniques work well for their current users, and a new solution should respect this and not serve to replace them. Without this requirement, any users who benefit from the new solution will be offset by those who are now inconvenienced or can no longer use their existing techniques.

Flexible Deployment at Scale

The solution must be capable of flexible deployment at scale. Network operators should be able to deploy the network initiated solution in a broad range of network topologies to support the data transfer demands of any network scale.

5.2 SDN-based Solution

In this section, a high-level introduction to the SOS architecture is provided, followed by design details partitioned into three logical components, and a discussion on architecture scalability. The structure of this section is as follows. Section 5.2.1 shows how the proposed solution meets the requirements of any network based solution. Section 5.2.2 discusses how SDN is used to implement transparency to the end user. It includes fundamental network requirements for the successful deployment of SOS. Two common deployment scenarios are discussed in detail. Section 5.2.3 explains how SOS can improve the data transfer rates of TCP workflows through the use of SOS agents implementing parallel TCP. Section 5.2.4 presents the SOS OpenFlow controller, which manages and orchestrates the SOS deployment. The operation of the controller and management APIs are discussed in depth. Section 5.2.5 shows how the SOS architecture can scale through the deployment of multiple SOS agents and through layering with application parallel TCP techniques.

All source code for the SOS Floodlight controller can be found at `http://www.github.com/rizard/SOSForFloodlight/tree/smart-1.3`. The source code for the SOS agents can be found at `http://www.github.com/cbarrin/sos-agent`.

5.2.1 Addressing the Requirements of High Throughput Data Transfer

SOS is a network based data transfer solution, allowing users to use their own end-to-end data transfer application and define and implement their own storage solution. It implements data transfer through a combination of SDN and general purpose compute resources installed within the network.

5.2.1.1 Software Defined Networking

SDN is used to achieve end user data transfer transparency by careful control over the flow of selected end-to-end TCP data transfer traffic. This allows SOS to improve user end-to-end data transfers disadvantaged by existing data transfer techniques, all without end user intervention. The careful selection of TCP traffic also ensures SOS is enacted only on traffic of interest, causing no harm to other data transfers (edge or network based) and background traffic. Furthermore, through its selection criteria of TCP traffic, SOS is an application agnostic solution, supporting any data transfer application operating over TCP. The SDN also provides a management framework for network operators through the SDN controller, making it simple to deploy and manage at scale in a variety of network topologies.

5.2.1.2 Compute Resources

General purpose compute resources are used to implement data transfer in the SDN through the well known effectiveness of parallel TCP. Two of these servers sit at each side of the long distance link, using parallel TCP between them to make greater use of the available bandwidth. Multiple of these servers can be used in order to flexibly scale an SOS deployment. The use of compute resources for this task is a necessity, since traditional network forwarding devices cannot perform such complex operations at present. As SDN technology evolves, this parallel TCP operation can be migrated into the SDN fabric itself without the need for dedicated servers. Also due to limitations of present forwarding devices, Section 5.2.2.3 discusses how these compute resources can also serve to flexibly offload some of the SDN operations used to achieve transparency to a software network switch onboard the server.

5.2.2 SOS Architecture and Design Patterns

In order to meet the requirements of high throughput data transfer, SOS is composed of three key physical components: (1) an SDN consisting of at least two SDN switches, (2) an SDN controller, and (3) general purpose compute resources. The SDN allows for custom forwarding of traffic in order for SOS to manipulate data transfer TCP packets without user knowledge. The SDN controller orchestrates the packet manipulation through the installation of rules in the SDN switches, manages ongoing SOS data transfers, and provides a way for network operators to manage the SOS

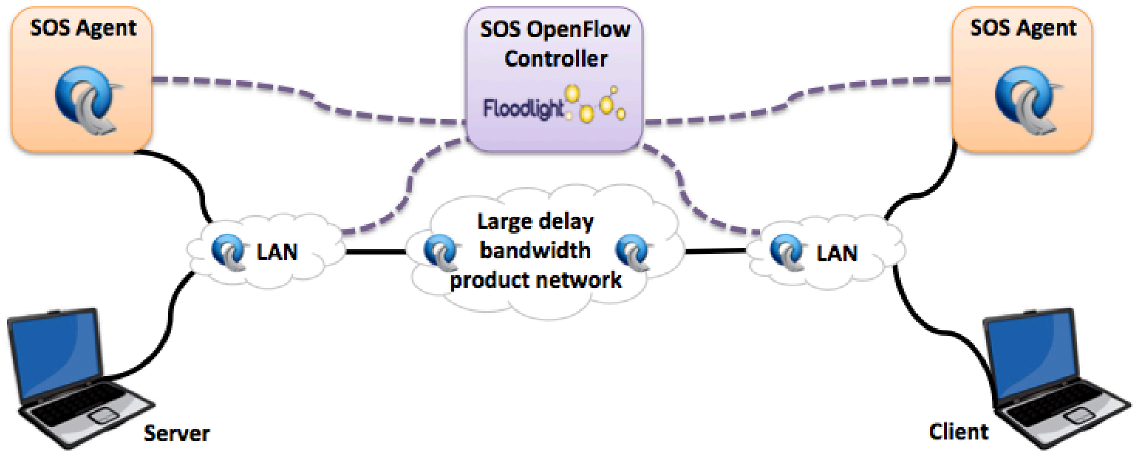


Figure 5.2.1: General SOS Architecture. The OpenFlow logo indicates possible network locations where SOS can support and leverage OpenFlow switches, both physical and software.

deployment. The compute resources, known as SOS agents, facilitate per-packet operations essential to SOS that cannot be performed on the SDN switches. SOS also leverages these agents to improve data transfer rates over long distance links using multiple TCP connections in parallel between two agents. Due to the widespread adoption and use of the OpenFlow protocol, SOS has been designed using OpenFlow as the enabling SDN technology; however, the SOS architecture can be adapted to any SDN protocol with equivalent features to those leveraged from OpenFlow.

In an SOS workflow, SOS agents, in conjunction with the underlying SDN and its controller, implement a transparent proxy between the client and server involved in the data transfer. This transparent proxy utilizes parallel TCP techniques between agents in order to improve utilization over the long link. Figure 5.2.1 depicts the SOS architecture. In the following two subsections, further details are revealed about how this transparent proxy is implemented in the SDN and how the agents implement parallel TCP transparent to the client and server end users.

In order for SOS to achieve end user transparency, the network architecture in which it is deployed must be conducive to the transparency implemented by the SDN. In the following subsections, the fundamental network requirements for an SOS deployment are explained. Deployment details are revealed through two simple yet common deployment scenarios.

5.2.2.1 Network Architecture

The SDN does not need to be entirely OpenFlow-enabled in order for SOS to function. Instead,

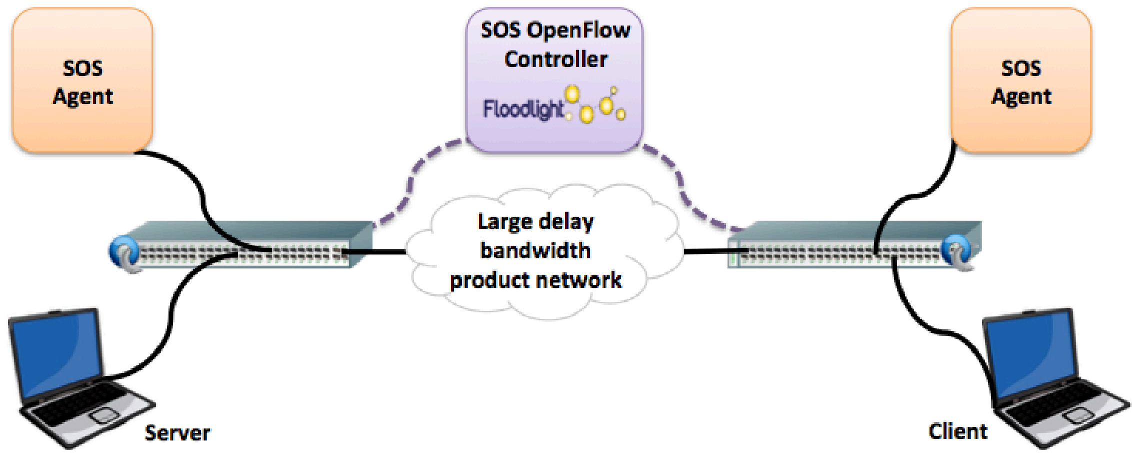


Figure 5.2.2: Simple SOS deployment with two OpenFlow switches

OpenFlow switches need to be strategically deployed to achieve service transparency. A fundamental requirement is the presence of an OpenFlow switch at each side of the path that the TCP connection will traverse – one in the path nearby the client and another in the path close to the server.

Figure 5.2.2 depicts this fundamental requirement in a simple SOS deployment. In this example, the SOS deployment consists of two OpenFlow switches, an SOS agent connected to each OpenFlow switch, and a client or server machine connected to each OpenFlow switch. Both OpenFlow switches are controlled by the SOS OpenFlow controller. When the client and the server hosts initiate and transfer data over TCP, SOS is automatically invoked in order to improve the throughput of the TCP connection.

Although this basic deployment is simple to comprehend, SOS can be deployed across more complex network topologies. In more complex networks, it may not be feasible for the client, server, or SOS agents to be directly connected to an OpenFlow switch. In this situation, recall the fundamental requirement of SOS, which is that the OpenFlow switches must exist in the path between the client and the server. Thus, the OpenFlow switch must simply exist someplace in the path in order to intercept the TCP packets for redirection to the nearby agent. The OpenFlow switch at the point of interception should ideally be as geographically close as possible to the client/server to ensure the single TCP connection between he client/server and the proxied SOS agent is able to efficiently utilize the link. The closer the agent is to the client/server, more of the long distance link will be utilized by the rapid inter-agent communication rather than the slow client-to-server communication.

5.2.2.2 Manipulating TCP Connections with OpenFlow

OpenFlow is used to seamlessly intercept and manipulate the TCP connection between two hosts – the client and the server. This interception is done when the TCP handshake begins and is terminated when the TCP connection is closed or upon a timeout in the case of an ungraceful close.

On the client side of the network, the OpenFlow switch redirects and rewrites TCP packets from the client to the client's SOS agent and back from the client's SOS agent to the client. The client's SOS agent serves as a transparent proxy for the remote server and relays any data sent from the client to the client's agent to the remote SOS agent nearby the server. The server's SOS agent will receive data from the client's SOS agent and will then establish a TCP connection with the server. The server's OpenFlow switch performs both TCP packet redirection and rewrite from the server's SOS agent to the server and from the server to the server's SOS agent. Due to the packet rewrite employed by the flows installed in the OpenFlow switches, the server thinks it is communicating directly with the client and the client thinks it is communicating directly with the server. In reality, SOS has created this illusion through (1) careful packet interception, rewrite, and redirection and (2) the use of SOS agent software to complete the TCP connection and proxy for the remote host.

The specific flows installed in the OpenFlow switches will vary depending on the network topology; however, they must consist of layer 2, layer 3, and layer 4 packet header matches and rewrites in order to achieve transparency. The SOS OpenFlow controller can determine the network topology and install the appropriate flows automatically.

5.2.2.3 Design Patterns

SOS is designed to support a variety of network configurations and topologies. To illustrate the specific OpenFlow flows used by SOS and to better understand how transparency is achieved with SOS, two common SOS deployment models are considered. The flows required to achieve transparency are discussed in detail. The two common deployment models are (1) where the hardware switches chosen as the in-path intercept switches support all the required OpenFlow header matches and rewrite actions, and (2) where the hardware switches chosen as the in-path intercept switches only support a subset of the required OpenFlow header matches and rewrite actions.

Hardware Switches Support All Required SOS OpenFlow Operations

In this section, an ideal SOS deployment scenario is provided. The goal is to demonstrate how transparency is achieved with the SDN and SOS agents through elimination of unnecessary topology-induced cognitive load.

As a simple example, let's assume the SOS deployment consists of a single OpenFlow switch at each side of the network with a directly connected SOS agent and a host for participating in the data transfer. Operating under the assumption the OpenFlow switch at each side of the network is capable of performing layer 2 through layer 4 packet header match and rewrite, the SOS controller can install flows on the client-side interception switch to match and intercept the packets from the client that are intended for the server. These flows send intercepted packets to the nearby SOS agent. The destination MAC and IP addresses and the destination TCP port of such intercepted packets are rewritten by the flow on the OpenFlow switch to the agent's MAC, IP, and open TCP port so that they may be received by the agent machine through all layers of the network stack. All TCP packets destined for the client from the agent's TCP port, such as TCP ACKs, will be addressed by the agent machine's network stack to the client's TCP port, IP address, and MAC address; however, to maintain transparency, the same interception OpenFlow switch on the client–side of the network rewrites the source MAC, IP, and TCP port to the server's MAC and IP addresses and TCP port. This bi-directional packet interception, rewrite, and redirection allows a TCP connection to be established from the client to the local SOS agent where the client "believes" it has actually connected to the server across the network.

Now, because SOS provides a transparent service, any data relayed to the the client-side SOS agent over the TCP connection must eventually make it to the real destination (the server) which knows the data semantics and can interact with the client appropriately over TCP. To accomplish this, the client-side SOS agent reliably relays any data to the server-side SOS agent. The server-side SOS agent establishes a TCP connection with the server. All TCP packets from the server-side SOS agent are addressed to the server machine's MAC, IP, and TCP port but contain server-side SOS agent source headers. The interception OpenFlow switch on the server-side of the network contains a flow installed by the SOS OpenFlow controller to match these packets and rewrite the source MAC and IP addresses and TCP port to those of the client machine. When these packets arrive at the server, the server will think they have originated from the client. Any TCP packets

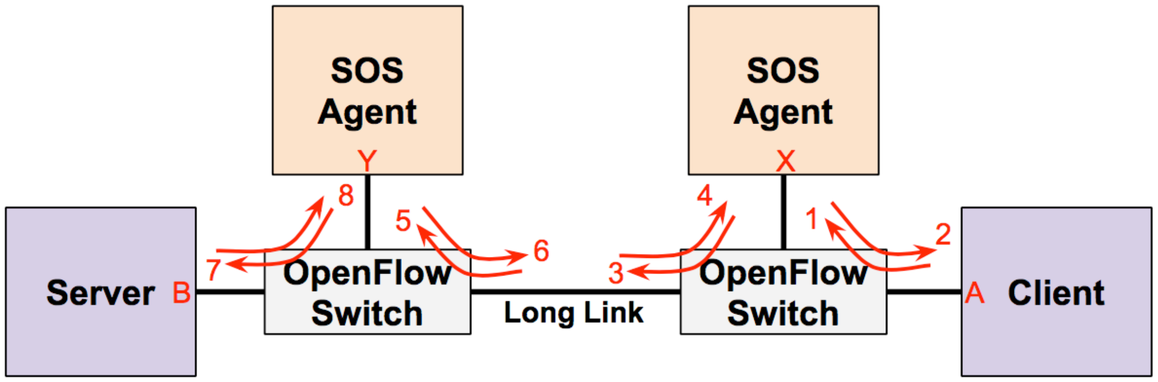


Figure 5.2.3: Logical flows in an SOS deployment

from the server to the client are intercepted by a similar flow in this same interception OpenFlow switch, their destination MAC address, IP address, and TCP port will be rewritten to those of the server-side SOS agent, and they will be redirected to the server-side SOS agent.

The SOS agents relay any data exchanged between the client and the server over the network between the two SOS agents. All communications between SOS agents do not involve any packet redirection or rewrite using OpenFlow.

The flows required for this example deployment are shown in Table 5.2.1. The Flow Number column indicates where the flow is installed in the SOS network topology, as shown and annotated in Figure 5.2.3.

Hardware Switches Support a Subset of the Required SOS OpenFlow Operations

In this section, it is shown how SOS can be deployed in a more realistic network topology with off the shelf white box OpenFlow switches. Such a deployment scenario aligns closely with modern production networks and physical OpenFlow switches.

OpenFlow defines a simple yet powerful set of primitives to perform packet matching and modification. However, a common misconception about OpenFlow is that all OpenFlow features are supported on an "OpenFlow" enabled forwarding device. The present reality of OpenFlow and OpenFlow implementations is that the most useful rewrite features are optional features and oftentimes are not actually supported in hardware, at line rate. This is in part due to available OpenFlow switches being implemented as updated firmware on top of pre-existing hardware pipelines not designed to perform many OpenFlow features. For SOS, in addition to the packet redirection to/from agents,

Flow	Flow Match	Flow Action	Preconditions	Postconditions	Description
1	in_port=port$_A$ ip ip$_{src}$=ip$_A$ tcp tcp$_{src}$=tcp$_A$	mac$_{dst}$=mac$_X$ ip$_{dst}$=ip$_X$ tcp$_{dst}$=tcp$_X$ output=port$_X$	TCP packets from A to B	Packets now appear to be from A to X	Rewrite and redirect from A->B to A->X
2	in_port=port$_X$ ip, ip$_{dst}$=ip$_A$ tcp tcp$_{dst}$=tcp$_A$	mac$_{src}$=mac$_B$ ip$_{src}$=ip$_B$ tcp$_{src}$=tcp$_B$ output=port$_A$	TCP packets from X to A	Packets now appear to be from B to A	Rewrite and redirect from X->A to B->A
3	in_port=port$_X$ ip ip$_{src}$=ip$_X$ ip$_{dst}$=ip$_Y$ tcp	output=port$_Y$	TCP packets from X to Y	Packets relayed unmodified from X to Y	Send unaltered inter-agent packets from X->Y
4	in_port=port$_Y$ ip ip$_{src}$=ip$_Y$ ip$_{dst}$=ip$_X$ tcp	output=port$_X$	TCP packets from Y to X	Packets relayed unmodified from Y to X	Send unaltered inter-agent packets from Y->X
5	in_port=port$_X$ ip ip$_{src}$=ip$_X$ ip$_{dst}$=ip$_Y$ tcp	output=port$_Y$	TCP packets from X to Y	Packets relayed unmodified from X to Y	Send unaltered inter-agent packets from X->Y
6	in_port=port$_Y$ ip ip$_{src}$=ip$_Y$ ip$_{dst}$=ip$_X$ tcp	output=port$_X$	TCP packets from Y to X	Packets relayed unmodified from Y to X	Send unaltered inter-agent packets from Y->X
7	in_port=port$_Y$ ip ip$_{dst}$=ip$_B$ tcp tcp$_{dst}$=tcp$_B$	mac$_{src}$=mac$_A$ ip$_{src}$=ip$_A$ tcp$_{src}$=tcp$_A$ output=port$_B$	TCP packets from Y to B	Packets now appear to be from A to B	Rewrite and redirect from Y->B to A->B
8	in_port=port$_B$ ip ip$_{src}$=ip$_B$ tcp tcp$_{src}$=tcp$_B$	mac$_{dst}$=mac$_Y$ ip$_{dst}$=ip$_Y$ tcp$_{dst}$=tcp$_Y$ output=port$_Y$	TCP packets from B to A	Packets now appear to be from B to Y	Rewrite and redirect from B->A to B->Y

Table 5.2.1: OpenFlow flows required to transparently insert SOS agents X and Y into a TCP connection between hosts A and B. These flows indicate the flows required if the hardware switches of the SOS deployment support all the required packet header matches and actions.

packet MAC address, IP address, and TCP port numbers must be rewritten in order to achieve end-user transparency. If a physical OpenFlow switch does not support such optional rewrite actions as defined in the OpenFlow specification, a software switch can be used to perform the rewrites instead. This software switch implementation can either (1) replace the physical switch and perform both in-path interception, rewrite, and redirection all in one, (2) rewrite packets after they have been intercepted and redirected from the in-path hardware OpenFlow switch, or (3) perform a careful combination of (1) and (2). Implementation (1) requires the use of only one OpenFlow switch; however, it might require other non-TCP packets to be handled by the software switch, which if not installed on an adequate general purpose server, could prove to be a bottleneck. On the other hand (2) allows the interception and redirect (i.e. packet selection) to occur in hardware at line rate and leaves the rewrite of the matched TCP packets to be handled by a separate software OpenFlow switch.

Option (3) is a practical compromise of (1) and (2) and is used in the experimental evaluation of SOS in Section 5.3. While most white box OpenFlow switches cannot perform IP and transport layer rewrite, they do support traditional switching features through OpenFlow at line rate, which includes layer 2 through layer 4 access control list (ACL) matching, as well as source and destination MAC address rewrite. If the OpenFlow switch used for packet redirection is used to also rewrite the MAC addresses of the intercepted packets, then any layer 2 network (OpenFlow and/or non-OpenFlow) can be used in between such a capable interception OpenFlow switch and the SOS agent. The MAC rewrites allow layer 2 networks to implement their MAC learning algorithms unimpeded. Otherwise, if packet redirect is performed without rewriting MAC addresses and such packets traverse an unsuspecting layer 2 network, then MAC learning could be incorrectly or inconsistently performed. As such, option (3) can be used to first intercept and rewrite the MAC addresses of the desired TCP packets, then send the packets in the direction of the SOS agent machine. Any layer 2 non-SDN between the interception switch and the SOS agent will be able to perform layer 2 forwarding using the newly-rewritten packet MAC addresses, which will ensure the packets arrive at the agent. Layers 3 and 4 can be rewritten using an in-path software switch closer to the agent. Of course, it is advantageous to perform as many required header rewrites in hardware rather than in the software switch. As shown in Figure 5.2.2, this software OpenFlow switch can be installed on the same machine as the SOS agent itself, greatly simplifying an SOS deployment.

The flows required for this example deployment are similar to those shown in Table 5.2.1, but

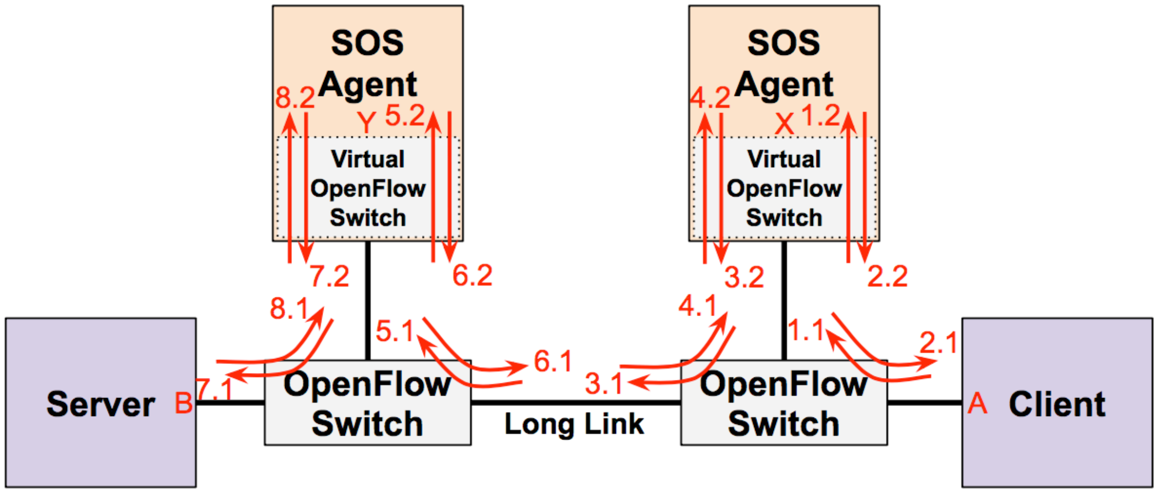


Figure 5.2.4: SOS flows in a deployment with white box switches

the matches and header rewrites are spread amongst capable switches in the path and compensated for by software switches installed onboard the SOS agents as indicated in Figure 5.2.4. Tables 5.2.2 and 5.2.3 show how the logical tasks of each flow defined in Table 5.2.1 can be divided into multiple flows to conform to physical switch capabilities.

5.2.3 SOS Agents

While the SDN implements service transparency, SOS agents are designed to improve the throughput of a TCP data transfer between the client and the server. Physically, they are installed as devices within the SDN on general purpose, commodity hardware. This hardware runs custom software that operates on user data in order to improve throughput. Custom software is used, since the operations performed on the user data cannot be performed on OpenFlow switches. As shown in Figure 5.2.5, an SOS agent is a piece of software that consists of a host data engine and an inter-agent data engine.

The host data engine is a TCP connection that interfaces directly with the client or server. It proxies for the remote host and handles data for the client's/server's TCP connection. The inter-agent data engine is responsible for implementing the throughput-improving technique between two agents.

The design allows data to flow bidirectionally and requires an agent at both the client and the server side of the long distance link. Since parallel TCP is a proven solution to improve data transfer rates over long links where a single TCP connection is unable to fill the pipe, the present design

Flow	Flow Match	Flow Action	Preconditions	Postconditions	Description
1.1	in_port=port$_{\mathrm{A}}$ ip ip$_{\mathrm{src}}$=ip$_{\mathrm{A}}$ tcp tcp$_{\mathrm{src}}$=tcp$_{\mathrm{A}}$	mac$_{\mathrm{dst}}$=mac$_{\mathrm{X}}$ output=port$_{\mathrm{X}}$	TCP packets from A to B	Packets now appear to be from A to B/X	Rewrite L2 and redirect from A->B to A->X
1.2		ip$_{\mathrm{dst}}$=ip$_{\mathrm{X}}$ tcp$_{\mathrm{dst}}$=tcp$_{\mathrm{X}}$ output=port$_{\mathrm{X}}$	TCP packets from A to B/X	Packets now appear to be from A to B	Rewrite L3 and L4 from A->B to A->X
2.1	in_port=port$_{\mathrm{X}}$ ip, ip$_{\mathrm{dst}}$=ip$_{\mathrm{A}}$ tcp tcp$_{\mathrm{dst}}$=tcp$_{\mathrm{A}}$	ip$_{\mathrm{src}}$=ip$_{\mathrm{B}}$ tcp$_{\mathrm{src}}$=tcp$_{\mathrm{B}}$ output=port$_{\mathrm{A}}$	TCP packets from X to A	Packets now appear to be from X/B to A	Rewrite L3 and L4 from X->A to B->A
2.2		mac$_{\mathrm{src}}$=mac$_{\mathrm{B}}$ output=port$_{\mathrm{A}}$	TCP packets from X/B to A	Packets now appear to be from B to A	Rewrite L2, and redirect from X->A to B->A
3.1	in_port=port$_{\mathrm{X}}$ ip ip$_{\mathrm{src}}$=ip$_{\mathrm{X}}$ ip$_{\mathrm{dst}}$=ip$_{\mathrm{Y}}$ tcp	output=port$_{\mathrm{Y}}$	TCP packets from X to Y	Packets relayed unmodified from X to Y	Send unaltered inter-agent packets from X->Y
3.2		output=port$_{\mathrm{Y}}$	TCP packets from X to Y	Packets relayed unmodified from X to Y	Send unaltered inter-agent packets from X->Y
4.1	in_port=port$_{\mathrm{Y}}$ ip ip$_{\mathrm{src}}$=ip$_{\mathrm{Y}}$ ip$_{\mathrm{dst}}$=ip$_{\mathrm{X}}$ tcp	output=port$_{\mathrm{X}}$	TCP packets from Y to X	Packets relayed unmodified from Y to X	Send unaltered inter-agent packets from Y->X
4.2		output=port$_{\mathrm{X}}$	TCP packets from Y to X	Packets relayed unmodified from Y to X	Send unaltered inter-agent packets from Y->X

Table 5.2.2: OpenFlow flows required to transparently insert SOS agents X and Y into a TCP connection between hosts A and B. These flows indicate the flows required if the hardware switches of the SOS deployment do not support all the required packet header matches and actions. In this case, the unsupported matches and actions can be supplemented through a software switch on the SOS agent. These flows are continued on Table 5.2.3.

Flow	Flow Match	Flow Action	Preconditions	Postconditions	Description
5.1	in_port=port$_X$ ip ip$_{src}$=ip$_X$ ip$_{dst}$=ip$_Y$ tcp	output=port$_Y$	TCP packets from X to Y	Packets relayed unmodified from X to Y	Send unaltered inter-agent packets from X->Y
5.2		output=port$_Y$	TCP packets from X to Y	Packets relayed unmodified from X to Y	Send unaltered inter-agent packets from X->Y
6.1	in_port=port$_Y$ ip ip$_{src}$=ip$_Y$ ip$_{dst}$=ip$_X$ tcp	output=port$_X$	TCP packets from Y to X	Packets relayed unmodified from Y to X	Send unaltered inter-agent packets from Y->X
6.2		output=port$_X$	TCP packets from Y to X	Packets relayed unmodified from Y to X	Send unaltered inter-agent packets from Y->X
7.1	in_port=port$_Y$ ip ip$_{dst}$=ip$_B$ tcp tcp$_{dst}$=tcp$_B$	ip$_{src}$=ip$_A$ tcp$_{src}$=tcp$_A$ output=port$_B$	TCP packets from Y to B	Packets now appear to be from Y/A to B	Rewrite L3 and L4 from Y->B to A->B
7.2		mac$_{src}$=mac$_A$ output=port$_B$	TCP packets from Y/A to B	Packets now appear to be from A to B	Rewrite L2 and redirect from Y->B to A->B
8.1	in_port=port$_B$ ip ip$_{src}$=ip$_B$ tcp tcp$_{src}$=tcp$_B$	mac$_{dst}$=mac$_Y$ output=port$_Y$	TCP packets from B to A	Packets now appear to be from B to Y/A	Rewrite L2 and redirect from B->A to B->Y
8.2		ip$_{dst}$=ip$_Y$ tcp$_{dst}$=tcp$_Y$ output=port$_Y$	TCP packets from B to Y/A	Packets now appear to be from B to Y	Rewrite L3 and L4 from B->A to B->Y

Table 5.2.3: OpenFlow flows required to transparently insert SOS agents X and Y into a TCP connection between hosts A and B. These flows indicate the flows required if the hardware switches of the SOS deployment do not support all the required packet header matches and actions. In this case, the unsupported matches and actions can be supplemented through a software switch on the SOS agent. This is a continuation of Table 5.2.2.

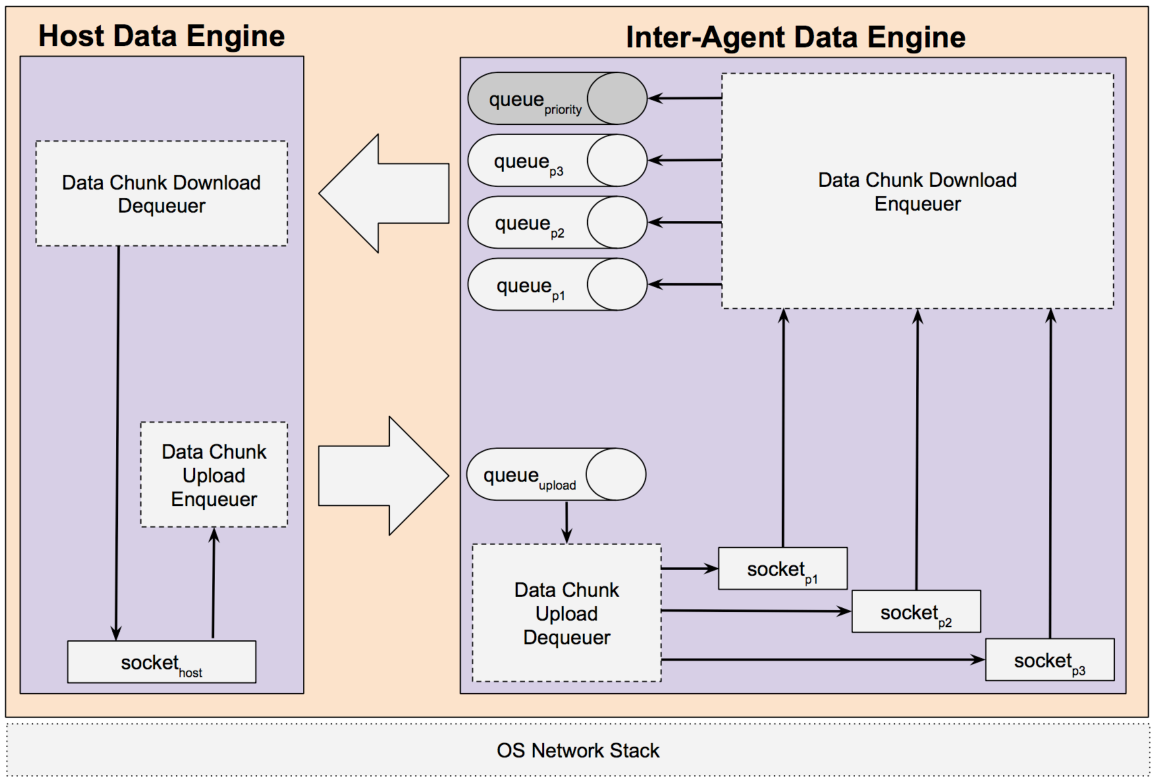


Figure 5.2.5: SOS agent data forwarding software architecture

of the agent uses multiple TCP connections in parallel as the throughput-improving mechanism within the inter-agent data engine. Data received from the host by the host data engine is broken into fixed-length chunks, is prepended with a 4-byte sequence number, and is placed into an upload queue in the inter-agent data engine. The inter-agent data engine processes this queue by sending data chunks contained within on one of many TCP connections open in parallel between the two SOS agents' inter-agent data engines. The TCP connection chosen is the first available connection. When the data is received by the inter-agent data engine at the corresponding agent, the data chunk is removed, and if the sequence number at the beginning of the data chunk is the next sequence number, it is transferred directly to the host data engine. Otherwise, the desired data chunk has not arrived yet, and the current data chunk and its sequence number is placed into a download queue associated with the given parallel TCP socket. Additional data chunks are received and queued in a similar manner until the data chunk with the desired sequence number arrives, at which point it is relayed directly from the inter-agent data engine to the host data engine. Each time the host data engine is provided with a data chunk, any subsequent data chunks that may have arrived out of order and are cached in the inter-agent download queues are dequeued and provided to the host data engine in order. Due to the queue implementation, only the sequence number of the first element of each download queue is queried.

The use of TCP in the inter-agent data engine guarantees data chunks relayed across and presented by each of the parallel TCP connections is done in order and is done reliably. Likewise, the sequence numbers and queues used by the inter-agent data engine guarantee data integrity across the aggregate inter-agent TCP connections with minimal overhead.

The SOS agent is capable of supporting multiple client/server TCP sessions simultaneously. However, since the SOS agent is a software application, CPU time of each session may be reduced as more sessions are handled simultaneously. On inadequate hardware, this can introduce a bottleneck. As such, the use of multiple agents in agent clusters is an encouraged scale deployment option discussed in detail in Section 5.2.5.

The SOS agent application used to gather results discussed later in this dissertation was written in C. However, there are no restrictions on which programming language the SOS agent can be implemented.

5.2.3.1 Selecting and Tuning SOS Agents

In this section, the SOS agent is discussed, in particular hardware requirements imposed by the desired data transfer capacity and the agent software design. Discussion is divided into three logical components: Section 5.2.3.2 discusses the compute requirements of the SOS agent machine, Section 5.2.3.3 explains the networking requirements of an SOS agent, and Section 5.2.3.4 shows how agent memory is related to its performance.

When attempting to fill the pipe over the large delay bandwidth product network, the ability of the agents to perform this task needs to be considered. Given that agents are software applications designed to run on general purpose hardware, this ability can be defined primarily by the the CPU speed and the memory installed in the system [81]. Modern system buses are high performance and are not bottlenecks when processing and simply moving data. In SOS, the storage capacity of an agent is of little concern, since the agents are not designed to cache data but instead quickly receive, process, and send data, as if the agent is a network forwarding device.

The amount of data an agent can handle at a given time is limited by the total RAM installed in the system. As an agent's installed RAM decreases, the agent will be less capable of handling large sets of data without the performance-hindering use of hard disk swap space. As such, the more RAM an agent has, the more data it will be able to handle at a given time in an attempt to keep up with the line rate speed of its network card. However, simply increasing the agent RAM is not going to yield high throughput. The processing ability of the CPU must also be capable of ushering data into and out of RAM as the agent moves data to/from its remote agent and its local client/server. The faster the CPU, the more data the agent is able to process per second.

The agent application has three parameters that can be tuned through the SOS controller REST API in order to fit the desired performance or hardware abilities of a particular agent:

1. The number of TCP sockets to be used by the inter-agent data engines, denoted by p
2. The data chunk size in bytes moved between agents, denoted by c
3. The data chunk queue capacity associated with each TCP connection, denoted by q

p exploits TCP parallelism benefits already proven effective in the related works. c can be used to balance the number of system calls required for network I/O. It also has a significant influence on the amount of RAM required by an SOS agent. Lastly, q can be used to improve performance in lossy networks or networks that are more likely to deliver data chunks out of order, such as those load balancing inter-agent data engine TCP connections across multiple paths. The queue defined by capacity q allows the agent to cache data chunks received from inter-agent data engine parallel connections in the event the data chunk received is not the next sequence data chunk requested from the host data engine. As such, q also has a significant impact on the amount of RAM required for the the SOS agent; however, it has little impact on the CPU.

Relationships between these three parameters and the hardware capabilities of an SOS agent in terms of CPU and RAM are discussed in the following subsections. Section 5.2.3.2 discusses how the CPU of the SOS agent relates to its performance. Section 5.2.3.3 discusses the network requirements and recommended configuration for an SOS agent, while Section 5.2.3.4 discuses the memory utilization and requirements of an SOS agent.

5.2.3.2 Compute

Due to the design of the agent software, more CPU time is required to receive and reorder a set of parallel TCP connections from the inter-agent data engine than it does to chunk/divide and send data at the source. Thus, between two comparably equipped agent machines, the sync of the data transfer is the bottleneck and should be used in lieu of the source in bottleneck calculations.

Because the goal of an SOS agent is to move data as quickly as possible between its remote agent peer and the local client or server, it is important that the compute resources of the agent be able to process the data delivered from the network at the speed the network can deliver it. There are many strategies one can execute in order to increase processing performance of an agent.

Modern CPUs are highly sophisticated and due to the variety of CPU architectures and their complexity, it is not feasible to derive a precise computation for an SOS agent's CPU requirements. However, an approximation can be determined based on the following key assumptions:

1) The bus between the network card (e.g. PCI express) is much faster than the CPU. This allows us to neglect the bus speed and assume it as instantaneous.

2) The network card uses direct memory access (DMA). DMA allows the card to have access to RAM and directly read/populate the send/receive buffers of the driver without CPU intervention.

3) The agent machine itself has at least three CPU cores – one for OS tasks and other processes,

one for the agent processes, and one for handling network card interrupts. Such task isolation resulted in a gain of approximately 1.5Gbps throughput using SOS on a test setup using an Intel Core i5 2400 CPU. The agent and network card interrupt handling cores should be on the same physical CPU to reduce latency. Note that the agent process is serial since network I/O is serial in nature; however, there is a parent process that forks a child for each client the agent is presently handling.

4) The agent machine is a 64-bit hardware architecture with a 64-bit Linux host OS with an up-to-date (2.6+) kernel. 32-bit architectures are problematic due to statically mapped kernel space memory (typically 3GB user and 1GB kernel for maximum of 4GB RAM), which prevents increasing the aggregate TCP memory to satisfy large delay-bandwidth-product networks when using multiple TCP connections simultaneously. 64-bit systems do not have such a limitation, and the kernel space memory can be defined arbitrarily high (up to physical RAM limitations) through the modification of system parameters [82].

5) The agent process is configured with a "nice" parameter of -20. This gives the agent process as much CPU time as possible on its dedicated CPU core [83]. Nice values can range from -20 to 19, where -20 indicates the process is high priority and should receive the most CPU time and 19 indicates the process is low priority and should receive the least CPU time.

Taking the above assumptions into account, a model can be derived to approximate the work per unit time required of the SOS agent process in order to effectively usher data from one TCP connection into another. As described by [84], the processing of data received can be broken into three main components:

1. Network Card and Driver

   (a) Orchestrates DMA to driver buffers in RAM

   (b) Triggers hard interrupt to invoke driver post data transfer

   (c) Queues soft interrupt for processing by kernel

2. Kernel Protocol Stack

    (a) IP processing, e.g. firewall, routing
    (b) Queues soft interrupt to transport protocol stack
    (c) Full copy of buffer for TCP processing

3. Data Receiving Process

    (a) OS scheduling
    (b) epoll notification of data present on a socket in p
    (c) Receive system call
    (d) Copy from TCP receive buffer to application. The copy length is dependent on the bytes requested from the application's receive system call.
    (e) Process the data in the SOS agent application
    (f) Send the data into another TCP socket from the SOS agent application

Note that each of (1), (2), and (3) above is executed in isolation on different CPU cores. The linkage between each distinct task is achieved through soft interrupts that are coalesced and processed periodically in batches. As such, the problem can be simplified by examining the throughput required by each core for each task.

The ability of (1) to move data is directly linked to the size of the driver's Ethernet frame receive buffer. The buffer is used most effectively when each position in the buffer can contain more data. The length of each position is simply the Ethernet MTU. As such, using the highest MTU the network card can afford is advantageous in that it gives the device driver the ability to buffer more data at any given time. Since it is assumed that DMA is used to transfer the data from the network card to RAM, the CPU is not used to perform the data transfer. As such, the CPU handling the network card interrupts should be capable of running the hard interrupt service routine for each frame received to initiate the data transfer, running the hard interrupt service routine for each frame transferred to terminate the data transfer and queue a soft interrupt, and batch-execute the soft interrupts, which inform the kernel running on another CPU of the data waiting for it in the network card's driver buffer. Since network cards contain mixed and proprietary implementations for their driver interrupt service routines and their DMA implementations, the required CPU core capability can be defined for (1) by the CPU clock cycles required to accomplish each of (1a), (1b), and (1c).

(2) is tasked with processing each unit of data (i.e. 1 IP MTU) through the network stack. This includes analyzing the packet against any firewall rules, applying IP forwarding and routing rules, queuing a soft interrupt for TCP, and copying the encapsulated TCP header and data into the TCP engine upon execution of the soft interrupt. These tasks can be optimized using techniques such as TCP segment offloading. The compute resources required are dependent on the exact optimizations used, whether or not the firewall is enabled, and so forth.

Regardless of the exact implementations of (2) and (3), the CPU cores should be capable of achieving the throughput expected by the SOS agent process on the third CPU core. A rule of thumb is to ensure the CPUs with optimizations and rules present can handle the IP throughput of the network card.

Last in the CPU discussion is (3) – namely, where SOS agent CPU utilization comes into play. As discussed, the SOS agent is a multiplexer/demultiplexer, ushering data to and from TCP connections. The receiving SOS agent is tasked with the heaviest load, which is to gather data from p parallel TCP connections, ensure proper data ordering via q, and send the data out the single TCP connection of the host data engine. This task can be broken into subtasks (3a) through (3f) as outlined above.

To show how the agent's CPU can impact the throughput of a data transfer, the time required to process a data chunk of size$c$ can be determined. This is the processing delay of a data chunk, denoted by d_{chunk}. d_{chunk} is directly related to the CPU frequency f_{cpu}. CPU executes SOS agent code in the form of CPU instructions, which vary in the number of CPU cycles required to complete each instruction. This can be described as the average clock cycles per instruction, denoted by f_{inst}. The specific instructions executed by the SOS agent code are dependent on how the SOS agent code is compiled and the hardware on which the SOS agent is installed.

To avoid ties to specific hardware, logically distinct SOS agent operations can be expressed in terms of the number of machine instructions required to complete each operation. Operations include (1) reading and copying a data chunk from a TCP socket, (2) writing the data chunk to a TCP socket, (3) overhead from system calls, (4) determining the significance of the data chunk and preparing it for transport, and (5) overhead from epoll. Because c is variable in size, the instructions required for (1), (2), and (3) can be expressed in terms of c and the instructions required to handle the data in increments of the bus width, commonly referred to as a word, w. Thus, d_{chunk} can be computed as:

$$d_{chunk} = \left(\frac{1}{(f_{cpu})(u_{agent})}\right)(f_{inst}) * \left[(i_{rx} + i_{tx})\left(\frac{c}{w}\right) + i_{analyze} + i_{epoll} + 2i_{syscall}\right]$$

where u_{agent} is the average CPU utilization percentage of the SOS agent process, i_{rx} and i_{tx} are the instructions required to receive and send one word w of data, $i_{analyze}$ is the number of instructions required to analyze the data chunk, i_{epoll} is the number of instructions necessary for epoll to monitor and signal the arrival of a data chunk including the soft interrupt queued by the kernel, and $i_{syscall}$ is the instructions required to handle the I/O system calls and context switches when reading and writing the data chunk. i_{epoll}, is not dependent on the number of parallel TCP connections p, unlike other polling mechanisms such as poll and select. As such, it has uniform overhead when detecting and dispatching the file descriptor write event triggered by the soft interrupt of data present [85]. i_{rx} is predominantly the time required to copy c bytes of data into the SOS application. Due to the implementation of the agent process, if there are not c bytes of data waiting when the receive system call is made (triggered by an epoll soft interrupt handler), the agent will block until the data is available. As such, for highest efficiency, it is important to optimize the system such that the read system call can copy all w bytes of data and return in one invocation. Repeated system calls to read the remaining bytes of c increases overhead due to context switches from the agent process to the kernel.

The processing delay of a data chunk, d_{chunk} can be used to determine how quickly an agent machine will be able to process a given chunk size of data, from reception to transmission. It can be used to diagnose performance bottlenecks for a chosen chunk size c, on a chosen system architecture defined by word size w (typically 8 bytes for modern 64-bit architectures) and f_{inst}, and the actual or target percentage of CPU being consumed for the agent process, u_{agent}. A code profiler can be used to determine the values of $i_{analyze}$, i_{epoll}, $i_{syscall}$, i_{rx}, and i_{tx} for the agent binary as compiled for a particular system and architecture.

5.2.3.3 Network

The inter-agent data engine of an SOS agent uses many TCP connections in parallel in order to achieve improved throughput over large delay-bandwidth-product links. This section discusses how this is accomplished and how the desired amount of throughput improvement can be tuned.

The default maximum TCP window size of the Linux kernel is 64KB [82]. This means that at most 64KB can be sent on the link, outstanding, without an acknowledgement. Modern, large delay-bandwidth-product links are capable of buffering far more than this. For example, a 10Gbps link between the states of South Carolina and Utah in the United States has an approximately 53ms round-trip latency as determined through a simple ping. This particular link can buffer 10Gbps x 0.053s or 63.18MB of data – far more than the maximum allowable TCP widow size. To eliminate TCP as a bottleneck, SOS agent Linux kernels are configured with TCP window scaling enabled, which expands the TCP window size to theoretical maximum of 1GB. However, achieving and sustaining this full 1GB window in practice is difficult due to TCP congestion control algorithms and network packet loss. The use of multiple TCP connections in parallel in conjunction with TCP window scaling between the SOS agents provides each TCP connection with the potential to achieve an average window size greater than the kernel default of 64KB while at the same time limiting the effect of packet loss to the window of the single TCP connection that experienced the loss. It is assumed in this discussion that all SOS agents are configured with TCP window scaling enabled in the kernel.

Let t_{agent} denote the maximum throughput in bits per second of an SOS agent. Likewise, let $t_{network}$ denote the maximum throughput of the network between the client, server, and the SOS agents. This is the minimum cut of the network graph of all path(s) used in the data transfer. It is a defined value based on link speeds. Let t_{sos} denote the total theoretical maximum throughput in bits per second of SOS for a given client-to-server TCP connection, where $t_{sos} = min(t_{agent}, t_{network})$. Clearly, $t_{network}$ cannot be physically surpassed, so in order to maximize throughput, one or more SOS agents can be configured such that cumulative t_{agent} approaches $t_{network}$. The use of multiple agents might be necessary depending on the SOS parameters used and/or the characteristics and limitations of individual SOS agent machines.

In order to avoid the need to account for differences in user data sizes, it is convenient to think of t_{agent} in terms of an SOS data chunk size c and the delay incurred by the agent to process c bytes as d_{chunk}:

$$t_{agent} = \frac{8c}{d_{chunk}}$$

t_{agent} is the throughput attainable by a single SOS agent when ushering data from one TCP connection to another. It is not indicative of the network throughput, since t_{agent} does not include overhead present at each layer in the network stack. The maximum throughput of an SOS agent in bits per second, t_{agent}, is useful to evaluate the proposed performance of and diagnose performance problems of an SOS agent. In particular, it helps show the relationship between the agent data chunk size c, and the actual or desired throughput. Note that the number of parallel connections, p, and queue size, q, do not influence this computation. Its scope is limited to the performance of a single SOS agent in an SOS deployment and not the performance between two SOS agents across the long distance link.

The total header overhead in bytes $h_{network}$ over an IP network over Ethernet is the sum of the byte overhead from Ethernet framing $h_{framing}$, the Ethernet header $h_{ethernet}$, the IP header h_{ip}, and the TCP header h_{tcp}:

$$h_{network} = h + h_{ethernet} + h_{ip} + h_{tcp}$$

As an example, for an Ethernet link using a single VLAN tag, IPv4, and TCP, this is precisely:

$$h_{network} = 24 + (14 + 4) + 20 + 20 = 82\, bytes$$

Now, in order for SOS to ensure data integrity, a 4-byte sequence number is appended to the beginning of each data chunk before it is sent on a parallel TCP connection. This is to ensure proper, ordered delivery on the receiving end of the connection. Thus, we must account for this additional overhead:

$$h_{agent} = 4$$

$h_{network}$ is applicable for each unit of data transferred over the network, which is restricted in size by the maximum transfer unit (MTU) in bytes over all links between the SOS agents. For convenience, let $m_{ethernet}$ denote the lowest MTU in bytes of the Ethernet network(s) traversed between the agents and m_{tcp} denote the maximum payload of application data that can be contained in a single TCP packet. We can relate m_{tcp}, m_{eth}, and h_{total} by:

$$m_{eth} = h_{network} + m_{tcp}$$

The total network utilization of the agent, u_{agent}, can then be defined as:

$$u_{agent} = \begin{cases} \frac{8(m_{eth}\lfloor\frac{c}{m_{tcp}}\rfloor + h_{agent})}{d_{chunk}} & c \bmod m_{tcp} = 0 \\ \\ \frac{8(m_{eth}\lfloor\frac{c}{m_{tcp}}\rfloor + h_{agent} + h_{network} + c \bmod m_{tcp})}{d_{chunk}} & otherwise \end{cases}$$

The total network utilization of an SOS agent, u_{agent}, is useful to determine the impact of network stack parameters on an SOS agent. In particular, note that u_{agent} demonstrates the significant impact of MTU on the network overhead. As m_{tcp} increases due to MTU increase, more chunk data can be included with a given, fixed-length header of $h_{network}$. The lower MTU becomes, the more headers of size $h_{network}$ will be required to transmit a data chunk c. Note that u_{agent} does not include the number of parallel connections, p. It more simply measures the raw "chunk moving" performance of an SOS agent. It is irrelevant if the chunk movement is from a single TCP connection to another single connection or from one TCP connection to one of many TCP connections in parallel. The performance of the agents over the long distance link is a function of the inter-agent engine – parallel TCP for the SOS agents discussed in this dissertation. The positive effects of parallel TCP have been well-studied and thus are not analyzed in this dissertation. With sufficient parallel TCP connections, p, the inter-agent data engine will be capable of keeping the pipe full at the rate defined by u_{agent}. With a value of p that is too low, u_{agent} will be greater than the combined throughput achievable by each TCP connection in p, and the bottleneck shifts from the SOS agent's compute resources to the TCP windowing problem over the large delay-bandwidth-product link.

5.2.3.4 Memory

In order to process data at a rate defined by t_{agent}, the agent machine must be able to allocate sufficient RAM for the handling of data as it is in transit from the parallel TCP connections of the inter-agent data engine to the single TCP connection of the host data engine. The host operating system (OS) must allocate two buffers of memory for each TCP socket that it will use to for reading and writing data, w_{tcp}. On the receiving side, this buffer should be large enough to hold an entire TCP window's amount of data under the condition that the agent process can read an entire TCP window's amount of data as soon as it becomes full. A SOS agent must be capable of serving as either the source or the destination of a file transfer, so the amount of RAM in bytes required for the OS TCP buffers for all parallel TCP connections p and the single TCP connection serving the client/server is defined by r_{tcp}. It consists of the buffer for the TCP window itself per socket, the OS read buffer, and the OS write buffer:

$$r_{tcp} = 3w_{tcp}(p + 1)$$

Once the agent process reads data from a parallel TCP connection, it must place it in a temporary buffer prior to transmission on the single TCP connection to ensure the proper ordering of data. This buffer must be able to contain the largest unit of data possible, which is c bytes, the data chunk size. As such, the minimum RAM required for the agent process can be defined as $r_{agent,min}$:

$$r_{agent,min} = r_{tcp} + pc$$

Under ideal circumstances, the agent will be able to receive and process data chunks of size c in order at all times, but a lost TCP packet or non-optimal OS scheduling could cause a delay in processing of the next data chunk while subsequent data chunks become available. Thus, the agent can support an internal queue for each TCP connection in the inter-agent data engine and the host data engine defined by length q. Each position of this queue holds an entire data chunk of data c as read into the agent application from the OS. Factoring in queuing, the bytes of RAM required by the agent process can be defined as r_{agent}:

$$r_{agent} = r_{tcp} + pwq$$

Note that when $q = 1$, $r_{agent} = r_{agent,min}$. A queue $q > 1$ is not necessary for SOS to function, however given sufficient RAM, it can improve performance on networks where contention is high or packet loss probability is relatively high.

To finish the discussion of RAM, since the agent is an application process on a general purpose machine, the OS and other critical processes on the agent machine require RAM to function. As such, the total system RAM requirements for the agent machine can be calculated as r_{total}, which is comprised of the agent application itself, r_{agent}, and other usage by the system, r_{system}:

$$r_{total} = r_{agent} + r_{system}$$

Any machine to be used as an SOS agent should undergo an evaluation to ensure it meets the minimum RAM requirements.

5.2.4 SOS Floodlight OpenFlow Controller

The SOS OpenFlow controller leverages SOS agents and OpenFlow switches in the network topology to provide an autonomous network service. Discussion of the SOS OpenFlow controller is partitioned into the following. Section 5.2.4.1 introduces the Floodlight controller and features leveraged by SOS. Following that, Section 5.2.4.2 presents SOS management APIs available to network operators. Next, Section 5.2.4.3 reveals the SOS controller's management strategy for SOS agents. Then, Section 5.2.4.4 shows how packets are handled that are not of interest to SOS. Such packets include background traffic on the network. And finally, Section 5.2.4.5 discusses how TCP packets of interest to SOS are handled in the controller.

5.2.4.1 The SOS Floodlight OpenFlow Controller

The SOS controller is implemented in Java on top of the open source Floodlight [74] OpenFlow controller.

Floodlight abstracts different features into Java classes called Floodlight modules. The controller comes equipped with a number of traditional network features, as well as features to provide useful abstractions to Floodlight module developers, such as an OpenFlow switch manager, packet processor, message listener, device manager, link manager, routing manager, and flow pusher. SOS is implemented as a Floodlight module and is incorporated within the Floodlight controller with minimal modification to Floodlight itself. Table 5.2.4 outlines the Floodlight services leveraged by the SOS Floodlight module.

The switch manger manages the OpenFlow switches throughout the SOS SDN. It takes care of the lower level, southbound details such as the OpenFlow handshake, flow table programming, and statistics gathering and presents logical switch abstractions to the SOS module. The SOS module uses these abstractions indirectly to instruct flows to be added or removed on the particular switch.

The packet processor does many core controller tasks, but in particular is responsible for receiving OpenFlow messages from connected OpenFlow switches. It dispatches these messages to modules

Floodlight Service	Exposed as IFloodlightService	Description
Switch Manager	IOFSwitchService	Handles switch connections and exposes as IOFSwitch objects.
Packet Processor	IFloodlightProviderService	Receives OpenFlow messages from IOFSwitch objects and notifies listeners.
Device Manager	IDeviceService	Listens for packet-in messages to monitor host locations in the network.
Link Discovery Manager	ILinkDiscoveryService	Discovers links between switches using LLDP through packet-in and packet-out messages.
Topology Manager	ITopologyService	Constructs a graph of the topology. Precomputes paths upon topology updates.
Routing Manager	IRoutingService	Looks up paths between two given endpoints.
Flow Pusher	IStaticEntryService	Provides flow management for modules.
REST API	IRestApiService	Implements a northbound REST interface.
Automatic Packet Forwarding	N/A	Listens for packet-in messages and reactively installs flows between the source and destination devices.

Table 5.2.4: Floodlight services used by the SOS Floodlight module

that register to receive notification of such message types. There are many message types defined by the OpenFlow protocol, but the packet-in message type is of interest to many modules. A packet-in message is a message from the switch to the controller that contains in its payload a packet from the data plane that either did not match any flows in the switch flow table or was explicitly sent to the controller by an output action in a flow. Packet-in messages are designed to allow controllers to monitor and react to data-driven network events. In Floodlight, packet-in messages are used by SOS and other modules to update the controller's understanding of the network state and to reactively install flows in the network.

In order to register for packet-in and other OpenFlow messages received, a module must register as a listener for a specific message type. Modules can specify they be notified before or after other Floodlight modules in order to form an ordered message processing chain. Modules that process the message early in the chain can influence whether subsequent modules should or should not process the message by returning a result of either "CONTINUE" or "STOP", respectively. The SOS module registers to receive packet-in messages after the link discovery manager, topology manager, and device manager and before the forwarding module. This allows SOS to handle TCP packets and agent packets to be processed by SOS, as discussed in Section 5.2.4.3. It also allows SOS to defer processing of non-whitelisted TCP connections to the forwarding module, as discussed in 5.2.4.4 and 5.2.4.5. The packet-in processing chain for the SOS Floodlight controller operates as depicted in Algorithm 5.1.

Algorithm 5.1 SOS Floodlight controller packet-in processing chain

```
1: $pktInListeners[0] \leftarrow LinkDiscoveryMgr$
2: $pktInListeners[1] \leftarrow TopologyMgr$
3: $pktInListeners[2] \leftarrow DeviceMgr$
4: $pktInListeners[3] \leftarrow SOS$
5: $pktInListeners[4] \leftarrow Forwarding$
6: loop$true$
7:     $packetInMsg \leftarrow$ getPacketInMsg
8:     for $pktInListener : pktInListeners$ do
9:         $result \leftarrow pktInListener$.receive($packetInMsg$)
10:        if $result == STOP$ then
11:            break
12:        end if
13:    end for
14: end loop
```

The device manager detects and manages hosts and their observed locations in the SDN. It depends on the switch manager and packet processor in order to be notified upon any packet-in message received from a switch. The device manager will then analyze the packet-in message to update its knowledge of the device based on the header source address(es) of the packet-in message's payload. The SOS module uses this information in order to learn the locations of the SOS agents, the clients, and the servers participating in SOS. These locations are used to assist in path finding operations, discussed below.

The link discovery manager also leverages the switch manager and uses link layer discovery protocol (LLDP) to discover links between OpenFlow switches. The topology manager leverages the links discovered by the link discovery manager and the switches given in the switch manager in order to create trees to represent the network topology, providing information on topology structure. The routing manager uses the trees computed by the topology manager to determine paths between points in the topology. The SOS module leverages the routing manager in order to query available network paths between clients and agents and servers and agents. It uses this information in order to algorithmically determine the closest SOS agent to a particular client or server. This is necessary, since SOS can only improve the throughput of the intercepted TCP connection if it gets forward to the local SOS agent. If the other/foreign SOS agent is inadvertently chosen, the TCP connection will still succeed, but it will require three traversals of the long link, most likely resulting in degraded performance.

There are many metrics that can be used to choose the closest SOS agent based on the paths provided by the routing manager. The routing manager was extended for SOS and contributed back to the Floodlight project in order to provide an option to choose the path cost most appropriate and reliable for the SOS deployment. Available path metrics are hop count, latency, bandwidth utilization, and bandwidth capacity. Tunnels, if present, can also be disadvantaged (i.e. penalized with a heavier weight) if desired. By default, the SOS module uses path latency in order to choose the closest SOS agent.

The flow pusher provides an method with which flows can be composed, assigned a unique name, and then installed on switches. It is a wrapper around the switch manager's integrated flow pusher functionality that provides in-controller memory of the flows installed. A module can reference a flow by its name defined in the flow pusher to update or remove the flow without needing to remember the specific flow matches, action, and other fields. SOS uses the data path identifier of the switch associated with each hop along the paths provided by the routing manager in order to instruct the

API	HTTP Method	Description
config	POST	Set global configuration
	GET	Get global configuration
agent	POST	Add an SOS agent
	DELETE	Remove an SOS agent
	GET	Get SOS agents
whitelist	POST	Add whitelist entry
	DELETE	Remove whitelist entry
	GET	Get whitelist entries
enable	POST	Enable SOS
disable	POST	Disable SOS
status	GET	Get SOS state and errors
stats	GET	Detailed performance stats

Table 5.2.5: REST APIs defined by the SOS Floodlight module

flow pusher to install flows along each switch in the path. SOS leverages the flow pusher's memory feature to programmatically remove all flow names associated with an SOS TCP connection upon connection termination.

5.2.4.2 SOS Management APIs

SOS is designed to be deployed in a production SDN and managed at a high level by service administrators. Floodlight includes a secure REST API, which can be readily extended by Floodlight modules. Modules define URIs and register these URIs with callbacks to be invoked when any HTTP method such as query, post, or delete is received at the registered path. The SOS module defines and adds several APIs to the Floodlight REST API in order to make SOS a service that can be managed directly via REST or with a REST-based user interface such as a command line or graphical/web interface. Table 5.2.5 shows the functionality exposed by the SOS REST API.

Global configuration consists of the number of parallel TCP connections to use by the SOS agent inter-agent data engines, the data chunk size, and the queue size – all of which can be used to fine tune SOS agent performance. The SOS agent APIs are used to add, remove, and query configured SOS agents. The whitelist APIs inform SOS about TCP connections on which it should operate. Non-whitelisted TCP connections will not be optimized by SOS. A whitelist entry consists of the server IP, client IP, the server TCP port number, and the longevity of the entry as expressed by a start time and a stop time. The SOS module can also be enabled or disabled entirely, which can be used to suspend SOS temporarily regardless of the whitelist entries. The runtime status of SOS can also be queried, giving useful information such as error conditions. And lastly, SOS statistics can be queried, giving detailed information about past and ongoing SOS-assisted data transfers, such as the participating server and client, the agents, the network paths used, and data transfer performance.

5.2.4.3 SOS Agent Management

After an SOS agent has been added to SOS using the APIs described above, the SOS controller must discover the location of the agent in the SDN. This is done using ARP where the controller sends ARP request packets within packet-out messages to all OpenFlow switches. Any ARP reply from an SOS agent sent in response is received by the SOS controller as a packet-in message and is processed by the device manager, which extracts and records the location of the host. As such the agent locations can be learned automatically. Furthermore, this provides the controller with the agent MAC address, which is critical for its use in the rewrite flows in order to maintain transparency. Once an agent is added and its network location and MAC address is learned, it can be used in an SOS-assisted data transfer.

There can be multiple SOS agents configured on an SOS controller. The controller can use these agents and their locations in order to select the best agent for the client and for the server when a whitelisted TCP connection is initiated. These agents can be deployed in clusters at opposing sides of a long-distance link, allowing the controller to utilize the aggregate abilities of the agents in the cluster to handle a multitude of TCP connections simultaneously. In the event there are multiple "best" agents available for a given client or server, the controller implements a simple load balancing algorithm and selects the agent with the least amount of load. Load is defined by the number of client-to-server TCP connections being handled. This algorithm is pluggable and can be interchanged with a more sophisticated agent selection strategy without impacting the rest of the SOS controller. A more sophisticated strategy might involve using simple network management protocol (SNMP) to gather agent CPU and other performance metrics indicative of load.

5.2.4.4 Non-SOS Packet Handling

Floodlight is also equipped to provide automated packet forwarding between hosts in an SDN mimicking a traditional (i.e. non-SDN) network. It accomplishes this using the learning switch and forwarding modules. The former installs flows as a result of a traditional learning switch algorithm that operates on packet-in events on a per-switch basis. The latter implements global device learning, and leverages all of the aforementioned Floodlight services in order to intelligently install flows proactively along the entire path between the communicating hosts. These modules exist primarily for Floodlight users to attain a functioning SDN more quickly, but they also assist developers by providing fallback mechanisms for cases where they wish to have automated, traditional packet forwarding employed on packets. The SOS module only wishes to intercept, rewrite, and redirect packets to be operated on by SOS. Other traffic that is either not TCP or is not to have its throughput improved gets ignored by SOS. Rather than dropping the undesired packets, SOS passes them to the forwarding module, which installs flows to forward the traffic to its intended destination. Forwarding's flows do not conflict with SOS's flows, since SOS utilizes a higher flow priority. Furthermore, Forwarding's flows and SOS's flows are precise to the point where there is no overlap that might cause a conflict.

5.2.4.5 SOS Packet Handling Workflow

The SOS Floodlight module registers itself with the packet processor as a packet-in message listener. Each time a packet-in is received, the SOS module is notified and has the opportunity to process the packet contained within the packet-in message's payload. SOS operates on TCP sessions that have been whitelisted and examines the IP and TCP headers of the packet in order to determine if the packet matches a whitelist entry. Recall, whitelist entries are used to specify the TCP connection on which SOS should operate and is defined as the client and server IP addresses and the server TCP port. If the TCP packet headers do not match a whitelist entry, the SOS module relinquishes the packet, returning "CONTINUE", and allows the Forwarding module registered later in the packet-in processing chain to automatically install flows that match and forward subsequent TCP packets in the session between the client and server without the assistance of SOS. However, if matching whitelist entry is located, the SOS module will select an agent nearby the client and another agent nearby the server, and will install flows in order to intercept subsequent TCP packets in the session and implement SOS for the TCP session.

The flows are pushed in two sets. The first set is pushed based on the first TCP packet (the SYN) received as a packet-in from the client to the server. This TCP packet and the agents selected contain sufficient information to compose the flow match conditions and actions between the client and the client-side agent and between the client-side agent and the server-side agent. However, in order to install the flows between the server-side agent and the server, the client TCP port of the server-side agent needs to be learned. Thus, a second TCP SYN packet relayed to the SOS Floodlight controller as a packet-in from the server-side agent to the server is required in order for the SOS module to learn this missing piece of information and install the flows between the server-side agent and the server.

When an agent is selected for use in an SOS-assisted TCP connection, the controller sends the agent a UDP information packet as a packet-out. The packet contains the information necessary for the agent to implement SOS for the given TCP connection. The client-side SOS agent is provided with the client IP address and TCP port, as well as the IP address of the corresponding server-side SOS agent. This allows the client-side agent to handle the TCP connection between the client and the client-side agent and establish TCP connections in parallel between the client-side agent and the server-side agent. Likewise, the server-side SOS agent is informed of the client-side agent IP address in order to handle the TCP connections in parallel between itself and the client-side agent. It is also provided with the server IP address and TCP port in order to establish a TCP connection between the server-side agent and the server. The information provided in each of these UDP information packets is determined from the client-side TCP SYN and from the agents selected thereafter.

The SOS packet handling workflow is depicted in Algorithm 5.2.

5.2.5 Deploying SOS at Scale

When scaling up SOS, a variable number of agents can be installed at flexible locations in the network. Each of them need not be tuned to have the same capabilities but together the total achievable throughput can grow up to the link capacity as agents are added. As SOS scales, the underlying network infrastructure must be capable of supporting the bandwidth consumed by the simultaneous data transfers. Given the use of high rate 10Gbps, 40Gbps, 100Gbps, bonded links, and multiple paths in the core of many enterprise and backbone networks, this bottleneck becomes increasingly negligible. Nevertheless, it still exists and should be kept in mind when designing an SOS deployment.

An individual agent can become a bottleneck if ill equipped to process available data at the desired or line rate. Since SOS improves TCP data transfers, a single client cannot harness the aggregate capability of multiple agents simultaneously with a single TCP connection; by design of TCP, point to multipoint is not supported. However, a client can leverage the combined throughput improvement of multiple nearby agents by utilizing parallel TCP, on top of an SOS deployment. In such a scenario, each of the parallel TCP connections is handled as a separate TCP file transfer

Algorithm 5.2 Packet handling by SOS Floodlight module

```
1: if SOS enabled then
2:     $packet \leftarrow$ getPayload($packetIn$)
3:     $headers \leftarrow$ getPacketHeaders($packet$)
4:     if isTcp($headers$) $\wedge$ isOnSosWhitelist($headers$) then
5:         $srcHdrs \leftarrow$ getSrc($headers$)
6:         $dstHdrs \leftarrow$ getDst($headers$)
7:         $srcDev \leftarrow$ getLearnedDevice($srcHdrs$)
8:         $dstDev \leftarrow$ getLearnedDevice($dstHdrs$)
9:         if $\neg$exists($srcDev$) then
10:            reportError
11:        else if $\neg$exists($dstDev$) then
12:            arpFor($dstDev$)
13:        else if $\neg$pendingFlowInstallComplete($headers$) then
14:            $clientAgent \leftarrow$ selectAgent($srcDev$)
15:            $serverAgent \leftarrow$ selectAgent($dstDev$)
16:            $caDev \leftarrow$ getLearnedDevice($clientAgent$)
17:            $saDev \leftarrow$ getLearnedDevice($serverAgent$)
18:            if $\neg$exists($caDev$) $\vee$ $\neg$exists($saDev$) then
19:                reportError
20:            else
21:                notifyAgent($clientAgent$)
22:                notifyAgent($serverAgent$)
23:                pushFlows($srcDev, caDev$)
24:                pushFlows($caDev, saDev$)
25:                sendPacketOut($packet$)
26:                SOS flow install pending complete
27:            end if
28:        else                                    ▷ finish installing flows on server side
29:            pushFlows($srcDev, dstDev$)
30:            SOS flow install complete
31:        end if
32:        return stopProcessing($packetIn$)
33:    end if
34: else
35:    return passToForwarding($packetIn$)
36: end if
```

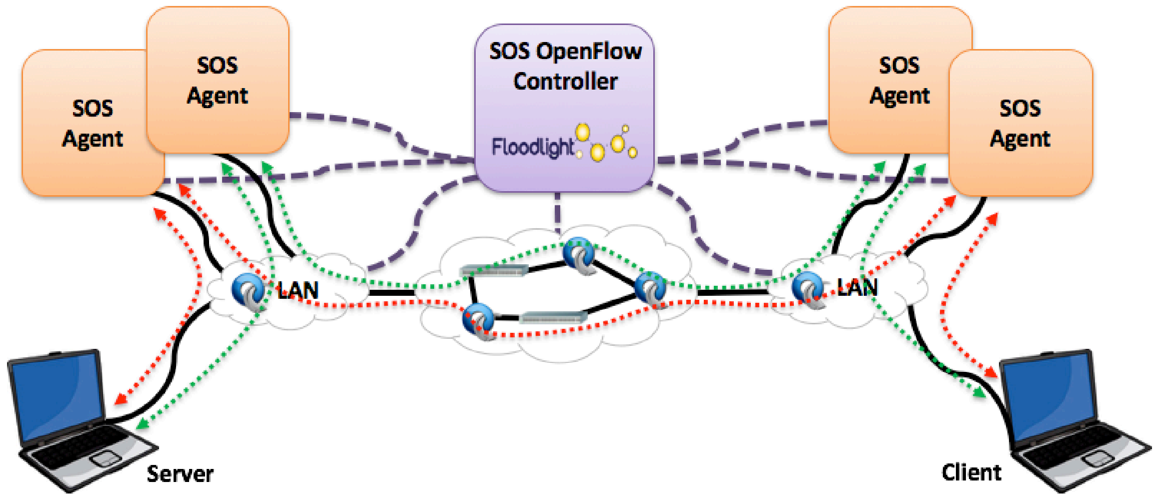


Figure 5.2.6: An example use case demonstrating SOS scalability using parallel TCP or multiple TCP connections layered on top of the SOS framework.

in the SOS framework, utilizing a separate SOS agent for each. Figure 5.2.6 provides an example to demonstrate how the use of parallel TCP by the end user data transfer application can leverage the ability of multiple SOS agent pairs simultaneously. Naturally, since SOS can support parallel TCP from a single end user, SOS can also scale out to support multiple simultaneous data transfers involving different end users.

When layering these parallel TCP techniques on top of SOS, there is no additional overhead within SOS, which treats each TCP connection as an individual data transfer. As discussed in Section 5.2.3, the agents themselves introduce a slight overhead through the use of sequence numbers in data relayed between SOS agents, but this overhead is present even for a single SOS data transfer. Any overhead present in the parallel TCP-implementing application layered on top of SOS is dependent on the application chosen by the end user. Such overhead is a tradeoff that should be considered by the end user independent of SOS.

The SOS architecture supports the layering of parallel TCP techniques on top of SOS, as discussed above. Below, we show how existing TCP-based solutions to long distance data transfer can work collaboratively with the SOS architecture to form a rich data transfer ecosystem. However, recall that the end user TCP connection is transparently proxied by the SOS agents. This means that SOS can work collaboratively with end user applications that employ throughput enhancing techniques as long as those techniques do not assume the use of a non-standard network stack [86]. Given a non-standard TCP/IP network stack on the client and server that is not also installed on the SOS agent, SOS may not be able to operate on the TCP connection, or the complementary TCP throughput enhancing technology might have limited or no impact on throughput achieved.

To overcome TCP windowing problems, the use of parallel TCP connections has been proposed and is proven as a viable solution to increasing the throughput of large data transfers over high latency networks [46]. pTCP is a parallel TCP solution that utilizes different network interfaces to transmit multiple TCP connections across multi-homed devices, thus making use of the aggregate bandwidth available [51]. There are many multi-path TCP implementations [52, 53] that take advantage of multipath topologies in order to increase aggregate bandwidth. MP-TCP requires the addition of metadata to TCP, which allows MP-TCP to manage the multiple TCP connections that span potentially multiple paths and involve multiple IP addresses. Specifically, MP-TCP uses secondary data sequence numbers within the TCP payload. Because these sequence numbers are included as a part of the TCP payload, they will be transparent to SOS – appearing simply as data – and will be able to pass from client to server without interference. However, MP-TCP requires custom TCP options to negotiate the number of parallel TCP connections to open, as well as to authenticate the establishment of the multiple TCP connections. These custom TCP options are included in the TCP handshake as a part of the TCP header and will be ignored by a TCP implementation not expecting such options, such as an SOS agent or any machine not equipped with MP-TCP. As a result, the custom TCP options required by MP-TCP will not propagate through SOS from the source MP-TCP equipped host to the destination. Since MP-TCP will fall back to traditional TCP, it is compatible with SOS, but it is important to note that the combined benefits of MP-TCP and SOS would not be visible; only SOS would be used to improve throughput. In general, TCP techniques that use an alternative TCP/IP network stack will not layer nicely with SOS unless the SOS agent machines are also equipped with the same network stack modifications.

Globus GridFTP [47] is a popular data transfer solution that supports parallel TCP and file striping from parallel servers. Like other TCP-based solutions discussed, GridFTP requires the installation of non-standard software on both the client and the server of the data transfer, which can be difficult for average users or those who do not have administrative access to the remote resource. Despite this limitation, it is worth noting that GridFTP and other standard TCP-based data transfer techniques can be layered on top of the SOS solution. From SOS's perspective, each GridFTP data stream is merely a TCP connection. SOS will perform data transfer optimization on each single TCP connection used by GridFTP.

WAN optimization through deduplication and data caching is also a popular data transfer solution. By caching commonly requested data in local stores, they avoid the retransmission of data from remote repositories and instead provide the locally-cached content to local users. However, to be effective, this technology relies on predictable network user behavior and frequent access to the same content. It requires more than one request to the remote data to be effective, since the initial request will be performed without any optimization. To fill this gap, SOS can also be layered with WAN optimization techniques to provide a more fully-featured data transfer solution. For frequent data requests, SOS will optimize the first request over the WAN, where data caching is then performed to serve subsequent requests. This allows the initial request to benefit from improved throughput. It also allows all requests that cannot be cached, such as those using encryption, to benefit from an improved data transfer experience.

Although it is not TCP-based, Aspera [48] has developed a rapid and reliable file transfer technology that operates using UDP for data transfer along with TCP feedback connections that ensure reliability. Like other solutions, Aspera requires custom software installed on both the client and the server conducting the data transfer. For the same reasons cited above, the use of Aspera by average network users can be problematic. Despite this, Aspera can be used in an SOS deployment between hosts equipped with Aspera. The Floodlight controller managing the SOS network will simply install flows to pass the UDP data and TCP control traffic between Aspera endpoints without the assistance of SOS.

5.2.6 Design Decisions and Assumptions

The design of the SOS architecture is founded on the agent-based framework. As shown in Figure 5.2.7, the SOS agents themselves implement framework agents and software forwarding devices, while the Floodlight controller implements the controller plugin through the SOS module.

5.2.6.1 Agent Discovery

A primary design goal of SOS is to make it highly and easily deployable with a large variety of supported network topologies. Given that SDN is still an emerging technology, unless an organization replaces their entire network with an SDN, it is unreasonable to expect an entirely SDN-enabled network on which SOS can operate. Thus, emphasis is placed in making SOS interoperable with

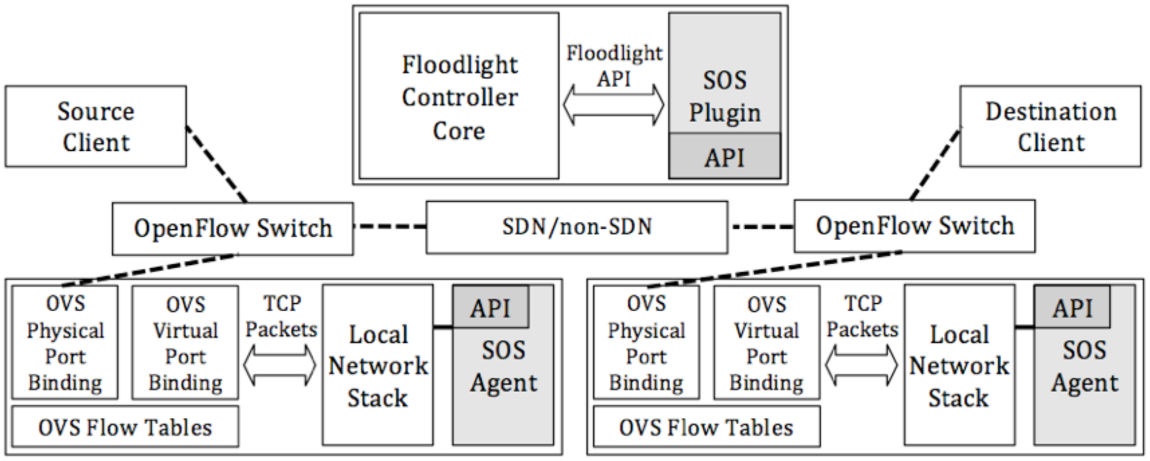


Figure 5.2.7: The influence of the agent-based framework in the design of the SOS architecture

traditional network forwarding devices in the data plane.

When an SOS agent is deployed, the controller needs to know about the agent, how to address it, and where it is located in the network topology. Naturally, there are two options: (1) implement an agent discovery protocol or (2) explicitly register the agents with the controller. An agent discovery protocol can be implemented by processing OpenFlow packet-in messages sent to the controller by OpenFlow switches in the SDN in which the agent is installed. The use of LLDP by the agents with custom TLVs is a possibility, but it will not propagate to the nearest OpenFlow switch if there is a non-SDN between the agent and the nearest OpenFlow switch. Likewise, other broadcast and multicast techniques are also not guaranteed to propagate through a non-SDN. Since the addition of an agent is a deliberate task, it is reasonable to require the agent be explicitly added to the controller, which was a design decision made for SOS.

5.2.6.2 TCP Connection Selection

In a similar manner, when designing SOS, a decision needs to be made on how to choose which TCP connections to enhance. Assuming operation in a busy network, there are likely to be many TCP connections in use. Some of these will be short lived, such as loading small web pages. Others might not operate at a very high or consistent bitrate, such as SSH or TELNET. Since SOS is designed to improve the performance of large data transfers over long links, filtering out TCP connections that do not meet these criteria is a requirement in order to not oversubscribe the SOS infrastructure with TCP connections it was not intended to benefit. As such, SOS requires explicit whitelisting of the client and server IPs, as well as the server TCP port. Because of this, it is also assumed this server port only serves or receives data that meets these criteria. It is understandable that this latter assumption might not hold true for all cases. Under such circumstances, the SOS controller's packet-in processing logic can be readily extended to perform deeper packet inspection into application headers. For example, HTTP headers could be inspected to select whitelist specific files or paths.

5.2.6.3 Agent Proximity and OpenFlow Switch Deployment

Although SOS can provide transparent performance improvement for end users, it must be deployed correctly in order for maximum performance to be achieved and for the TCP connection to be handled correctly. It is assumed that for each whitelisted TCP connection, network administrators have deployed an SOS agent nearby the client and another SOS agent nearby the server. Furthermore, it is assumed that there is an OpenFlow switch in the path between the client and server at each side of the long distance link to perform the flow operations required.

5.2.6.4 Layer 3 Operation

Lastly, although the SOS architecture supports layer 3 networks between the SOS agents and between the clients, servers, and nearby SOS agents, the SOS controller discussed in this dissertation does not currently support layer 3 routing and path-finding. The implementation of these features is not necessary to deploy SOS and to discuss general architecture, scalability, and performance, which is the focus of this part of this dissertation. Future work is planned to integrate layer 3 routing support into the Floodlight controller, a necessary prerequisite for the SOS module.

5.2.6.5 Multi-Agent Use

Since SOS is designed around the TCP windowing problem and only applies to TCP (and similarly windowed and reliable protocols), a single client cannot utilize multiple local agents simultaneously for a single TCP connection. By design of TCP, the client can only relay data to/from a single machine over a single TCP connection. However, a client can increase throughput beyond what a single SOS agent can provide and up to the link capacity by utilizing other proven throughput enhancing methods, such as parallel TCP, on top of an SOS deployment. However, it is important to note a fundamental requirement for the layering of SOS with other throughput enhancing solutions.

The complementary TCP throughput improvement technologies must not modify the transport or lower layers in the OSI model [86] if they are to be used in conjunction with SOS. Because SOS operates on standard TCP connections, such modifications to the protocol stack on the client and/or server might not transparently pass through the SOS middleware. As such, either SOS might not be able to operate on the TCP connection, or the complementary TCP throughput enhancing technology might have limited or no impact on throughput achieved.

5.3 Evaluation

5.3.1 Cloud-Based Environment

The value of SOS is not limited to high speed networks dedicated to data transfers. In fact, SOS has shown promise as a solution to increase file transfers over the public Internet. Figure 5.3.1 shows SOS deployed using GENI resources. An SOS agent, client, and a redirection software OpenFlow switch were deployed at MAX InstaGENI. Another SOS agent, a server, and a redirection software OpenFlow switch were deployed at Stanford InstaGENI. These aggregates were chosen due to the large propagation delay between them.

Ubuntu 14.10 64-bit VMs with Open vSwitch (OVS) 2.3.0 preinstalled were used for the SOS agent machines. The "physical" OpenFlow switches used for packet redirection were emulated with VMs configured using the same Ubuntu 14.10 VM images with OVS 2.3 preinstalled. The client and server were also VMs but were using Ubuntu 14.10 64-bit VM images without OVS. Each VM shared a portion of the VM server's compute resources (Intel Xeon CPU E5-2450 2.10GHz) and contained 489MB of available RAM. No optimization was done to the network stack at any VM. However, each machine's IP MTU was lowered from the default of 1500 bytes to 1410 bytes to accommodate the additional overhead of a GRE tunnel linking the two sites as shown in Figure 5.3.1.

The SOS Floodlight OpenFlow controller was running on a lab machine with a public IP at Clemson. Each of the four OVSs in the GENI topology were connected to this controller. A series of disk-to-disk DNA data transfers were conducted between the VMs both using the SOS solution and operating in non-OpenFlow mode. For this test topology, non-OpenFlow mode is defined as each OVS instance disconnected from the controller and operating independent MAC learning algorithms.

Ten tests were conducted using wget transferring a tarred 10GB DNA data file to ascertain the

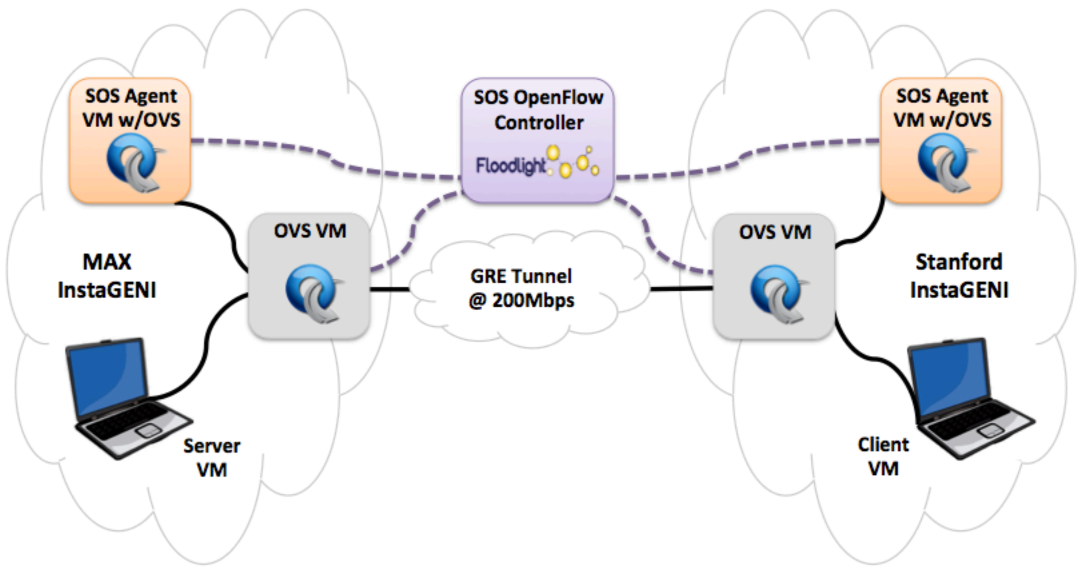


Figure 5.3.1: SOS on MAX InstaGENI and Stanford InstaGENI

Non-SOS Data Transfer	SOS-Assisted Data Transfer
15.6 Mbps	84.8 Mbps

Table 5.3.1: A performance comparison of a disk-to-disk, 10GB DNA file data transfer over a GRE tunnel, between VMs, on Stanford InstaGENI and MAX InstaGENI

performance of an SOS assisted data transfer as compared to that of a traditional, non-SOS assisted data transfer in a traditional networking environment. Results are given in Table 5.3.1.

As shown in Table 5.3.1, the same file transfer with SOS resulted in an average throughput increase of eight times that without the use of SOS. SOS was conducted with 32 parallel connections in between the agents, a queue capacity of 1000, and a buffer size of 8192 bytes. It was noted that agent RAM utilization was approximately 90% during the tests with SOS.

5.3.2 Production Network Environment

To ascertain how effective SOS is at increasing TCP throughput on high speed networks and to better determine the limiting factors of an SOS agent, SOS was deployed over a 10Gbps link connecting Clemson University in Clemson, SC to the University of Utah in Salt Lake City, Utah. The link was a VLAN-stitched, layer 2 link within the AL2S network with a 10Gbps end-to-end link. Each machine described below was equipped with a Myricom 10Gbps Ethernet card with TCP segment

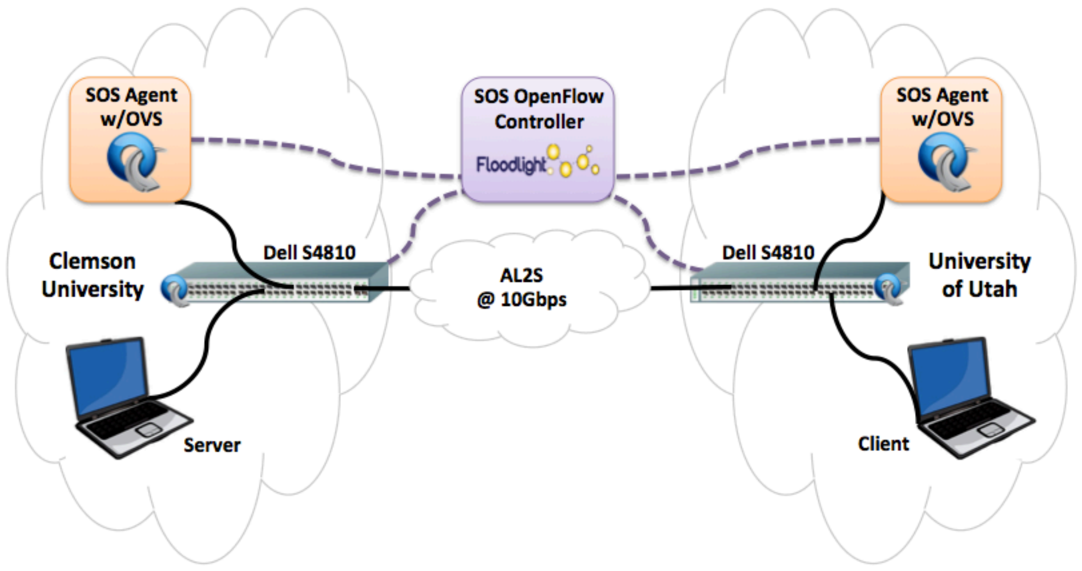


Figure 5.3.2: SOS on AL2S

offloading enabled (default configuration) and with the maximum possible IP MTU of 8976 bytes. The transmit queue length of the network cards on the SOS agents were also configured to be 1000. And lastly, on the agent machines only, the TCP buffer, send, and receive queue lengths were configured to be more suitable for large delay bandwidth product networks [87].

As shown in Figure 5.3.2, the file server was hosted at Clemson, while the client that downloads the large files was hosted at Utah. An SOS agent was also deployed at each location. Ubuntu 14.10 64-bit was installed on the Clemson server and the Clemson agent, while CentOS 6.7 64-bit was installed on both the Utah client and SOS agent. The Clemson server and SOS agent were equipped with Intel Core i5 2400 processors and 4GB and 8GB of RAM, respectively. The Utah client and SOS agent were each equipped with two (per machine) Intel Xeon E5620 processors and 24GB of RAM.

On the Utah agent, since there were 16 available CPU cores, the network card interrupts were distributed evenly across 13 of these cores, the agent application itself was assigned to a single core, and all other OS processes were assigned to the last two cores such that all cores were being utilized and the system load was efficiently distributed. Likewise, on the Clemson agent, since there were 4 available CPU cores, the network card interrupts were assigned to one, the agent application processes were assigned to another, and OS processes were assigned to the other two.

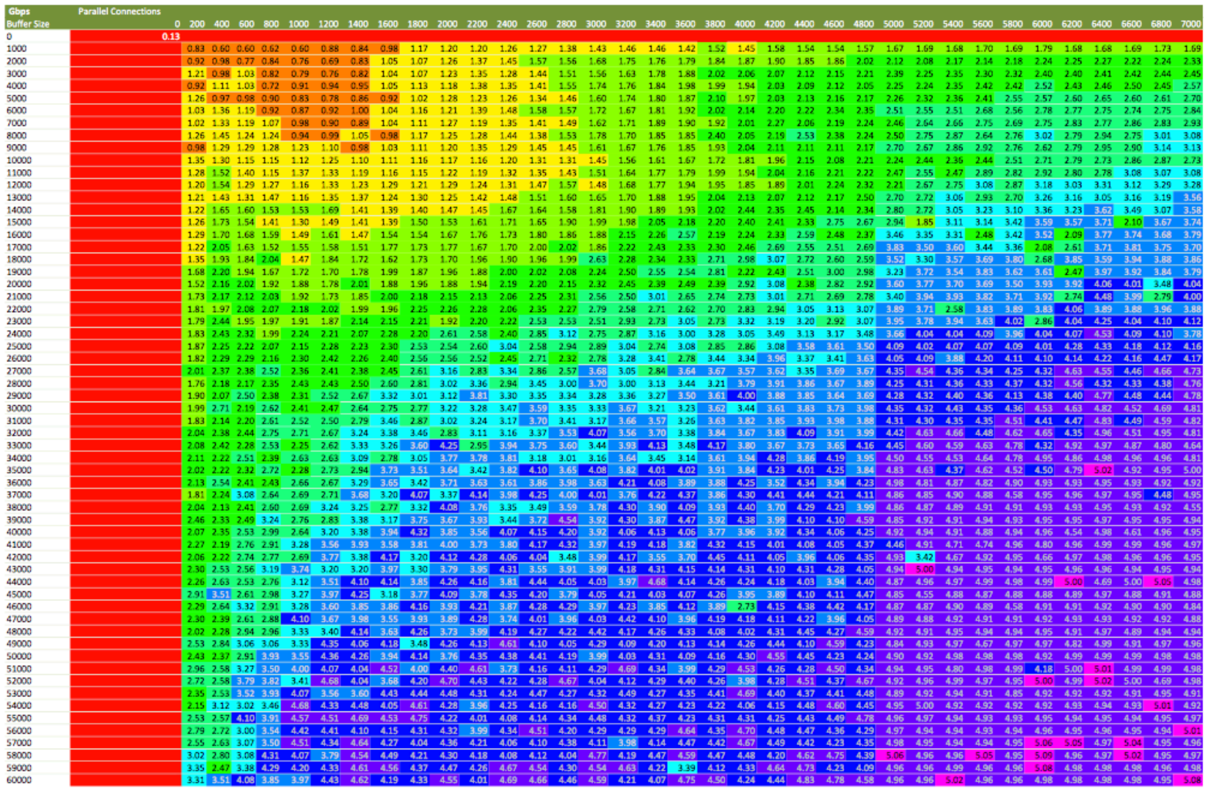
Note that the client and server machines were not modified in any way with the exception of the installation of iperf for network throughput tests.

At the Clemson end of the network, the server and the SOS agent were directly connected via their 10Gbps Myricom network cards to an OpenFlow-enabled Dell S4810 network switch using a 10Gbps SFP network cable. The same physical configuration was mirrored in Utah with another OpenFlow-enabled Dell S4810 switch directly connected to the client and the second SOS agent. The S4810 at Clemson and the S4810 at Utah were connected through a layer 2 link traversing AL2S. VLAN translation was performed within this link, causing Clemson to be on one VLAN, while Utah resided on another. This required modifying the default MAC learning algorithm of the SOS Floodlight OpenFlow controller.

A series of tests were conducted running an iperf server at the Utah server and running an iperf client at the Clemson client. This resulted in a flow of data from Clemson to Utah. (By default, iperf tests transfer data from the client to the server – emulating a file upload.) These tests were designed to better gauge the impact SOS parameters have on performance. Specifically, the number of TCP sockets to use between the agents and the number of bytes to send/receive at a given time on a given TCP socket were of interest. A third parameter – the TCP socket receive queue lengths – was set to a constant length of 5, since this parameter has no impact on throughput performance but simply serves as a "safety" in the case of significant packet loss.

Figure 5.3.3 demonstrates the impact the number of parallel sockets and the buffer size used on the agents can have on a data transfer. Without SOS, the data transfer rate between the Clemson client and the Utah server was an average of 130Mbps. With SOS, the maximum average data transfer rate achieved was 5.08Gbps, which was conducted with 60,000 byte buffer sizes and 7,000 parallel TCP connections on the agent machines at both Clemson and Utah. It was noted that the Clemson agent CPU utilization was quite high in the mid-90% range during the tests resulting in the greatest throughput. This is an indication that the agent was becoming the bottleneck. The scalability of SOS can be leveraged to overcome such situations where a single agent is unable to keep up with the desired network throughput.

For the tests in this section, the Clemson client and Utah server were not modified in any way to increase performance. The default TCP algorithm of TCP-cubic [40] was used on each machine, and the Linux default TCP windowing parameters were used. On the other hand, the agents were configured to use h-TCP [42] – designed for high speed and long distance networks – and with increased TCP windowing parameters according to [87].



Gbps	Parallel Connections																		
Buffer Size	0	200	400	600	800	1000	1200	1400	1600	1800	2000	2200	2400	2600	2800	3000	3200	3400	3600
0	0.13																		
1000		0.83	0.60	0.60	0.62	0.60	0.88	0.84	0.98	1.17	1.20	1.20	1.26	1.27	1.38	1.43	1.46	1.46	1.42
2000		0.92	0.98	0.77	0.84	0.76	0.69	0.83	1.05	1.07	1.26	1.37	1.45	1.57	1.56	1.68	1.75	1.76	1.79
3000		1.21	0.98	1.03	0.82	0.79	0.76	0.82	1.04	1.07	1.23	1.35	1.28	1.44	1.51	1.56	1.63	1.78	1.88
4000		0.92	1.11	1.03	0.72	0.91	0.94	0.95	1.05	1.13	1.18	1.38	1.35	1.41	1.55	1.74	1.76	1.84	1.98
5000		1.26	0.97	0.98	0.90	0.83	0.78	0.86	0.92	1.02	1.28	1.23	1.26	1.34	1.46	1.60	1.74	1.80	1.87
6000		1.03	1.36	1.19	0.92	0.87	0.92	1.00	1.04	1.16	1.21	1.39	1.48	1.58	1.57	1.72	1.67	1.81	1.92
7000		1.02	1.33	1.19	1.07	0.98	0.90	0.89	1.04	1.11	1.27	1.19	1.35	1.41	1.49	1.62	1.71	1.89	1.90
8000		1.26	1.45	1.24	1.24	0.94	0.99	1.05	0.98	1.17	1.25	1.28	1.44	1.38	1.53	1.78	1.70	1.85	1.85
9000		0.98	1.29	1.29	1.28	1.23	1.10	0.98	1.03	1.11	1.20	1.35	1.29	1.45	1.45	1.61	1.67	1.76	1.85
10000		1.35	1.30	1.15	1.15	1.12	1.25	1.10	1.11	1.16	1.17	1.16	1.20	1.31	1.31	1.45	1.56	1.61	1.67
11000		1.28	1.52	1.40	1.15	1.37	1.33	1.19	1.16	1.15	1.22	1.19	1.32	1.35	1.43	1.51	1.64	1.77	1.79
12000		1.20	1.54	1.29	1.27	1.16	1.33	1.23	1.29	1.21	1.29	1.24	1.31	1.47	1.57	1.48	1.68	1.77	1.94
13000		1.21	1.43	1.31	1.47	1.16	1.35	1.37	1.24	1.30	1.25	1.42	1.48	1.51	1.60	1.65	1.70	1.88	1.95
14000		1.22	1.65	1.60	1.53	1.53	1.69	1.41	1.39	1.40	1.47	1.45	1.67	1.64	1.58	1.81	1.90	1.89	1.93
15000		1.26	1.73	1.54	1.41	1.30	1.49	1.41	1.39	1.50	1.53	1.61	1.71	1.65	1.90	1.99	1.98	2.05	2.18
16000		1.29	1.70	1.68	1.59	1.49	1.61	1.47	1.54	1.54	1.67	1.76	1.73	1.80	1.86	1.88	2.15	2.26	2.57
17000		1.22	2.05	1.63	1.52	1.55	1.58	1.51	1.77	1.73	1.77	1.67	1.70	2.00	2.02	1.86	2.22	2.43	2.33
18000		1.35	1.93	1.84	2.04	1.47	1.84	1.72	1.62	1.73	1.70	1.96	1.90	1.96	1.99	2.63	2.28	2.34	2.33
19000		1.68	2.20	1.94	1.67	1.72	1.70	1.78	1.99	1.87	1.96	1.88	2.00	2.02	2.08	2.24	2.50	2.55	2.54
20000		1.52	2.16	2.02	1.92	1.88	1.78	2.01	1.88	1.96	1.88	1.94	2.19	2.20	2.15	2.32	2.45	2.39	2.49
21000		1.73	2.17	2.12	2.03	1.92	1.73	1.85	2.00	2.18	2.15	2.13	2.06	2.25	2.31	2.56	2.50	3.01	2.65
22000		1.81	1.97	2.08	2.07	2.18	2.02	1.99	1.96	2.25	2.26	2.28	2.06	2.35	2.27	2.79	2.58	2.71	2.62
23000		1.79	2.44	1.95	1.97	1.91	1.87	2.14	2.15	2.21	1.92	2.20	2.22	2.53	2.53	2.51	2.93	2.73	3.05
24000		1.83	2.43	2.32	1.99	2.24	2.21	2.07	2.28	2.20	2.61	2.58	2.40	2.85	3.12	2.75	2.87	3.16	3.00
25000		1.87	2.25	2.22	2.07	2.15	2.28	2.23	2.30	2.53	2.54	2.60	3.04	2.58	2.94	2.89	3.04	2.74	3.08
26000		1.82	2.29	2.29	2.16	2.30	2.42	2.26	2.40	2.56	2.56	2.52	2.45	2.71	2.32	2.78	3.28	3.41	2.78
27000		2.01	2.37	2.38	2.52	2.36	2.41	2.38	2.45	2.61	3.16	2.83	3.34	2.86	2.57	3.68	3.05	2.84	3.64
28000		1.76	2.18	2.17	2.35	2.43	2.43	2.50	2.60	2.81	3.02	3.36	2.94	3.45	3.00	3.70	3.00	3.13	3.44
29000		1.90	2.07	2.50	2.38	2.31	2.52	2.67	3.32	3.01	3.12	3.81	3.30	3.35	3.34	3.28	3.36	3.27	3.50
30000		1.99	2.71	2.19	2.62	2.41	2.47	2.64	2.75	2.77	3.22	3.28	3.47	3.59	3.35	3.33	3.67	3.21	3.23
31000		1.83	2.14	2.20	2.61	2.52	2.50	2.79	3.46	2.87	3.02	3.24	3.17	3.70	3.41	3.17	3.66	3.57	3.26
32000		2.04	2.38	2.44	2.75	2.71	2.67	3.24	3.38	3.46	2.83	3.11	3.16	3.37	3.53	4.07	3.56	3.70	3.38
33000		2.08	2.42	2.28	2.53	2.25	2.62	3.33	3.26	3.60	4.25	2.95	3.94	3.75	3.60	3.44	3.93	4.13	3.48
34000		2.11	2.22	2.51	2.39	2.63	2.63	3.09	2.78	3.05	3.77	3.78	3.81	3.18	3.01	3.16	3.64	3.45	3.14
35000		2.02	2.22	2.32	2.72	2.28	2.73	2.94	3.73	3.51	3.64	3.42	3.82	4.10	3.65	4.08	3.82	4.01	4.02
36000		2.13	2.54	2.41	2.43	2.66	2.67	3.29	3.65	3.42	3.71	3.63	3.61	3.86	3.98	3.63	4.21	4.08	3.89
37000		1.81	2.24	3.08	2.64	2.69	2.71	3.68	3.20	4.07	3.37	4.14	3.98	4.25	4.00	4.01	3.76	4.22	4.37
38000		2.04	2.13	2.41	2.60	2.69	3.24	3.25	2.77	3.32	4.08	3.76	3.35	3.49	3.59	3.78	4.30	3.90	4.09
39000		2.46	2.33	2.49	3.24	2.76	2.83	3.38	3.17	3.75	3.67	3.93	3.44	3.72	4.54	3.92	4.30	3.87	4.47
40000		2.07	2.35	2.53	2.99	2.64	3.20	3.38	3.94	4.32	3.85	3.56	4.07	4.15	4.20	3.92	4.06	4.13	4.06
41000		2.27	2.19	2.76	2.91	3.28	3.56	3.93	3.58	3.81	4.00	3.73	3.80	4.17	4.32	3.97	4.19	4.18	3.82
42000		2.06	2.22	2.74	2.77	2.69	3.77	3.38	4.17	3.20	4.12	4.28	4.06	4.04	3.48	3.99	4.17	3.55	3.70
43000		2.30	2.53	2.56	3.19	3.74	3.20	3.20	3.97	3.30	3.79	3.95	4.31	3.55	3.91	3.99	4.18	4.31	4.15
44000		2.26	2.63	2.53	2.76	3.12	3.51	4.10	4.14	3.85	4.26	4.16	3.81	4.44	4.05	4.03	3.97	4.68	4.14
45000		2.91	3.51	2.61	2.98	3.27	3.97	4.25	3.18	3.77	4.09	3.78	4.35	4.20	3.79	4.05	4.21	4.03	4.07
46000		2.29	2.64	3.32	2.91	3.28	3.60	3.85	3.86	4.16	3.93	4.21	3.87	4.28	4.29	3.97	4.23	3.85	4.12
47000		2.30	2.39	2.61	2.88	4.10	3.67	3.98	3.55	3.93	3.89	4.28	3.74	4.01	3.96	4.03	4.42	4.10	3.96
48000		2.02	2.28	2.94	2.96	3.33	3.40	4.14	3.63	4.26	3.73	3.99	4.19	4.27	4.22	4.42	4.17	4.26	4.33
49000		2.53	2.84	3.06	3.06	3.33	4.35	4.06	4.18	3.48	4.26	4.13	4.61	4.10	4.05	4.29	4.09	4.20	4.13
50000		2.43	2.37	2.91	3.93	3.55	4.36	4.26	3.94	4.14	3.76	4.35	4.38	4.41	4.19	3.99	4.03	4.31	4.09
51000		2.96	2.58	3.27	3.50	4.00	4.07	4.04	4.52	4.00	4.40	4.61	3.73	4.16	4.11	4.29	4.69	4.34	3.99
52000		2.72	2.58	3.79	3.82	3.41	4.68	4.04	3.68	4.20	4.70	4.43	4.22	4.28	4.67	4.04	4.12	4.29	4.40
53000		2.35	2.53	3.52	3.93	4.07	3.56	3.60	4.43	4.44	4.48	4.31	4.24	4.47	4.27	4.32	4.49	4.27	4.35
54000		2.15	3.12	3.02	3.46	4.68	4.33	4.48	4.05	4.61	4.28	3.96	4.25	4.16	4.16	4.50	4.32	4.27	4.23
55000		2.53	2.57	4.10	3.91	4.57	4.51	4.69	4.53	4.75	4.22	4.01	4.08	4.14	4.34	4.48	4.32	4.37	4.23
56000		2.79	2.72	3.00	3.54	4.42	4.41	4.10	4.15	4.31	4.32	3.99	4.34	4.51	4.20	4.29	4.29	4.29	4.64
57000		2.55	2.63	3.07	3.50	4.51	4.34	4.64	4.27	4.04	4.36	4.21	4.06	4.10	4.38	4.11	3.98	4.14	4.47
58000		3.02	2.80	3.08	4.31	4.07	3.79	4.54	4.49	4.21	4.30	4.18	4.08	4.12	4.04	4.77	4.19	4.47	4.59
59000		3.35	2.47	3.38	4.29	4.20	4.33	4.61	4.56	4.37	4.47	4.26	4.67	4.54	4.30	4.54	4.63	4.22	3.39
60000		3.31	3.51	4.08	3.85	3.97	4.43	4.62	4.19	4.33	4.55	4.01	4.69	4.66	4.46	4.59	4.21	4.07	4.75

Gbps																	
Buffer Size	3800	4000	4200	4400	4600	4800	5000	5200	5400	5600	5800	6000	6200	6400	6600	6800	7000
0																	
1000	1.52	1.45	1.58	1.54	1.54	1.57	1.67	1.69	1.68	1.70	1.69	1.79	1.68	1.68	1.69	1.73	1.69
2000	1.84	1.87	1.90	1.85	1.86	2.02	2.12	2.08	2.17	2.14	2.18	2.24	2.25	2.27	2.22	2.24	2.33
3000	2.02	2.06	2.07	2.12	2.15	2.21	2.39	2.25	2.35	2.30	2.32	2.40	2.40	2.41	2.42	2.44	2.45
4000	1.99	1.94	2.03	2.09	2.12	2.05	2.25	2.24	2.35	2.42	2.42	2.52	2.43	2.46	2.50	2.45	2.57
5000	2.10	1.97	2.03	2.13	2.16	2.17	2.26	2.32	2.36	2.41	2.55	2.57	2.60	2.65	2.60	2.61	2.70
6000	2.02	2.14	2.20	2.22	2.34	2.35	2.51	2.55	2.51	2.68	2.56	2.78	2.77	2.75	2.74	2.75	2.84
7000	1.92	2.01	2.27	2.06	2.19	2.24	2.46	2.64	2.66	2.75	2.69	2.75	2.83	2.77	2.86	2.83	2.93
8000	2.40	2.05	2.19	2.53	2.38	2.24	2.50	2.75	2.87	2.64	2.76	3.02	2.79	2.94	2.75	3.01	3.08
9000	1.93	2.04	2.11	2.11	2.11	2.17	2.70	2.67	2.86	2.92	2.76	2.62	2.79	2.95	2.90	3.14	3.13
10000	1.72	1.81	1.96	2.15	2.08	2.21	2.24	2.44	2.36	2.44	2.51	2.71	2.79	2.73	2.86	2.87	2.73
11000	1.99	1.94	2.04	2.16	2.21	2.22	2.47	2.55	2.47	2.89	2.82	2.92	2.80	2.78	3.08	3.07	3.08
12000	1.95	1.85	1.89	2.01	2.24	2.32	2.21	2.67	2.75	3.08	2.87	3.18	3.03	3.31	3.12	3.29	3.28
13000	2.04	2.13	2.07	2.12	2.17	2.50	2.70	2.72	3.06	2.93	2.70	3.26	3.16	3.05	3.16	3.19	3.56
14000	2.02	2.44	2.35	2.45	2.14	2.34	2.80	2.72	3.05	3.23	3.10	3.36	3.23	3.62	3.49	3.07	3.58
15000	2.20	2.40	2.41	2.33	2.75	2.67	2.94	1.85	3.11	3.14	3.42	3.59	3.57	3.71	2.10	3.67	3.74
16000	2.19	2.24	2.33	2.59	2.48	2.37	3.46	3.35	3.31	2.48	3.42	3.52	2.09	3.77	3.74	3.68	3.79
17000	2.30	2.46	2.69	2.55	2.51	2.69	3.83	3.50	3.60	3.44	3.36	2.08	2.61	3.71	3.81	3.75	3.70
18000	2.71	2.98	3.07	2.72	2.60	2.59	3.52	3.30	3.57	3.69	3.80	2.68	3.85	3.59	3.94	3.88	3.86
19000	2.81	2.22	2.43	2.51	3.00	2.98	3.23	3.72	3.54	3.83	3.62	3.61	2.47	3.97	3.92	3.84	3.79
20000	2.39	2.92	3.08	2.38	2.82	2.92	3.60	3.77	3.70	3.69	3.50	3.93	3.92	4.06	4.01	3.48	4.04
21000	2.74	2.73	3.01	2.71	2.69	2.78	3.40	3.94	3.93	3.82	3.71	3.92	2.74	4.48	3.99	2.79	4.00
22000	2.70	2.83	2.94	3.05	3.13	3.07	3.89	3.71	2.58	3.83	3.89	3.83	4.06	3.89	3.88	3.96	3.88
23000	2.73	3.32	3.19	3.20	2.92	3.07	3.95	3.78	3.94	3.63	4.02	2.86	4.04	4.25	4.04	4.10	4.12
24000	3.28	3.05	3.49	3.13	3.17	3.48	3.66	4.04	4.04	4.09	3.96	4.04	4.07	4.53	4.09	4.10	3.78
25000	2.85	2.86	3.08	3.58	3.61	3.50	4.09	4.02	4.07	4.07	4.09	4.01	4.28	4.33	4.18	4.12	4.16
26000	3.44	3.34	3.96	3.37	3.41	3.63	4.05	4.09	3.88	4.20	4.11	4.10	4.14	4.22	4.16	4.47	4.17
27000	3.67	3.57	3.62	3.35	3.69	3.67	4.35	4.54	4.36	4.34	4.25	4.32	4.63	4.55	4.46	4.66	4.73
28000	3.21	3.79	3.91	3.86	3.67	3.89	4.25	4.31	4.36	4.33	4.37	4.32	4.56	4.32	4.33	4.38	4.76
29000	3.61	4.00	3.88	3.85	3.64	3.69	4.28	4.32	4.40	4.36	4.13	4.38	4.40	4.77	4.48	4.44	4.78
30000	3.62	3.44	3.61	3.83	3.73	3.98	4.35	4.32	4.43	4.35	4.36	4.53	4.63	4.82	4.52	4.69	4.81
31000	3.63	3.82	3.85	3.93	3.98	3.88	4.31	4.30	4.35	4.35	4.51	4.41	4.47	4.83	4.49	4.59	4.82
32000	3.84	3.67	3.83	4.09	3.91	3.99	4.42	4.63	4.66	4.48	4.62	4.65	4.35	4.96	4.51	4.95	4.81
33000	4.17	3.80	3.67	3.73	3.65	4.16	4.45	4.60	4.59	4.63	4.78	4.32	4.92	4.97	4.87	4.80	4.64
34000	3.61	3.94	4.28	3.86	4.19	3.95	4.50	4.55	4.53	4.64	4.78	4.95	4.86	4.98	4.96	4.96	4.81
35000	3.91	3.84	4.23	4.01	4.25	3.84	4.83	4.63	4.37	4.62	4.52	4.50	4.79	5.02	4.92	4.95	5.00
36000	3.88	4.25	3.52	4.34	3.94	4.23	4.98	4.81	4.87	4.82	4.90	4.93	4.93	4.95	4.93	4.92	4.92
37000	3.86	4.30	4.41	4.44	4.21	4.11	4.86	4.85	4.90	4.88	4.58	4.95	4.96	4.97	4.95	4.48	4.95
38000	3.93	4.40	3.70	4.29	4.23	3.99	4.86	4.87	4.89	4.91	4.91	4.93	4.93	4.95	4.93	4.92	4.55
39000	3.92	4.38	3.99	4.10	4.10	4.59	4.85	4.92	4.91	4.94	4.93	4.95	4.95	4.97	4.95	4.95	4.94
40000	3.77	3.96	3.92	4.34	4.06	4.25	4.92	4.94	4.91	4.88	4.94	4.96	4.54	4.98	4.61	4.96	4.95
41000	4.32	4.15	4.01	4.08	4.05	4.37	4.46	4.91	4.71	4.74	4.96	4.80	4.96	4.99	4.99	4.96	4.97
42000	4.45	4.11	4.05	3.96	4.06	4.35	4.93	3.42	4.67	4.92	4.95	4.66	4.97	4.98	4.96	4.96	4.95
43000	4.14	4.31	4.10	4.31	4.28	4.05	4.94	5.00	4.94	4.95	4.94	4.95	4.96	4.96	4.94	4.95	4.94
44000	4.26	4.24	4.18	4.03	3.94	4.40	4.87	4.96	4.97	4.99	4.98	4.99	5.00	4.69	5.00	5.05	4.98
45000	4.26	3.95	3.89	4.10	4.11	4.47	4.85	4.55	4.88	4.87	4.88	4.88	4.89	4.97	4.88	4.91	4.88
46000	3.89	2.73	4.15	4.38	4.42	4.17	4.87	4.87	4.90	4.89	4.58	4.91	4.91	4.92	4.90	4.90	4.84
47000	4.19	4.18	4.11	4.22	3.96	4.05	4.89	4.88	4.92	4.91	4.91	4.92	4.93	4.93	4.92	4.92	4.88
48000	4.08	4.02	4.31	4.45	4.27	4.59	4.92	4.91	4.95	4.94	4.94	4.95	4.91	4.97	4.89	4.94	4.94
49000	4.14	4.26	4.44	4.10	4.59	4.23	4.84	4.93	4.97	4.67	4.97	4.97	4.82	4.99	4.91	4.97	4.96
50000	4.16	4.30	4.55	4.45	4.23	4.24	4.90	4.92	4.98	4.98	4.98	4.92	4.99	4.99	4.99	4.98	4.98
51000	4.29	4.53	4.26	4.28	4.50	4.34	4.94	4.95	4.80	4.98	4.99	4.18	5.00	5.01	4.99	4.99	4.98
52000	4.26	3.98	4.28	4.51	4.37	4.67	4.92	4.96	4.99	4.97	4.95	5.00	4.99	5.02	5.00	4.69	4.98
53000	4.41	4.69	4.40	4.37	4.41	4.48	4.89	4.92	4.94	4.91	4.85	4.92	4.92	4.92	4.91	4.95	4.91
54000	4.22	4.06	4.15	4.48	4.60	4.45	4.95	5.00	4.92	4.92	4.92	4.92	4.93	4.94	4.93	5.01	4.92
55000	4.31	4.31	4.25	4.43	4.49	4.78	4.96	4.97	4.94	4.93	4.93	4.95	4.94	4.95	4.94	4.95	4.97
56000	4.35	4.70	4.48	4.47	4.36	4.29	4.97	4.94	4.94	4.93	4.94	4.96	4.95	4.96	4.95	4.94	5.01
57000	4.42	4.67	4.49	4.42	4.23	4.35	4.98	4.95	4.94	4.94	4.95	5.06	5.05	4.97	5.04	4.95	4.96
58000	4.47	4.48	4.20	4.62	4.75	4.39	5.06	4.96	4.96	5.05	4.95	5.09	4.96	4.97	5.02	4.95	4.97
59000	4.12	4.33	4.64	4.73	4.23	4.09	4.96	4.96	4.99	4.96	4.96	5.08	4.98	4.98	4.98	4.96	4.98
60000	4.50	4.24	4.44	4.83	4.78	4.58	4.96	4.96	5.02	4.96	4.96	4.98	4.98	4.98	4.98	4.95	5.08

Figure 5.3.3: SOS agent number of parallel sockets and send/receive data chunk size parameter sweeps

5.3.3 Scale-Up Datacenter Environment

In this section, SOS is evaluated for its performance and scalability.

SOS has been evaluated in production network environments and in the GENI testbed [26]. Previous work [88] demonstrated the SDN components of SOS as proof of concept in GENI. Here, it was shown how SDN can be used to implement a transparent proxy between the two hosts. The concept and prototype of the SOS agent was also revealed in this work. As SOS evolved, the need for scalability became apparent given the agent bottleneck. In more recent works [78, 79], a scalable SOS architecture was proposed and a prototype was evaluated in the GENI testbed. Results revealed that SOS can improve the throughput of a TCP data transfer over five times in a virtualized environment. Furthermore, results showed that SOS can scale linearly as additional agents are deployed. These results have demonstrated proof of concept for both SOS and its scalability; however, they were not conducted with modern link speeds and in some cases used GRE tunnels as links. Although this shows SOS's viability in even non-ideal networks, it does not show that SOS can keep up with the speeds of modern network links in excess of 1Gbps.

Evaluation in the following series of tests consists of a real-world scalability evaluation using modern datacenter network link speeds. Experiments were designed to show how the SOS architecture can scale by simply adding and using additional agents. Tests were conducted using a single SOS agent at each side of a large delay-bandwidth-product link, followed by two agents at each side, and three agents at each side. In order to provide a simple demonstration of these scenarios, the following evaluation ran multiple data transfer applications in parallel on and between a client and a server. This conveniently shows how SOS can (1) be layered underneath parallel TCP applications and (2) be used my multiple clients and server applications simultaneously, where the bottleneck becomes per-agent capacity.

The remainder of this section is structured as follows. Section 5.3.3.1 introduces CloudLab and provides hardware specifications for the machines used in the evaluation. Section 5.3.3.2 discusses how the SOS deployment was configured. Section 5.3.3.3 explains how TCP traffic was generated. Section 5.3.3.4 provides the results of the CloudLab experimentation for single and multiple agents in order to demonstrate scalability.

5.3.3.1 CloudLab Setup

The CloudLab testbed is a testbed designed to conduct distributed and datacenter experiments at scale. It consists of datacenter network and compute resources at Clemson University, the University of Utah, and the University of Wisconsin all interconnected by a layer 2 network supported by Internet2's Advanced Layer 2 Service (AL2S) layer 2 network. While at the time of this series of tests, CloudLab at the University of Wisconsin did not support SDN experimentation on hardware switches, Clemson University and the University of Utah were equipped with Dell S6000, Z9000, and Z9100 OpenFlow-enabled switches. The University of Utah also supported SDN experimentation through its HP Moonshot 45XGc and FlexFabric 12910 OpenFlow-enabled switches.

The distributed nature of CloudLab across the three aforementioned locations and the fact that SDN experimentation is supported on hardware OpenFlow switches make CloudLab an ideal setup for the evaluation of SOS at scale. Furthermore, its compute resources consist of general-purpose, commodity hardware, and the OpenFlow switches used are white box switches not supporting much of the optional OpenFlow matches and actions. These two implementation choices add to the strengths of CloudLab for SOS experimentation, since they demonstrate SOS's support for agents running on commodity hardware and for SDN on non-specialized OpenFlow switching hardware more likely to be already present in a production network.

To evaluate SOS, a CloudLab profile was created, which defined the compute and network resources required. To show scalability, the profile consisted of three agent nodes at Clemson and three agent nodes at Utah. A single node at Clemson and another at Utah served as the client and server, respectively. The four nodes at Utah were defined in the APT [89] cluster, and consisted of Dell PowerEdge c6220 servers. Each was equipped with two Xeon E5-2650v2 processors (8 cores each at 2.6Ghz), 64GB RAM (8 x 8GB DDR-3 RDIMMs at 1.86Ghz), two 1TB SATA 3.5" 7,200 RPM hard drives, four 1Gbps Broadcom Ethernet NICs, and one Intel X520 PCIe dual port 10Gb Ethernet NIC. The four nodes at Clemson were defined in CloudLab proper and consisted of Dell PowerEdge c8220 servers, each with two Xeon E5-2660v2 processors (10 cores each at 2.2Ghz), 256GB RAM (16 x 16GB ECC, DDR-4 dual rank RDIMMs at 1,600MT/s), eight 1TB SATA 3.5" 7,200 RPM hard drives, twelve 4TB SATA 3.5" 7,200 RPM hard drives, four 1Gbps Broadcom Ethernet network interface cards (NICs), and one Intel X520 PCIe dual port 10Gb Ethernet NIC. Each node was running Ubuntu 14.10 server 64-bit.

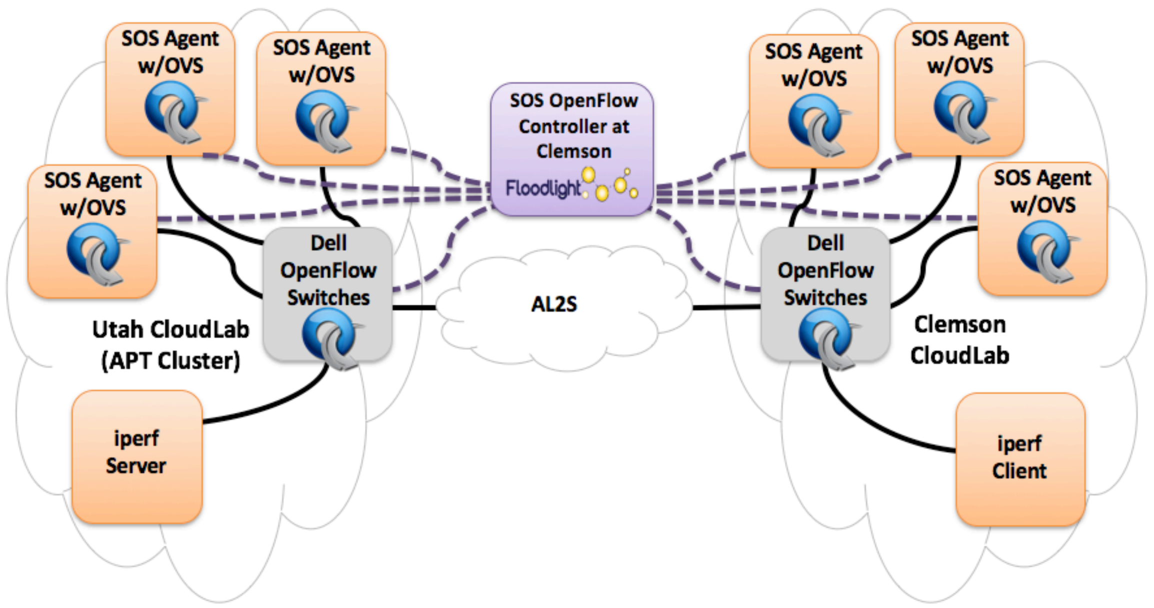


Figure 5.3.4: The scalable SOS deployment in the CloudLab testbed

A 1Gbps Ethernet NIC was used for control on each machine, while one 10Gbps NIC was used for data. The 10Gbps data NICs were all connected to OpenFlow-enabled Dell switches. Utah APT consisted of a single Z9000, while Clemson, consisted of a Z9100 and and two S6000 switches. The AL2S [24] network interconnected the Utah and Clemson portions of the testbed via the Z9000 and Z9100 switches, providing an approximately 53ms round trip delay over the private layer 2 WAN.

For convenience, the public Internet was used as the control network between the SOS Floodlight OpenFlow controller and the Dell OpenFlow switches. The OpenFlow switches, being behind local firewalls, connected to the public controller open to the firewall over TCP. Note that the use of the public Internet was done for convenience only and is not a recommended SOS control plane deployment. General networking and SDN security practices should be enacted, which is an effort disjoint from the deployment and operation of SOS.

All hosts – client, server, and agents – were assigned unique IPs in the same 10.0.0/24 network. Figure 5.3.4 shows SOS deployed using CloudLab resources.

5.3.3.2 SOS Configuration

SOS was configured through the controller's REST interface. A data chunk size of 15,000 bytes, four inter-agent data engine TCP connections, and a queue size of three were all set to apply to future SOS data transfers. Furthermore, all six SOS agents – three at Clemson and three at Utah – were added to the controller. To conduct tests using one, two, and three agents at each site, three whitelist entries were added to the controller, one for each iperf server and client pair.

The SOS agents were installed on three of the nodes reserved at Utah and three of the nodes at Clemson. OVS version 2.3.1 was installed on each agent machine in kernel mode. The use of OVS was necessary, because although the the Dell OpenFlow switches in CloudLab could perform MAC address rewrite, they were not capable of performing the IP and TCP header rewrites required to achieve transparency. Each agent machine's 10Gbps data plane interface was configured with the maximum MTU supported by the testbed, which was experimentally determined to be 1500 bytes. Because the SOS agent is single-threaded application, it was assigned to run on one CPU core, OVS was assigned another, OS and other tasks were assigned a third core, and the 10Gbps NIC interrupts were assigned to the remaining cores. Next, the Linux TCP stack was configured with window scaling enabled. And finally, because CloudLab is a shared testbed, tc is configured by default to throttle traffic on the 10Gbps data plane NICs. Since the goal was to show throughput improvement by SOS at scale, the tc queues were removed to allow the 10Gbps links to achieve up to line rate.

The client and server machines were equipped with iperf, discussed below. No optimizations were performed; however, the tc queues were also removed as discussed above.

5.3.3.3 TCP Traffic Generation

The iperf utility was used to generate user TCP traffic to be optimized by SOS. For each test, iperf was run over TCP for a duration of thirty seconds. At the conclusion of each test, the average throughput in bits per second was recorded. Thirty seconds was chosen for a couple of reasons. First, it is a relatively short period of time, and allows for quick, successive trials. Given that the CloudLab testbed is a shared resource, conducting the tests in a short time block reduced the likelihood of skewed data due to network utilization by other CloudLab users and other users of AL2S, which is a best-effort network. Thirty seconds also was shown in test trials as sufficient to demonstrate the performance improvements SOS provides. Longer iperf test durations showed an even greater performance gain, since the TCP window "ramp up" period had less of an impact on the overall throughput achieved. However, the longer test duration was not necessary to show the same net result – performance improvement and scalability of SOS.

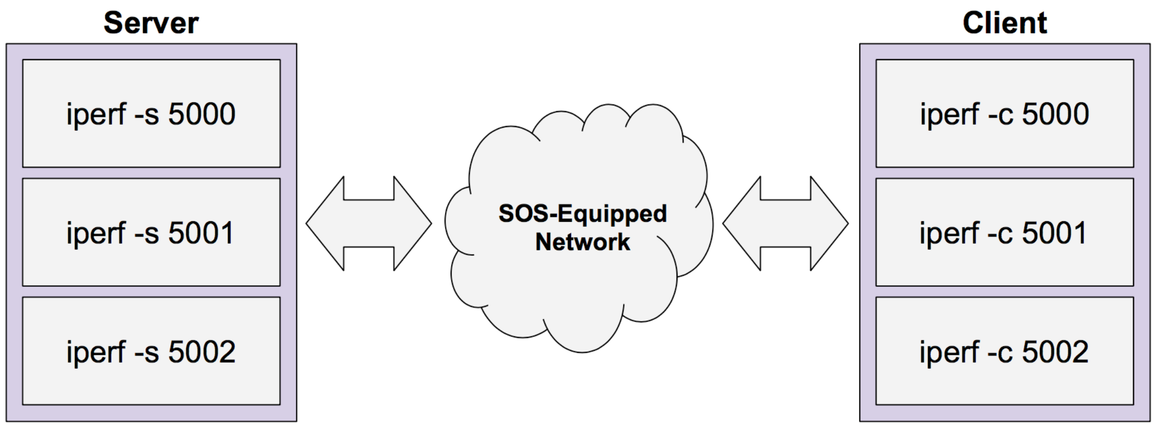


Figure 5.3.5: TCP traffic generation between the client and the server

iperf has many benefits that make it a good fit for the evaluation of SOS. First, it attempts to fully saturate the pipe using the protocol of choice, TCP in the case of our evaluation. This is ideal, since as a benchmark, the application should continuously have data available for transfer. Without this property, the performance of the data source must be considered. This leads us to the third benefit of iperf, which is that it easily emulates the transfer of arbitrary user data without regard to the data being transferred. The specific data set and its source is beyond the scope of SOS. It is very important to keep in mind that SOS is designed to increase network throughput, not disk-to-disk throughput. Evaluation of SOS with disk-to-disk data transfers would be hindered by the typically slower performance of the disks. Since SOS only operates at the network level, it can readily be integrated with disjoint disk throughput improving techniques, such as SSD arrays and parallel file systems.

To conduct scale tests with multiple SOS agents, multiple iperf instances were run in parallel on the client and server machines. The server iperf instances were run on different TCP ports 5000, 5001, and 5002, respectively. This configuration is depicted in Figure 5.3.5. A second set of tests were run using iperf on different client and server machines, four in total.

5.3.3.4 Results

Tests were conducted in sequence first using a single client-side agent and a single server-side agent. Following this, two client-side and server-side agents were used to handle two TCP connections simultaneously – one per pair of agents. Then, similar to the two-agent tests, three client-side and server-side agents were used to handle three TCP connections simultaneously. In all three of these test cases, iperf was used between a client machine and a server machine as described in Section 5.3.3.3. As shown in Figure 5.3.6, an iperf TCP data transfer without the use of SOS achieved approximately 330Mbps over the large link between Clemson and Utah. This average was obtained by disabling SOS in the Floodlight controller. The net result was the Forwarding module reactively installing flows. This has an initial setup delay during the TCP SYN packet-in/packet-out and during the flow installation; however, it does not impact the data transfer rate reported by iperf, which initiates data collection after it has established the TCP connection between the client and the server iperf applications.

Based on prior tests, it was expected that a single pair of SOS agents would significantly increase this data transfer rate and that for each additional pair of agents, the performance should exhibit an approximately linear increase. The use of a single SOS agent pair was capable of improving the non-SOS data transfer to an average rate of 2.07Gbps over the same duration. This is also shown in Figure 5.3.6.

Multiple iperf sessions between the same client and server were then initiated. The SOS controller selected a pair of agents for each of the TCP connections. In this case, the data transferred between the client and server achieved an average aggregate data transfer rate of 5.13Gbps. This was split between two pairs of agents, where each pair achieved half of the aggregate data transfer. This was approximately the data transfer rate achieved by a single pair of SOS agents. Also shown in Figure 5.3.6, a greater than linear increase in performance was shown between one and two pairs of SOS agents

In a third series of tests, three pairs of SOS agents were used to assist in data transfer between the client and the server. Three iperf sessions between the same client and server were initiated. The SOS controller reacted to each connection individually and assigned each a pair of SOS agents. The data transferred between the client and server was assisted by the three pairs of SOS agents, which were able to achieve an average aggregate data transfer rate of 8.46Gbps. This was also more than a linear increase in data transfer performance between the client and the server and is depicted in Figure 5.3.6.

A final test was conducted to test the performance of SOS with multiple client and server pairs using it simultaneously. Two iperf sessions between different clients and servers were initiated. The SOS controller reacted to each connection individually and assigned a each a pair of SOS agents. The data transferred between the client and server was assisted by the two pairs of SOS agents,

Single Client/Server Pair	Two Client/Server Pairs
8 Gbps	16 Gbps

Table 5.3.2: Use of two independent clients and servers in an SOS architecture

which were able to achieve an average aggregate data transfer rate of 16Gbps. This is shown in Table 5.3.2. This increase in performance was due to reduced load on the CloudLab testbed at the time, and better shows the capabilities of the SOS agents to scale to modern link capacities.

An interesting observation in these results is that the addition of a second, then a third SOS agent to Clemson and Utah achieved more than a linear increase in performance. Although we would like to claim it is due entirely to SOS, this result can be explained by the fact that CloudLab is a shared testbed. Traffic from other users of CloudLab and AL2S is typically negligible relative to the link capacities, but its impact on results cannot be ignored. A higher incidence of background traffic and thus link contention during the tests conducted between a single pair and two pairs of SOS agents would give the appearance of greater than linear increase in performance over all three sets of tests. Architecturally, SOS is not capable of achieving a greater than linear increase in performance when adding SOS agents unless the agents added are capable of achieving data transfer rates higher than those individual agents already installed.

As SOS scales beyond the tests conducted here, it is interesting to consider potential bottlenecks. The bottleneck of an individual SOS agent has proven to be irrelevant given the scale-out design of the SOS architecture. Additional agents can simply be added as the utilization of currently deployed SOS agents reach their capacity. Furthermore, for an individual application between a single client and server can utilize parallel TCP in the application in order to scale beyond the capacity of a single pair of SOS agents. This allows the application to use multiple pairs of SOS agents.

As more clients and servers use SOS simultaneously, network link capacity will become a bottleneck. This can be extrapolated through the results obtained, which show that only three of our chosen SOS agents and SOS configuration use approximately 85% of the 10Gbps link capacity. The addition of a fourth pair of SOS agents (not done here due to CloudLab resource constraints) would result in continuous contention on the 10Gbps link.

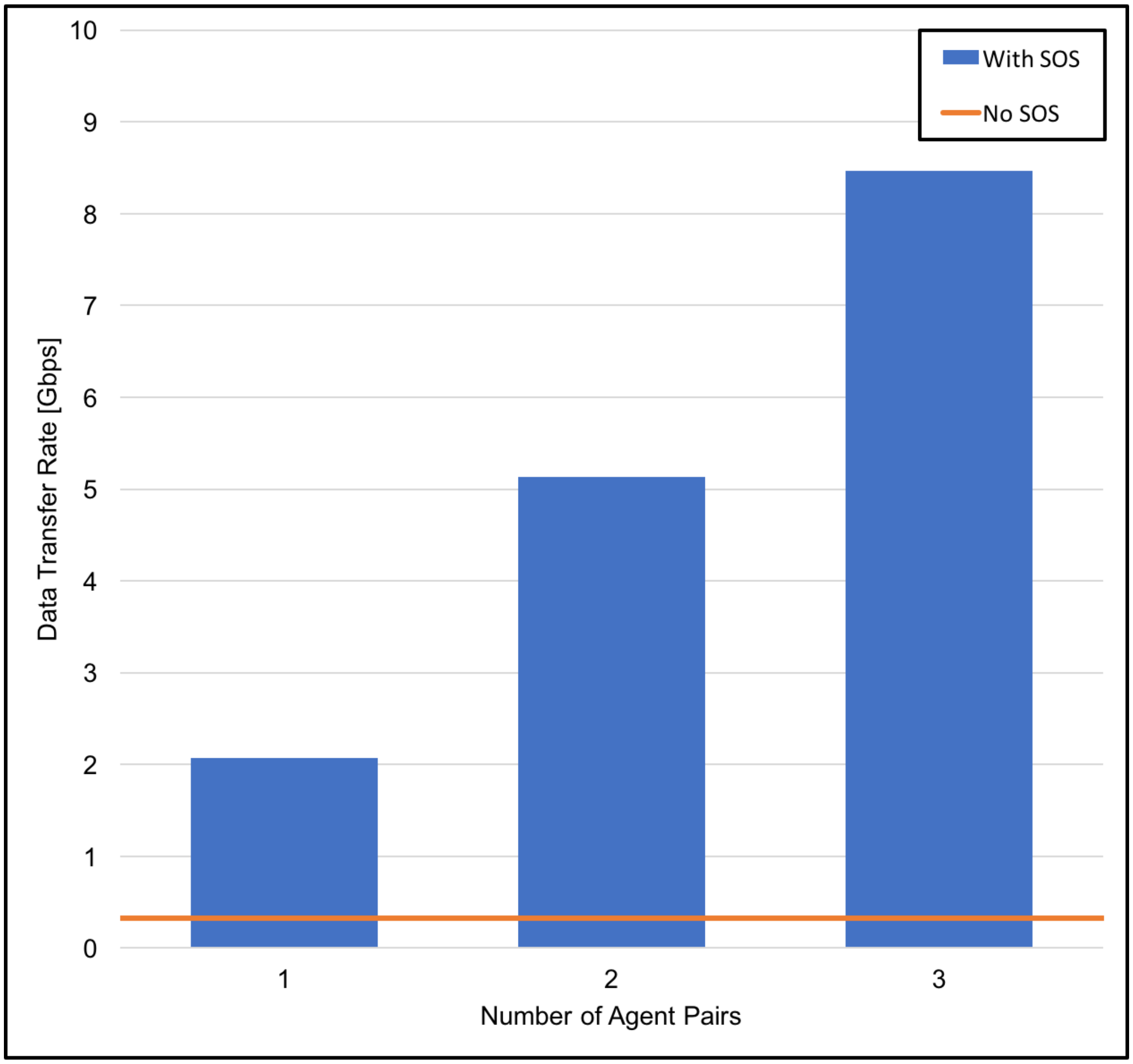


Figure 5.3.6: Performance scales linearly as agents are added to an SOS deployment

Chapter 6

Efficient and Scalable Content Distribution

Today's average network user is highly focused on multimedia content. There are users who wish to provide multimedia – often video streams – for many to access. There also exist users who wish to consume existing video content. These two classes of users are referred to as producers and consumers. The overarching goal is to relay the live video data from a producer to the consumers who wish to watch it. This poses a great networking challenge.

SDN can be used to as an alternative to IP multicast for the efficient and scalable distribution of content from producers to consumers. Its knowledge of the global network topology, as well as the deep programmability exposed can be exploited to provide a content distribution service that keeps bandwidth consumption both in the core and on the edge to a minimum and also achieves a minimal latency between content delivery when toggling between content from different producers. Furthermore, SDN can be used to address the security flaws present in IP multicast. The key topics addressed by the SDN solution proposed are as follows:

- Minimal bandwidth consumption and latency. The solution proposed optimizes bandwidth consumption in the core and on the edge, and also achieves a low latency when fetching, switching between, and terminating content.
- Capacity-aware content distribution. The solution proposed can route content in the core

without oversubscribing links.

- Scalability. The solution proposed is scalable to any number of producers and consumers.

Work has been published in [90], [91], and [92].

This section is structured as follows. Section 6.1 discusses the current state of the art content distribution techniques and the ongoing problem of content distribution. Section 6.2 discusses the SDN-based solution called GENI Cinema. Section 6.3 evaluates the proposed solution in real world deployments and provides an analytical comparison to IP multicast.

6.1 Content Distribution Problem

The dissemination of content to users at scale can be done using a variety of solutions. The solution chosen primarily depends on the type of content being served – static or dynamic.

6.1.1 Static Content Distribution

Static content is pre-distributed to geographically distributed, redundant edge servers in content delivery networks (CDNs). As the name implies, and as shown in Figure 6.1.1, these edge servers live close to the edge of the network, serving content to consumers who are nearby. Each of the edge servers determines and updates its available content based on the content pushed by the origin server.

A request for content download is typically a lightweight operation that is handled by a front end server, such as a web server where the consumer interacts with the service and with available content. This front end server determines where the consumer is located and assigns an appropriate edge server located near the consumer. This edge server is where the consumer fetches its data. The advantage to such a design is twofold. First, bandwidth in the network core is conserved, since content requests and download responses are redirected to and served at the network edge. Second, the design is a "natural" load balancer for the a given piece of content. As the number of consumers grow that wish to simultaneously consume a given piece of content, additional edge servers can be added to the CDN deployment to handle the additional load. Without the redundant edge servers,

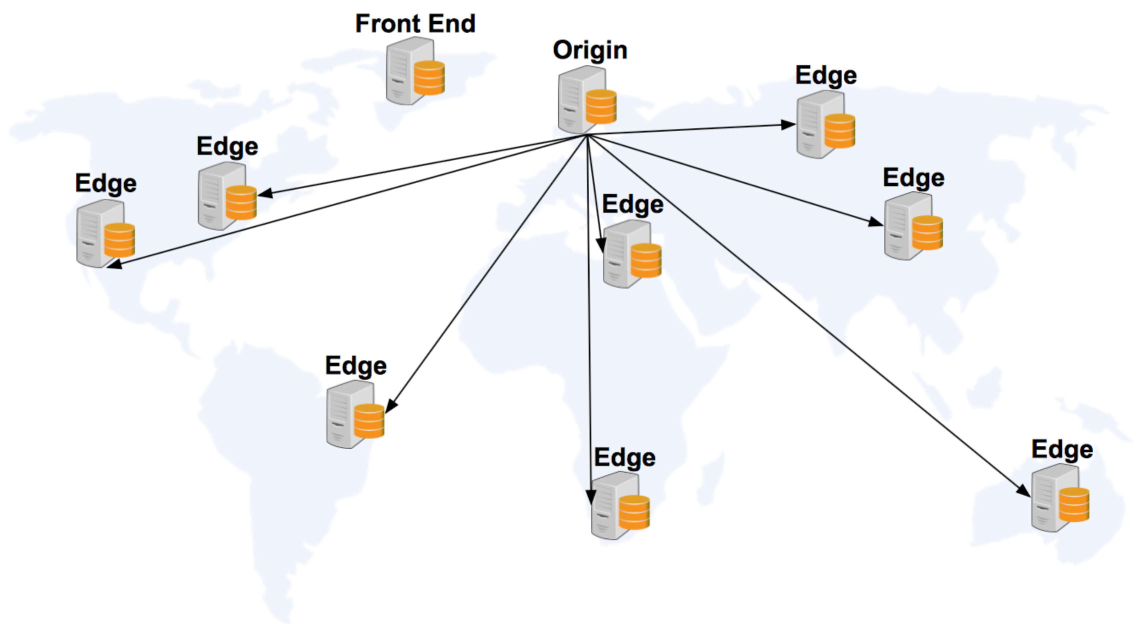


Figure 6.1.1: The architecture of a content delivery network

the origin server would be responsible for handling individual content requests and would quickly become a bottleneck.

CDNs have been adopted by many of today's popular content providers and web services. Static video streaming services, for example, use CDNs to host video content like movies and television shows on edge servers, where a front end server or cluster of servers provide a single "pane of glass" where the consumers interact with the service using a graphical user interface. In an operation that is typically masked from the consumers, the CDN delivers requested video content from an edge server close to the consumer, rather than from the front end server with which the consumer is interacting. Although they are widely deployed in today's networks, since they require the pre-deployment of content on edge servers, CDNs are ineffective for dynamic content distribution.

6.1.2 Dynamic Content Distribution

Dynamic content is content with characteristics such as location and size that are unknown prior to requests for the content. Live video streaming is an example of dynamic content. It does not have a fixed size, and in an environment where the producers of the streams can come and go from indeterministic locations, live video streaming does not have a predefined content location or source.

As such, it cannot be deployed over a CDN. For dynamic content distribution, there are two popular solutions in today's network architectures, namely IP unicast and IP multicast.

The use of unicast results in a large number of traffic flows in the network from a producer to every interested consumer. This limits the number of consumers a producer can support as a function of the video bitrate and the available network bandwidth of the network's minimum cut between the producer and all consumers – possibly the link between the producer and the network. IP multicast attempts to overcome this bottleneck by distributing a consumer's data throughout the network to interested consumers by constructing a multicast tree. This allows a single producer to serve an arbitrarily large number of consumers, where the maximum number of consumers is no longer dependent on the available network bandwidth at the producer. It is instead a function of available capacity at each independent edge. Although efficient for the distribution of content from one to many throughout the network, IP multicast has limitations.

First, it is not desired for switching rapidly from one IP multicast group to another. If a consumer wishes to watch a video stream – whether as an initial subscription or transitioning from one video to another – it must join the IP multicast group of the desired content. This join is not guaranteed to complete in a short amount of time, especially if the producer does not have its multicast tree constructed close to the requesting consumer.

Second, although IP multicast constructs trees in the network core to efficiently distribute groups without redundancy, it does not make efficient use of bandwidth when consumers transition between groups. If a consumer is to rapidly switch between videos in IP multicast groups, it must join and leave the groups. There is the potential for many of these groups to fetch data for the consumer in parallel until a timeout occurs or until the user's local router sends a leave message to the upstream router. This results in bandwidth being used unnecessarily both locally and between the upstream router and the consumer's local router. Fundamentally, this is due to the reliance on timers that prune consumers and branches of the multicast tree. Timers are used, since multicast is distributed and does not maintain an accurate state of the exact demand for multicast content. With the use of timers, bandwidth can be wasted in the case where there is no longer downstream demand.

Third, IP multicast does not use the lowest cost path for the dissemination of content. It requires a broadcast tree be first implemented via the rendezvous point of the particular multicast group being joined. Then, after this tree has been completed and the content is being delivered, a second tree is constructed following the paths defined in routing tables from the consumer to the producer. During this time and until the rendezvous point tree is deconstructed, the content is delivered simultaneously along the rendezvous point tree and the shortest path tree.

Fourth, IP multicast is not easy to deploy. There are many flavors of IP multicast for the distribution of content within and across administrative domains. Each is complex to implement correctly and requires hop-by-hop configuration. For this reason, many choose to forego IP multicast. Furthermore, although there have been proposals to improve IP multicast security, IP multicast is inherently insecure. A denial of service attack could be performed on a network by issuing different, rapid, and sequential IP multicast group joins (with or without leave notifications). This poses a great security risk in an IP multicast implementation and is part of the reason IP multicast is not widely deployed across administrative domains where precise control over usage is not always possible.

And lastly, IP multicast does not take into consideration network capacity. Because IP multicast is a collection of distributed protocols, not one forwarding device in the network is able to determine the best path to forward a group's content to downstream consumers. The construction of the IP multicast trees is done independently for each video source without regard for other network users. As such, IP multicast can be dangerous where network bandwidth is limited or where there is the risk of oversubscription. On a similar note, IP multicast allows anyone to publish to a group, where the consumers are responsible for sorting out the data they are interested in. This can result in excessive bandwidth use, since such undesired yet still transmitted data will inevitably be thrown away by consumers.

6.2 The GENI Cinema Solution

GENI Cinema (GC) is a live video delivery service that operates using SDN and is realized using GENI compute and network resources on the GENI nationwide testbed. Being a geographically distributed testbed, the GENI infrastructure lends itself nicely to GC in order to implement a content delivery network for the efficient delivery of video content to the consumer at the edges. Combined with SDN, this allows both network and compute resources to be conserved while consumers choose between the available video "channels" hosted by GC.

GC also places a soft quality of service guarantee on the video being streamed to each consumer.

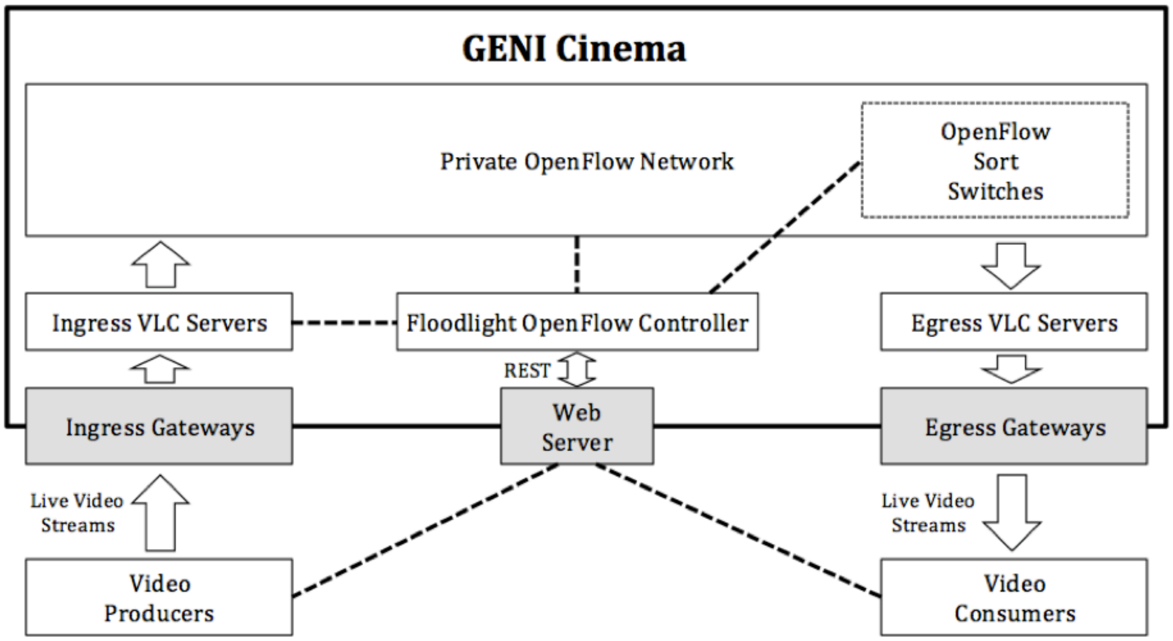


Figure 6.2.1: GENI Cinema logical architecture

With GC, there is a minimal delay when changing video channels. The architecture provides a "soft" guarantee, since each consumer must connect to GC using the public Internet, or if they are on a participating campus they may use their campus network. The underlying assumption is that there must be sufficient network resources within the path the video traverses between GC to the consumer over the public Internet or the campus network. Both of these networks are beyond the control of GC and are assumed to be capable of supporting video streaming (~1Mbps / HD stream).

As shown in Figure 6.2.1, GC consists of many functional blocks that come together to form the GC service. There are ingress and egress gateways for receiving and sending video streams, ingress and egress VideoLAN Client (VLC) [93] servers for hosting video streams on the backend and providing them on the front end, a Floodlight global resource and OpenFlow controller, both physical and software OpenFlow switches for controlling the flow of video streams internal to the GC service, and a public web server for consumer interaction. The following sections describe these functional blocks and how they interact with each other.

All source code for GENI Cinema can be found at `http://www.github.com/rizard/GENICinema`.

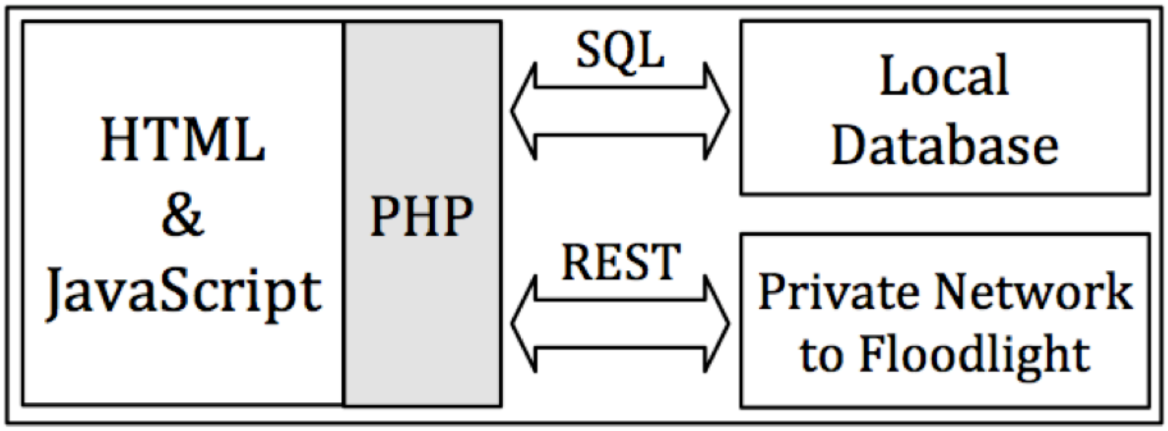


Figure 6.2.2: GENI Cinema web server

6.2.1 System Architecture

6.2.1.1 Web Server

The public web server hosts a website that allows video producers to specify live video streams as input to the GC service, as well as allows video consumers to subscribe to, watch, and switch between all available video content. The web server abstracts away all the low-level network and content management details from the consumer to provide a straight-forward and seamless experience.

The web server also detects video consumer disconnects, both graceful and ungraceful, and notifies the controller for resource reallocation. The web server is implemented with combination of HTML, PHP, and JavaScript, as depicted in Figure 6.2.2. A MySQL database on the backend serves as a repository of user account information and also caches current GC operating state information from the controller. To communicate with the Floodlight controller, the web server uses REST.

6.2.1.2 Floodlight OpenFlow Controller

The OpenFlow controller is implemented in Floodlight v1.1 [74] (which supports OpenFlow 1.0 - 1.4). As shown in Figure 6.2.3, the Floodlight controller is equipped with a GC module responsible for managing all video content and routing it to the video consumers via the GC private network. The web server communicates to the Floodlight controller through its REST API in order to add, remove, modify, stream out (watch), and query for videos. All communication exchanged between the web server and Floodlight is done in JSON for portability and ease of use. Floodlight's REST API and the web server can also be used with HTTPS for added security in the event the JSON

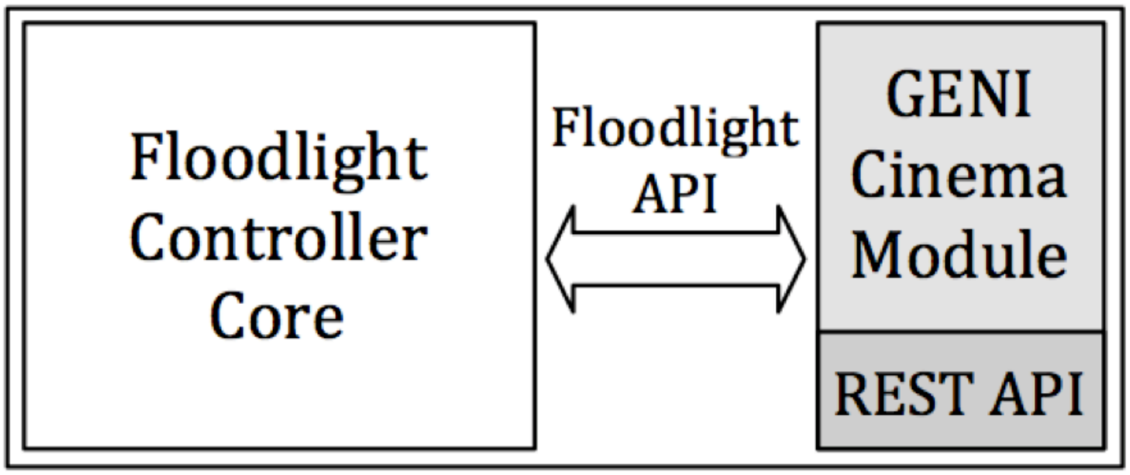


Figure 6.2.3: GENI Cinema Floodlight OpenFlow controller

data must be relayed over an untrusted network or the controller's REST API could be accessible by potentially untrusted clients.

6.2.1.3 Ingress and Egress Gateways

The points of video stream input into and output from the GC service are referred to as ingress and egress gateways. These gateways are implemented as nodes reserved at various aggregates in GENI and are designed to serve as firewalls and stream liveness detectors for video producers. These two functions have not been implemented and are not a proposed part of this dissertation, but they are included in the overall design of GC.

The firewall serves to protect the internal GC components from unauthorized use and can be implemented using standard techniques like Linux IP tables and also with OpenFlow. IP tables rules can be installed to permit only authorized or registered video streams to enter the GC service. This would prevent, for example, an attacker from observing the IPs and ports used to receive streams and performing a denial of service attack on these ingress points. In a similar manner, OpenFlow flows can be installed in an OpenFlow switch at the ingress point of GC. These flows would match the desired video stream sources at the ingress point and allow them to be passed into the GC service. All other attempts to stream video into GC will match a default flow to drop the packets.

The purpose of the stream liveness detectors is to determine if and when a video producer terminates a stream unexpectedly. In such an event, the GC service can react and free the resources for use by other video producers or take other appropriate actions. Liveness detectors can be

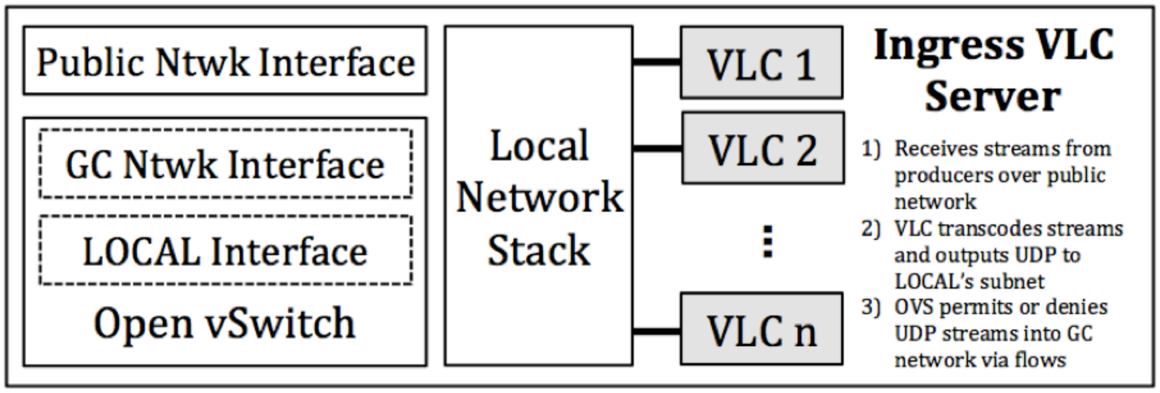

Public Ntwk Interface
GC Ntwk Interface
LOCAL Interface
Open vSwitch
Local Network Stack
VLC 1
VLC 2
VLC n
Ingress VLC Server
1) Receives streams from producers over public network
2) VLC transcodes streams and outputs UDP to LOCAL's subnet
3) OVS permits or denies UDP streams into GC network via flows

Figure 6.2.4: GENI Cinema Ingress VLC server

implemented using OpenFlow 1.3 meters, OpenFlow 1.0 idle timeout notifications (send_flow_rem flag set in flow mods), or by periodic counter polling for applicable flows. The use of liveness detectors is a GC feature not discussed as a part of this dissertation and has not been implemented.

The firewall and stream liveness detectors can also be used to augment all points that streams egress the GC service. In a similar manner, the firewall would only permit registered clients to view stream(s) authorized for that particular client. A liveness detector can also be used to detect when a client unexpectedly stops consuming a video stream.

The design of ingress and egress gateways were included in the design of GC, but are not critical to the operation of the service. As such, the ingress and egress VLC servers presently function as the ingress and egress gateways directly without the extra layer.

6.2.1.4 Ingress VLC Servers

When a video producer sends a video to GC, it must be hosted for video consumers to access. Furthermore, it must be accomplished such that the video producers and consumers do not directly interact with one another (i.e. for privacy and scalability reasons). As such, an ingress VLC server fulfills this requirement by receiving the video stream from the producer via the ingress gateway and hosting it for access by video consumer(s).

As shown in Figure 6.2.4, the ingress VLC server consists of multiple instances of VLC running in parallel. Each VLC instance listens on a transport port exposed to the video producers. As a result, each ingress VLC server can host as many live streams as there are VLC instances running.

After a VLC instance receives a video stream, it then transcodes the video stream to a constant format, and outputs the video stream into the private GC network over UDP. A constant format and

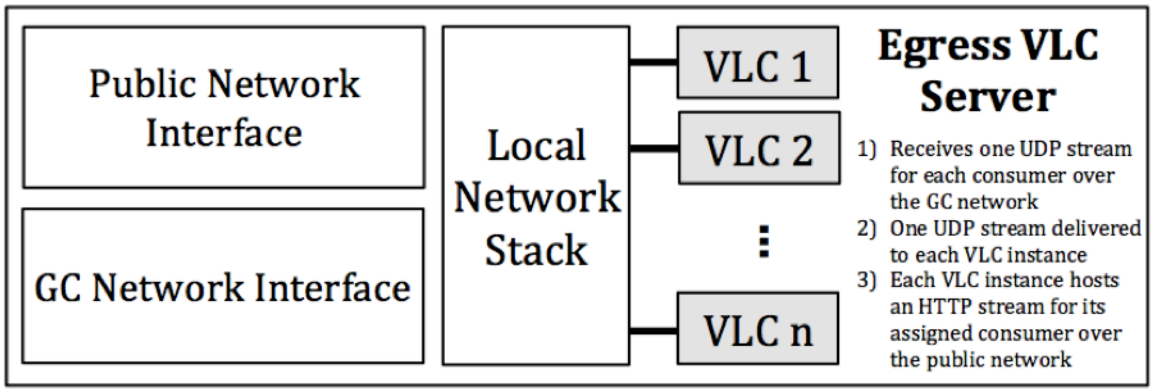

Public Network Interface
GC Network Interface
Local Network Stack
VLC 1
VLC 2
⋮
VLC n
Egress VLC Server
1) Receives one UDP stream for each consumer over the GC network
2) One UDP stream delivered to each VLC instance
3) Each VLC instance hosts an HTTP stream for its assigned consumer over the public network

Figure 6.2.5: GENI Cinema Egress VLC server

encoding is used for each video stream in order to make the channel changing process more seamless to the video consumer in the same manner a television provider scales channels to the same aspect ratio and sometimes quality.

Also shown in Figure 6.2.4, each ingress VLC server also contains a single onboard Open vSwitch (OVS) instance. This OVS serves as an "on/off" switch for each video stream. It switches a channel "off" if there is presently no demand for the channel by any video consumers. Likewise, it switches a channel "on" if there is at least one consumer wishing to watch that particular channel. This is accomplished using the Floodlight controller and simple allow/drop OpenFlow flows in the OVS from it's LOCAL port to the port attached to the private GC network – one action per hosted video stream. The LOCAL port is the interface OVS uses to tap into the local network stack of the machine. The ingress VLC server routes packets from the VLC instances to this LOCAL port (exposed as an interface in the local network stack), which then allows the video packets to enter the private GC network (allow/drop flow permitting).

6.2.1.5 Egress VLC Servers

Like the ingress VLC servers, the egress VLC servers run copies of VLC in order to receive and send video streams. However, the egress VLC servers, unlike their ingress counterparts, interact directly with the video consumers instead of the video producers.

As shown in Figure 6.2.5, each egress VLC server consists of multiple instances of VLC running in parallel. Each VLC instance is assigned to a video consumer and listens on a transport port within the private GC network. The video stream the video consumer has chosen is directed to this transport port, is received by the VLC instance listening, and is served to the video consumer on the public-facing interface of the server. Each egress VLC server can host as many video consumers as there are VLC instances running.

6.2.1.6 Private Network

The private OpenFlow network in GC is the link between the ingress and egress VLC servers. It is responsible for routing the video stream of each video hosted by an ingress VLC server to each video consumer wishing to watch that particular video on each egress VLC server. The topology of the private GC network is arbitrary; however, it must be capable of handling the bandwidth utilized by the video streams as they traverse the network from the ingress VLC servers to the egress VLC servers. In general, there are X ingress VLC servers and Y egress VLC servers. The network must be able to handle any arbitrary mapping from one-to-zero to one-to-many of video stream x in X to video consumer(s) y in Y. A fat tree topology with ingress and egress VLC servers at the edges naturally lends itself to this scenario, but any capable network will do ranging from a fat tree to a fully connected network.

All video traffic output from the ingress VLC servers into the private GC network is unicast UDP in order to allow fast video stream switching without regard for the connection state, sequence, or source as TCP would impose. Each UDP video stream is directed through the network toward all egress VLC servers where there is at least one video consumer wishing to watch that particular stream. Prior to each egress VLC server is an OpenFlow switch called a "sort switch."

This sort switch can be included on the VLC server itself as an OVS; however, it can also reside within the network. It is recommended that the sort switch be on a physically separate device from the egress VLC server, since both require real-time video stream processing. Each sort switch is responsible for taking the UDP video streams supplied as input on its interface(s) leading to the ingress VLC servers, duplicating these streams if appropriate, and sending them to the VLC instances on the associated egress VLC server. The involves rewriting the destination MAC, IP, and/or UDP port numbers in order for the network stack on the egress VLC server to accept the packets and send them to the VLC instances running as applications. The use of the sort switch reduces the duplicate transmission of video streams until the last hop at the egress point where the consumers are connected.

When a video consumer selects a channel to watch, the sort switch is responsible for selecting the appropriate input stream and sending it to the VLC instance on the egress VLC server associated

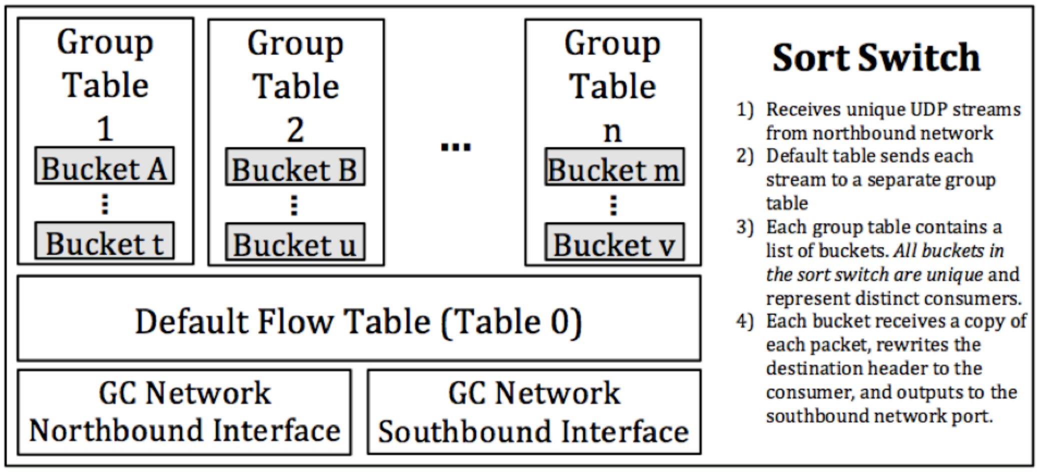

Group Table 1
Bucket A
Bucket t
Group Table 2
Bucket B
Bucket u
...
Group Table n
Bucket m
Bucket v
Sort Switch
1) Receives unique UDP streams from northbound network
2) Default table sends each stream to a separate group table
3) Each group table contains a list of buckets. All buckets in the sort switch are unique and represent distinct consumers.
4) Each bucket receives a copy of each packet, rewrites the destination header to the consumer, and outputs to the southbound network port.
Default Flow Table (Table 0)
GC Network Northbound Interface
GC Network Southbound Interface

Figure 6.2.6: GENI Cinema "sort" switch. The sort switch uses OpenFlow 1.1+ groups and buckets to duplicate direct video streams to the users based on present user demand.

with that video consumer. OpenFlow 1.1+ groups and buckets are used on the sort switch to implement this channel changing feature. As shown in Figure 6.2.6, every video is classified as an OpenFlow group, and every video consumer has a unique and single OpenFlow bucket. An OpenFlow bucket is a list of actions, and in the case of the buckets used for GC, each action list rewrites the destination MAC, IP, and/or UDP port headers of the packets. If a video consumer switches video channels, its bucket is removed from the previous video group of the channel it was viewing, and the bucket is simply added to the group of the video channel it wishes to watch. In this way, only one connection and video stream per consumer is ever present at a given time within the private GC network, and zero connections are set up or torn down on the ingress and egress VLC servers upon a channel change. This reduces the latency incurred during a channel change by switching the producer stream in the network rather than in the application as accomplished by other solutions. It also guarantees only one stream is sent to a consumer at a time, which reduces bandwidth consumption and removes the security concern present in an IP multicast implementation. Figure 6.2.7 depicts the sort switch more abstractly in the form of clients and channels.

6.2.1.7 The Agent-Based Framework in GENI Cinema

The design of the GC architecture is based on the agent-based framework. As shown in Figure 6.2.8, the ingress VLC servers, egress VLC servers, and sort switches themselves implement framework agents and software forwarding devices, while the Floodlight controller implements the controller plugin through the GC module.

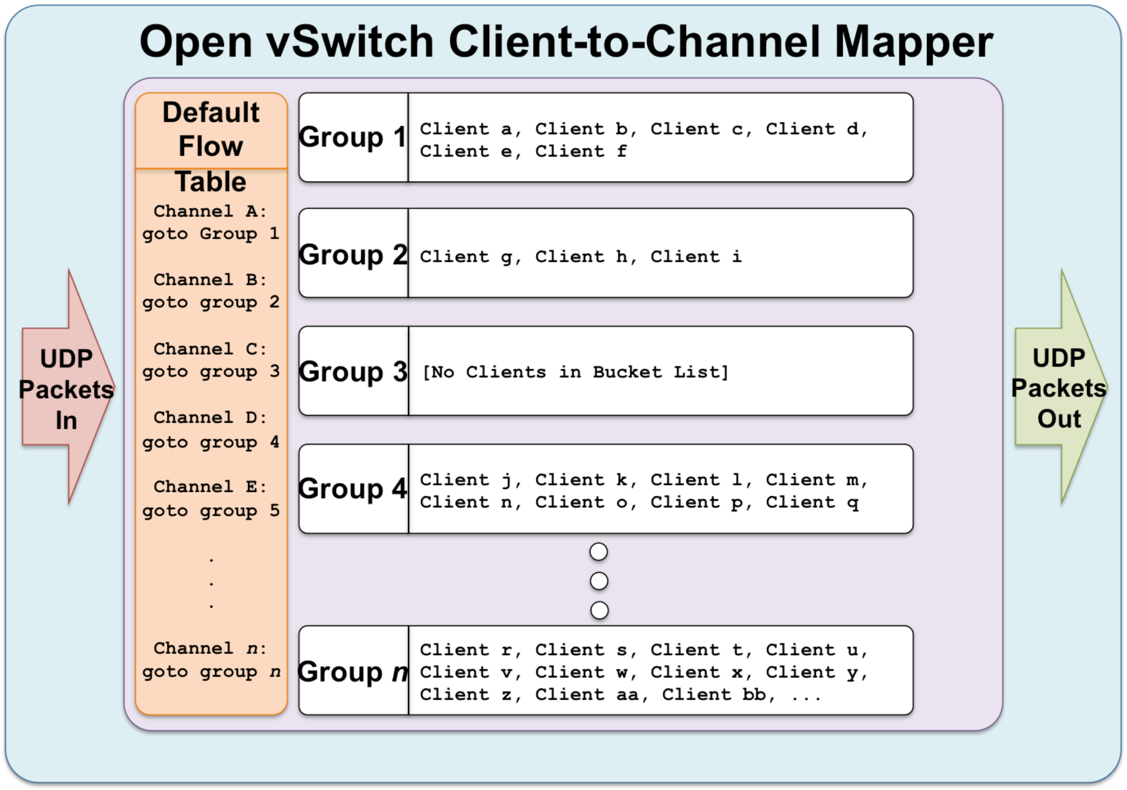

Open vSwitch Client-to-Channel Mapper
Default Flow Table
Channel A: goto Group 1
Channel B: goto group 2
Channel C: goto group 3
Channel D: goto group 4
Channel E: goto group 5
Channel n: goto group n
Group 1
Client a, Client b, Client c, Client d, Client e, Client f
Group 2
Client g, Client h, Client i
Group 3
[No Clients in Bucket List]
Group 4
Client j, Client k, Client l, Client m, Client n, Client o, Client p, Client q
Group n
Client r, Client s, Client t, Client u, Client v, Client w, Client x, Client y, Client z, Client aa, Client bb, ...
UDP Packets In
UDP Packets Out

Figure 6.2.7: How channels and clients relate to the GENI Cinema “sort” switch

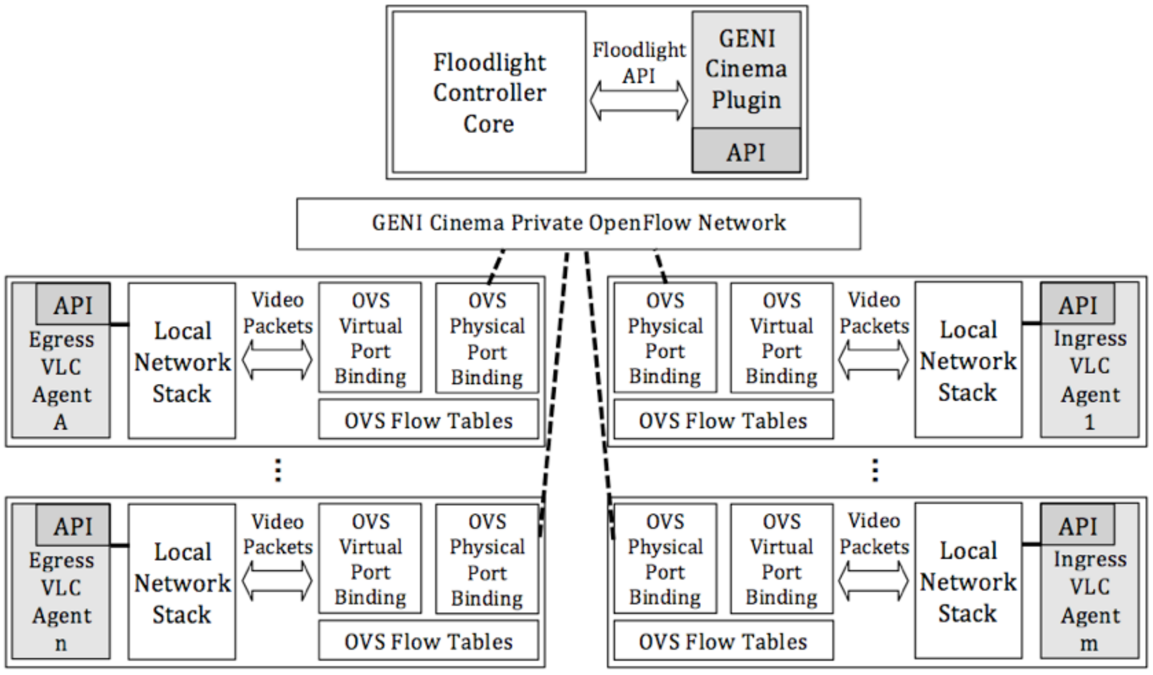


Figure 6.2.8: The influence of the agent-based framework in the design of the GC architecture

6.2.2 Capacity-Aware Content Distribution

GC controls the distribution of the video streams throughout the network core and to the edge. Because it exercises such precise control, GC is aware of the flows installed on each switch and thus the bandwidth consumed on each link. Assuming each video stream is a known, constant bitrate – guaranteed by the ingress VLC servers – GC can compute the available bandwidth on any link at any time based on the current distribution of video streams. As such, given the link capacity of each link in the network core from the ingress VLC server to the sort switches and egress VLC servers, GC can ensure link oversubscription does not occur.

GC does not control the distribution from the edge to the consumer, however. The assumption is that each egress VLC server is capped at the number of consumers it can support based on its non-GC public network capacity.

At both the edge where GC meets the public network and in the private GC core network, it should be noted that the networks need not be used entirely for GC video traffic. Each link can be configured with an artificial maximum capacity to allow for third party network users to coexist on the same network infrastructure. A common complaint of IP multicast users is the potential for IP multicast streams to "step on" other network users due to their greedy and typically unrestrained use of network resources. With GC and artificial link capacities, bandwidth can be allocated for both GC video traffic and for other network users, ensuring an exceptional quality of service for all constituents.

6.2.3 Workflow

When a video producer wishes to register a video as input into GC, it does so via the website. The website relays the request to Floodlight, which responds with an appropriate ingress gateway IP address and transport port number to which the video producer can connect and send the video. Upon reception of the video stream, the ingress gateway relays it to an ingress VLC server where the live stream is hosted.

When a video consumer on the website requests to watch a particular video, the website relays the request to Floodlight, which responds with an appropriate egress gateway IP address, and transport port number where the video consumer can connect and watch the video. The video selected is routed, duplicated, and rewritten within the private network of GC from the ingress VLC server on which it is being hosted to the private interface of the egress VLC server where the consumer is connected. A VLC instance on this egress VLC server outputs the video on the public interface and relays it to the video consumer. Video requests are presently served over HTTP in order to display the live stream within a VLC plugin in the video consumer's web browser. However, other protocols may be used to suit other video players at the video consumer. This would require modifying the egress gateway's VLC server parameters to use the appropriate encoding and transmission protocol as well.

6.2.4 Example

The following demonstrates the operations performed and the network flows present during a consumer and producer join, switch, and leave. Each of the following subsections indicates a time step in the example join, switch, and leave operations.

6.2.4.1 Single Consumer Waiting (t=0)

Assume the initial state of the system is such that a single GC consumer is waiting on its selected channel to become available. In this case, there are zero flows installed in the network, as depicted in Figure 6.2.9.

6.2.4.2 Desired Producer Joins (t=1)

After some time, the video stream on which the consumer is waiting is provided by a producer. At this point in time, the GC controller will install flows along the tree rooted at the ingress VLC server and branched to all consumers, as depicted by green arrows in Figure 6.2.10.

6.2.4.3 Another Consumer Joins (t=2)

After some time, another consumer joins. This time, assume the producer was already present, so the consumer does not wait; flows are installed for the stream at the time of the join, as depicted by red flows in Figure 6.2.11. This operation does not disrupt ongoing video streams, such as the green stream.

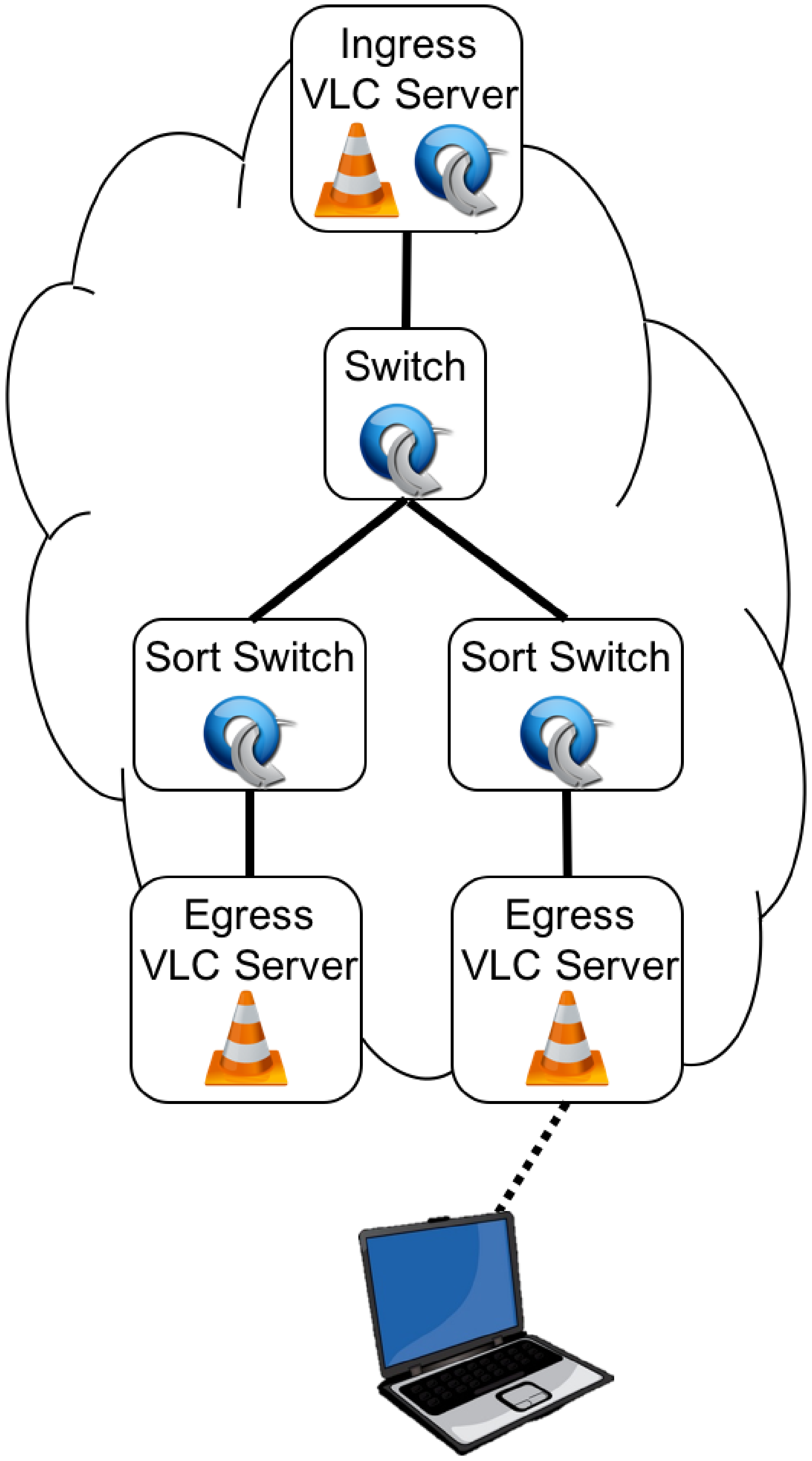


Figure 6.2.9: Zero producers and a single waiting consumer results in no flows installed

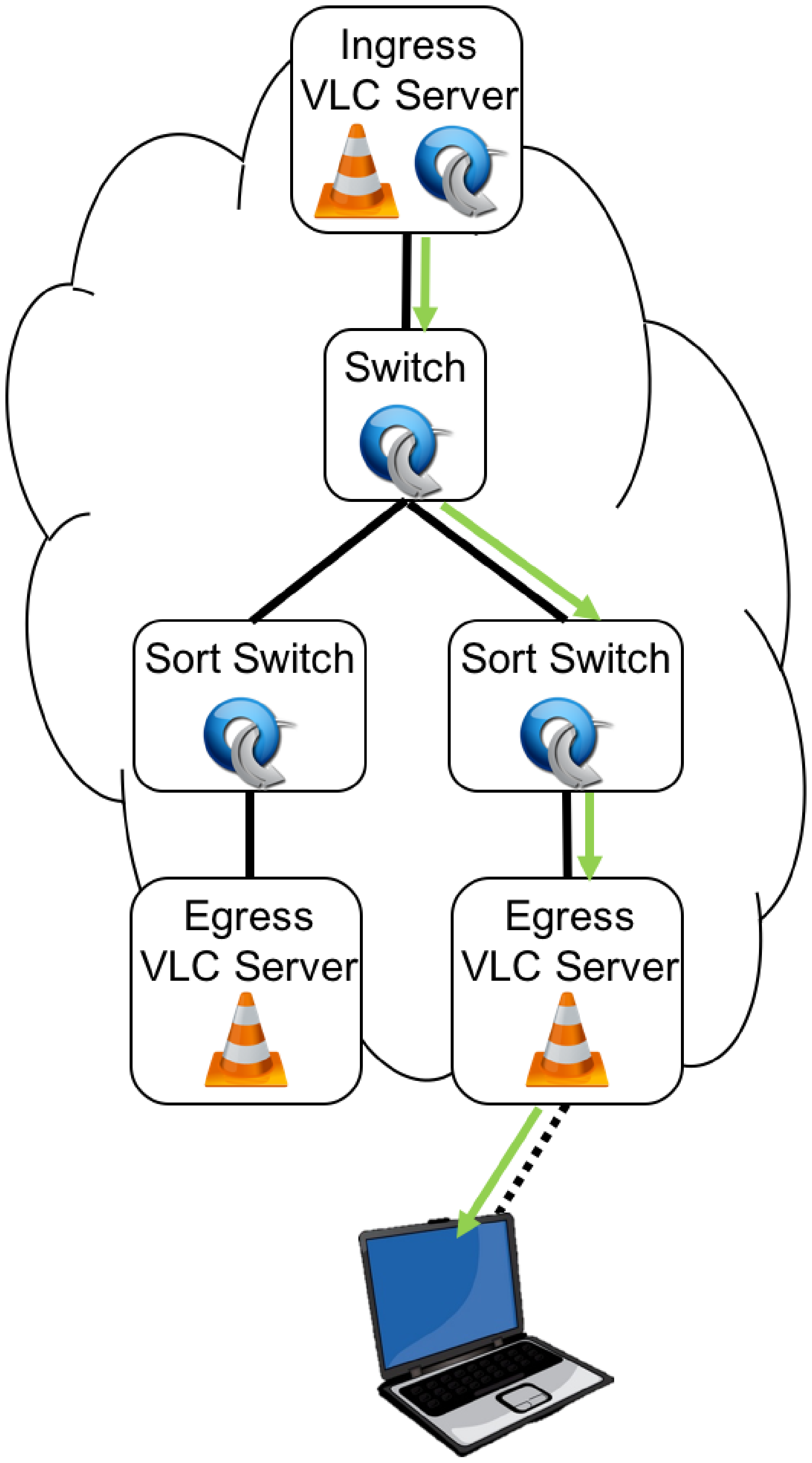


Figure 6.2.10: The producer on which the consumer was waiting joins, and the stream is routed via flows

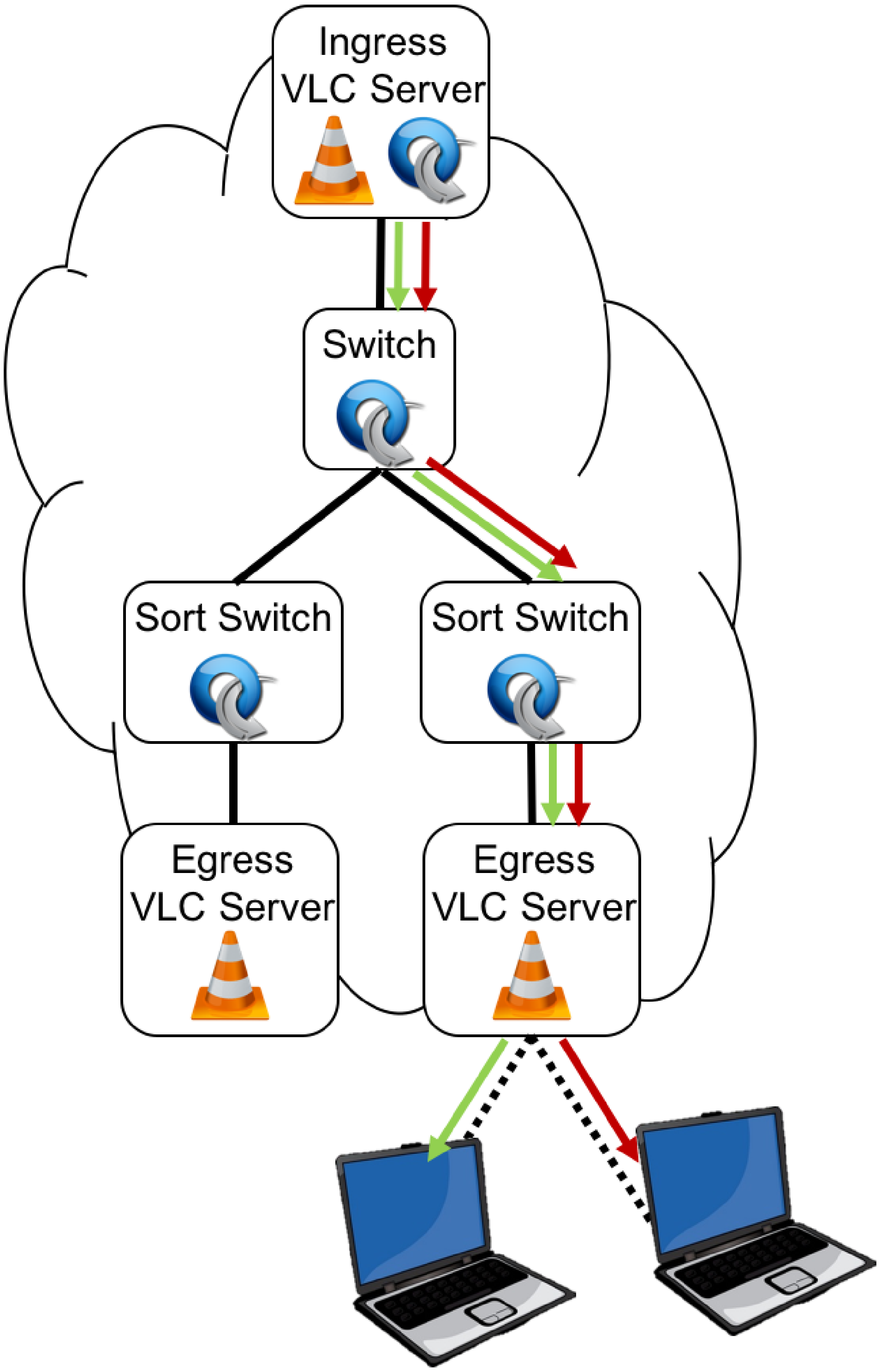


Figure 6.2.11: A new consumer joins and requests a different stream. This stream was already present, so flows are installed.

6.2.4.4 One Consumer Switches Streams (t=3)

The consumer watching the green stream decides to switch to the red stream. Because the red stream already has a tree of flows installed in the network, the consumer is simply "attached" to the tree via flow modification and add operations. In this case, the modifications involve removing the first consumer from the green stream's bucket and adding it to the red stream's bucket at the sort switch. The green stream no longer has any consumers, so it is removed from the network through flow delete operations. The end result is shown in Figure 6.2.12.

6.2.4.5 Another Consumer Joins at a Different Egress Point (t=4)

Another consumer joins GC and wants to watch the same red stream as the current consumers. Due to the new consumer's location, it is assigned a different egress VLC server. Because the red stream already has a tree of flows installed in the network, the consumer is simply "attached" to the tree via flow modification and add operations. The new consumer is not associated with a sort switch where the red stream is already being watched, so it joins the tree at the nearest point and a new group and bucket for the red stream is installed on the local sort switch. The end result is shown in Figure 6.2.13.

6.2.4.6 Consumer Switches to Different Producer (t=5)

The most recent consumer to join on the left switches to the green stream. Because there are no longer any consumers of the red stream at its sort switch, the red stream tree is pruned via flow deletes. Simultaneously, the green stream flows are installed in a tree from the producer at the ingress VLC server to the consumers, as depicted in Figure 6.2.14.

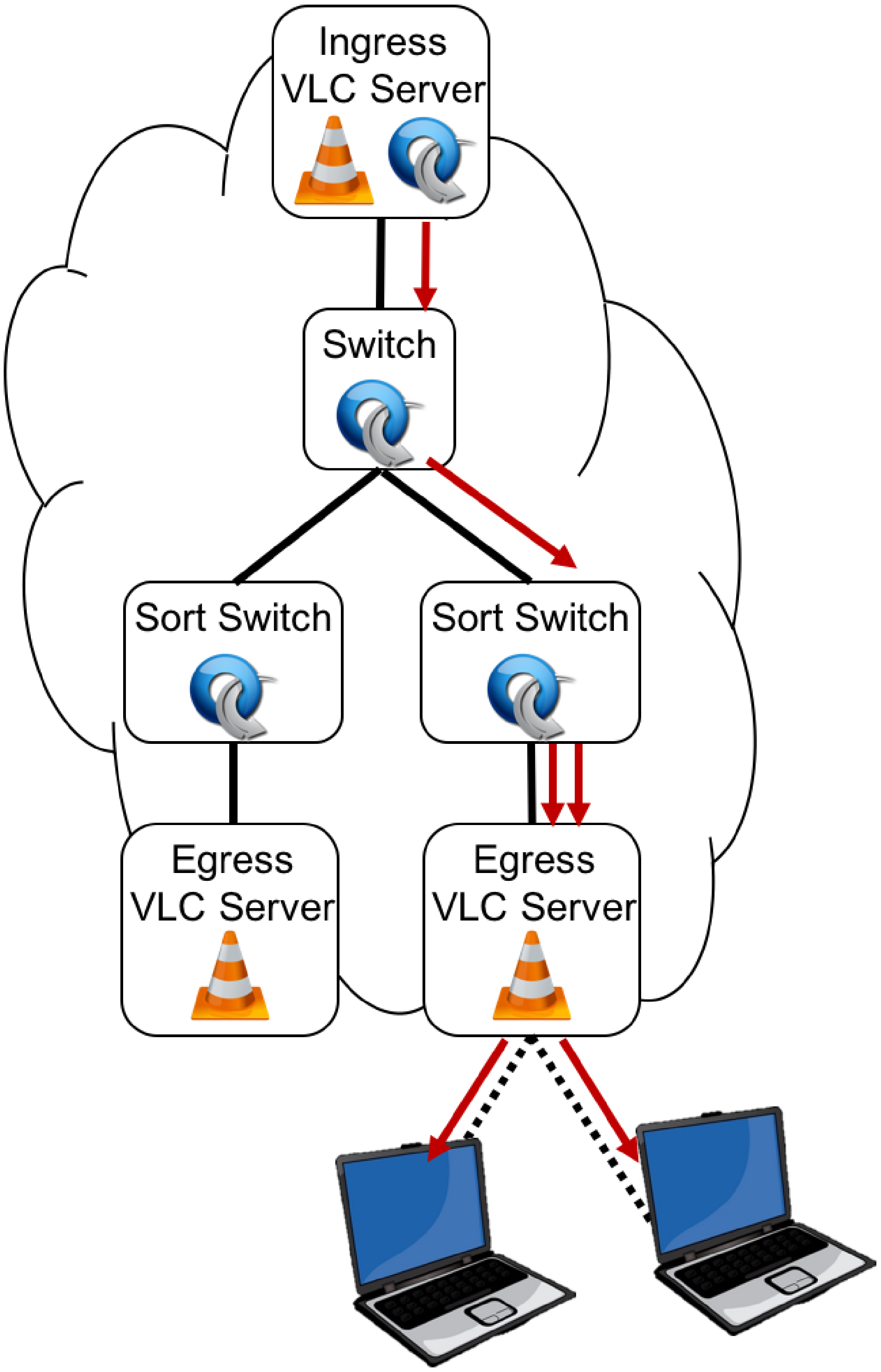


Figure 6.2.12: A consumer switches to another stream. The stream is already at the sort switch, so only a group and bucket operation is required.

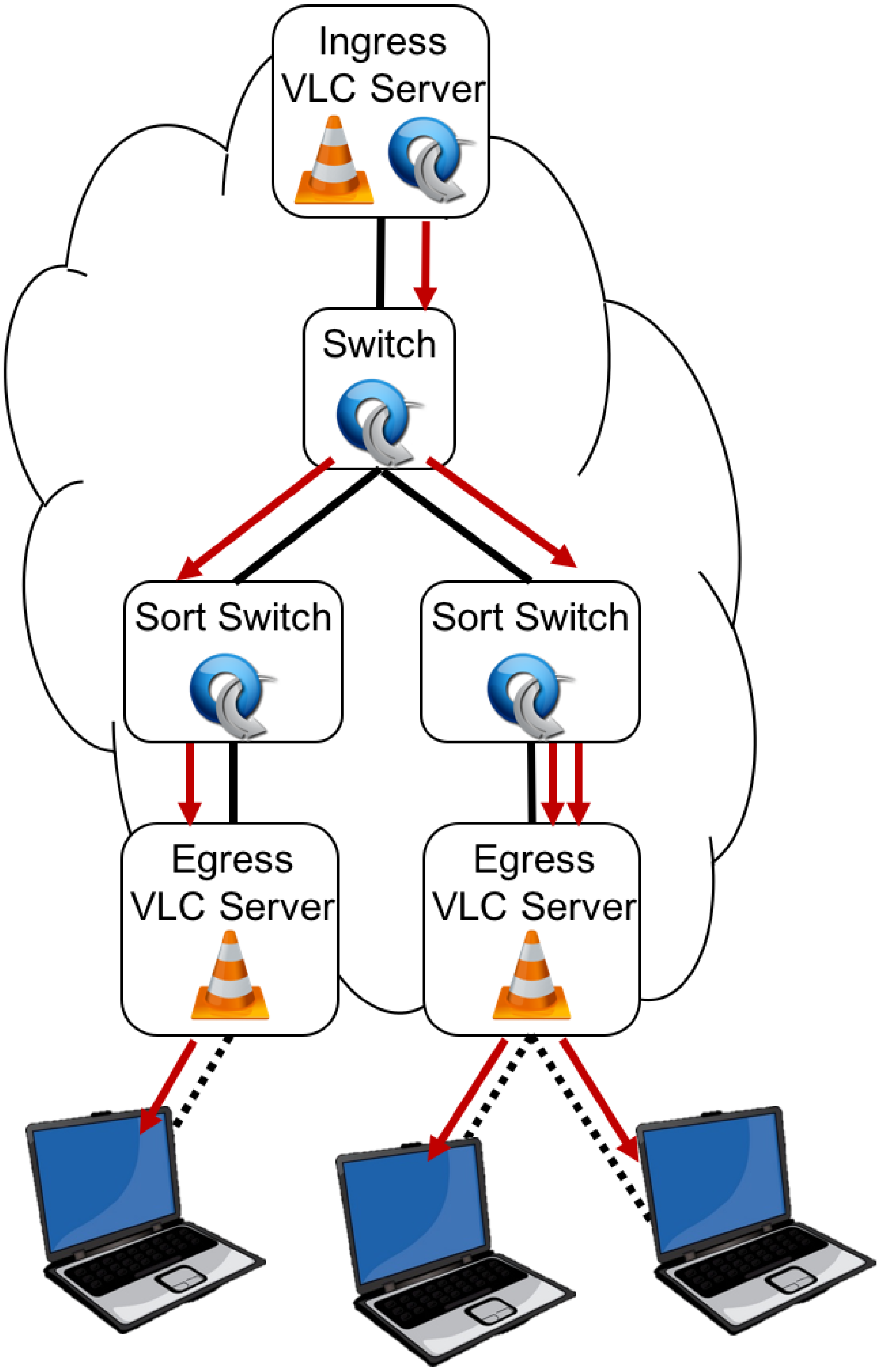


Figure 6.2.13: Another consumer joins, involving a second sort switch

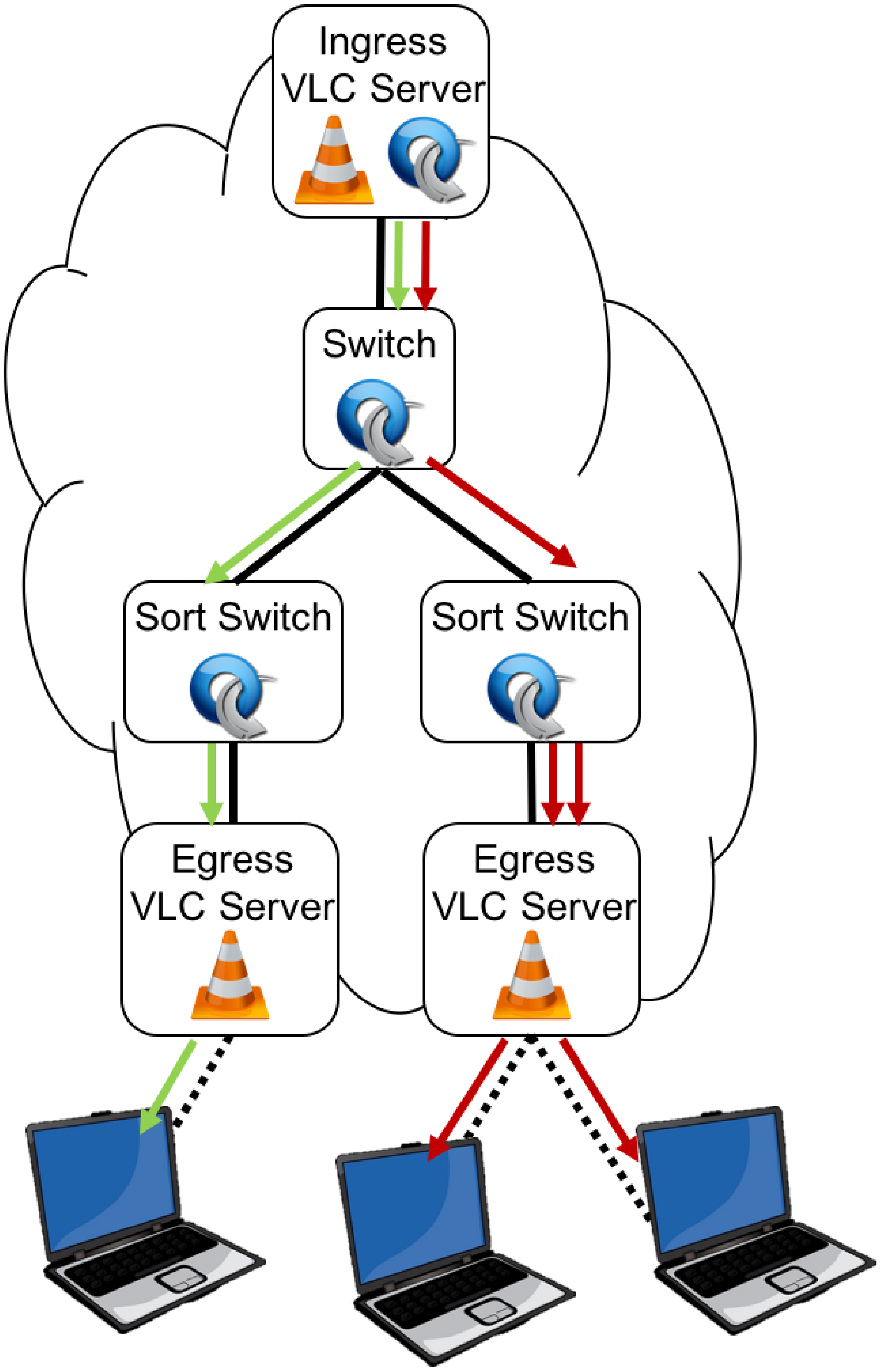


Figure 6.2.14: A consumer switches to the green stream.

6.3 Evaluation

6.3.1 Bandwidth Consumption and Latency

In order to achieve a quick transition between one video stream and another, there are a few options, each which has advantages and disadvantages.

1. Prefetch both streams and perform the switch locally at the consumer. This is the solution that will always achieve the most rapid switch; however, it requires up to double the bandwidth consumption, where only a fraction of the bandwidth consumed at any given time is used to show the user the active video stream.
2. Stop the first stream and start the second. This results in the lowest amount of bandwidth usage; however, the "gap" between visible content due to the fetch and buffering results in a quality of service that is unacceptable to the consumer.
3. Start the second stream and stop the first. This results in a temporary increase in bandwidth as the second stream is fetched. When the second video stream is buffered at the consumer, the first stream can then be terminated, bringing bandwidth consumption back to that of a single stream. Another side effect of this approach is the need to wait for buffering of the second stream to complete at the consumer.
4. Inject the second stream into the socket the consumer is using for the first stream. This results in a zero increase in bandwidth consumption, zero additional buffering, no perceived "gap" in content, and a minimal latency.

Options 1 through 3 are implemented by traditional video streaming methods, including IP multicast. Option 4 is implemented in GC and is made possible by OpenFlow.

The bandwidth consumption and latency incurred by GC is minimal during a video stream fetch, switch, and termination. During a fetch, because GC in knowledgable of all producer locations, it can precisely route the video stream being fetched from the producer to the consumer. The path chosen is the shortest path of the tree constructed for this video stream or the video stream source. This is analogous to IP multicast; however, no search is involved, since the controller can directly

Technology	Channel Change Time (s)
GENI Cinema	2
Digital Television	4
Analog Television	< 1
Satellite Television	6
Internet Streaming	5

Table 6.3.1: Channel switching delay as perceived by the consumer

compute the route and install the flows. The latency for a join is the time required to compute and install the flows for the route, propagation delay for the video to traverse the route, plus any buffering done on the egress VLC server and on the consumer.

The bandwidth consumption during a video stream switch is constant from the egress VLC server to the consumer. In the network core, the second stream is fetched in the same manner as described above. In parallel with this, the consumer's OpenFlow group bucket is removed from the old stream group and added to the new stream group on the consumer's sort switch. Also done in parallel, the flows used by the old stream's distribution tree are pruned if the consumer was the last consumer watching that stream at its egress location. The buffer used on the egress VLC server allows for a brief fluctuation in the video streams as they are switched in the core. This buffer accounts for the propagation delay of the second stream through the network core to the VLC instance on the egress VLC server.

With a one second buffer at the user's video player (VLC web plugin) and a 200 millisecond buffer on the egress VLC server, the average perceived delay experienced when switching live, high-definition 720p streams is approximately two seconds. This delay is on par with and sometimes less than delays experimentally derived from modern digital cable and satellite television services of four seconds and six seconds, respectively. In addition, when switching between high-definition 720p streams, GC outperforms contemporary live video streaming solutions for the Internet such as YouTube Live at approximately five seconds. These results are depicted in Table 6.3.1.

Lastly, when a consumer wishes to stop watching video streams all together, it will terminate its session. GC will simply remove the consumer's bucket from the OpenFlow group of the video channel the consumer was last watching. This will terminate the stream to the consumer. If the consumer was the only consumer watching the video stream, the group for that stream will be removed from the sort switch, and the flows for this video stream will be pruned back in the distribution tree for the stream. As such, the bandwidth consumed for this user will be stopped immediately after the group and flow modification operations complete. The latency upon a leave is the time for this operations to complete.

6.3.2 Scalability

To show scalability of GC, a prototype deployment included two ingress gateways, a private OpenFlow-based core network, and seven egress gateways, as shown in Figure 6.3.1. Each ingress and egress gateway were hosted on dedicated and capable InstaGENI servers, where 100 VLC agents were run in parallel to support a total of 200 video producers and 700 video consumers. Each video stream was capped at 1Mbps such that an ingress gateway could potentially receive a maximum of 100Mbps aggregate video traffic from video producers and thus send into the private GC network a maximum of 100Mbps. If more than 200 video producers are desired, additional ingress gateways can be added to the service, each which would support an additional 100 video streams. Likewise, if more than 700 video consumers are needed, additional egress gateways can be added to GC, each which would support an additional 100 video streams. As such, the architecture of GC allows for an elastic deployment, where gateways are added or possibly removed as dictated by demand. As will be discussed below, there is no theoretical limit to the number of video consumers GC can support. Video producers are limited by the aggregate link bandwidth of the minimum cut of the private GC network. Increasing this minimum cut bandwidth can allow GC to scale to an arbitrarily large number of video producers and an arbitrarily large number of video consumers.

Within the private, OpenFlow-based GC network, each video stream is intelligently routed from each ingress gateway to every egress gateway where there is presently demand for that particular stream. As such, a minimum of 0Mbps and a maximum of 100Mbps bandwidth is required leading from each ingress gateway to the private GC network to satisfy the cases of zero demand and full-demand for the ingress gateway's video streams, respectively. Similarly, a minimum of 1Mbps and a maximum of 100Mbps bandwidth is required to reach each egress gateway for the cases where all video consumers are watching the same video stream and all are watching different streams, respectively. The private, OpenFlow-based network core is arbitrary but can scale using (1) a fat tree topology with the ingress and egress gateways as the leaves and the center of the core network as the root, or (2) a fully-connected network from all ingress gateways to all egress gateways where each link carries 100Mbps.

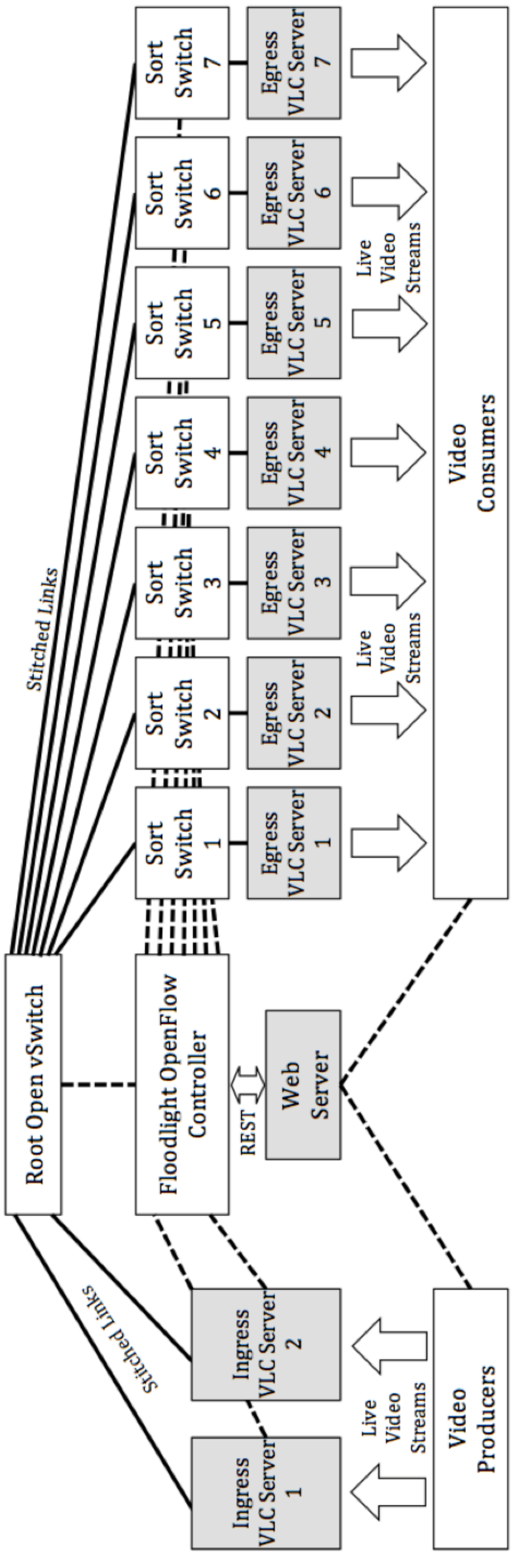


Figure 6.3.1: Example GENI Cinema deployment

The sort switch, which is responsible for duplicating the video streams for clients is done at the egress edge of GC prior to each egress VLC server and gateway. This guarantees that no streams are transmitted in duplicate over any given link in the GC private network architecture. There are two edge cases to consider when analyzing the load of a sort switch:

1. All consumers allocated to the sort switch have selected the exact same producer's video stream
2. All consumers allocated to the sort switch have selected unique video streams from different producers

Considering the first cast, if all 100 video consumers wish to watch the same video stream, a single stream will be sent by the private GC network to the sort switch, using 1Mbps bandwidth. This single stream will be made into 100 copies where each copy's destination headers are rewritten such that the packets are routed to the VLC instance of each video consumer on the associated and nearby egress VLC server. This means 1Mbps of traffic enters the sort switch and 100Mbps exits.

On the contrary, considering the second case, if there are 100 video consumers and each wishes to watch a different video of those available, there will be 100 unique video streams – the total channels available – entering the sort switch for a total of 100Mbps. The sort switch will make zero copies of each stream and simply rewrite the destination headers of each stream to send it to the VLC instance of the video consumer that wishes to watch that particular stream. In this case, the total exit traffic is still 100Mbps leaving sort switch. The exit traffic is directly proportional to the number of video consumers presently attached to that particular egress gateway. The traffic entering can be no more than the minimum of either the total number of video channels available or the number of video consumers connected to the associated egress gateway.

In general, GC can scale in arbitrary topologies. Additional ingress and egress servers can be added to satisfy increased compute requirements and greater demand for ingress and egress locations. Within the private GC network, additional links can be added to drive the greater bandwidth requirements of additional consumers. An example is shown in Figure 6.3.2.

6.3.3 IP Multicast

GC stands apart from IP multicast in the following ways:

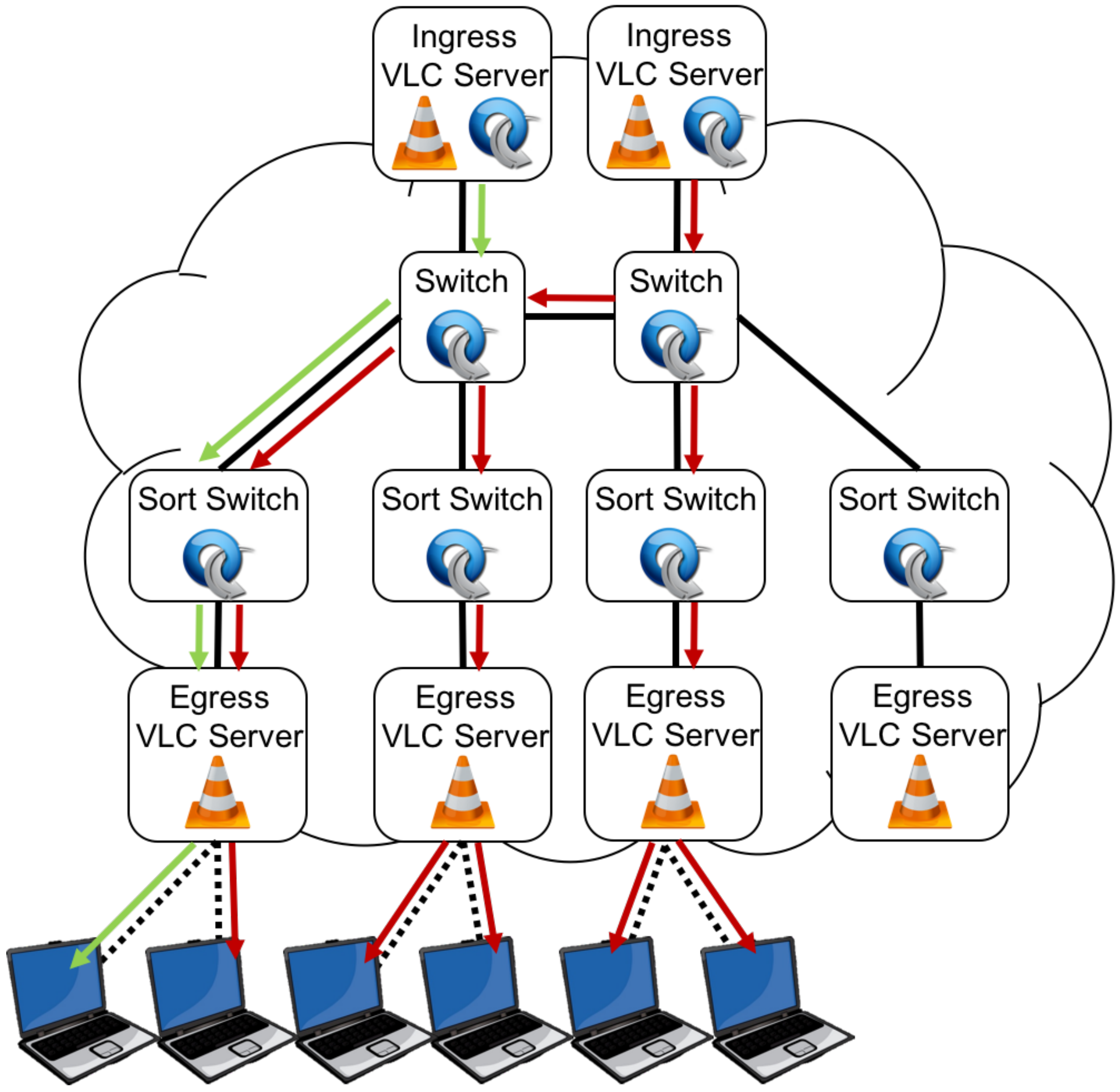


Figure 6.3.2: The GC architecture can scale through the addition of ingress and egress servers as the numbers of producers and consumers grow. Network bandwidth consumption must scale with the number of consumers through the addition of links or utilization of higher capacity links.

1. GC is inherently more secure. The control plane of GC is the only means by which flow rules can be installed for a video publisher to use and provide content. If the content is not approved (based on criteria beyond the scope of this dissertation), GC does not allow it ingress the service by drop rules. Contrast this to IP multicast, which allows any host to publish to and subscribe to any IP multicast group. Similarly, GC limits content subscription to authorized users using multi-layer security features, such as OpenFlow and HTTPS.

2. GC is a cloud-based service that operates on top of existing network infrastructure. It does not mandate specific hardware requirements for a domain to participate. IP multicast must be supported within and between ISP and domain networks in order for content to be shared and distributed. IP multicast also does not penetrate firewalls and NAT without explicit allow rules. GENI Cinema avoids NAT and firewalls given the distributed, cloud-based architecture of the system and the fact that TCP can be used at the edges to request streams if a firewall exists between a subscriber and the nearest egress gateway.

3. GC's global network knowledge provides an architecture where various video distribution algorithms can be used to achieve different optimizations. For example, IP multicast does not have congestion control. With GC, each video's tree is known, and each video stream's bandwidth is known. With this information, GC can precisely compute the available bandwidth and (re)route video streams as necessary to support subscribers according to the algorithm used or the optimization desired. It is also feasible for GC to carve out and reserve bandwidth for non-GC traffic as to not "step on" others. However, since it is presently deployed as a private network in the cloud, other traffic is a non-issue. There is no limitation though that prevents GC from being deployed on a network and co-existing with other third-party traffic.

4. GC's global subscriber knowledge allows for precise video stream termination. When a subscriber leaves an IP multicast group, the first-hop router can stop the subscriber's stream locally; however, IGMP has a default leave processing interval of 1 second. The stream would still exist for up to 1 second after the client's leave is queued. GC has no such processing interval and will immediately modify OpenFlow flows to effectively terminate the subscriber's stream from the edge to the subscriber as soon as the subscriber terminates its video stream's TCP connection. Furthermore, GC will immediately remove the client's OpenFlow bucket from its edge "sort" switch's OpenFlow group as soon as the client leaves. A more significant

improvement though over the IP multicast algorithm is that GC will also terminate the video stream from the edge “sort” switch to where the video stream attaches to the tree if the leave notification came from the last subscriber at that geographic egress location. Contrast this to IP multicast, which collects IGMP subscription notifications over a time interval. If no subscription notifications were received for a particular content group during the interval, the router will then inform its upstream multicast source that it no longer wishes to receive the content. This implementation is necessary, since each router is stateless and does not actively track subscribers. A default interval of (2)(125s) + (1/2)(100) or 300s or 5 minutes [19] must elapse from the last time a join was made to the multicast group (or a downstream router sent a report with a positive group membership) before the local router determines nobody else wants the stream. This delay is a maximum of 5 minutes in the worst case, minimum 0 seconds in the best case – both edge cases. When this timer expires, the local router will inform the upstream router that it no longer wants the stream. This is when the "old" stream will officially cease consuming bandwidth from the upstream router to the local router. This implies a multicast stream could consume bandwidth for up to 5 minutes after the last subscriber leaves. GC's global subscriber knowledge allows streams to be terminated as soon as the last subscriber leaves.

In order to compare GC performance to IP multicast, the IP multicast specification and related works are used to derive and reference performance edge cases present in IP multicast. Cases to consider with respect to bandwidth consumption and latency are:

- Join
- Leave
- Switch

Both IP multicast and GC operate in a steady state when consumers are not performing any of the above operations. Steady state means video stream bandwidth consumption is at a minimum and is not in flux. An edge case analysis for both IP multicast and GC will yield the relative performance of each of the primitives given above. Section 6.3.3.1, Section 6.3.3.2, and Section 6.3.3.3 discuss the moving parts that must be considered when analyzing IP multicast and GC for join, leave, and switch, respectively.

6.3.3.1 Join

IP Multicast

When a user wishes to join an IP multicast group, an IGMP join request is sent to the user's local router that supports IP multicast. There are two things that can happen at the router upon the reception of such a request:

1. The router determines that it already is participating in the multicast group. In this case, the router simply sends the requested multicast group data to the user. This delay can be modeled as the RTT between the user and its first hop router.
2. The router is not already receiving the multicast group. In such an event, the router will use one or more IP multicast discovery protocols to ask its neighbors if they have the requested IP multicast group. The group discovery process is distributed in nature and does not have a finite convergence time. In fact, the requested IP multicast group is not guaranteed to be found. Assuming a router with the group is located, the router will relay the group "downstream" to the requesting router, which will do the same and relay the group to the router from which it received the request. This process will repeat until the group is relayed to the original requesting router, which will then send the group to the user who requested it. Given this process, the delay to join an IP multicast group is dependent on the latency on each link for each hop the request must traverse until it reaches a router that is in possession of the group in question. The data in the group must then propagate back to the user, which will traverse the same links. As such, the delay can be modeled as two times the latency between the source router and the user.

Let J_{mc} denote the time to join a video stream using IP multicast.

$$J_{mc} = 2l_{dr} + \begin{cases} 2\sum_{i=1}^{j} l_i + 2\sum_{m=1}^{n} l_m & \textit{initial} \\ 0 & \textit{local} \\ 2\sum_{i=1}^{j} l_i & \textit{remote} \end{cases}$$

l_{dr} is the latency between the consumer and the designated router. The *initial* case is the worst case scenario and occurs when a join request occurs and the source currently has no consumers. It accounts for the latency of each hop between the designated router and the rendezvous point with l_i and from the rendezvous point to the designated router of the source with l_m. The *local* case is the best case scenario and occurs when the requested multicast group is already being served by the designated router of the consumer. The *remote* case occurs when the requested multicast group is not present at the consumer's designated router but is present at the rendezvous point or at some router en route to the rendezvous point.

An IP multicast join does not have to end here though. Depending on the exact IP multicast implementation, further work might be done. Clearly, using the rendezvous point of the multicast group might not result in the lowest cost content distribution tree from the producer to the consumers. Typical IP multicast deployments will construct a secondary content distribution tree after a consumer joins. After the construction of this more optimal tree between the producer and consumer, the old tree via the rendezvous point is decommissioned. The additional time to perform this operation is defined by T_{spt} as $T_{spt} = T_{build} + T_{decom}$, where T_{build} is the time to construct the new shortest path tree and T_{decom} is the time to decommission or tear down the old tree routed through the rendezvous point. T_{build} can be computed as the time for the consumer's designated router to request to join the producer. This request is done using normal layer 3 routing with the designated router's routing table and the producer's IP. This repeats for each hop en route to the producer. Once at the producer's designated router, the content will be relayed back along the path towards the consumer, resulting in an overall $T_{build} = 2\sum_{s=1}^{t} l_s$. After the producer's designated router begins to relay the content to the consumer along this new tree, it will begin to tear down the old tree by issuing leave requests in the path towards the rendezvous point. Each router along the way will continue to propagate the leave message if it does not have any local consumers. Eventually, the leave will reach the rendezvous point and the consumer's designated router, decommissioning the entire tree. At this point, all the content will be relayed via the new shortest path tree. The time to perform the entire decommission process T_{decom} can be computed as $T_{decom} = 2\sum_{u=1}^{v} l_u$ where the path u to v is the path from the producer's designated router to the v hop where there are still consumers of the content that is being terminated. Best case, v is the consumer's designated router; worst case it is the rendezvous point.

During this transient period of shortest path tree construction and old tree decommission, IP multicast uses bandwidth to transmit the content redundantly. The redundant bandwidth consumed can be computed by analyzing the shortest path tree construction and decommission process hop-by-hop and factoring in the average bitrate of the content being transmitted, b. First, when the shortest path tree is being constructed, bandwidth utilization is that of the path through the rendezvous point, $B_{rp} = b\sum_{i=1}^{j} l_i + b\sum_{m=1}^{n} l_m$ plus the additional and incremental bandwidth consumed while the shortest path tree is constructed, $B_{spt} = b\sum_{s=1}^{t} sl_s$. This clearly is inefficient, since construction of the shortest path tree in the first place would have been ideal. As such, the wasted bandwidth can be defined as the difference between the bandwidth consumed by the initial tree through the rendezvous point and the bandwidth consumed by the shortest path tree. Furthermore, if the rendezvous point is not a part of the shortest path tree, bandwidth for the content will be consumed along the path from the producer's designated router to the rendezvous point. This will continue to be consumed regardless whether or not the rendezvous point has any consumers, even after the rendezvous point tree is pruned. Figure 6.3.3 indicates the links utilized for content distribution prior to the construction of the shortest path tree. Figure 6.3.4 shows the final distribution tree after the shortest path tree has been constructed and the rendezvous point tree has been pruned. Bandwidth is continuously used along the shorter of the path from the producer's designated router to the rendezvous point or from an upstream router from the producer that is participating in the shortest path tree to the rendezvous point, regardless whether or not the rendezvous point has any downstream consumers.

GENI Cinema

When a user wishes to initially watch a GC channel, a HTTP post is sent from the user to the web server, which then relays the request to the OpenFlow controller. This delay is the latency between the user and the web server plus the latency between the web server and the controller. The latter can be neglected assuming the web server and the controller are operating on the same machine or are on a LAN. After this, one of two cases can occur:

1. If there is another user on the requester's edge switch that is also watching the requested

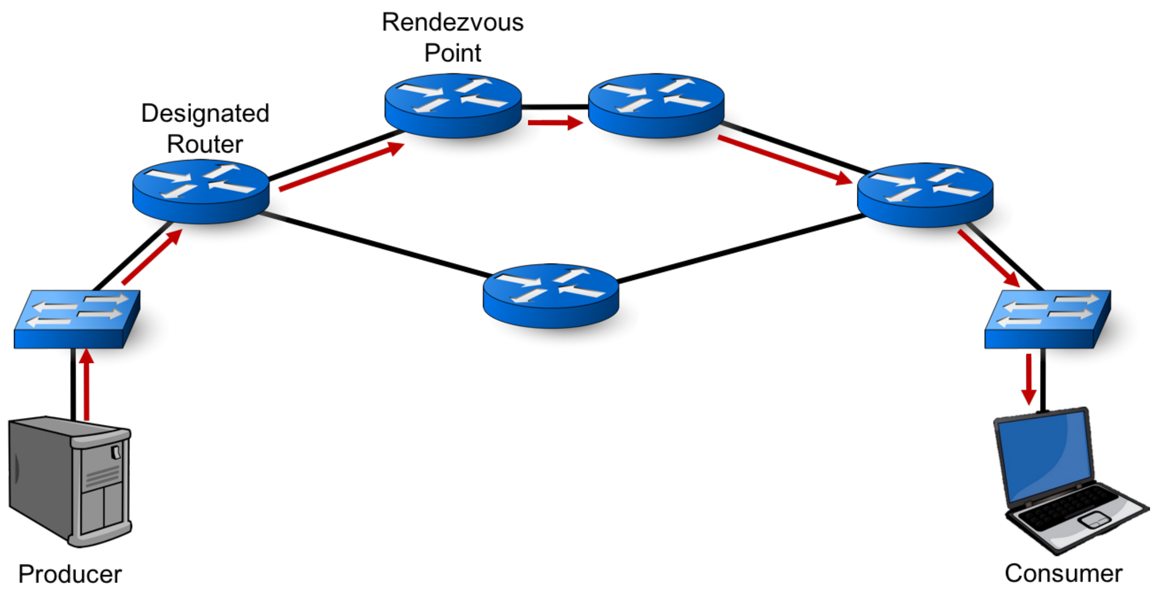


Figure 6.3.3: Example PIM-SM rendezvous point content distribution tree

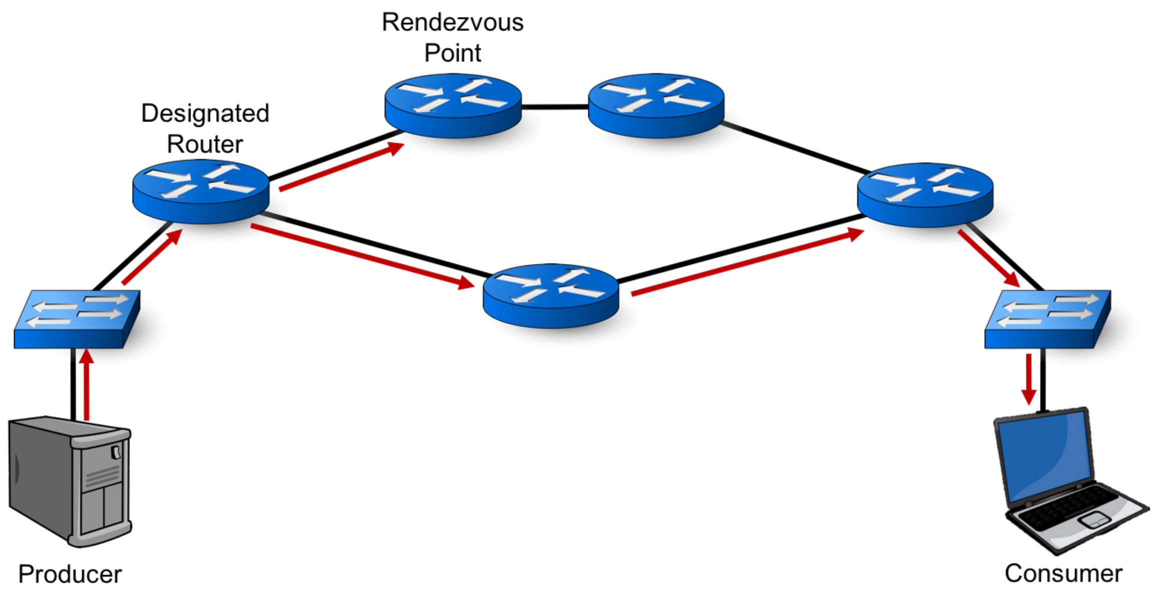


Figure 6.3.4: Example PIM-SM shortest path content distribution tree

channel, the flows on the edge switch will be modified to duplicate the stream and forward it to the user. The time to accomplish this is the control messaging latency from the OpenFlow controller to the user's edge switch plus the time for the stream to propagate from the edge switch to the user.

2. If the edge switch where the user is attached is not in possession of the channel requested, and because the OpenFlow controller has knowledge of the switches and links through which the requested video stream is traversing, the lowest cost path from the edge where the user requesting the channel is located to a switch in possession of the channel can be determined. This switch, referred to as the "join" switch, has its flow table modified to duplicate the stream and relay it to the edge switch along the shortest path determined. Flows are inserted along this shortest path to forward the stream hop by hop to the edge switch where the user is attached. The delay in this case consists of the time required for the flow modification and add messages to be sent from the controller to each switch involved. These flow table operations are performed in parallel, since they are done on separate switches. As such, the delay consists of the longest latency between the controller and all involved switches. After the flow tables are modified, the video stream in the requested channel must propagate from the join to the edge switch and from the edge switch to the user itself. This delay is the sum of the link latencies between the switches from the join to the edge plus the latency from the edge switch to the user.

Let J_{gc} denote the time to join a video stream using GC. $J_{gc} = T_{http} + T_{flows} + l_{egress}$, where T_{http} is the time to notify the web server of the join intent and to establish the clients connection to the assigned egress gateway, T_{flows} is the time for the OpenFlow controller to install flows for the video stream, and l_{egress} is the latency between the egress gateway and the consumer. Note that $l_{egress} = l_r$ in functionality. Below, we explore how T_{flows} can be computed from a combination of the control plane latencies between the OpenFlow controller and OpenFlow switches and the latencies of the data plane links between the OpenFlow switches.

Let L_i be the latency for a video stream to arrive to switch i in the tree from the root, where the root is switch $i = 0$. Let c_i be the control plane latency between switch i and the controller. And, let l_i be the switch link latency between switch i and switch $i + 1$.

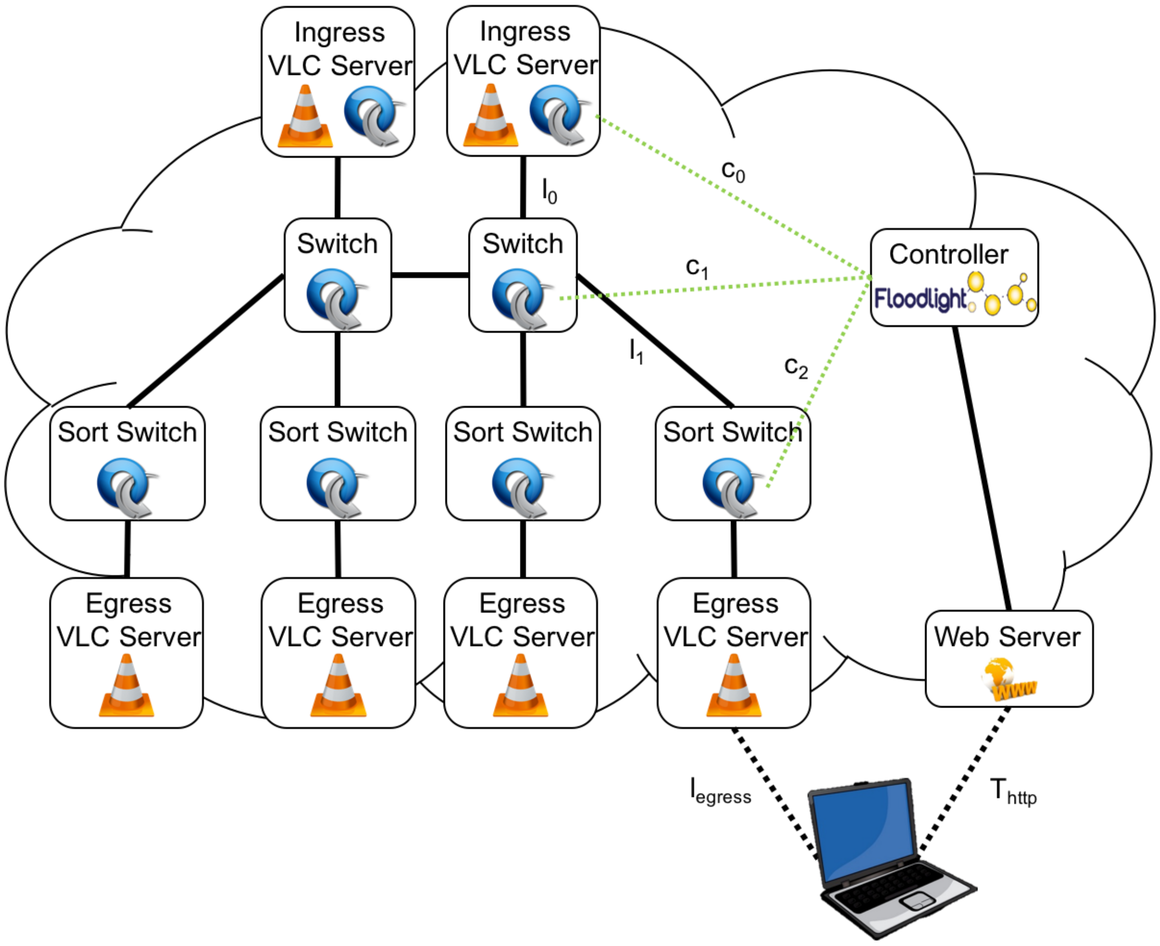


Figure 6.3.5: Components of a GC join

Consider the first case for L_0. The time for the video stream to get to switch $i = 0$ is simply the time for the controller to program that switch, c_0. This is due to the fact that the video stream is being hosted by the same machine that is running software switch $i = 0$. The second case is of course more complex. To aid in understanding, let's consider switch $i = 1$, which is the switch attached to switch $i = 0$ over link l_0. The delay for the video stream to reach this switch can be derived from the maximum of either (1) the time for the video stream to reach switch $i = 0$ and the latency of the data plane link between switches $i = 0$ and $i = 1$, namely $L_0 + l_0$, or (2) the latency of the control plane link of switch $i = 1$, c_1. Once flow mods are installed in switch $i = 0$, the video stream packets will start to traverse the link between the two switches. At the same time the flow mod is installed to switch $i = 0$, the controller concurrently sends similar flows to switch $i = 1$ to forward the video packets.

If $c_0 = c_1$, then the control plane links are equal in latency and latency for the video packets to reach switch $i = 1$ is simply $c_0 + l_0$ or $c_1 + l_0$. In general terms, this can be expressed between two adjacent switches i and $i - 1$ as $L_i = L_{i-1} + l_{i-1}$.

If $c_0 > c_1$, then the video packets will not leave switch $i = 0$ until c_0 has elapsed. The flows for switch $i = 1$ will be installed before the video packets leave switch $i = 0$, making control plane latency c_1 irrelevant. Thus, the latency for the video packets to reach switch $i = 1$ is simply $c_0 + l_0$. In general terms, this can be expressed between two adjacent switches i and $i-1$ as $L_i = L_{i-1} + l_{i-1}$

If $c_0 < c_1$, then the video packets will leave switch $i = 0$ before the flows are programmed into switch $i = 1$. If $c_1 > L_0$ or the control plane latency of switch $i = 1$, c_1, is larger than the time for the video stream to get to switch $i = 0$, L_0, then at time L_0, there is still $c_1 - L_0$ time remaining for the flows to be installed in switch $i = 1$. This time competes with the latency of the link from switch $i = 0$ to switch $i = 1$, l_0.

If $c_1 - L_0 > l_0$, then the video packets will arrive to switch $i = 1$ before the flows are programmed and will be dropped until $c_1 - L_0 - l_0$ additional time has elapsed. The total time for switch $i = 1$ to to receive the video packets is thus the time for the video stream to reach the previous switch, L_0, plus the residual control plane latency to program the flows on switch $i = 1$, $c_1 - L_0$. This can be summarized as $L_0 + c_1 - L_0$ or more simply just c_0. In general, between two adjacent switches i and $i - 1$, this can be written as $L_i = c_i$.

If $c_1 - L_0 \leq l_0$, meaning the link latency between switch $i = 0$ and $i = 1$, l_0, is greater than or equal to the residual residual control plane latency to program the flows on switch $i = 1$, $c_1 - L_0$, the flows will be programmed at switch $i = 1$ before the video packets arrive and will thus be received by switch $i = 1$ after the video packets finish traversing l_0 at time $L_0 + l_0$. More generally put, this can be written as $L_i = L_{i-1} + l_{i-1}$between any two adjacent switches i and $i - 1$.

Putting it all together, L_i can be expressed as a constant c_i for the base case where $i = 0$ given that this switch is the root of the tree. Between any two adjacent switches in the tree i and $i+1$, L_i is at least L_{i-1} plus the slowest of either the residual control plane latency for switch i, $c_i - L_{i-1}$ or the link latency between switches i and $i - 1$. This allows us to conclude with a formulation for L_i:

$$L_i = \begin{cases} c_i & i = 0 \\ max\,(L_{i-1} + l_{i-1},\ c_i) & i > 0 \end{cases}$$

Thus, $T_{flows} = L_{sort} - L_{join} + l_{sort}$, where i indicates each switch along the path from the join switch where the video stream is already present to the sort switch. An additional l_{sort} is added to account for the latency between the sort switch and the egress gateway; however, note that l_{sort} can be neglected ($l_{sort} \approx 0$) due to the pairing and close proximity of the sort switch and the egress gateway. This indicates the join time for GC is:

$$J_{gc} = T_{http} + (L_{sort} - L_{join}) + l_{egress}$$

Comparison

As is evident, the delays for case 1 and case 2 for both IP multicast and GC are similar in nature. If there is already a consumer watching a stream in an IP multicast deployment and a new consumer watches who is at the same designated router, $J_{mc} = 2l_{dr}$. Similarly for GC, if there is already a consumer watching a stream and a new user joins at the same egress gateway, $J_{gc} = T_{http} + c_{sort} + l_{egress}$. Assuming similar topologies for both GC and IP multicast, $l_{egress} = l_{dr}$. In order for GC to produce the quickest join time, $T_{http} + c_{sort} < l_{dr}$. In a real world deployment, $T_{http} + c_{sort}$will most likely be greater than the one-way latency between the designated router and the consumer. IP multicast wins in the case where greater than one consumer at the same local site wish to watch the same content and there is at least one consumer already watching the content. The join for IP multicast is limited to interaction with the local designated router, whereas GC requires interaction with the web server and controller, which is fixed overhead.

In the case of an initial join where the designated router or egress gateway does not already have the desired content, GC is more likely to outperform IP multicast. In order for GC to achieve a quicker join time, $T_{http} + L_{sort} < l_{dr} + 2\sum_{i=1}^{j} l_i + 2\sum_{m=1}^{n} l_m$. If GC control plane latencies are all less than the aggregate data plane path latency between the ingress and egress gateways / designated routers, then GC will outperform IP multicast if $T_{http} < l_{dr} + \sum_{i=1}^{j} l_i + \sum_{m=1}^{n} l_m$. In large, remote networks, this is likely to be the case. Similarly, for remote cases where the desired content is being consumed by other consumers on different egress gateways or designated routers, GC will outperform IP multicast if $T_{http} < l_{dr} + \sum_{i=1}^{j} l_i$. Although not as likely as the previous case, a nearby GC web server can have a lower T_{http} than the time for the video stream to propagate from the nearest participating router of an IP multicast deployment, especially in large networks.

As for bandwidth consumption, it is straightforward to see how the bandwidth consumed before this initial join for both IP multicast and GC is zero. During the join, GC constructs a shortest path tree from the content producer to the consumer. Subsequent consumers are attached as branches or leaves to the existing tree. On the other hand, IP multicast must construct a tree to the content source through the rendezvous point, which might not be the most efficient path. To optimize for this common scenario, after the stream has been established, the consumer will attempt to fetch the stream through a better path than the rendezvous point by routing to the producer's address. A shortest path tree with the content is constructed from the producer directly to the consumer while the rendezvous point tree still exists. Clearly, the entire rendezvous point tree, if it is not optimal in the first place, is an inefficient use of resources. Not only is bandwidth consumed, but additional control messaging must occur for the construction and decommission of the trees. From a complexity, control message overhead, and bandwidth consumption perspective, GC outperforms IP multicast for an initial join.

6.3.3.2 Leave

IP Multicast

When a user wishes to completely disconnect and stop watching video channels, in IP multicast, an IGMP leave request is sent to the user's router. This request will be received by the user's router where the router will remove the user from the multicast group; however, the router will wait until a timer to expire to tell the upstream router delivering the IP multicast group to discontinue sending the group. This time could be anywhere from 0 seconds to 5 minutes depending on when the leave was received relative to the timer expiration. The delay incurred from this leave message until the video is stopped is the latency between the user and its edge router. One of two events will occur on a leave:

1. The user was not the only user subscribed to the IP multicast group on the edge router. In this case, the algorithm is efficient, since the continued delivery of the IP multicast group data is required for other edge router users. The delay to perform this leave is simply the latency between the user and the edge router that receives the IGMP leave.
2. The user was the only user subscribed to the IP multicast group on the edge router. In this case, the edge router will continue to receive the IP multicast group even though there is not any demand for it at the users it serves. As such, there is excessive use of bandwidth in the topology from the edge router upstream to the where the IP multicast group meets a branch of the broadcast tree constructed during the IP multicast group distribution algorithm. The delay however to accomplish the leave is the same as (1) above. The timer is not included in the delay, since it does not impact the leave time from the user's perspective; the user will stop receiving the group as soon as the router receives the IGMP leave.

GENI Cinema

Upon a disconnect in GC, the user issues a HTTP post to the web server, which relays the message to the OpenFlow controller. At this point, one of two events will occur:

1. The user was not the only user subscribed to the channel on the sort switch. In this case, the sort switch where the user is attached at the corresponding egress gateway will have the user's bucket removed from the channel's OpenFlow group. This will discontinue streaming the video to the user but will retain the streams of other users watching the same channel on the edge. The time to remove the user's bucket is the latency for the HTTP post between the user and the web server, the web message from the web server to the OpenFlow controller, and the group modification message from the controller to the user's edge switch. Neglecting the

web server to OpenFlow controller latency, the latency of a leave in GC is the same as that of a join.

2. The user was the only user subscribed to the channel on the sort switch. In this case, the entire OpenFlow group will be removed from the sort switch along with all flows up to the first join in the tree detected starting from the edge up to the root. This will ensure the video stream is discontinued along all unnecessary links in the network, freeing up bandwidth for other video streams or other data. The time to accomplish the aforementioned operations includes the HTTP post from the user to the web server, the message from the web server to the OpenFlow controller, and the group delete to the user's edge router and flow deletes along the path in the tree from the join to the user's sort switch. This is the same time required to perform a join.

Comparison

Comparing the two algorithms, GC has the edge over IP multicast when it comes to the last user at an edge switch leaving a video channel. GC can quickly terminate the video stream upstream the edge router, while IP multicast must wait for a timer to expire. However, it should be noted that IP multicast has an edge over GC in this case as well. Not with regard to bandwidth utilization, but in potential control overhead and possible join delay. Consider the case when another user wants to watch the same video shortly thereafter and before the IP multicast timer expires. IP multicast will not have to locate and route the group to the edge router again, since it's still present. This results in reduced join latency and control overhead. On the other hand, GC will be required to determine the join point in the video distribution tree and insert the flows once again, maintaining the same join overhead and latency as described above.

6.3.3.3 Switch

IP Multicast

When a user wishes to switch seamlessly between one IP multicast group to another, the user must first join the new group as discussed in Section 6.3.3.1 and then leave the old group as discussed in Section 6.3.3.2. The bandwidth for both groups will be consumed at the edge router until the old group timer expires, as previously discussed. The same inefficiencies are present for a switch in the event the timer does not immediately expire.

GENI Cinema

With regard to GC, a switch from one channel to another is similar to a join followed by a leave. However, the user's bucket is removed from the old OpenFlow group and added to the new group such that only one stream is ever forwarded from the edge sort switch to the consumer. This mirrors the effect of the IGMP join followed by the IGMP leave, with the added advantage to GC that two channels destined for one user will never egress the edge sort switch. This improves upon IP multicast by reducing the bandwidth consumed between the edge sort switch/router and the consumer by the old video bitrate for one RTT between these two points. If this consumer was the last consumer watching the old channel, the same bandwidth reductions are achieved as discussed in Section 6.3.3.2.

Note that GC has another unique feature where it will utilize the same consumer sockets on the egress VLC server and on the consumer itself, thus avoiding the need to create a new socket, which requires additional overhead, and buffer the newly selected video stream.

Chapter 7

Conclusions and Future Work

This dissertation introduces three key problems relevant to data movement across computer networks. They are (1) persistent connectivity over heterogeneous wireless networks, (2) high throughput data transfer over large networks, and (3) the efficient distribution of content to users. All three problems introduce scalability concerns with respect to the network architecture. The recent strides taken by SDN due to the advent and popularity of OpenFlow provide a new way to approach these problems. This dissertation proposes and demonstrates solutions to the aforementioned problems using scalable approaches rooted in OpenFlow.

7.1 User and Application Mobility

This dissertation details a framework that leverages SDN features to enable handovers for IPv4 heterogeneous wireless networks. Based on the framework, a testbed has been designed and implemented that can be used to explore ideas related to vertical handovers. The framework testbed was implemented as a part of the GENI project which identified the need for an easily deployable method of achieving seamless vertical handovers in an IPv4 OpenFlow-enabled heterogeneous wireless environment. The framework design presented in this dissertation meets these requirements and, as a proof-of-concept, has been implemented across wireless resources at the Clemson campus. The result is an IP mobility solution that is achieved solely through the use of OpenFlow features. Analysis of the framework's ability to conduct handovers across administrative domains was also performed.

For future work, the handover project should be extended and evaluated on LTE. The handover project began when there was not a clear “winner” between WiMAX and LTE. Now that LTE has achieved nation and global adoption, the solution should be explored in an LTE network architecture.

7.2 High Throughput Data Transfer

This dissertation discusses Steroid OpenFlow Service as a solution to fill the void between the transfer of increasingly large data sets over large, long-distance networks and the end users who wish to acquire the data in a timely manner with minimal effort. By providing an automatic, transparent network service, SOS solves the well-studied TCP windowing problem through the use of OpenFlow and parallel TCP on in-network SOS agents. SOS requires zero installation, configuration, or customization of software or settings on end user machines, which makes rapid data transfer accessible to any user execution environment and any data transfer application that uses TCP.

SOS management is designed to work in production networks with SOS-assisted traffic, non-SOS-assisted traffic, and data transfers already assisted by other data transfer techniques. The SOS Floodlight controller leverages existing controller features to implement SOS in an unobtrusive manner to other and background traffic. The SOS module is equipped with a REST API to provide easy integration into other REST-based network management applications and for easy administration of the SOS deployment.

The SOS architecture can be easily, flexibly, and scalably deployed in a variety of network topologies – both on-premise and cloud-based environments. Evaluation in the GENI testbed revealed how a cloud-based SOS deployment can scale and improve data transfer rates between remote sites without access to high-performance network infrastructure. An SOS deployment on the Clemson University and University of Utah campuses demonstrated how SOS can be deployed in a real-world environment with white box switches and inexpensive desktop-grade hardware. Further evaluation in CloudLab showed that the SOS architecture can scale elastically and linearly in a datacenter networking environment with high capacity links.

Together with existing data transfer technologies, SOS is a critical piece of a the greater data transfer ecosystem. It works seamlessly with existing popular data transfer technologies, not harming what works already, while providing a boost to unassisted transfers. Non-TCP-based technologies will simply bypass use of SOS. TCP-based technologies that use a standard TCP network stack, such as parallel TCP, will operate on top of SOS – through a joint effort they can be layered to achieve a greater data transfer rate than with either component alone. For those data transfers that must be conducted using simple, contemporary TCP-based data transfers, SOS will provide improved throughput that they would otherwise not enjoy.

SOS is near production ready with respect to the controller, however, the agent application is merely a proof of concept. Work needs to be done to transform the parallel TCP implementation into a production ready application. It would also be worthwhile to design the agent application such that different data transfer technologies, such as GridFTP, Aspera, etc. can be used in a pluggable agent framework. In this sense, the network operators choose the implementation best suited to their deployment.

7.3 Efficient and Scalable Content Delivery

This dissertation explores the use of SDN as a mechanism to efficiently distribute streaming multimedia content to users at scale. The solution proposed, called GENI Cinema, combines the effectiveness of a distributed architecture proposed by content delivery networks with novel use of SDN in place of IP multicast. The result is an architecture that makes efficient use of available network bandwidth for the delivery of content to the user. It can be deployed in the cloud to ride over top of backbone networks and ISPs, eliminating the need for integration into physical forwarding devices. GENI Cinema has been deployed across the country using the GENI network for use in education and as a tool to distribute conference video content.

Going forward, GENI Cinema needs to be automated. At present, the framework requires manual configuration, which can be tedious when deploying GENI Cinema in a new environment. Much was learned during the automation of the SOS solution. These same automation techniques can be applied to GENI Cinema. Furthermore, in a real world deployment, network forwarding elements, ingress and egress gateways, and sort switches should be capable of elastic deployment. At present, these are statically configured in the GENI Cinema controller. Similar to the automation techniques used in SOS, GENI Cinema can also benefit from the same flexibility orchestrated through the GENI Cinema Floodlight module.

Bibliography

[1] P. Reinbold and O. Bonaventure, “A Comparison of IP mobility protocols,” *IEEE Symposium on Communications and Vehicular Technology*, 2001.

[2] A. G. Valkó, “Cellular IP: a new approach to Internet host mobility,” *ACM SIGCOMM computer communication review*, vol. 29, no. 1, pp. 50–65, 1999.

[3] A. Calvagna, G. Morabito, and A. Pappalardo, “WiFi mobility framework supporting GPRS roaming: design and implementation,” vol. 1. IEEE, 2003, pp. 116–120.

[4] Cisco, “Configuring Mobility Groups,” 2015, [Online; accessed 2-November-2015]. [Online]. Available: http://www.cisco.com/c/en/us/td/docs/wireless/controller/7-0MR1/configuration/guide/wlc_cg70MR1/cg_mobility.pdf

[5] C. Perkins, “IP mobility support for IPv4, revised,” Tech. Rep., 2010.

[6] A. H. Talaat, N. A. Hussein, H. A. Elsayed, and H. Elhennawy, “Real-time Traffic Performance for WiFi Handovers over MIPv4 versus MIPv6,” *SIMULATION*, vol. 10, p. 11, 2011.

[7] K. Chowdhury, B. Patil, and P. M. IPv, “Network Working Group S. Gundavelli, Ed. Request for Comments: 5213 K. Leung Category: Standards Track Cisco V. Devarapalli Wichorus,” 2008.

[8] C. Guimaraes, D. Corujo, R. L. Aguiar, F. Silva, and P. Frosi, “Empowering software defined wireless Networks through Media Independent Handover management.” IEEE, 2013, pp. 2204–2209.

[9] T. Melia, G. Bajko, S. Das, N. Golmie, and J. Zuniga, “IEEE 802.21 mobility services framework design (MSFD),” Tech. Rep., 2009.

[10] O. N. Foundation, "OpenFlow Switch Specification," 2012, [Online; accessed 2-November-2015]. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf

[11] K. Xu, "Cloud-based Strategies for Robust Connectivity and Efficient Transmission in Vehicular Networks," 2015.

[12] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. S. Wolff, "The past and future history of the Internet," *Communications of the ACM*, vol. 40, no. 2, pp. 102–108, 1997.

[13] Y.-T. Li, D. Leith, and R. N. Shorten, "Experimental evaluation of TCP protocols for high-speed networks," *Networking, IEEE/ACM Transactions on*, vol. 15, no. 5, pp. 1109–1122, 2007.

[14] M. Scharf and A. Ford, "Multipath TCP (MPTCP) application interface considerations," Tech. Rep., 2013.

[15] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet inter-domain traffic," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 75–86, 2011.

[16] S. Cass, "The age of the zettabyte Journal: the future of internet traffic is video [Dataflow]," *Spectrum, IEEE*, vol. 51, no. 3, pp. 68–68, 2014.

[17] F. T. Leighton and D. M. Lewin, "Content delivery network using edge-of-network servers for providing content delivery to a set of participating content providers," Apr. 22 2003, uS Patent 6,553,413.

[18] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the IP multicast service and architecture," *Network, IEEE*, vol. 14, no. 1, pp. 78–88, 2000.

[19] W. C. Fenner, "Internet group management protocol, version 2," Tech. Rep., 1997.

[20] S. Das, G. Parulkar, and N. McKeown, "Why OpenFlow/SDN can succeed where GMPLS failed." Optical Society of America, 2012, pp. Tu–1.

[21] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Van Reijendam, P. Weissmann, and N. McKeown, “Maturing of OpenFlow and Software-defined Networking through deployments,” *Computer Networks*, vol. 61, pp. 151–175, 2014.

[22] S. J. Vaughan-Nichols, “OpenFlow: The next generation of the network?” *Computer*, no. 8, pp. 13–15, 2011.

[23] Internet2, “Internet2,” 2015, [Online; accessed 2-November-2015]. [Online]. Available: http://www.internet2.edu

[24] ——, “Advanced Layer 2 Services,” 2015, [Online; accessed 2-November-2015]. [Online]. Available: http://www.internet2.edu/products-services/advanced-networking/layer-2-services/

[25] ——, “Advanced Layer 3 Services,” 2015, [Online; accessed 2-November-2015]. [Online]. Available: http://www.internet2.edu/products-services/advanced-networking/layer-3-services/

[26] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, “GENI: A federated testbed for innovative network experiments,” *Computer Networks*, vol. 61, pp. 5–23, 2014.

[27] R. Amin, “Towards Viable Large Scale Heterogeneous Wireless Networks,” Ph.D. dissertation, Clemson University, 2013.

[28] A. Ulvan, R. Bestak, and M. Ulvan, “The study of handover procedure in lte-based femtocell network.” IEEE, 2010, pp. 1–6.

[29] J. Hare, L. Hartung, and S. Banerjee, “Beyond deployments and testbeds: experiences with public usage on vehicular WiFi hotspots.” ACM, 2012, pp. 393–406.

[30] J. Ormont, J. Walker, S. Banerjee, A. Sridharan, M. Seshadri, and S. Machiraju, “A city-wide vehicular infrastructure for wide-area wireless experimentation.” ACM, 2008, pp. 3–10.

[31] B. Angoma, M. Erradi, Y. Benkaouz, A. Berqia, and M. C. Akalay, “A vertical handoff implementation in a real testbed,” *Mobile Computing*, vol. 1, no. 1, 2012.

[32] M. M. Uddin, A.-S. K. Pathan, S. Haseeb, and M. Ahmed, “A test-bed analysis for seamless MIPv6 handover in heterogeneous environment.” IEEE, 2011, pp. 89–94.

[33] K. Xu, R. Izard, F. Yang, K.-C. Wang, and J. Martin, "Cloud-based handoff as a service for heterogeneous vehicular networks with OpenFlow." IEEE, 2013, pp. 45–49.

[34] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," Tech. Rep., 2009.

[35] S. Floyd, J. Mahdavi, M. Mathis, and A. Romanow, "TCP selective acknowledgment options," Tech. Rep., 1996.

[36] S. Bakthavachalu, S. Bassi, X. Jianxuan, M. Labrador *et al.*, "An Additive Increase Smooth Decrease (AISD) Strategy for Data and Streaming Applications." IEEE, 2007, pp. 27–34.

[37] M. Handley, S. Floyd, J. Padhye, and J. Widmer, "TCP friendly rate control (TFRC): Protocol specification," Tech. Rep., 2002.

[38] S.-C. Tsao, Y.-C. Lai, and Y.-D. Lin, "A fast-converging tcp-equivalent window-averaging rate control scheme." IEEE, 2012, pp. 1–6.

[39] L. Xu, "Extending equation-based congestion control to high-speed and long-distance networks," *Computer Networks*, vol. 51, no. 7, pp. 1847–1859, 2007.

[40] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[41] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks," vol. 4. IEEE, 2004, pp. 2514–2524.

[42] D. Leith and R. Shorten, "H-TCP: TCP for high-speed and long-distance networks," vol. 2004, 2004.

[43] S. Ci, S. W. Turner, and H. Sharif, "An energy-efficient TCP quick timeout scheme for wireless LANs." IEEE, 2003, pp. 193–197.

[44] F. A. Feltus, J. R. B. III, J. Deng, R. S. Izard, C. A. Konger, W. B. L. III, D. Preuss, and K.-C. Wang, "The Widening Gulf between Genomics Data Generation and Consumption: A Practical Guide to Big Data Transfer Technology," *Bioinformatics and Biology Insights*, pp. 9–19, 09 2015. [Online]. Available: www.la-press.com/the-widening-gulf-between-genomics-data-generation-and-consumption-a-p-article-a5092

[45] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, no. 7453, pp. 255–260, 2013.

[46] T. J. Hacker, B. D. Athey, and B. Noble, "The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network." IEEE, 1993, pp. 10–pp.

[47] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP framework and server." IEEE Computer Society, 2005, p. 54.

[48] Aspera, "Aspera Technology," 2015, [Online; accessed 19-November-2015]. [Online]. Available: http://www.asperasoft.com

[49] "UDT: Breaking the Data Transfer Bottleneck," 2017, [Online; accessed 21-March-2017]. [Online]. Available: http://udt.sourceforge.net

[50] "Fast Data Transfer," 2017, [Online; accessed 21-March-2017]. [Online]. Available: http://monalisa.cern.ch/FDT/

[51] H.-Y. Hsieh and R. Sivakumar, "pTCP: An end-to-end transport layer protocol for striped connections." IEEE, 2002, pp. 24–33.

[52] K. Rojviboonchai and A. Hitoshi, "An evaluation of multi-path transmission control protocol (M/TCP) with robust acknowledgement schemes," *IEICE transactions on communications*, vol. 87, no. 9, pp. 2699–2707, 2004.

[53] J. R. Iyengar, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths," *Networking, IEEE/ACM Transactions on*, vol. 14, no. 5, pp. 951–964, 2006.

[54] D. Bansal and Y. Wong, "System and method for ECMP load sharing," Mar. 8 2011, uS Patent 7,903,654.

[55] D. Lu, Y. Qiao, P. Dinda, F. E. Bustamante *et al.*, "Modeling and taming parallel TCP on the wide area network." IEEE, 2005, pp. 68b–68b.

[56] M. Sandri, A. Silva, L. A. Rocha, and F. L. Verdi, "On the Benefits of Using Multipath TCP and Openflow in Shared Bottlenecks," 2015.

[57] Y. Xiao, X. Du, and J. Zhang, "Internet protocol television (IPTV): the killer application for the next-generation internet," 2007.

[58] M. R. Macedonia and D. P. Brutzman, "MBone provides audio and video across the Internet," *Computer*, vol. 27, no. 4, pp. 30–36, 1994.

[59] D. Farinacci, C. Liu, S. Deering, D. Estrin, M. Handley, V. Jacobson, L. Wei, P. Sharma, D. Thaler, and A. Helmy, "Protocol independent multicast-sparse mode (PIM-SM): Protocol specification," Tech. Rep., 1998.

[60] A. Adams, J. Nicholas, and W. Siadak, "Protocol independent multicast-dense mode (PIM-DM): Protocol specification (revised)," Tech. Rep., 2004.

[61] J. Moy, "Mospf: Analysis and experience," Tech. Rep., 1994.

[62] S. E. Deering, "Distance vector multicast routing protocol," Tech. Rep., 1988.

[63] D. Meyer and B. Fenner, "Multicast source discovery protocol (MSDP)," Tech. Rep., 2003.

[64] D. Thaler, "Border gateway multicast protocol (BGMP): Protocol specification," Tech. Rep., 2004.

[65] P. Sharma, E. Perry, and R. Malpani, "IP multicast operational network management: design, challenges, and experiences," *Network, IEEE*, vol. 17, no. 2, pp. 49–55, 2003.

[66] J. Widmer and M. Handley, *Extending equation-based congestion control to multicast applications.* ACM, 2001, vol. 31, no. 4.

[67] D. Kreutz, F. Ramos, P. Verissimo, C. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceeding of the 7th USENIX conference on Networked System Design and Implementation*, vol. 103, no. 1, pp. 14–76, 2015.

[68] M. Al-Fares, S. Radhakrishnam, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," pp. 19–19, 2010.

[69] S. Shahreza and Y. Ganjali, "FleXam: Flexible Sampling Extension for Monitoring and Security Applications in OpenFlow," *Proceedings of the second ACM SIGCOMM workshop on Hot Topics in Software Defined Networking*, pp. 167–168, 2013.

[70] C. Hyunjeong, K. Saehoon, and L. Younghee, "Centralized ARP proxy server over SDN controller to cut down ARP broadcast in large-scale data center networks," *International Conference on Information Networking (ICOIN)*, pp. 301–306, 2015.

[71] Y. Nakagawa, K. Hyoudou, and T. Shimizu, "A Management Method of IP Multicast in Overlay Networks Using OpenFlow," *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, 2012.

[72] "SDN Architecture," 2011, [Online; accessed 16-October-2014]. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf

[73] "Open Network Operating System (ONOS)," 2015, [Online; accessed 4-January-2015]. [Online]. Available: http://onosproject.org

[74] "Floodlight OpenFlow Controller," 2015, [Online; accessed 2-November-2015]. [Online]. Available: http://www.projectfloodlight.org

[75] "Open vSwitch," 2015, [Online; accessed 2-November-2015]. [Online]. Available: http://www.openvswitch.org

[76] R. Izard, A. Hodges, J. Liu, J. Martin, K.-C. Wang, and K. Xu, "An OpenFlow testbed for the evaluation of vertical handover decision algorithms in heterogeneous wireless networks." Springer, 2014, pp. 174–183.

[77] "CyberTiger," 2015, [Online; accessed 24-November-2015]. [Online]. Available: http://cybertiger.clemson.edu

[78] R. Izard, J. Deng, Q. Wang, K. Xu, and K.-C. Wang, "An agent-based framework for production software defined networks," *International Journal of Communication Networks and Distributed Systems*, vol. 17, no. 3, pp. 254–274, 2016.

[79] R. Izard, C. G. Barrineau, Q. Wang, J. Zulfiqar, and K.-C. Wang, "Steroid openflow service: A scalable, cloud-based data transfer solution." IEEE, 2106, pp. 447–452.

[80] "The OrangeFS Project," 2016, [Online; accessed 3-August-2016]. [Online]. Available: http://www.orangefs.org

[81] L. Rizzo, L. Deri, and A. Cardigliano, “10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals,” 2012.

[82] “Tcp(7) linux programmer’s manual.” [Online]. Available: http://man7.org/linux/man-pages/man7/tcp.7.html

[83] M. Ohlin and M. A. Kjaer, “Nice resource reservations in linux,” Online, 2007.

[84] W. Wu, M. Crawford, and M. Bowden, “The performance analysis of Linux networking–packet receiving,” *Computer Communications*, vol. 30, no. 5, pp. 1044–1057, 2007.

[85] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, “Comparing and evaluating epoll, select, and poll event mechanisms,” vol. 1, 2004.

[86] H. Zimmermann, “OSI reference model–The ISO model of architecture for open systems interconnection,” *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 1980.

[87] D. Beloslyudtsev, D. Bulgakov, J. Bernard, M. Cannon, E. B. Duffy, F. A. Feltus, C. Ferrier, F. Gao, C. A. Konger, B. Lee, Y. Li, K. Mace, D. Odom, B. Parker, J. Pepin, D. Preuss, R. Schwartzkopf, and K.-C. Wang, “Configuring a 100 Gbps Internet2 Connection Between Two Institutions: Practical Advice and Implications for Genomics,” 2014.

[88] A. Rosen, “Network service delivery and throughput optimization via software defined networking,” 2012.

[89] Apt. [Online]. Available: https://www.aptlab.net

[90] Q. Wang, K. Xu, R. Izard, B. Kribbs, J. Porter, K.-C. Wang, A. Prakash, and P. Ramanathan, “GENI Cinema: An SDN-Assisted Scalable Live Video Streaming Service.” IEEE, 2014, pp. 529–532.

[91] R. McGeer, M. Berman, C. Elliott, and R. Ricci, “The geni book,” 2016.

[92] R. Izard, Q. Wang, B. Kribbs, J. Porter, K.-C. Wang, S. Gupta, A. Prakash, and P. Ramanathan, “Openflow-based live video streaming with geni cinema.” IEEE, 2016, pp. 1039–1040.

[93] “VLC,” 2015, [Online; accessed 2-November-2015]. [Online]. Available: http://www.videolan.org

Appendix A

SDN Development Troubleshooting Guide

Through the development, deployment, and troubleshooting of the handover solution, SOS, and GENI Cinema, a collection of best practices and troubleshooting techniques have been learned. Issues encountered have ranged from within layer 1 all the way to layer 7. These issues consumed much research and development time and energy, and could have been prevented given a formal troubleshooting guide. Through the formal presentation of said issues, a network application troubleshooting approach has been derived for the benefit of future SDN developers, as well as network administrators of SDNs.

A.1 Physical Switch Selection

Deploying SOS or any distributed application can be challenging. Introducing SDN and OpenFlow to a network enables features that foster innovation, but it also can lead to more complex debugging and troubleshooting processes. Over the course of SOS's development, a list of useful information and techniques to make deploying SDN applications more streamlined has been compiled.

First, physical OpenFlow switches, as touched on earlier, do not necessarily implement all OpenFlow actions and matches. When procuring an OpenFlow switch to serve as the redirection point in an SOS deployment, it is important to consider the surrounding topology. If redirected packets must traverse a non-OpenFlow forwarding device, MAC learning confusion onboard the device could result in unpredictable and unexpected behavior. To avoid such situations, the physical OpenFlow switch that is performing the redirection must be capable of rewriting the MAC addresses of packet headers. If SOS is being deployed in a layer 3 network or a subnet-switched layer 2 network, the redirection switch might also need to be capable of rewriting IP addresses. These are features one should explicitly look for, if necessary, in any physical OpenFlow switch. Not only this, but one should also ensure any rewrite is performed in hardware. Some OpenFlow switches will perform matches and actions in the switch CPU (i.e. the slow path) if the hardware is not capable of performing such a match or action. SOS is designed for high throughput data transfer. Using the slow path of a hardware switch for packet match and/or rewrite could result in very slow performance, or even worse, crash the switch. In summary of this point, any physical OpenFlow switch should be meticulously inspected for the desired and necessary features (in hardware) prior to being installed. Failing to do this so can result in unnecessary headache.

A.2 Physical Switch Operating Mode

Next, when configuring any OpenFlow switch – hardware or software – it's important to note the available operating mode(s) of the switch. Many switches support OpenFlow/non-OpenFlow hybrid operation where control over the flow of traffic can be delegated to either the OpenFlow controller, the switch-local forwarding engine, or a combination of the two. In order to establish the OpenFlow versus non-OpenFlow partitions on the switch, many vendors choose to allow for multiple OpenFlow "instances" to be instantiated onboard the switch – typically up to a low power of two number of instances. (Note that not all switches support multiple instances.) These instances can be configured with an OpenFlow controller. What this controller controls is defined by how the instance attaches to the data plane.

Some vendors support VLAN instances, where any port configured on VLAN X will be under the control of the instance's OpenFlow controller. If the port is in access mode for VLAN X, the switch will tag ingress packets with VLAN X prior to delivering them to the instance's forwarding table; likewise, it will pop the tags from all packets egress this port. On the other hand, if the port is in trunk mode, only those ingress packets with VLAN tag X will be delivered to the OpenFlow instance; now, packets egress such a trunk port from an OpenFlow instance will sometimes have the tag pushed automatically if it is not already present. Other times, the controller will be responsible for pushing the tag as necessary within any installed flows. Since switch vendors vary in their implementations of this mode, it is recommended one consult the OpenFlow operating guide of the physical switch to ensure it supports the desired mode.

Another popular operating mode is port-based. In such an implementation, the OpenFlow instance is configured with a set of ports. This instance is responsible for controlling all traffic on these ports, regardless of the VLANs used.

It's common to see both VLAN and port based OpenFlow instance partitioning on physical OpenFlow switches. In VLAN mode, a single port can support both OpenFlow controlled and non-OpenFlow controlled traffic. The two sets would belong to different VLANs where one or more VLANs would go to an OpenFlow instance and the other(s) would be handled by the onboard switch forwarding logic. In VLAN mode, special care must be taken to ensure all packets are handled correctly. If care is not taken, chaos can ensue.

Take link layer discovery protocol (LLDP) as an example. Many OpenFlow controllers implement LLDP in order to discover the links in an OpenFlow network. LLDP operates on top of VLANs (i.e. is untagged), since it is traditionally a protocol used between switches, which might share many VLANs over a given link. When using physical OpenFlow switches configured in VLAN mode and an LLDP module in the OpenFlow controller, the controller will likely send untagged packet-out messages to the switch. If the switch does not push the VLAN instance's tag automatically, the LLDP packets will be sent out untagged. This isn't necessarily bad; in fact, it's as LLDP was intended. But, consider the case where another adjacent physical OpenFlow switch has an OpenFlow instance configured in VLAN mode on the same VLAN. And, for clarity, let's assume all switch ports attached to the OpenFlow instance are in trunk mode. If the first switch sends an untagged LLDP packet out its OpenFlow instance and this instance does not automatically push the tag, the untagged LLDP packet will traverse the link to the adjacent OpenFlow switch. This switch will not deliver the packet to the OpenFlow instance, since the switch is set to use VLAN X as to multiplex traffic between OpenFlow instances and its own forwarding engine. As such, the controller connected to both physical switches will not discover the link that is in fact present.

Another note about physical switch operation/configuration is that some vendors support a different kind of hybrid OpenFlow/non-OpenFlow operating mode, where an OpenFlow instance can have traffic filtered by the switch prior to being exposed to the controller. For example, some switches have a layer 2 hybrid mode where the switch will not deliver any packets to the controller that can be handled by its MAC learning implementation. This is a desirable behavior in some cases, but it can lead to confusion from the controller's perspective if the controller is expected to receive all packets. Similar implementations exist for access control list policies configured on the switch. If enabled, any rules installed on the switch might preempt any flows installed by the controller in the flow table. It is very important to keep this in mind, especially when first deploying a switch – sometimes these features are turned on by default and must be manually disabled.

Lastly, some physical OpenFlow switches give the user control over TCAM carving, and oddly enough might set the default size of controller-exposed flow tables to zero entries. If this occurs, the controller will likely receive an error message from the switch when it tried to insert a flow.

A.3 MAC Learning

It has already been discussed that with SOS, special care must be taken to ensure the physical OpenFlow switch is able to perform layer 2 source and destination rewrite if SOS is to be deployed within a non-OpenFlow network with devices that conduct their own MAC learning. This is certainly a concern and something to watch out for, but OpenFlow setups are not immune to MAC learning issues either.

Many OpenFlow controllers implement some form of a device/host manager that learns devices that are connected to the OpenFlow network. Oftentimes, as is the case with the Floodlight controller, the device attachment points observed are used for controller-based learning switch implementations when inserting flows. Information such as device MAC address and VLAN (and sometimes IP address) are recorded, as would be the case for a traditional MAC learning implementation on an Ethernet switch. If an OpenFlow network, such as an SOS deployment, is to operate on top of another network, such as on a campus or on the GENI network, then the controller might need to know about some characteristics of the network that have been or will be configured out of band from the OpenFlow controller.

VLAN translation is oftentimes used on local and perhaps more often on long-distance layer 2 links that traverse many networks under different administration. Translation is used when using the same VLAN tag across multiple networks is not feasible. If there are OpenFlow switches on each end of a VLAN-translated link that are connected to an OpenFlow controller that is learning devices and/or implementing a learning switch algorithm, the controller might learn the device on multiple VLANs. Now, a single MAC address should not exist on multiple VLANs on a single network. As such, the controller will likely think the device is moving from say VLAN A on one side of the VLAN translated link to VLAN B on the other side of the link. In such a scenario, if the controller is not "told" about the VLAN translation or does not ignore VLAN tags when learning devices, device attachment point "flapping" might occur between both VLAN A and B. This can result in instability or inaccurate device information. It can also cause controller MAC learning algorithms to fail. The Floodlight controller has the ability for the user to configure what header fields should be used to define logical networks. The simple solution for SOS on the AL2S network where VLAN translation was used was for Floodlight to ignore VLAN tags when learning devices such as clients, servers, and SOS agents.

MAC learning can also become an issue when operating an SDN over top of a traditional network infrastructure. If the logical links between OpenFlow switches consist of non-SDN-enabled (i.e. traditional) forwarding devices, the SDN might need to be aware of this to avoid MAC learning issues. GC is implemented on top of the GENI infrastructure and uses stitched Ethernet links between GENI racks. A stitched Ethernet link is essentially a logical layer 2 link between two GENI racks, which may consist of multiple traditional forwarding device hops in the AL2S network. Stitching programs the traditional forwarding devices along the path of the stitched link, typically performing VLAN forwarding and/or translation where necessary to make the link appear as a layer 2 link between the devices reserved by the GENI experimenter at each rack. When deploying an SDN application with multiple stitched links, such as GC, the AL2S aggregate manager might allocate and reserve link paths through the same physcial forwarding devices. If the SDN application sends a packet with the same source MAC address along multiple stitched links that traverse at least one common forwarding device, that device's MAC learning algorithm (if not disabled) will become "confused" after observing the given source MAC on more than one VLAN. The typical behavior of a traditional forwarding device when it sees the same MAC on mulitiple VLANs is to allow the MAC to "flap" a few times between the VLANs but if the "flapping" occurs too frequently, shut down all but one of the impacted VLANs. Such a feature is often referred to as loop protection. This problem occurred a few times in GC, since a video stream is flooded from an ingress VLC server to all subscribed egress VLC servers. The paths between the ingress and egress VLC servers consisted of stitched links. The solution to the problem was to rewrite the source MAC addresses of the stream packets to some link-unique MAC address prior to sending the packet across a given stitched link. Without this workaround solution, stitched links will appear to randomly shut down and cease to forward GC traffic until a timeout of approximately 15 minutes occurred. This is simply another type of MAC learning issue that can occur when deploying SDN applications on a traditional underlay network. The problem and solution required a fundamental understanding of how the traditional networking and SDN elements worked together.

A.4 MTU

It is a common network engineering task to configure link and end host MTUs. However, it is relatively uncommon for SDN developers to be concerned with MTU, as MTU is more of a link layer configuration and SDN manipulates the data that is already in transit. Furthermore, many SDN use cases do not require or experiment with MTUs above 1500 bytes, where most network forwarding devices are configured by default to support such MTUs.

One must take care to confirm and/or configure the MTU of each switch port and end host if jumbo frames are to be used. Jumbo frames can be and are recommended in a layer 2 SOS deployment. Checking for consistent MTU values is not enough in all cases. Different vendors have different definitions for what "MTU" means. Of course, the definition is "maximum transfer unit," but the "unit" is sometimes inferred as layer 2, other times as layer 3. One should consult switch documentation for the proper definition and configuration of MTUs.

If using a software switch, MTU is most likely defined as IP (layer 3) MTU, such as the MTU value exposed via ifconfig or ip link in Linux. This MTU must be set keeping in mind the requirements of layer 2 below. This might seem to break the convention of the OSI model by crossing layers; however, it is required to properly set the MTU, as the link layer is what will carry the network layer as its payload.

Consider an example where it is desired to configure an end host for a network that supports jumbo frames on a VLAN-tagged interface. End host network card IP MTU is commonly restricted to 9000 bytes. Assuming this is true (must check specific network card limitations) and assuming the layer 2 network supports such an IP MTU, then an IP payload can be no larger than 9000 bytes. Configuring 9000 bytes as the IP MTU will likely result in a problem though. Consider the minimum defined Ethernet frame size of 64 bytes. If one is to use IPv4 and TCP, then the minimum possible frame size (e.g TCP ACK) would be 6 bytes source MAC + 6 bytes destination MAC + 2 bytes ethertype + 4 bytes VLAN tag + 20 bytes IPv4 header + 20 bytes TCP header, which sums to 58 bytes. This is of course less than the 64-byte minimum frame size. As such, the network stack will append an extra 6 bytes of padding to bring the total frame size to 64 bytes. This implies that the IP MTU should actually be 9000 bytes minus 20 bytes of TCP header minus another 6 bytes of padding, which lowers the end host MTU to 8974 bytes.

On a final note, MTU should also be considered when using tunnels. SOS, GC, and the handover project have been tested in GENI over GRE tunnels between GENI aggregates. The IP MTU of the end hosts and agents was lowered to 1410 bytes in order to accommodate the extra headers involved when transmitting the TCP packets through the tunnel.

A.5 Linux

When using Linux as a platform to implement software defined networking solutions, care must be taken to ensure Linux itself is not inadvertently impacting performance. Things to watch out for are firewall rules, traffic control queues, and NIC queues.

A.5.1 Firewall Rules

In Linux, firewall rules are implemented in the form of IP table and EB table rules. Each of these can be used to filter and rewrite network traffic based on match criteria or certain trigger conditions. Depending on the Linux distribution or your privileges on the system, there might be rules in place that could impact traffic on the SDN. It is important to check these rules when debugging and deploying an SDN that uses Linux to run components of the SDN in the data plane.

A.5.2 Traffic Control Queues

Traffic control queues are implemented with tc qdisc policies. Although this is a critical networking component of the Linux kernel, many sysadmin and developers are unaware of this functionality. What it does is enable user-defined queuing policies on NICs, which can be used to implement quality of service on a Linux box.

On GENI and CloudLab, tc qdisc is used to enforce per-experiment bandwidth limits. Many experimenters do not know they can lift these restrictions themselves. They incorrectly assume, as I did myself for quite some time, that per-experiment network bandwidth restrictions were enforced using the hypervisor, physical, and software switches in the GENI/CloudLab infrastructure. When performing high-bandwidth experiments that go beyond the limits imposed, one can coordinate with other experimenters, GENI, and CloudLab to (politely and carefully) remove such limitations to use more available bandwidth. This was especially useful for SOS, where machines were limited by tc qdisc to 20Mbps and 100Mbps by default on GENI and CloudLab, respectively.

Another place queues are commonly used is with OVS GRE tunnels. If a tunnel is capable of higher bandwidths yet you notice you seem to be artificially throttled (i.e. machine resource utilization is low), check tc qdisc for any policies installed on the physical or virtual interfaces associated with the OVS tunnel.

A.5.3 NIC Queues

Similar to traffic control queues, NIC queues can impact the performance of SDN applications. NIC queues are configured by the Linux OS as a means to buffer bursts of packets destined for the NIC. Care must be taken to set this appropriately for your network type and SDN application using the network. The Linux kernel sets the txqueuelen to 1000 by default, however note that this means 1000 extra frames will be buffered when the NIC's buffers become saturated. For applications like TCP that have feedback loops to determine their transmit rate, this extra buffer means a packet will take longer to leave the host en route to the destination upon link congestion. TCP might retransmit thinking the packet has been dropped when in fact it has been buffered in an excessively long buffer.

It is important to experiment with txqueuelen if you notice many transmit drops or issues with throughput. Increasing this value will mitigate transmit drops, but it might put an artificial ceiling on TCP window sizes due to increased transmit latency. Note that high transmit drops might indicate that the application is generating packets faster than the NIC can transmit them either due to network congestion or NIC speed.

A.6 CPU Affinity and NUMA Architectures

In NUMA architectures, it is imperative to consider where the software and hardware data paths meet between the application and the network. When receiving data from the network, most modern NICs will generate a hard interrupt to start the transfer of received frames from the NIC to memory via DMA. This happens in most platforms over the PCI bus. In NUMA architectures, the PCI bus is often split between the two CPUs with particular lanes wired to each. If the NIC transfers frames over PCI to memory for an application running on a different CPU socket than the PCI lanes are wired to, it will requires the CPUs retransmit the data again using the front side or local bus between the CPUs – DMI or QPI for modern Intel architectures.

Although this is fast, it can quickly become the bottleneck and requires unnecessary utilization of cycles from both CPUs. With proper planning, the interrupt service routine for the NIC, the NIC's PCI slot (and thus lanes), and CPU core(s) to which the application is pinned will all be associated with the same CPU socket. This will eliminate any data transfer over the local bus, reduce latency, and eliminate it as a potential bottleneck. With SOS, a non-trivial 1-2Gbps performance improvement was routinely observed by simply taking the time to carefully pin NIC interrupts and the SOS agent applications to the same socket.

A.7 Underlay and Network Equipment Problems

A.7.1 Faulty Hardware

When experimenting with SDN and OpenFlow on a network, the first thing questioned when something goes awry is the controller. This is a good assumption, since controllers are software-defined and are inherently more prone to error than traditional hardware switching devices. This is especially true during the development and debugging phase on an SDN project. However, as was learned through the development of SOS, traditional network can fail as well. It is very important to keep this in mind when debugging challenging SDN problems.

Best explained through a short narration, one day SOS was running great over an AL2S deployment; the next day, SOS was no longer working. The symptoms were:

- A trunk port disappeared on a physical switch VLAN-delineated OpenFlow instance. This was due to a power failure and user error in committing recent changes to the switch's memory.

Easily fixed.

- All frames of any size and layer 3/layer 4 type traverse AL2S link in one direction
- The other direction:
  - Pings with a payload over 18 bytes are lost; less than 18 byes succeed
  - TCP packets with don't fragment (DF) flag set, even ACKs, are lost
  - UDP and TCP without DF flag set are truncated by 192 bytes, regardless of actual payload size
- No errors or drop reports on any switches, both OpenFlow and non-OpenFlow

Unfortunately, the timing was a week or two after rolling out a new code revision in the controller, so all energy was channeled into debugging the controller. The first symptom, as noted, was easily fixed and was the combination of a power failure and not saving a permanent change to the switch's configuration. The other problems though were not as easy to solve. In fact, after two weeks of discussion with IT at Clemson University and the other endpoint at the University of Utah, the root cause was at last determined to be a module failure on the main 100Gbps aggregation services router at Utah. Overnight maintenance of the router solved all problems.

Although this issue was ultimately unrelated to an SDN aspect of SOS, it is a valuable experience worth sharing, since precious time was devoted to trying to find a problem in the controller and no consideration was initially given to a non-OpenFlow component failure in the network. A similar, less time-consuming situation occurred at the Clemson side of the network a year prior, where one of the 10Gbps twinax cables failed connecting an SOS agent to the physical OpenFlow switch. The moral of these stories is to always explore each possibility and never underestimate or discount anything, especially in the physical network layer on which SDN depends for its operation.

A.7.2 Buggy Software

Similar to the faulty hardware case above, it cannot be assumed the software chosen to implement an SDN is perfect. Consider the release cycle for product hardware and software, network switches in particular. When a new switch is released, it consists of the switching hardware along with a bundled OS called the firmware. Historically, each hardware platform has many firmware revisions that are released for it over its lifetime. Each subsequent firmware release might add new features driven by consumer demand. However, an often overlooked purpose of firmware revisions is to provide a mechanism for the company to address bugs discovered in the field, in real-world consumer deployments. Unfortunate as it may be, there was a bug that plagued SOS deployments, on and off, for years. In the end, it turned out to be caused by a firmware bug in our chosen Dell OpenFlow switches running their Force10 OS (FTOS). Since then, a ticket was opened and resolved with Dell to patch the issue for future firmware releases.

The symptoms of the problem were straightforward: ping (and other IP traffic) would sometimes not be forwarded across the network. The most troubling part of this statement is "sometimes". A bug that is not easily reproducible or inconsistent is very difficult to solve in many cases.

A.7.2.1 Increase Available Information

To discover the root cause of the problem, control plane monitoring was performed at the Dell switches themselves and at the controller, along with data plane monitoring via tcpdump at the edge hosts. This allowed us to gather information about how the switch thought it was communicating with the controller, how the controller thought it was communicating with the switch, and what packets the hosts were generating and receiving. One would think that the controller and switch debug logs would indicate the same information, but depending on where the logs are implemented, events might occur that are not captured by one but visible on the other.

A.7.2.2 Narrow Down the Problem

In an SDN, flows can be installed proactively or relatively. In the case of the former, all flows are pre-installed in the data plane, reducing the number of packet-in messages the controller must process. Installing flows proactively is a systematic way to eliminate the data plane as the source of the problem and possibly indicate a control plane problem.

For Dell OpenFlow switches, ARP requests and replies cannot be processed in the data plane with flows, nor can ARP opcode be matched by a flow. Thus, to effectively use them in a proactive setup, IP flows can be pushed to the switches, and packet-in followed by a flooded packet-out for all ARP packets can be performed to achieve host-to-host IP connectivity. Alternatively, static ARP entires can be installed on the hosts, as discussed later in this section.

In this proactive setup, it was noticed that the IP flow counters never increased, indicating they never matched an IP packet from the ping. This indicated that the problem might be related to ARP, as further evidenced by continuous ARP requests in the source hosts's tcpdump. So, analyzing the ARP packet-in and packet-out messages revealed that the first hop Dell switch sent an ARP request packet-in, the controller received the packet-in, generated a flood packet-out, and relayed it to the switch, where it was also seen. The switch then flooded the packet-out, or so it claimed. The adjacent hosts on this switch saw the flooded ARP request packet in their tcpdump logs, indicating the flood was a success. Furthermore, adjacent Dell and other switches received the ARP request and generated packet-in messages of their own in a similar workflow described above.

However, there was one adjacent Dell switch en route to the intended destination host of the ARP request that did not appear to see the ARP request packet after the first hop switch flooded it. This indicated that (1) the ARP request never made it to the switch, meaning the link itself dropped the packet, (2) the receiving switch decided to not send the ARP request as a packet-in to the controller, or (3) the sending switch did not actually send the ARP request out the port of the link between the two switches.

Although it was a possibility, (1) was quickly ruled out as the source of the problem, since LLDP used by the controller to discover links was able to consistently be sent as a packet-out, propagate across the link, and was consistently received as a packet-in at the adjacent switch. To examine (2) and (3), further digging was required. For (2), the receiving switch logs did not indicate the packet-in was received. However, there is no indication where in the packet processing pipeline in the switch OpenFlow agent these messages are displayed. So, we could not rule out the possibility that the receiving switch was dropping the ARP requests before it could generate a packet-in. For (3), switch logs indicated the flooded ARP packet-out was sent out the port of the adjacent switch. But, for the same reasons cited for (2), we could not assume this log was accurate. The most certain way to was place a tap in the link to monitor the packets that traversed it. This revealed that (3) was actually the culprit, and contrary to the sending switch logs, the ARP request packet-out message was not actually being sent on the link.

This posed the questions: Why was LLDP (also from a packet-out on the source switch) being sent out the link, but ARP was not? Were there other types of packets that could be sent as packet-outs, such as IPv4? To test these questions, static ARP entries were installed in the hosts involved in the ping, and the controller was changed back to reactive flow installation (to force IPv4 packet-in and packet-out messages). This test resulted in a ping that worked, and the IPv4 packets were sent from the packet-out messages.

However, recall that ARP request packet-outs were seen on some of the devices adjacent to the switch. The question then became, what was special about this port or link where the ARP request packet-out did not get sent but the IPv4 and LLDP did? To answer this, the switch's OpenFlow instance config was consulted, and it revealed that this particular link was a port channel link, while the others that worked were 10GbE, 40GbE, and 100GbE. Given that there was no indication in the customer support documentation that port channel links were not compatible with OpenFlow, a support ticket was opened with Dell, and after half a year of additional debugging, they were able to reproduce the problem and formulate a solution in the form of a firmware update.

A.7.2.3 Lessons Learned

The reason this bug was so hard to track down was: (1) We incorrectly and consistently kept assuming it was a software bug on our controller or in our flows. (2) It did not occur on all Dell OpenFlow instances on a given switch, although it was eventually determined to impact every OpenFlow instance fairly consistently except OpenFlow instance #2 (in most cases, although it still did arise infrequently). This instance #2 data was ultimately valuable information that led Dell to be able to reproduce the issue and determine the bug fix. (3) Occasionally, every few minutes, the port channel link would send the ARP request packet-out, and the ping would then work. But after the ARP cache entries expired, the bug would surface again. This generated much churn in the network in the form of device relearning on the controller, flow removal and reinstallation, and made it very difficult to find the source of the problem and further led us astray to the controller as the source of the problem.

In short, the lesson learned was that one cannot rely on third party software to work, even paid solutions from a reputable company. Bugs and mistakes happen in the real world, and the investigation of the underlying SDN-enabling technology should always be a "checkbox item" to consider when debugging SDN and general networking problems. It's worth repeating that Dell did come through with a solution to the problem, and we extend great thanks to their support engineers for quickly releasing a patch.