1-1-2012

NETWORK SERVICE DELIVERY AND THROUGHPUT OPTIMIZATION VIA SOFTWARE DEFINED NETWORKING

Aaron Rosen
Clemson University, arosen@clemson.edu

NETWORK SERVICE DELIVERY AND THROUGHPUT OPTIMIZATION VIA SOFTWARE DEFINED NETWORKING

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Aaron O. Rosen
May 2012

Accepted by:
Dr. Kuang-Ching Wang, Committee Chair
Dr. Harlan B. Russell
Dr. Sebastien Goasguen

Abstract

In today's world, transmitting data across large bandwidth-delay product (BDP) networks requires special configuration on end users' machines in order to be done efficiently. This added level of complexity creates extra cost and is usually overlooked by users unknowledgeable to the issues. This is one example problem which can be ameliorated with the emerging software defined networking (SDN) paradigm. In an SDN, packet forwarding is controlled via software controllers. In an OpenFlow SDN, a controller can control the forwarding, rewriting, and dropping of packets based on their header attributes. The ability to handle packets in customizable ways in software has significant implications for both users and operators of the network.

Via SDN, network providers can easily provide services to enhance users' experience of the network. Steroid OpenFlow Service (SOS) is presented as a solution to seamless enhancement of TCP data transfer throughput over large BDP networks without any modification to the software and configurations on users' machines. SOS utilizes OpenFlow to redirect application specific traffic to application specific service agents. SOS uses service agents on both ends of the connection to seamlessly terminate a user's TCP connection, launch a set of parallel TCP connections, and leverage multiple paths when available to maximize throughput.

Acknowledgments

I would first like to thank my advisor, Dr. Kuang-Ching Wang, for his helpful guidance and valuable knowledge during my time at Clemson. Dr. Wang was always very supportive in sharing his experiences and provided valuable guidance to direct me in the right direction. In addition I would like to thank Dr. Russell and Dr. Goasguen for providing additional insight and guidance while working on my thesis. I would like to especially thank Ke Xu and Nick Watts for letting me bounce my ideas and theories off them and for providing valuable discussions. Also, I would like to thank my friends and family for their love and support. Lastly, I'd like to thank the GENI Project Office for providing support in running my experiments on GENI and NSF for sponsoring my research efforts.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In this work Software Defined Networking (SDN) is leveraged as a solution for seamless throughput over large bandwidth-delay product (BDP) networks. As computer networks continue to expand across the globe, data needs to travel a farther distance between endpoints, resulting in higher round trip time (RTT) and packet loss. Transmission Control Protocol (TCP) is the de facto protocol that provides reliable and ordered delivery across the Internet. Unfortunately, TCP's performance degrades severely in situations where RTT and packet loss are high [1]. Much work has gone into improving TCP's efficiency over large BDP networks, though in most cases, special manual configuration and software is required in order to achieve good results [2]. Since most end users of TCP are not knowledgeable about these issues at hand, TCP over large BDP networks continues to suffer.

Software defined networking (SDN) is a network architecture which consists of separating the control and data plane of an arbitrary set of Ethernet switches [3]. This allows the ability for the control plane to manipulate how the data plane forwards packets via a software controller. In an OpenFlow SDN, a software controller has the ability to easily make forwarding decisions and can even instruct the data plane to perform packet header rewrite on specific flows of packets. This ability allows service providers to seamlessly deploy new services

programmatically to improve user experiences while remaining compatible with legacy protocols and network services [4].

As mentioned earlier, lots of work has been done on improving throughput over large BDP networks, but unfortunately much of this work is not widely taken advantage of. The reason for this is that these improvements generally require tedious modifications to end hosts in addition to use of special software [5]. These modifications in turn create additional complexity and costs in training end users and deploying special software to each user's machine. Through the use of SDN, these costs and complexities can be eliminated and services can be seamlessly provided to clients. This is achieved by using SDN to forward connections from a user of a given traffic type to a traffic specific performance enhancing software agent. This ability eliminates the cost and complexities for end users and allows service providers more control over their networks.

1.2 Motivation

In today's world, people often find themselves transmitting data over networks that span multiple continents. For example, due to economic reasons many companies are centralizing their data centers to one location. As a result, clients have to send packets a farther distance, resulting in higher RTT and packet loss, which TCP is quite sensitive to [6]. Typical solutions to this problem require special configuration and software in order to move data quickly. One example where this is being done is in scientific projects like the Large Hadron Collider (LHC), which generates petabytes of data each year. In order to efficiently move this amount of data across the world for researches to access, special software and hardware infrastructure such as

the Globus GridFTP [7] is needed. The need for specialized software and training in order to efficiently move data adds complexity and cost for end users.

The motivation for this work is to eliminate the complexities and costs from the users to a service provided by network providers. In order to accomplish this, special software agents on middleware boxes and software defined networking (SDN) are used together to deploy and develop new services seamlessly over the existing infrastructure. In this work, Steroid OpenFlow Service (SOS) is presented as such a service to improve TCP throughput over large BDP networks. SOS does this through the use of OpenFlow, an emerging SDN solution. The OpenFlow protocol defines a standardized interface for software controllers to communicate with and control the data plane of an arbitrary set of Ethernet switches [3]. SOS leverages OpenFlow switches in the network in order to detect long range TCP connections and redirects them to a special software agent along the path. The software agent first terminates a user's TCP connection using header rewrite and then launches an optimized transport protocol connection to another counterpart agent near the end host. The end agent reverses the process and restores the TCP connection to the intended destination. This process is completely transparent to the end hosts except the noticeable increased throughput. SOS therefore removes the added complexities and costs from end users in order to efficiently move data.

Chapter 2

Background and Related Work

In the past, a great deal of work related to improving end-to-end throughput over large BDP networks has been done. These works range from protocol optimization, compression, caching, and use of special agents along the forwarding path. In this section, a brief overview of these past works is given. Although this thesis does discuss means for performance improvements the main focus is the ability to decouple users transport protocol from the network via Software Defined Networking.

2.1 Software Defined Networking

Software defined networking (SDN) is a network architecture in which networking devices can be controlled by software outside the devices to manipulate their packet handling actions. OpenFlow is a communication protocol for software controllers to communicate with Ethernet switches (control plane communication) about their packet handling rules (data plane actions). The idea behind OpenFlow is that vendors implement the OpenFlow protocol on their devices, which allow researchers to program the devices via software. Allowing researchers to program Ethernet switches via software greatly lowers the barrier to try out and implement new ideas at a large scale [9].

OpenFlow is a centralized paradigm where switches (also referred to as datapaths) connect to a controller that tells them how to handle packets. OpenFlow uses a flow table within the

switch to match packets. This flow table allows packets to be matched based on their L2 (dl_src, dl_dst, dl_type), L3 (nw_src, nw_dst, nw_proto), and L4 (tp_src, tp_dst) headers, in addition to the physical port the packet arrived on. When a packet enters an OpenFlow switch, the switch first checks to see if the packet matches an existing flow entry. If it does not, the packet is forwarded to the controller via the control plane which is referred to as a packet_in.

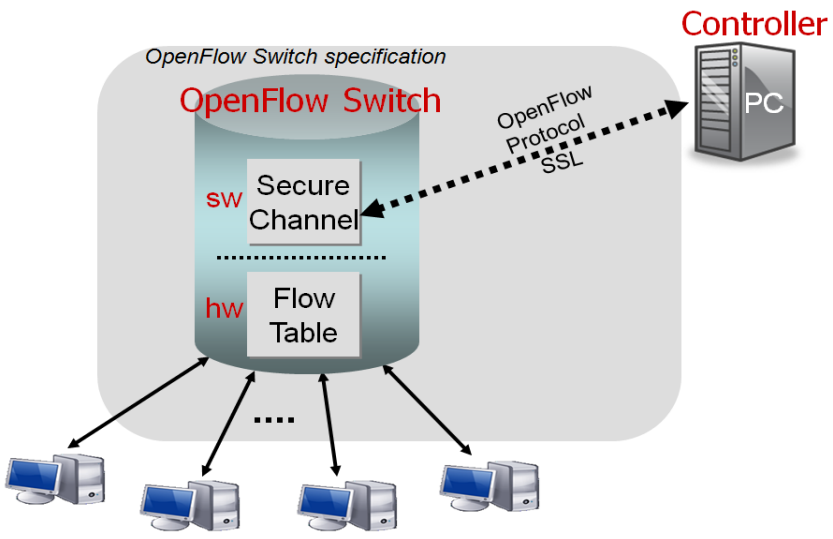

Figure 1 OpenFlow Network Architecture [3]

The controller now has the ability to tell the switch how to handle these types of packets via flow entry. Associated with each flow entry is an action that can range from dropping, outputting to port(s), and rewriting packet headers (via mod_*field* action). Once the controller installs a flow entry on the switch, packets of this match are no longer sent to the controller and are handled at the switch directly at line rate [9]. Flow entry details shown in Figure 2.

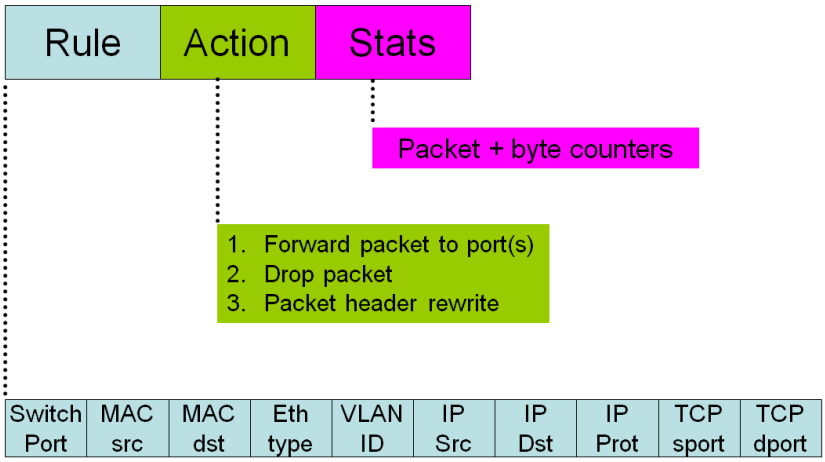

Figure 2 OpenFlow Flow Entry [3]

OpenFlow allows multiple researchers to try out different forwarding ideas via slicing. Slicing can be done via a special controller called FlowVisor. Researchers inform FlowVisor the subset of traffic they intend to control by specifying the L2, L3, L4 headers and/or physical port numbers; FlowVisor then transparently sends traffic matching that type to the specific researcher's own software controller [18]. This allows multiple researchers to try out ideas at once without requiring their own hardware or taking turns. In addition, it now allows the production network to become the test bed. For example, on the same network a production slice can be created to handle normal traffic and an experimental slice can be created to allow a researcher to test out new ideas without risking the stability of the network.

Steroid OpenFlow Service (SOS) presented in section 3 makes use of OpenFlow to provide a seamless service for clients to increase TCP throughput over large BDP networks. This is done by making use of parallel TCP. Currently SOS does not implement any type of other optimizations such as caching and compression which is later described. The main focus was to demonstrate the seamless ability to decouple a users' transport protocol from a network

providers' choice of protocol. That said, caching and compression are things that could be easily added to SOS.

2.2 Transmission Control Protocol

TCP is the de facto protocol for transmitting data reliably between two endpoints. Since the available bandwidth in a network can change at any time due to others using the network, TCP tries to be fair and share the bandwidth in the network. TCP does this through the use of its congestion control algorithm. The standard TCP congestion control algorithm works via a windowing mechanism. The sender maintains a congestion window (*cwnd*), in order to keep track of the amount of data that can be sent into the network before it has to stop to wait for an acknowledgement [10]. In addition, the receiver maintains a receiver window (*rwnd*), which advertises the amount of data that he is willing to receive. Through *rwnd*, the receiver is able to throttle the sender's data rate by modifying this value. The sender increments his *cwnd* by *1/cwnd* for each received acknowledgement, which is considered a sign that there is more available bandwidth in the network to use. Likewise, when a TCP sender does not receive an acknowledgement for a sent packet, it considers it a sign of congestion and reduces its *cwnd* to *cwnd/2.*

TCP's performance degrades as the RTT between endpoints increases [2]. The reason for this is that TCP's window growth algorithm is based on RTT [8]. As a result, with high RTT the *cwnd* growth is much slower than in low RTT settings. When congestion in the network is detected, the *cwnd* is halved and then must start the growth process again. One TCP variant that tries to solve this issue is Scalable TCP. Scalable TCP modifies the congestion control algorithm in order to be more aggressive. It does this by increasing *cwnd* by 0.01 instead of *1/cwnd* for each

acknowledgement. This change makes the *cwnd* growth only proportional to RTT instead of both RTT and *cwnd* such as in traditional TCP. In Scalable TCP, when congestion is detected, instead of halving the *cwnd* it takes a more aggressive approach by decreasing *cwnd* by *0.125*cwnd*. This keeps the *cwnd* at a higher value.

In addition to Scalable TCP, there are many other TCP variants such as BIC-TCP and CUBIC. Both BIC-TCP and CUBIC use a logarithmic convex function in order to grow the *cwnd* quickly when there is no congestion [10]. When congestion is detected, BIC-TCP performs a binary search which selects a value for *cwnd* that is a midpoint between where packet loss occurred and the last *cwnd* that did not have packet loss. Then, *cwnd* growth resumes using a slow logarithmic concave function. This method allows BIC-TCP to be quite stable, though it is slow to react to changes in available bandwidth after a packet loss event. This is due to the fact that it solely uses a concave function to increase *cwnd* after a congestion period.

CUBIC is currently the default TCP algorithm used by the Linux kernel (version 2.6.18), replacing BIC-TCP [10]. CUBIC takes a different approach in handling congestion. In CUBIC, when a loss occurs, *cwnd* is divided by a set factor (similar to standard TCP). Then, *cwnd* is grown using a convex function up until the point where packet loss had previously occurred. After reaching that value, CUBIC then switches to a concave window growth. This method allows for faster recovery than BIC-TCP in addition to maintaining high utilization.

2.3 Parallel TCP

Parallel TCP is a widely used technique in order to achieve high throughput over large BDP networks. Globus GridFTP uses parallel TCP connections in order to move data quickly

between endpoints [11]. Parallel TCP works by stripping data across multiple TCP connections and then reassembling the data at the endpoint. Parallel TCP helps to improve TCP's performance by improving window growth relating to RTT, recovery from packet loss, and window size limits. As mentioned before, TCP increases its sending window size for each acknowledgment received. A larger RTT takes a TCP connection more time to grow. In parallel TCP, multiple TCP connections are used, thus allowing the ability to exploit the growth rate by a factor of the number of connections used [12]. This is extremely beneficial in scenarios where high packet loss is a factor.

Besides growth rate, TCP often suffers from a lack of buffer space to hold the unacknowledged packets in the network. This limits the window size and in turn limits the ability to achieve high throughput in large BDP networks. This buffer size is a parameter set by the operating system which limits the amount of memory allowed for each connection. Using multiple TCP connections allows an application to circumvent the window limit since the limit is only applied to each connection and not the aggregate of them. The down side of parallel TCP is that one must implement the logic for it within the application which adds complexity for the application developers [13].

2.4 Caching

Caching is an approach to speed up response time and reduce the amount of traffic needed to be sent across a network. This is done by storing frequently accessed data at locations closer to the end host. As a result, performance can be improved for clients in addition to reducing network utilization to end servers [14]. Caching is usually implemented using one of the following strategies: hierarchical, distributed, and hybrid.

Hierarchical caching is generally implemented with four caching levels: bottom, institutional, regional, and national. In this model, when requests are made, they are directed through each level of caching starting from the bottom until the result is found. Once the document is found, the reply travels back down the chain, leaving a copy at each level. Therefore, the next time the document is requested, the request will not have to travel all the way to the national provider in order to retrieve the document. In the distributed model, it uses institutional level caches to exchange metadata about the location of documents rather than using a hierarchical chain to retrieve the document. Hybrid caching works similar to distributed caching except it is slightly smarter in the way that documents are fetched from caches. For example, it uses the lowest RTT location to retrieve the document. This helps to improve performance by avoiding fetching documents from farther caches when closer are available. This being said, caching has several apparent downsides. One key downside is that a proxy may serve stale data as a result of a document change at the root level [15].

2.5 Compression

Compression is a technique used in order to reduce the size of data by removing redundancy. Reducing the size of data needed to be sent across the link helps to reduce congestion since the link does not need to be utilized for as long [2]. Compression can be done offline before sending the data or on the fly when sending the data. There are several key components that affect the performance of data compression such as the type of data being sent, the compression method, and the amount of system resources devoted to compression. Data compression requires the CPU to perform a lot of work in order to compress the data. In

cases where the CPU is very busy compressing the data before sending, it might be even slower than just transmitting the data.

In [16], the authors proposed and implemented a dynamic compression scheme that was able to send compressed and uncompressed data at the same time. The scheme tries to use up all of the bandwidth by sending as much compressed data as they can. In cases when it could not compress data fast enough, sends uncompressed data to use the remainder of the available bandwidth. Via this process, it was shown that this dynamic approach provides significant gains in effective throughput.

2.6 Special Agents

There has been a lot of work in using special agents to help facilitate and/or provide network services. The Control for High-Throughput Adaptive Resilient Transport (CHART) project is one example that uses special agents to provide network quality of service (QoS) and accelerate throughput via special agents. CHART works by configuring clients to proxy their connections though CHART agents. Then, CHART is able to determine the available bandwidth in the network by sensing port utilization of network devices. Using that information, signaling can be done in order to speed up or slow down connections through CHART. A connection through CHART uses a special TCP implementation called the TCP-Trinity protocol stack which bypasses slow-start and congestion avoidance phases of TCP. Via the Trinity protocol stack, it allows TCP connections over large BDP networks to perform better by avoiding the congestion avoidance phase. In addition, CHART is able to schedule multiple connections over these links so they do not compete with each other similar to RSVP [5].

In [17], a different approach for improving end-to-end TCP performance was taken. In this work, modifications were made to routers along the path between the end hosts. This allowed their service to provide TCP acceleration transparently to end hosts (unlike in CHART which required proxy configuration on end hosts). In order to perform TCP acceleration, they used a split TCP approach which toke a TCP connection and terminated it at each router and then starts a new connection to the next router. Splitting the TCP connection into multiple TCP connections helps control congestion at each segment, in addition to reducing the BDP for each segment allowing each segment to obtain a higher *cwnd*, which in turn improves throughput.

Common Internet File System (CIFS) is a protocol developed by Microsoft in order to share files between different remote systems. When designed, it was intended to be used within a local area network where latency is low and bandwidth is high. Clearly, the assumption is no longer valid when people need to remotely access files when they are not within the local area network. In order to solve this problem, CIFS acceleration agents were developed. CIFS does not work well over large BDP networks due to its chattiness: when CIFS sends a request, it waits for the response before sending out another request. In order to speed up this process many CIFS accelerators have been developed using agents to respond to requests. Agents are able to predict the series of CIFS calls that are going to be made from the first request, and transmit the responses and additional requests ahead of time to speed up the process. In addition to this, some CIFS accelerators support write-back and read ahead in which the agents get the data ahead of time and store it locally (caching) in order to reduce transmission time [2].

Chapter 3

Steroid OpenFlow Service

Steroid OpenFlow Service (SOS) is a service that operates completely behind the scenes to improve TCP throughput over large BDP networks. SOS is an example of seamlessly decoupling end users' choice of protocols from network providers' choice of protocols. In order for SOS to do this, it needs to be able to manipulate the flow of traffic within the network. This is done via a software controller that speaks the OpenFlow protocol to Ethernet switches (i.e, datapaths) in the network. The SOS OpenFlow controller is responsible for handling all types of traffic in the network in addition to detecting and selecting specific types of traffic that are good candidates for SOS. The SOS agents, on the other hand, are located on machines that are in the forwarding path of end hosts and are also equipped with a software OpenFlow switch.

When the SOS controller detects a connection that is a good candidate for SOS, it first checks to see if there are agents near the end hosts of each side of the connection. If so, the controller notifies the agents about the connection and installs flow entries on switches on both ends of the connection and software switches located within the SOS agents so the traffic is redirected to the nearby agent. The agents relay the client's data using a high throughput connection between each other. The far end agent delivers the data via a TCP connection to the desired end host. This whole process is completely seamless to clients due to header rewrite (mod_*field* actions) and is depicted in a high level view in Figure 3. In order for SOS to be

dynamically deployed, work in any topology, and handle multiple connections from multiple clients, several key components are needed.

These components are topology, agent, and host discovery done via the control plane, agent controller communication, agent connection setup, agent data transfer and finally flow setup. In the following section these key components and their implementations are discussed. The SOS OpenFlow controller was built using the Network Operation System (NOX) [20] API which provides an interface to manage datapath connections and an interface to marshal data to send (flow entries, packet_outs) and receive (packet_in) from datapaths.

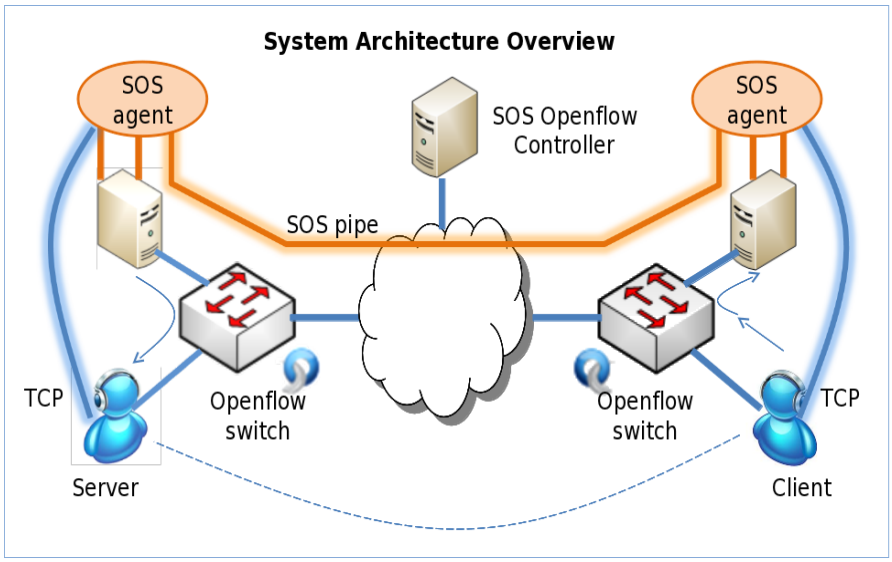

Figure 3 High Level View of SOS

3.1 Topology Discovery

As mentioned before, when a packet arrives at the OpenFlow controller, the controller must decide how to handle the packet. Since the packet in the network generally passes through several datapaths (OpenFlow switches) on its way to the destination, it makes sense to install flow entries on all the datapaths that the packet will pass through at the first notification of the packet (from packet_in). Doing this alleviates the need for controller involvement at each datapath that the packet passes through. In order to determine up front the datatpaths that the packet will need to pass through topology discovery must be done to learn the topology.

In order to discover the datapath topology the controller sends out Link Layer Discovery Protocol (LLDP) packets out each port of each datapath. Each packet contains the datapath ID and physical port it is sent from. Then, when these packets enter the ports of another datapath, they are forwarded to the controller as a packet_in event. From this, the controller knows there is a link between the two datapaths via the information contained in the LLDP packet and the packet_in event (which contains the port and datapath ID the packet arrived on). Since datapaths can disconnect and the topology can change, the controller periodically sends out these LLDP packets so that it can keep track of the topology. If links are not detected within a certain time out period, the controller then forgets about them since the link is no longer valid. This method of discovery is not an optimal solution since the controller needs to send out a separate packet for each port connected to each datapath. A much better approach for this would be if the discovery protocol was implemented on datapath, but unfortunately this has not yet been done [20].

3.2 Host Discovery and Broadcast Transmission with Loops

In order to install end-to-end flow entries the controller needs to know the topology in the network in addition to where hosts are in the network. In order to discovery hosts, Address Resolution Protocol (ARP) is used when the location of the end host is not known. ARP consists of broadcasting a packet to the entire network. In traditional networks there are generally no loops in the broadcast domain since loops can allow packets to continue to be retransmitted. In most cases, Spanning Tree Protocol (STP) is used in order to ensure a loop-free topology. In this case, loops are desired such that multiple paths are available for traffic to be directed. Due to this, broadcast packets need to be handled in a specific manner. When a broadcast packet is first seen, it is forwarded to the controller. Then, the controller tries to handle the request. For example, if the broadcast packet is an ARP request, the controller will send back a unicast ARP reply to the host (if the controller knows the MAC address of the desired end host). If the controller does not know the MAC address of the host the packet must be broadcasted to the network. Then, when the host replies the controller learns the hosts MAC address and which datapath they are connected to.

In order to safely do this the source address of the layer 2 packet (dl_src) is modified to identify that the packet came from the controller. Once this packet is flooded to the network it will then be returned to the controller due to loops (interconnected datapaths). At this point the controller then checks if the source MAC address of the packet is its own, and if so, the packet is dropped, stopping the packet from looping. This implementation is suboptimal since the control

plane is used in order to handle broadcast transmissions and could possibly be a bottle neck. Figure 4 outlines the ARP handling process.

As one can notice broadcasting to the entire network in order to learn a host's location definitely seems like a costly operation. That said, in most cases, the controller does not need to send out ARP requests broadcasting to the entire network since the location of hosts will generally be learned initially when a host tries to contact the end party (and the controller only needs to broadcast once in order to learn the hosts location). Lastly, the controller also needs to send ARP broadcast requests in the case that the controller is reset and the end hosts know each other's addresses but the controller does not know their location.

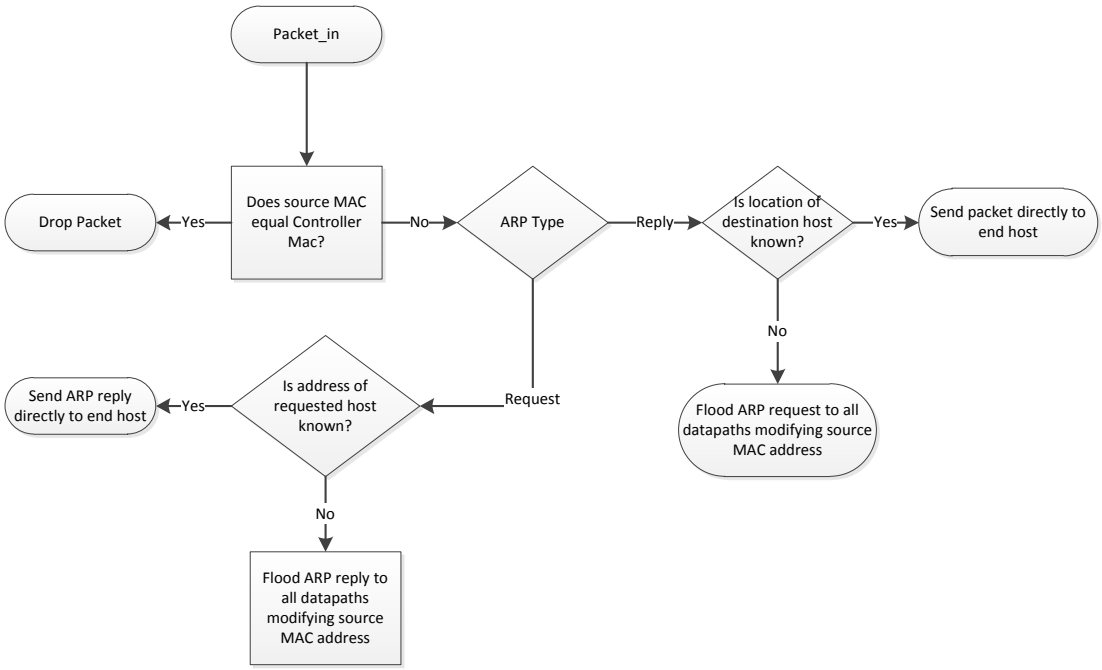

Figure 4 ARP Packet Processing

3.3 Agent Discovery

Since agents can be dynamically brought up and down in a network in addition to being available for use by multiple controllers (via FlowVisor), a discovery method for them is needed. In the current implementation the agents periodically send out UDP packets to a specific non-existent end host IP address and UDP port. The agent discovery mechanism uses UDP rather than LLDP (layer 2 protocol) for discovery so that if an agent does not share a layer 2 path with an OpenFlow datapath (i.e: is behind a router), the agent can still be learned if an OpenFlow datapath is located upstream.

When the agent sends out a discovery packet it is forwarded to the controller via packet_in at the first OpenFlow datapath it comes in contact with. Contained in the discovery packet are features that inform the controller about what each agent supports. This allows for different types of agents to be implemented using the same framework providing different services. Since the agent advertises himself by sending out packets instead of directly contacting the controller, multiple parties can make use of the agent without any additional change or configuration to the agent. In order to accomplish this, one would need to add a copy of the agent discovery packets to their slice via FlowVisor. The controller also times out the agents after not hearing from them in a time period, to allow for agents to be dynamically brought up and down.

3.4 Controller to Agent Communication

In order for the agents to know the correct hosts to connect to, communication between the controller and the agent is needed. When the controller decides to makes use of

an agent's service, the controller informs the agent of the incoming connection and what parameters should be used to handle the connection. The controller sends this information to the agent by injecting a packet into the forwarding plane of the port the agent is connected (setting the packet headers to that of the agent). The reason this is done is so that the agent does not need to directly communicate with the controller over a single socket since it is desired for multiple controllers to utilize a single agent.

The controller informs the agent near the connecting host of the incoming connection providing NW_SRC, TP_SRC, and TP_DST of the client and a universally unique identifier (UUID). When the client's connection makes it to the agent this information is confirmed to ensure the client is connected to the correct end hosts. This is especially important to ensure SOS connects them to the correct end host since SOS could connect them to the wrong host if two connects came in at the same time and this information was not checked(or if one wanted to be malicious and eavesdrop on a connection). Connecting the client to a different host would be completely transparent to the client unless the application was able to detect this (for example, ssh which uses host keys to warn the user, though many other applications do not). In addition to that information, the controller also sends some parameters for how to handle the connection such as number of sockets to use, block size for transmission, and size of queue to use.

3.5 Agent Connection Setup

In order for the agent to handle multiple simultaneous connections at once and also have good performance, the agents use asynchronous I/O coupled with epoll() - an I/O event

notification facility provided by the Linux Kernel. When the agent software is started, first the program binds on two types of sockets: sockets used for agent to agent communication and sockets for host to agent communication. After this is done, the agent enters a poll loop waiting for connections and periodically advertises itself. There are two types of connections that can occur next: a host connecting to the agent and an agent connecting to the agent. Whenever either of these two events happen, the controller sends the agent information about how to handle the connection.

In the case of a host side event, the agent accepts the socket and then issues a connect using the parallel sockets to the correct end agent and returns to the poll loop to service others. Next, an event will fire when the sockets are connected. After this occurs, the agent transmits the UUID (received from the controller) over the parallel agent sockets. The end agent then receives the UUID and knows which group of sockets to use for the transfer and then connects to the correct end host. The transfer of the UUID is necessary in order for the agent to handle multiple concurrent requests at once by grouping each connection by UUID. After everything has connected, the agent forks a process off to handle this transfer. Figure 5 is a flow diagram that displays the connection details.

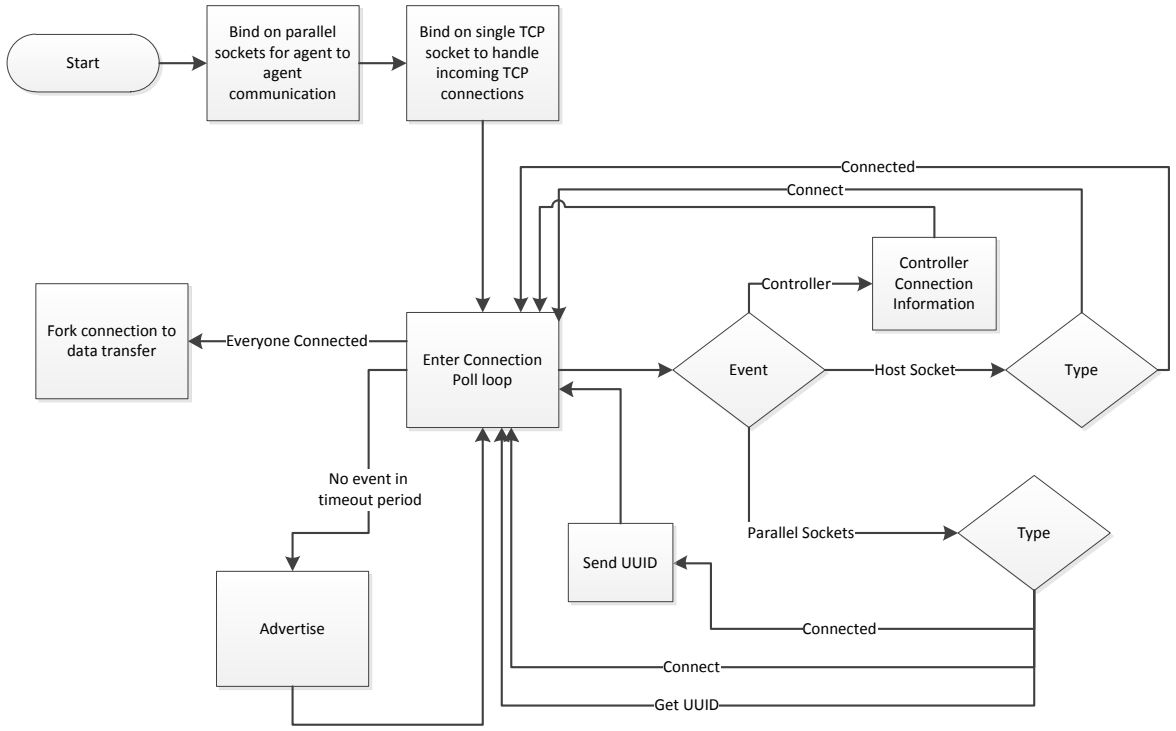

Figure 5 Connection Setup Process

3.6 Agent Data Transfer

In this version of SOS the agent uses multiple TCP streams in order to relay data between agents. Since the agent to agent traffic is independent of client and server any protocol services providers desire to use could be substituted in place of parallel TCP. In fact a version of SOS has also been developed using UDT [24] a high throughput protocol optimized for high bandwidth delay networks. In addition to that other transport methods such as RDMA [25] could be applicable as well.

Once everything has been connected, the agent must handle the data transfer. In this step another poll loop is used. This poll loop checks to see if there is data ready to be read at any of the sockets (pollin) or if the sockets are writable (pollout). If data is ready to be read at the host socket, the agent then checks if any of the parallel sockets are writable. If they are not, the pollin on the host socket is removed from the event loop and pollout on the agent sockets is added. This is done so CPU cycles are not wasted by constantly checking for pollout on the agent sockets due to pollin. Once one of the agent sockets becomes writeable, data is read from the host and sent to the agent.

Before sending the data, some bookkeeping needs to be done in order to reassemble the data on the other end. First, before sending the data, the size of the data is transmitted in addition to a sequence number. The size of the data needs to be transmitted first so the other agent knows how much data is part of the packet. This is required since streams are being used and not datagrams. The sequence number is needed in order to assemble the data in the correct order since there can be out of order arrivals across the multiple sockets. The transmitted packet is shown in Figure 6.

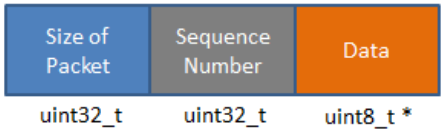

Figure 6 Packet Framing

On the receiving agent side, the packet is read once queue space is available for it. After reading the packet, the agent checks to see if the sequence number of the packet is the next sequence number needed to be transmitted. If it is, the packet is transmitted. If not, the packet

is stored in a hash table using the sequence number as the key for later transmission. During any time when packets are being transmitted or received, EAGAIN (errno return code) may be returned due to the fact that resources are unavailable. During this time, the number of bytes received/sent is saved in order to return to this location once resources are available. This occurs due to the fact that non-blocking (asynchronous I/O) is used. In synchronous I/O, the system call would block until the resources become available. This is not what is desired since work can be done on other sockets. Figure 7 shows a flow chart outlining the I/O process.

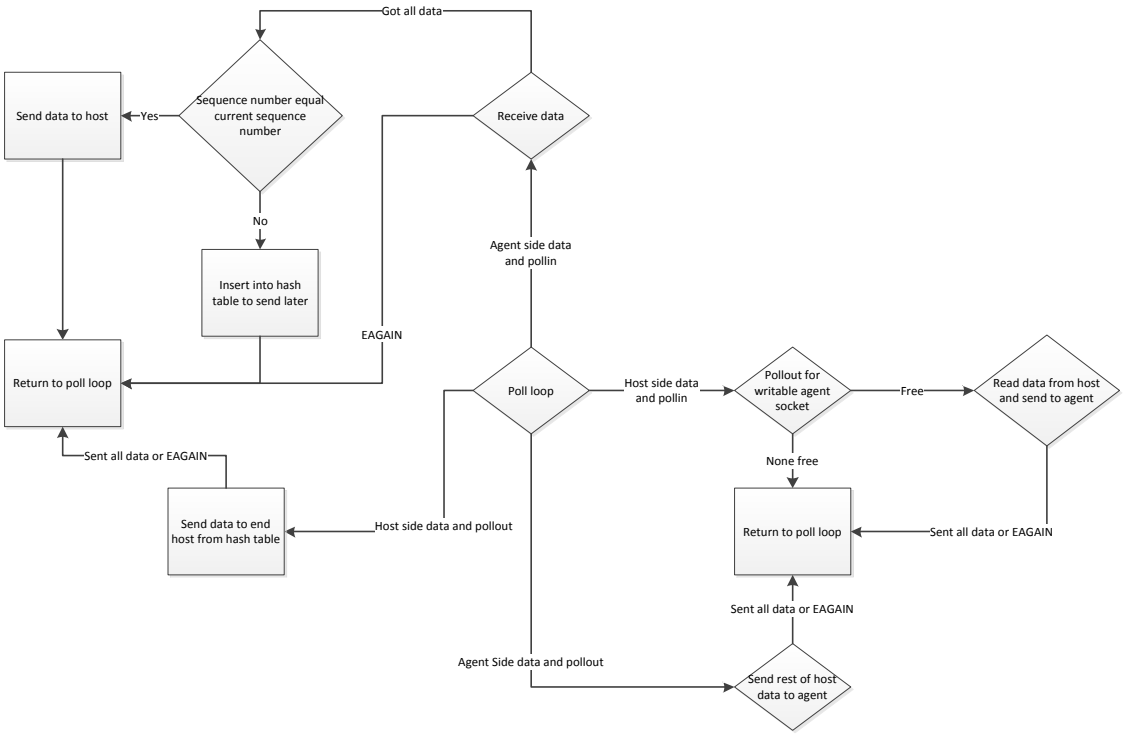

Figure 7 Agent Poll Loop

3.7 Controller Flow Setup

The controller is responsible for installing flow entries in order to handle traffic. Packets are forwarded to the controller when datapaths do not have an existing flow entry that matches a packet (packet_in). Once the packet is sent to the controller, the controller decides how the datapath should handle the packet. As mentioned in the previous section, the controller keeps track of the host and datapath topology information in the network. Using this information, the controller will perform a full end to end flow installation. This is done only if the location of the end host and a path between the two parties is known. Otherwise, the packet is dropped. If the location of the end host is not known, the controller sends out an ARP request in an attempt to learn the location.

Once the controller knows the location of the end hosts in the network and a path through the network the controller then decides how to handle the packet. First the controller looks at the packet to determine if it is suitable for the SOS service. It does this by checking if the packet is a TCP packet and coordinates to a specific port number. If so, the controller checks to see if an agent is located near the source and destination of the packet. If there is no agent available, the controller installs entries on all end-to-end datapaths in the path for normal L2 forwarding to occur.

If there are agents near both endpoints, flow entries are installed in order to make use of the agent. Below this process is outlined and shows the flow entries installed during each stage of the process for *Client* (10.0.0.11) connecting to *Server* (10.0.0.13). The flow entries given in this section correspond to the topology shown in Figure 8.

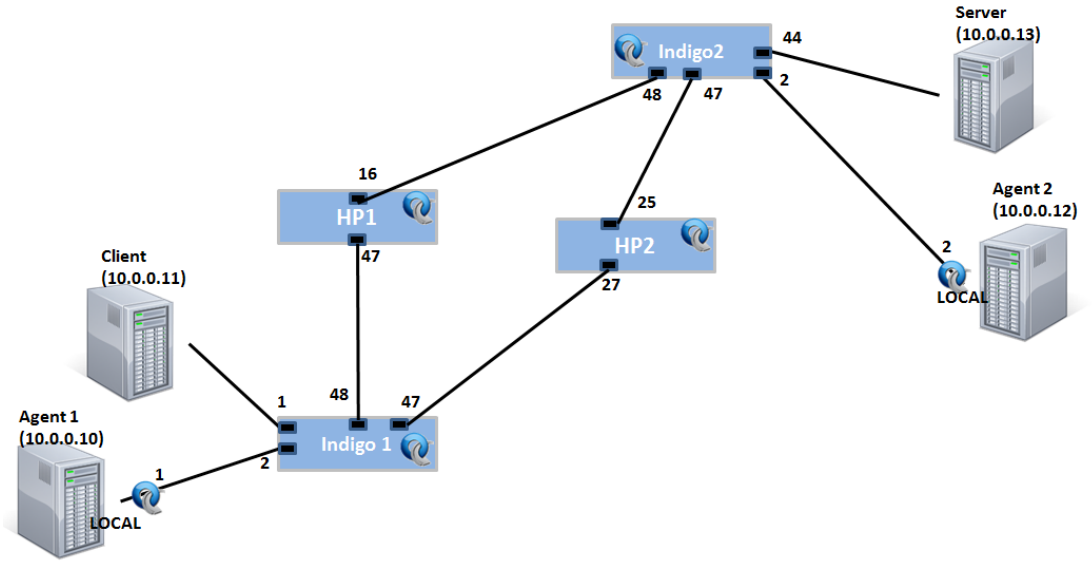

Figure 8 Local Test Bed Topology

1. ***Redirecting traffic from Client to Agent 1***: In order to make use of SOS, traffic from *Client* needs to be directed to its closest agent (in this case to *Agent 1)*. To do that, the set of flow entries below are installed on *Indigo 1* in order to forward *Client's* stream to *Agent 1* and back.

Table 1 Redirecting Traffic from Client to Agent 1

In_port=1,dl_vlan=0xffff,dl_vlan_pcp=0x00,dl_src=00:1f:29:32:91:99,dl_dst=00:1b:21:6a:85:88, nw_src=10.0.0.11,nw_dst=10.0.0.13,nw_tos=0x00,tp_src=41922,tp_dst=5003,actions=output:2
in_port=2,dl_dst=00:1f:29:32:91:99,nw_src=10.0.0.13,nw_dst=10.0.0.11,tp_src=5003,tp_dst=41922,actions=output:1

2) ***Modifying Client/Server headers for Agents:*** Now that *Client's* stream is able to reach *Agent 1,* the following flow entries must be installed on *Agent 1's* and *Agent* 2's

software OpenFlow switch. These flow entries rewrite the packet headers so the agents will be able to accept the stream. To do this, the fields DL_DST, NW_DST, TP_DST, are modified (to the agent's addresses) and then outputted to the agent's local networking stack. A subsequent flow entry is also installed in the opposite direction modifying DL_SRC, NW_SRC, TP_SRC, since the connection is bidirectional (not yet for *Agent 2)*.

Table 2 Modifying Client/Server Headers to Agents

Flow entries Installed on Agent 1
in_port=1,vlan_tci=0x0000,dl_src=00:1f:29:32:91:99,dl_dst=00:1b:21:6a:85:88,nw_src=10.0.0.11,nw_dst=10.0.0.13,nw_tos=0,tp_src=41922,tp_dst=5003 actions=mod_dl_dst:00:1f:29:32:92:4d,mod_nw_src:10.0.0.11,mod_nw_dst:10.0.0.10,mod_tp_dst:9877,LOCAL
in_port=65534,dl_src=00:1f:29:32:92:4d,nw_src=10.0.0.10,nw_dst=10.0.0.11,tp_src=9877,tp_dst=41922 actions=mod_dl_src:00:1b:21:6a:85:88,mod_nw_src:10.0.0.13,mod_tp_src:5003,output:1
Flow entries Installed on Agent 2
in_port=2,nw_src=10.0.0.13,nw_dst=10.0.0.11,tp_src=5003 actions=*mod_dl_dst:00:1b:21:6b:50:df*,mod_nw_dst:10.0.0.12,LOCAL

3) ***Flow Entries on Agent's Software OpenFlow switch for Agent to Agent Communication:*** At this point, *Client* is connected to *Agent 1* and believes it's connected to *Server.* This is due to the header rewrite performed (*mod_*field). Now, more flow entries need to be installed to handle the agent to agent parallel connections. Before this is done, the controller must decide which path it wants to send the agent to agent traffic on. SOS is able to utilize multiple paths at once. In order to limit the number of flow entries needed in the core of the network (datapaths between agents, in this case physical datapaths: *Indigo1, Indigo2, HP1, HP2*), the agent modifies the L2 header, changing the MAC address of the packet to reflect the path selected. This allows matching in the core of the network to be done solely on L1 and L2 (L3 is also matched

due to hardware limitation on some switches [26]) information rather than using L4 fields. This would require a flow entry for each parallel socket used (since L1-L3 headers all have the same values). The last flow entry installed rewrites the changed MAC address to the original value for traffic flowing from *Agent 1* to *Agent 2* and from *Agent 2* to *Agent 1*.

Table 3 Flow Entries on Agents for Agent to Agent Communication

Flow entries installed on Agent 1
Agent1 to Agent 2 flow entries using 4 streams and 2 different paths (see different mod_dl_src/dst)
in_port=65534,nw_src=10.0.0.10,nw_dst=10.0.0.12,tp_dst=9881 actions=*mod_dl_src:00:1f:29:32:92:4f,mod_dl_dst:00:1b:21:6b:50:e1*,output:1
in_port=65534,nw_src=10.0.0.10,nw_dst=10.0.0.12,tp_dst=9880 actions=*mod_dl_src:00:1f:29:32:92:4e,mod_dl_dst:00:1b:21:6b:50:e0*,output:1
in_port=65534,nw_src=10.0.0.10,nw_dst=10.0.0.12,tp_dst=9879 actions=*mod_dl_src:00:1f:29:32:92:4f,mod_dl_dst:00:1b:21:6b:50:e1*,output:1
in_port=65534,nw_src=10.0.0.10,nw_dst=10.0.0.12,tp_dst=9878 actions=*mod_dl_src:00:1f:29:32:92:4e,mod_dl_dst:00:1b:21:6b:50:e0*,output:1
Agent 2 to Agent 1 flow entry resetting the MAC addresses to be the correct values
in_port=1,nw_src=10.0.0.12,nw_dst=10.0.0.10 actions=*mod_dl_src:00:1b:21:6b:50:df,mod_dl_dst:00:1f:29:32:92:4d*,LOCAL
Flow entries installed on Agent 2:
Agent 2 to Agent 1 flow entries using 4 streams and 2 different paths (see different mod_dl_dst)
in_port=65534,nw_src=10.0.0.12,nw_dst=10.0.0.10,tp_src=9880 actions=*mod_dl_src:00:1b:21:6b:50:e0,mod_dl_dst:00:1f:29:32:92:4e*,output:2
in_port=65534,nw_src=10.0.0.12,nw_dst=10.0.0.10,tp_src=9878 actions=*mod_dl_src:00:1b:21:6b:50:e0,mod_dl_dst:00:1f:29:32:92:4e*,output:2
in_port=65534,nw_src=10.0.0.12,nw_dst=10.0.0.10,tp_src=9881 actions=*mod_dl_src:00:1b:21:6b:50:e1,mod_dl_dst:00:1f:29:32:92:4f*,output:2
in_port=65534,nw_src=10.0.0.12,nw_dst=10.0.0.10,tp_src=9879 actions=*mod_dl_src:00:1b:21:6b:50:e1,mod_dl_dst:00:1f:29:32:92:4f*,output:2
Agent 1 to Agent 2 flow entry resetting the MAC addresses to be the correct values
in_port=2,nw_src=10.0.0.10,nw_dst=10.0.0.12 actions=mod_dl_src:00:1f:29:32:92:4d,mod_dl_dst:00:1b:21:6b:50:df,mod_nw_src:10.0.0.10,LOCAL

4) ***Flow Entries on Core for Agent to Agent Communication:*** Now, flow entries are installed on all the datapaths between *Agent 1* and *Agent 2* in order to forward traffic between them through the core of the network. Again, only one flow entry is needed per path per direction.

Table 4 Flow Entries on Core for Agent to Agent Communication

Flow entries installed on Indigo 1:
in_port=2,dl_src=00:1f:29:32:92:4f,dl_dst=00:1b:21:6b:50:e1nw_src=10.0.0.10,nw_dst=10.0.0.12,actions=output:48
in_port=2,dl_src=00:1f:29:32:92:4e,dl_dst=00:1b:21:6b:50:e0,nw_src=10.0.0.10,nw_dst=10.0.0.12,actions=output:47
in_port=48,nw_src=10.0.0.12,nw_dst=10.0.0.10,actions=output:2
in_port=47,nw_src=10.0.0.12,nw_dst=10.0.0.10,actions=output:2
Flow entries installed on HP 1:
in_port=16,dl_src=00:1b:21:6b:50:e1,dl_dst=00:1f:29:32:92:4f,nw_src=10.0.0.12,nw_dst=10.0.0.10,actions=output:47
in_port=47,dl_src=00:1f:29:32:92:4f,dl_dst=00:1b:21:6b:50:e1,nw_src=10.0.0.10,nw_dst=10.0.0.12,actions=output:16
Flow entries installed on HP 2:
in_port=25,dl_src=00:1b:21:6b:50:e0,dl_dst=00:1f:29:32:92:4e,nw_src=10.0.0.12,nw_dst=10.0.0.10,actions=output:27
in_port=27,dl_src=00:1f:29:32:92:4e,dl_dst=00:1b:21:6b:50:e0,nw_src=10.0.0.10,nw_dst=10.0.0.12,actions=output:25
Flow entries installed on Indigo 2:
in_port=2,dl_src=00:1b:21:6b:50:e1,dl_dst=00:1f:29:32:92:4f,nw_src=10.0.0.12,nw_dst=10.0.0.10,actions=output:48
in_port=2,dl_src=00:1b:21:6b:50:e0,dl_dst=00:1f:29:32:92:4e,nw_src=10.0.0.12,nw_dst=10.0.0.10,actions=output:47
in_port=48,nw_src=10.0.0.10,nw_dst=10.0.0.12,actions=output:2
in_port=47,nw_src=10.0.0.10,nw_dst=10.0.0.12,actions=output:2

5) ***Flow Entries to Forward between Server and Agent 2:*** At this point, flow entries are installed on *Indigo 2* in order to forward the stream between *Server* and *Agent 2.*

Table 5 Flow Entries to Forward between Server and Agent 2

Flow entries installed on Indigo 2
in_port=2,nw_src=10.0.0.11,nw_dst=10.0.0.13,tp_dst=5003,actions=output:44
in_port=44,nw_src=10.0.0.13,nw_dst=10.0.0.11,tp_src=5003,actions=output:2

6) ***Last Flow Entry for Agent 2 to Server:*** Now, all flow entries are installed to forward the stream between *Client 1* to *Agent 1* to *Agent 2* to *Server* but not returning from *Agent 2* to *Server*. Lastly, *Agent 2* will initiate a last hop TCP connection to *Server* (which the controller informed him). When this is done, a packet_in is sent to the controller and a flow entry is installed, rewriting the DL_SRC and NW_SRC of the flow so that *Server* is tricked to believe the stream is coming from *Client*.

Table 6 Last Hop Flow entry

Flow entry installed on Agent 2
in_port=65534,vlan_tci=0x0000,dl_src=00:1b:21:6b:50:df,dl_dst=00:1b:21:6a:85:88,nw_src=10.0.0.12,nw_dst=10.0.0.13,nw_tos=0,tp_src=47489,tp_dst=5003 actions=mod_dl_src:00:1f:29:32:91:99,mod_nw_src:10.0.0.11,output:2

Finally, all of the flow entries are installed so the stream from *Client* is forwarded to *Agent 1* then *Agent 2,* which delivers it to *Server* via the original transport protocol used by *Client*. One might be curious why the previous flow entry was not installed up front while the others were installed. The reason for this is there is not enough information known from the original packet from the *Client* to uniquely match it with the desired actions for each client. Therefore, an agent would not be able to service multiple clients since each client would require

a flow entry with the exact same match but different actions. To solve this problem, the controller makes note that there is an expected stream coming from *Agent 2* toward a specific IP and port number. When this occurs, the controller installs a flow entry that is able to match on TP_SRC from *Agent 2*(which was unknown before) and adds the correct actions. Figure 9 outlines the process in a flow diagram.

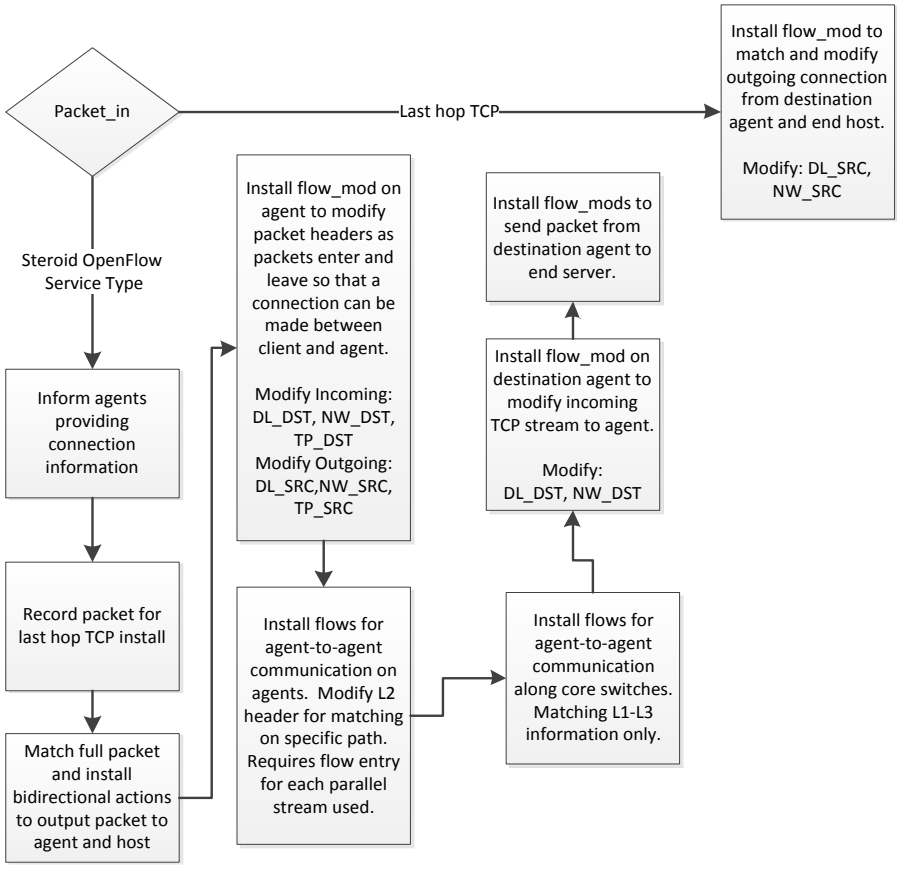

Figure 9 Agent Flow_Mod Process

Chapter 4

Experimental Analysis

The main focus of Steroid OpenFlow Service was to demonstrate the ability of decoupling a user's choice of protocol from the network. That said, in this section several experiments were conducted in order to verify the performance of Steroid Openflow Service. These experiments were conducted on both a local test bed and the NSF Global Environment for Network Innovations (GENI) future internet test bed. In the local test bed, different available bandwidth, latency, and amount of packet loss were emulated using Linux's tc tool to determine their impacts to the effectiveness of Steroid OpenFlow Service.

4.1 Local Test Bed

The local test bed used to gather results contained four different machines with their hardware specs displayed in Figure 10. Each machine ran the exact same OS: Linux 2.6.39-gentoo-r3 in an attempt to keep consistency since different kernel versions may have slight differences in TCP implementations and other features that may affect performance.

Processor	Ram	Network Interface	IP Address
Intel ®Core i5-2500k CPU @ 3.30Ghz	8Gb	Intel 82574L Gigabit Network Connection	10.0.0.12/24
Xeon ® CPU X3220 @2.40Ghz	4Gb	Broadcom NetXtreme BCM5715 Gigabit Ethernet (rev a3)	10.0.0.10/24
Xeon ® CPU X3220 @2.40Ghz	4Gb	Broadcom NetXtreme BCM5715 Gigabit Ethernet (rev a3)	10.0.0.11/24
Intel ® Core™2 CPU 6600 @ 2.40Ghz	8GB	Intel 82574L Gigabit Network Connection	10.0.0.13/24

Figure 10 Server Specifications

These machines were all connected in the topology shown in Figure 8 using two HP ProCurve J8693A switches 3500yl-48G running OpenFlow version 2.02w and two Pronto 3290 switches running 2011.12.05-iods: User mode. Stanford-LB9A. In addition to the hardware switches, Open vSwitch (a software OpenFlow switch) was also used (from git commit: 551a2f6ca9a90a577bc25cdd913f6df6bd8d9b23).

4.2 TCP Latency and Buffer Effects on Throughput

In this section, several tests were performed to determine the effect that latency and buffer size has on a TCP connection. The first parameter tested was the effect of latency on a normal end-to-end TCP connection. For this test, a TCP connection was run between hosts varying the RTT value. The default TCP settings on the hosts were left unchanged with both set to a 4MB maximum sending/receiving window value. In order to vary the RTT to simulate added latency to the link, Linux's tc tool was used. For each test, in order to achieve a specific RTT, such as to get 20ms RTT, 10ms of delay was added on each end host to the other. In addition to that, tc was configured to buffer 1Gbps (the max link rate of the hosts) amount of data multiplied by the (Xms) delay for each test. To ensure this was correct, a 1Gbps UDP flow was sent between hosts at each different latency setting. This confirmed that the added latency had no effect on the UDP throughput, which was to be expected.

Table 7 shows the tc configuration used for this experiment. First the configuration creates a root qdisc and attaches a handle with a buffer the size of the BDP of the link. Next, a filter is added to this qdisc that only applies this latency to traffic that matches the filter. This configuration was setup on two hosts, adding half of the RTT at each end point so the link latency would be symmetrical.

Table 7 TC Configuration for Latency

```
# performed on host 10.0.0.10

tc qdisc add dev eth1 root handle 1:0 prio
tc qdisc add dev eth1 parent 1:1 handle 10: netem delay Xms limit  (X/1000*125000000)
tc filter add dev eth1 protocol ip parent 1:0 u32 match ip dst 10.0.0.12/32 flowid 1:1

# performed on host 10.0.0.12

tc qdisc add dev eth1 root handle 1:0 prio
tc qdisc add dev eth1 parent 1:1 handle 11: netem delay Xms limit (X/1000*125000000)
tc filter add dev eth1 protocol ip parent 1:0 u32 match ip dst 10.0.0.10/32 flowid 1:1
```

Figure 11 presents the throughput of TCP connection between two end hosts varying the RTT. From this figure, it can be seen that latency has a large effect on TCP throughput. The main reason for this is that each host does not have enough buffer space in order to buffer all of the packets in flight in the network. In order to prove that buffer size was a factor, another test was performed to investigate the effect of this.

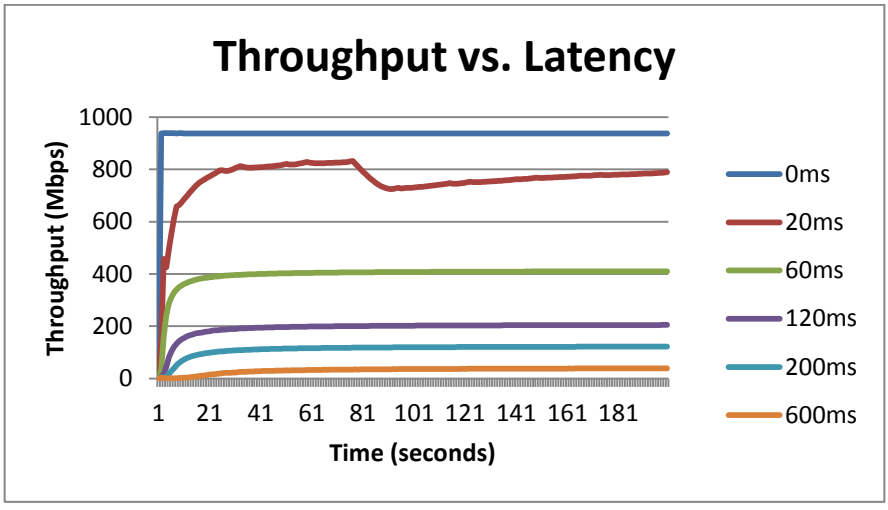

Figure 11 Throughput vs. Latency

In order to determine the effect of buffer size on a TCP connection, an experiment was setup where the buffer size was increased from 4MB to 16MB and 32MB. In this test, the sending window size, which corresponds to the number of unacknowledged bytes the sender is allowed to have, was recorded. This information was obtained using kprobes in the Linux kernel.

The results of buffer effect are shown in Figure 12. It can be noted that for both TCP connections, the sending window continues to grow exponentially at nearly the same rate until the maximum buffer allocated is reached. The connection with a 32MB buffer is able to obtain twice the sending window size as the connection with 16MB, which is expected. In addition to that, the connection with a 32MB buffer obtained a throughput of twice the 16MB connection. This test confirms that one of the limiting factors to throughput is TCP buffer size.

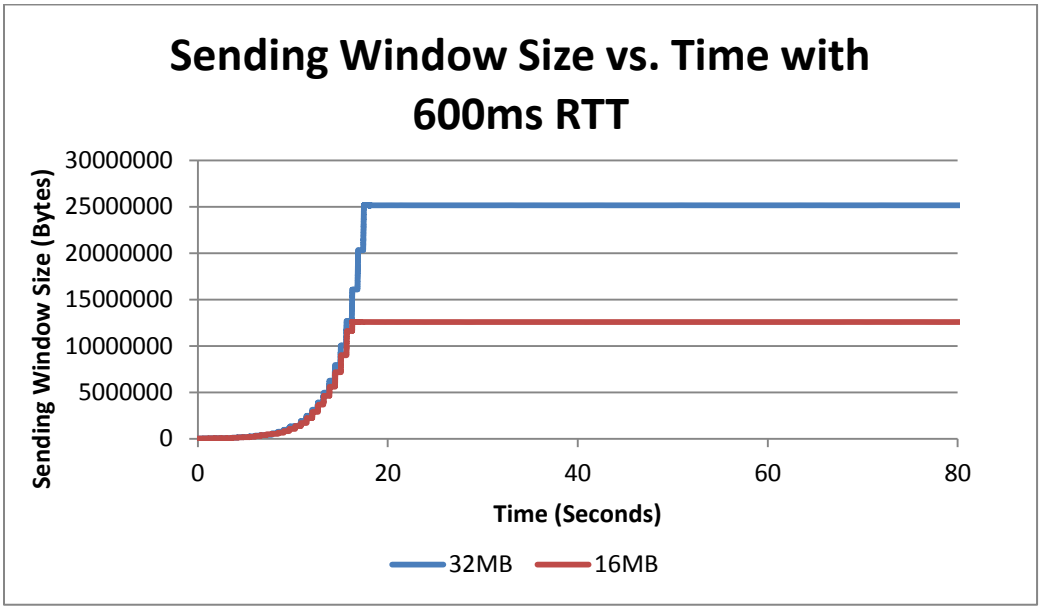

Figure 12 Sending Window Size vs. 600ms RTT

Besides buffer size, latency also has an effect on throughput. As mentioned before, TCP increments its sending window based on acknowledgements. With a higher RTT, it takes a

longer amount of time for the sending window to grow. In order to confirm this, the throughput for two TCP connections were measured over time, varying the RTT between 10ms and 600ms. The results in Figure 13 show the drastic effect RTT has on the throughput growth. The 10ms connection is able to immediately reach its maximum sending rate after 1 second, whereas the 600ms connection is still slowly growing even after 160 seconds.

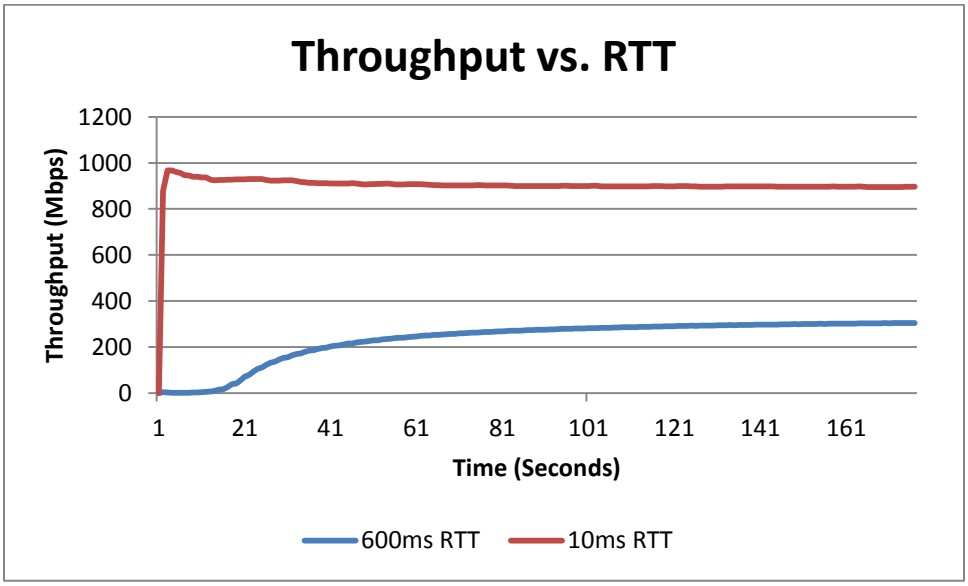

Figure 13 Throughput vs. RTT

4.3 Number of Parallel TCP Flows

SOS uses parallel TCP as its transport protocol in order to relay traffic between endpoints. Figure 14 shows the results of the aggregated throughput over time using 1, 2, 3, 4, 5, 10, 20, and 40 TCP connections with a 200ms RTT. In addition, the agents were configured with the default 4MB TCP buffer, a 1,000 packet queue size per stream, and a blocksize of 4096 bytes was used. The queue size coordinates to number of packets per stream the agent is

allowed to buffer when receiving packets. The block size is the number of bytes each agent tries to read/write when able to.

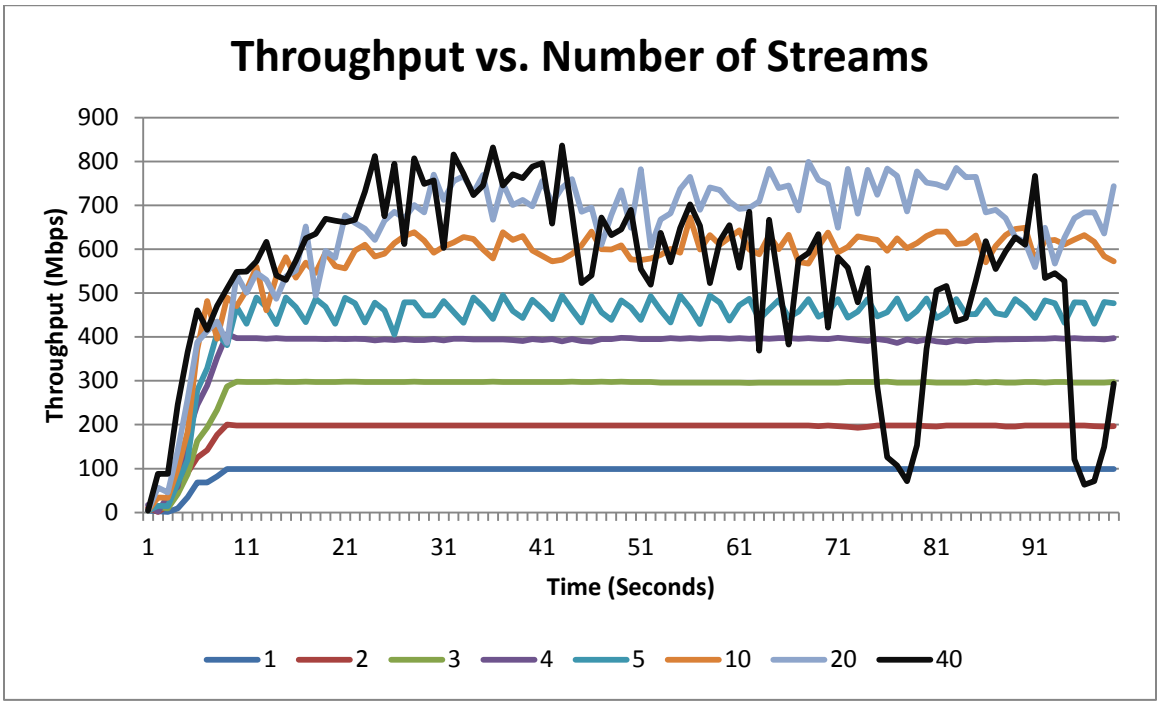

Figure 14 Throughput vs. Number of Streams

The results in Figure 14 show that the throughput continues to increase linearly for the number of streams used until a certain point. The results show that throughput grows by 100Mbits for each stream used up until 5 streams. After this, congestion occurs due to competing TCP streams. That being said, TCP throughput still continues to increase even with 20 TCP streams. After using 40 TCP streams it can be seen that performances starts to degrade due to oversaturating the network.

4.4 Multipath with Identical Paths

Steroid OpenFlow Service is able to make use of multiple paths at once in a network due to its parallel transport protocol. This ability allows SOS to exploit all the available bandwidth in

the network. Table 8 shows that using two TCP streams on path 1 or path 2 will achieve nearly the same throughput (proving that in the topology shown above both paths are close to identical). Though, by sending each TCP stream down a separate path, the average throughput is slightly lowered. It is hypothesized that the throughput is slightly lower using both paths at once due to a slight overhead in the switch when dequeuing packets from two different queues into one. The results below show the average throughput obtained after 10 minutes of runtime at each setting. That being said, this test was run multiple times and each time yielded similar results.

Table 8 Single Path vs. Multipath Throughput

Streams 2, Throughput (Mbps)		
Path	**Queue Size 1000**	**Queue Size 10,000**
Path1	930	929
Path2	929	930
Path 1 and 2	926	926

4.5 Queue Size

When packets arrive out of order, it affects the performance of TCP since it must wait until the next ordered packet arrives before passing data to the application (head of line blocking). There are several reasons for out of order packet arrival to occur. One reason is due to a packet being dropped and then retransmitted, thus arriving after other packets had already arrived. Other reasons are due to packet-level multipath routing, route fluttering, and many other reasons [21].

Parallel TCP suffers from these same issues in addition to out of order arrival among different TCP streams at the application level. The reason for this is because data is stripped

across multiple streams. When a missing segment of data has not reached an agent (at the application level), the agent has to wait until this segment arrives before it can reassemble the data and pass it on to the client. Since other streams can have data ready to be read, the agent goes ahead and reads those streams into a queue until the missing segment arrives. This is done in order to keep data moving and to avoid slowing down TCP connections if possible. Since the agent is unable to buffer an infinite amount of data, a queue is assigned to each stream. The queue specifies the number of packets that they are able to hold where each packet is of block size, which is the size unit that the agent tries to read/write when data is available. This allows the agent to keep reading from streams with data up until the queue fills. This is especially useful when dealing with a multipath network where paths widely vary in latency.

In this section, the effect of queue size on end-to-end performance is evaluated. Table 9 shows the throughput achieved using only one TCP stream between agents. In this setup, a block size of 8196 bytes was used in addition to a 100ms RTT. The results show the added queue size did not significantly improve throughput when only one connection was used. This is because with only one stream, it is not possible to have out of order packets arrive at the application level. The reason why the average queue utilization is not 1/1000 when a queue is provided and only 1 stream is used is due to bursty moments when packets are being received faster than they are being sent. Another reason is when the agent goes to transmit but is unable to do that. When this occurs, the agent needs to continue to hold the data until the sending buffer is able to receive more data. During this time, more packets may still be received from the end agent and are queued.

Table 9 Path Bandwidth vs. Throughput with Queue

Path Bandwidth (Mbit)	100		200		300		400		500	
Allocated Queue Size	1	1,000	1	1,000	1	1,000	1	1,000	1	1,000
Throughput (Mbps)	95	95	190	191	198	199	199	199	199	199
Average Queue Utilization	1	1.000011/ 1000	1	1.000166/ 10000	1	1.000011/ 1000	1	1.000339/ 1000	1	1.000362/ 1000

Table 10 shows the results of altering the number of sockets over a multipath network with two 512Mbit paths and a RTT 100ms. For this series of tests, a block size of 1024bytes was used and each test was run for 60 seconds. As the results show, there is a slight performance increase in average throughput when a queue is provided and multiple streams are used. The queue allows the agent to buffer out of order packets as they arrive over each stream. The average queue utilization for the 1,000 packet queue size is displayed. This value was found to be fairly dynamic in cases where each stream ran over a path that shared the same bandwidth and latency characteristics, even when the test was run for a longer period of time. The standard deviation for average queue utilization between streams is also displayed and proves to be much lower when identical paths are used (as seen later using multiple paths with different characteristics). The standard deviation of sent packets between streams shows that when a larger queue size is used, some streams end up sending more packets than other streams. This is due to the fact that with a queue provided, some streams are able to send packets while others suffer from congestion due to packet loss.

Table 10 Multiple Streams Queuing Effect

Number of Streams	1	2	3	4	5	6	7	8	9	10
Throughput (Mbps) with Queue Size 1	76	150	225	296	370	441	514	524	522	526
Throughput (Mbps) with Queue Size 1,000	76	151	227	299	376	451	514	540	550	529
Average Queue Utilization with Size 1,000	.001	.034	.03322	.07518	.0259	.03006	.02151	.0228	.02587	.04479
Standard Deviation of Average Queue Utilization with Queue Size 1,000	0	14.45	10.77	15.98	8.966	7.64	8.52	12.3	9.66	14.7
Standard Deviation of Sent Packets Queue Size 1	0	1.432	.304	.739	.651	.819	1.095	1.493	1.647	2.491
Standard Deviation Sent Packets Queue Size 1,000	0	2704	2266	1962	1094	1738	3371	4235	3066	4906

Table 11 shows the average throughput for a network with two 300Mbit paths. In this experiment, the goal was to see how difference of path latency affected throughput when a queue size of 1 and 1,000 was used. The results show that there is generally an increase in throughput when a queue is provided. For example, when 10 streams are used and there are two paths with latency 400ms and 25ms, using a queue of size 1,000 improves performance by 75%. In addition to that the results show that as the difference in latency between paths grows the more beneficial the added queue space is.

Table 11 Multiple Streams Queuing Effect Varied Path Latency

Throughput (Mbps) vs. Number of streams												
# Streams	1		2		4		6		8		10	
Queue Size	1	1000	1	1000	1	1000	1	1000	1	1000	1	1000
25-25 (ms)	291	291	580	580	580	584	580	585	580	584	581	584
50-25 (ms)	198	199	432	486	567	562	569	572	569	568	572	573
100-25 (ms)	97	96	304	380	452	478	524	541	539	523	544	555
200-25 (ms)	44	44	157	250	309	336	371	390	400	442	456	458
400-25 (ms)	19	19	60	122	139	209	183	278	226	310	258	346

*Note: When 1 stream is used, the stream ran over the path with a higher latency.

Table 12 shows the average queue utilization when two streams are used and each stream goes over a different path. As one can see, when the paths share a similar latency, the average queue utilization is about the same for each stream (for example, when both paths have a latency of 25ms the standard deviation of queue utilization is .00024). As the difference in latency between paths grow, the average queue utilization also grows. In order to achieve a high throughput, it becomes obvious that one needs a large enough queue to buffer the difference in latency between both paths in order to fairly utilize the bandwidth on both links.

Table 12 Average Queue Length vs. Multipath Latency

Average Queue Length (Bytes) Comparing Paths of Varied Latency									
25-25(ms)		50-25(ms)		100-25(ms)		200-25(ms)		400-25(ms)	
.00176	.00142	.00101	.04039	.00115	.32321	.01184	.623	.1411	.4805

The next series of tests performed compared the effect the queue had when multiple paths that have different available bandwidths are used. Each test below was run for 60 seconds with a block size of 8192 bytes. In addition to that, tc was used to limit each connection. The tc configuration in Table 13 creates a root qdisc that contains the maximum bandwidth of the link. Then, two more qdiscs are attached to the root that specifies their maximum bandwidth. Lastly,

a filter is added, which matches on port and aggregates each stream to the desired qdisc which represents the path.

Table 13 Tc Configuration for Different Path Bandwidths

```
# performed on host 10.0.0.10

tc qdisc add dev eth1 root handle 1:0 htb
tc class add dev eth1 parent 1:0 classid 1:1 htb rate 1024mbit ceil 1024mbit
tc class add dev eth1 parent 1:1 classid 1:2 htb rate 100mbit ceil 100mbit
tc qdisc add dev eth1 parent 1:2 handle 10: netem delay <latency>ms limit latency/1000*125000000)
tc class add dev eth1 parent 1:1 classid 1:3 htb rate <bandwidth>mbit ceil <bandwidth>mbit
tc qdisc add dev eth1 parent 1:3 handle 11: netem delay <latency>ms limit latency/1000*125000000)
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.12/32 match ip dport 9877 0xffff flowid 1:2
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.12/32 match ip dport 9878 0xffff flowid 1:3
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.12/32 match ip dport 9879 0xffff flowid 1:2
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.12/32 match ip dport 9880 0xffff flowid 1:3
#…. For each connection

# performed on host 10.0.0.12

tc qdisc add dev eth1 root handle 1:0 htb
tc class add dev eth1 parent 1:0 classid 1:1 htb rate 1024mbit ceil 1024mbit
tc class add dev eth1 parent 1:1 classid 1:2 htb rate <bandwidth>mbit ceil <bandwidth>mbit
tc qdisc add dev eth1 parent 1:2 handle 12: netem delay <latency>ms limit latency/1000*125000000)
tc class add dev eth1 parent 1:1 classid 1:3 htb rate <bandwidth>mbit ceil <bandwidth>mbit
tc qdisc add dev eth1 parent 1:3 handle 13: netem delay <latency>ms limit latency/1000*125000000)
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.10/32 match ip sport 9877 0xffff flowid 1:2
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.10/32 match ip sport 9878 0xffff flowid 1:3
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.10/32 match ip sport 9879 0xffff flowid 1:2
tc filter add dev eth1 parent 1:0 protocol ip u32 match ip dst 10.0.0.10/32 match ip sport 9880 0xffff flowid 1:3
#… For each connection
```

The results displayed in Table 14 and 15 show the throughput obtained using a queue size of 1 and 10,000. When viewing the results, it can be seen that the added queue size significantly improves performance in cases where paths greatly differ in available bandwidth. For example, throughput is increased 255% when 10 streams are used with 2 paths of 500Mbit and 100Mbit when a queue size of 10,000 is allocated compared to when only a queue size of 1 is used. The reason for this is because when paths vary in bandwidth, it allows one stream to send data faster than the other. Allocating a queue helps to mitigate the difference in bandwidth to reduce blockage.

Table 14 Throughput Queue Size 1 Multipath Varied Bandwidth

Throughput (Mbps) vs. Number of Sockets Queue length 1					
	Path Bandwidth (Mbit)				
# Streams	100-100	200-100	300-100	400-100	500-100
1	95	190	199	198	199
2	190	240	240	238	238
4	192	267	268	272	267
6	193	261	230	198	279
8	171	221	230	260	239
10	195	200	194	236	204

Table 15 Throughput Queue Size 10,000 Multipath Varied Bandwidth

Throughput (Mbps) vs. Number of Sockets Queue length 10,000					
	Path Bandwidth (Mbit)				
# Streams	100-100	200-100	300-100	400-100	500-100
1	96	190	198	199	199
2	191	286	293	293	293
4	193	288	376	413	425
6	194	289	382	460	499
8	195	290	382	472	547
10	196	294	385	469	521

* In Table 14 and Table 15, when 1 stream is used, it runs over the path of higher available bandwidth.

4.6 Parallel TCP protocol overhead

In order to reassemble the data stripped across the multiple TCP streams, a framing mechanism is needed. In order to do this, the size of the segment of data being transmitted followed by a sequence number is sent over each stream before the data. Since TCP is stream based (unlike UDP/SCTP), the size of each segment needs to be known in addition to the sequence of segments in order to correctly reassemble the data. Since this information is needed, a slight overhead to the protocol is added. The amount of overhead added largely

depends on the block size used, in addition to other factors, such as amount of data a client sends, number of streams, packet queue allocated size, and network characteristics such as latency and available bandwidth.

Since SOS uses an unsigned 4 byte number to hold the payload size and sequence number, this adds 8 bytes of overhead to each segment. For example, if a block size of 4096 bytes is used, that would coordinate to an overhead of .195% (8/4096). This being said, this is not always true. The block size coordinates to the maximum amount of bytes that the agent will read and write each time. Though, if there are not that many bytes to read, the agent will be forced to send a smaller amount of data, thus contributing to a higher overhead. Since the agents sit transparently between the client and server, they have no knowledge of how much data and when the clients will send data. Therefore, they forward data as soon as they receive it. This means that if a client sent packets with a payload of 1 byte, every few seconds the agents would immediately forward that one byte along as soon as it was received. This would add 8 additional bytes to transmit that 1 byte. That being said, TCP also suffers from this issue since in order to send one packet, it requires (at least) 54bytes for the packet header (L2, L3, L4). In the parallel TCP overhead case, the overhead could be slightly improved by implementing a Nagle's like algorithm like TCP does, but at the application level. Another option is to implement some kind of notification to dynamically inform the end agent that framing is off in times of low network utilization and only use one stream. But for simplicity, this was not done since this added overhead is still very low.

In order to evaluate the effect of block size on throughput, several tests were run. Table 16 shows the results of running 1 stream over a 100ms RTT network with no bandwidth cap. As the results show, the larger the block size yielded a better average throughput obtained.

Table 16 Varied Block Size Effects

Varying Block Size Results					
Block Size (Bytes)	**512**	**1024**	**2048**	**4096**	**8192**
Throughput (Mbps)	155	156	169	188	198
# Application Level Packets	2311930	1152818	622046	345238	182105
% Overhead	1.5625	0.781251	0.390626	0.195313	0.097609
Average Packet Size Send/Received (Bytes)	511.99	1023.99	2047.99	4095.97	8195.92

Referring to Table 16, the added overhead percentage does not equate to the loss of throughput. For example, with a block size of 512 and 8196 bytes, the throughput achieved was 155 and 198Mbps, respectively. Accounting for the 1.5625% overhead would put the throughput at 158Mbps for the 512 block size connection. The reason why these numbers do not match up is due to overhead added by TCP.

Table 17 TCP Segment Size vs. Block Size

Average Segment Size vs. Block Size					
Block Size (bytes)	**512**	**1024**	**2048**	**4096**	**8192**
Average Segment Size (bytes)	1493	1494	2107	4193	8354
# Packets Sent	792416	789710	604612	337199	178640

Table 17 displays the average segment size and total number of sent packets (not counting ACKs) at each various block size. The average segment size is the average size of TCP payload in each packet sent for this stream. This information was obtained using tcptrace [22]. After reviewing these results it was noticed that the average segment size was larger than the maximum transmission unit (MTU) configured for each host (1500bytes). This was due to TCP segmentation offloading (TSO) which is a feature on the NIC card which allows the operating

system to pass down up to 64KB chunks to the NIC which breaks the data into MTU sized segments. Since this operation is performed on the NIC the host is unable to obtain the actually segmentation size sent on the wire. This feature drastically reduces the CPU time needed to transmit data since the segmentation is performed in hardware.

From running many experiments it was found that using a block size of 8192 bytes yielded the highest average throughput. The reason hypothesized that that sending a larger block size reduces the amount of system calls and context switches between kernel and userspace needed. Thus reduces CPU utilization and allows for a slightly higher throughput to be achieved.

4.7 Parallel TCP with Lossy Path

Using multiple TCP connections in order to transmit data is incredibility helpful, not only when dealing with high RTT, but also in the case of a lossy path. When packet loss occurs in parallel TCP, the multiplicative decrease only effects one of the streams, thus the overall sending window size is reduced to 1/(2 * number of streams) rather than reducing it by 1/2 . To demonstrate this, tc is used to create a link with a 20ms RTT and a 1% packet loss. The tc configuration shown below is similar to the one in section 4.2 except it specifies a loss value associated with the qdisc.

Table 18 Tc Lossy Configuration

```
# performed on host 10.0.0.10

tc qdisc add dev eth1 root handle 1:0 prio
tc qdisc add dev eth1 parent 1:1 handle 10: netem delay 10ms limit 1250000  loss 0.500000
tc filter add dev eth1 protocol ip parent 1:0 u32 match ip dst 10.0.0.12/32 flowid 1:1

# performed on host 10.0.0.12

tc qdisc add dev eth1 root handle 1:0 prio
tc qdisc add dev eth1 parent 1:1 handle 11: netem delay 10ms limit 1250000  loss 0.500000
tc filter add dev eth1 protocol ip parent 1:0 u32 match ip dst 10.0.0.10/32 flowid 1:1
```

Table 19 shows the results of this test. For this test, a block size of 8192 bytes was used and each connection was run for 3 minutes. Using parallel TCP with a large number of streams allows the aggregate throughput of all the streams to perform quite well, even in the case of a lossy path.

Table 19 Lossy Path vs. Nonlossy Throughput

1% Lossy Path vs. Nonlossy Path Throughput (Mbps)				
# Sockets	**Lossy Path Throughput (Queue Size 1)**	**Lossy Path Throughput (Queue Size 1,000)**	**Nonlossy Path Throughput Mbps (Queue Size 1)**	**Nonlossy Path Throughput Mbps (Queue Size 1,000)**
1	16	29	805	897
2	30	32	902	877
4	51	90	900	904
6	93	110	884	899
8	93	124	902	902
10	121	136	883	888
20	135	149	902	904
30	213	316	905	906
40	251	484	902	904
50	334	632	903	902
60	348	740	905	901
70	392	796	906	897
80	415	865	905	897
90	404	867	902	894
100	448	874	902	892

4.8 NSF Global Environment for Network Innovations

In order to demonstrate SOS on a nationally deployed network, SOS was run on the NSF Global Environment for Network Innovations (GENI) future internet test bed [23]. The GENI future internet test bed is a multipath network that spans the United States, consisting of several OpenFlow enabled switches, using both National Lambda Rail (NLR) and Internet2 (I2). This network also connects to several Universities (Clemson, Georgia Tech, Indiana, Kansas State, Stanford, Rutgers, Washington, and Wisconsin University). The GENI test bed is a fully programmable network which allows users to create a slice containing the network traffic they want to control. In addition to network resources, GENI also provides several different types of compute nodes that allow experimenters to run their experiment.

To evaluate SOS, four Protogeni[27] nodes were reserved, two at Clemson University and two at GENI Project Office (GPO), located in Cambridge, Massachusetts. Protogeni nodes are nodes that allow users full control, allowing them to change the OS/Kernel if desired. This was needed since Open vSwitch requires inserting a kernel module, in addition to allowing us the ability to change the network stack configuration. In an attempt to keep consistency, all of these nodes were installed with 2.6.39-gentoo-r3 (as in the local test bed before), since different kernel versions may have slightly different implementations of TCP that may skew the performance results.

Tables 20 and 21 were gathered using these nodes and used two paths provided by NLR, shown in Figure 15. One thing to note about this series of tests is that the nodes located at GPO were attached to a switch with hardware limitations, which did not allow multipath ability. Therefore, traffic from GPO to Clemson was limited to only using Path 2 to send on. That said,

sending from Clemson to GPO was able to utilize both Path 1 and Path 2. In the test below the client using the SOS service was located at Clemson and transmitted data to the server located at GPO. This allowed the majority of the data to be sent across both paths and the returning acknowledgements over one path. Each test below was the average throughput obtained after running for 3 minutes.

Figure 15 GENI Topology

Table 20 GENI Link Characteristics

	Path1	Path2
Latency (ms)	54	160
Available Bandwidth UDP (Mbps)	952	952

Table 21 GENI Throughput Results

# STREAMS		Path1			Path2			Path1 & Path2		
	Queue Size	1	1,000	10,000	1	1,000	10,000	1	1,000	10,000
	1	295	329	309	95	98	99			
	2	630	632	642	146	220	214	156	161	456
	4	653	685	744	447	434	434	127	261	491
	6	666	679	766	529	634	627	134	221	596
	8	623	711	802	783	776	788	127	259	660
	10	591	833	793	356	371	570	164	271	647
	12	563	737	826	351	401	614	173	251	684
	16	590	756	869	393	424	644	183	318	675
	20	584	758	789	406	646	672	201	327	689
	32	596	780	802	440	595	715	263	447	703

Tables 20 and 21 gathered from the GENI network show a similar trend of those gathered on the local test bed. The trend shows that using multiple paths of varied latency performance is increased when additional queue space is allocated. In addition, when a large number of streams are used, the network becomes over saturated and as a result overall performance degrades.

4.9 Detecting Over Saturated Network

From the results presented in the local and GENI test bed, it can be seen that there is not one magic number for the number of streams that can be used in every scenario. For example, the results in Table 22 show different statistics obtained over a network with two 300Mbit paths and a RTT of 20ms (obtained using the local test bed). In this series of tests, the agents are told to use a block size of 8192bytes and a queue size of 1,000 packets. In Table 21, the field Rexmt bytes is the total number of bytes contain in packets that had to be

retransmitted. The sent data column contains the retransmitted data, in addition to the successfully transmitted packets. When reviewing the data below, it can be seen that in this network, using only 5 streams yields a much better throughput. Reviewing the packet trace using tcptrace[22] shows that when 100 streams are used, 0.0185% of the data sent was retransmitted data, whereas no data needed to be retransmitted when 5 streams were used. In addition to this, the average queue utilization was almost 9 times higher when 100 streams were used. Furthermore, a lower average received block size was obtained, which is the average amount of data the agent receives each time the agent tries to read the stream. In this series of tests, the agent tries to read 8192 bytes (block size), though if 8192 bytes are not available, it receives what is available. The average queue utilization is higher when 100 streams are used since a large number of packets need to be retransmitted. This causes the agent to queue up packets since it must periodically wait longer for a packet that has been retransmitted. Lastly, the average received block size is lower due to the fact that there is less data that can be read.

Table 22 Over Saturated Network Equal Bandwidth

# Streams	Throughput (Mbps)	Average Received Block Size (Bytes)	Average Queue Utilization	Rexmt (MB)	Sent Data in 5 minutes (GB)
5	597	8180.2	.07026	0	20.78
100	512	7217.98	.62778	3.4	17.94

Table 23 shows the results obtained from a network consisting of a 20Mbit and 300Mbit path both with a RTT of 20ms where agent used 20 streams with a queue length of 1,000 packets and a block size of 8192bytes. In this setup, an average throughput of 111Mbps was obtained. Reviewing the statistics in Table 22 shows that the 300Mbit had an average queue

utilization of .921, which means that this set of streams spends most of its time blocking and waiting on packets from the 20Mbit stream.

Table 23 High difference in Average Queue Utilization

Path	Average Block Size(Bytes)	Average Queue Utilization
20Mbit	8181.1	.13529
300Mbit	8168.3	.92177

The results show that there is possibly some information the agent could make use of in order to better maximize throughput. For example, in Table 22 the agent could periodically review the statistics for each stream comparing it to each other's, and determines that there is an overall high average amount of queued packets and that the average block size is below a threshold from the controller told block size. Using this information, the agent could communicate to the other agent in order to slowly disable a number of sockets to reduce the average queue utilization and increase the average received block size. In addition to reducing the number of sockets, the agent could also try to periodically enable sockets in case available bandwidth increases. This same strategy could also be done using the difference in average queue utilization in Table 23 to disable streams, since performance would be better if less streams used the 20Mbit path. This being said, these features have not been implemented and evaluated.

4.10 Lessons learned

One lesson learned during the development process was that our OpenFlow-enabled hardware switches had a number of limitations that greatly affected performance. These limitations were: 1) the switches were unable to modify layer 2, 3 and 4 header fields of a

packet at line rate and 2) installing a large number of flow entries causes the traffic to be processed at the switch's slow path (software processing), and takes seconds to be moved to hardware forwarding. The rewrite issue was solved by adopting Open vSwitch(OVS) at each agent machine to perform rewrite before sending packets out to the hardware switches. The large number of flow entries installed was solved by using OVS on the agents to rewrite the MAC addresses, which gave the core switches a field to differentiate on to make use of multiple paths. Doing this greatly reduced the number of flow entries required.

This said, unfortunately some of the core switches (HP) do not check the dl_src/dl_dst of packets against the flow entries installed within the switch [26]. This was an issue that was not initially known by us and deemed to be quite troublesome to discover/debug. In addition to this some of the switches (Pronto, NEC) did check dl_src and dl_dst. Therefore the issue that occurred was that flow entries would be properly installed such that multipath would work correctly but would only actually worked in one direction and returning packets would only actually take one path. Since some switches checked dl_dst/dl_src packets that were supposed to return on the same path but took that other path ended up being dropped due to missing flow entries. The solution to this was to stop matching on the dl_src and dl_dst on core switches. This made SOS only able to use multiple paths if the agents were connected to an OpenFlow datapath that checks dl_src/dl_dst in addition to being directly connected to multiple paths at the datapath an agent was connected.

Lastly, the importance of reading/writing strategy and use of socket polling proved to be extremely important. An early implementation of the agent used the SCTP transport protocol, which is message based, unlike TCP (stream based). This allowed the agent to read and write

data in a round robin fashion, allowing the order of data to be maintained correctly. This approach was easy to implement, although it lead to performance loss since it required the agent to wait for the next file descriptor to be available to write/read before moving on. In addition to this, OpenFlow does not yet support modifying or application matching of SCTP yet therefore the service could not be implemented completely seamless. Thus, TCP was adopted in place of SCTP, and the reading and writing strategy was changed to using polling. The agent uses epoll() to be informed when any file descriptor becomes available for read/write. This avoided any busy socket from blocking the read/write progress of the entire flow.

Chapter 5

Conclusions and Future Work

This thesis shows how through the use of software defined networking user's choice of protocols can be decoupled to use a network provider's choice of protocols in order to provide a better experience for end users. The implementation details and performance results of Steroid OpenFlow Services were then presented showing its performance running in a local test bed and a large scale test bed deployed across the United States. From the results presented, it showed that SOS was able to overcome the short comings in TCP and alleviated the configuration complexities for end users since it requires no changes on their end to be made.

There is much future work in optimizations that could be added to SOS such as compression, caching, and automatic performance tuning to avoid over saturating the network. SOS presented above is just one example of using software defined networking in order to provide a seamless service, in this case to overcome the pitfalls in TCP while transmitting over a large BDP network. Future work will investing other approaches to leverage software defined networking to provide other seamless services. Some examples of include network security enhancements, delay tolerant network services, and user mobility in addition to a number of other applications.

Bibliography

[1] Yee-Ting Li; Leith, D.; Shorten, R.N.; , "Experimental Evaluation of TCP Protocols for High-Speed Networks," Networking, IEEE/ACM Transactions on , vol.15, no.5, pp.1109-1122, Oct. 2007

[2] Zhang, Y.; Ansari, N.; Wu, M.; Yu, H.; , "On Wide Area Network Optimization," Communications Surveys & Tutorials, IEEE , vol.PP, no.99, pp.1-24, 0

[3] Stanford Clean Slate Program, http://www.openflow.org/, last accessed in Nov. 2011.

[4] Jeongkeun Lee; Sharma, P.; Tourrilhes, J.; McGeer, R.; Brassil, J.; Bavier, A.; , "Network Integrated Transparent TCP Accelerator," Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on , vol., no., pp.285-292, 20-23 April 2010

[5] J. Brassil, et. al., "The CHART System: A High-Performance, Fair Transport Architecture Based on Explicit-Rate Signaling," ACM SIGOPS Review, February 2009.

[6] Sangtae Ha and Injong Rhee. 2011. Taming the elephants: New TCP slow start. Comput. Netw. 55, 9 (June 2011), 2092-2110. DOI=10.1016/j.comnet.2011.01.014

[7] Globus GridFTP, http://www.globus.org/toolkit/data/gridftp/, last accessed in Nov. 2011.

[8] Tom Kelly, Scalable TCP: Improving Performance in Highspeed Wide Area Networks. Computer Communication Review 32(2), April 2003

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[10] Sangtae Ha, Injong Rhee and Lisong Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant" in ACM SIGOPS Operating System Review, July 2008.

[11] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus Striped GridFTP Framework and Server, SC'05," ACM Press, 2005.

[12] Soohyun Cho; Bettati, R.; , "Adaptive Aggregated Aggressiveness Control on Parallel TCP Flows Using Competition Detection," Computer Communications and Networks, 2006. ICCCN 2006. Proceedings.15th International Conference on , vol., no., pp.237-244, 9-11 Oct. 2006

[13] H. Sivakumar, S. Bailey, and R. Grossman, "PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks," in Proc. ACM/IEEE Conf. Supercomputing (SC), Nov. 2000, p. 38.

[14] R. T. Hurley and B. Y. Li, "A performance investigation of web caching architectures", in ACM C3S2E'08 Conference", 2008

[15] J. Wang, "A survey of Web Caching Schemes for the Internet", *ACM Computer Communication Review,* October 1999.

[16] Pu, C.; Singaravelu, L.; , "Fine-Grain Adaptive Compression in Dynamically Variable Networks," Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on , vol., no., pp.685-694, 10-10 June 2005 doi: 10.1109/ICDCS.2005.37

[17] T. Worlf, S. You, and R. Ramaswamy, "Transparent TCP Acceleration Through Network Processing", in Proc. IEEE Global Telecommunications Conference (GLOBECOM), Dec. 2005, pp. 750-754

[18] FlowVisor, http://flowvisor.org. Last accessed in Mar. 2012.

[19] Open vSwitch, http://openvswitch.org. Last accessed in Mar. 2012.

[20] Noxrepo, http://noxrepo.org/noxwiki/index.php/Discovery. Last accessed in Nov. 2011

[21] Ka-Cheong Leung; Li, V.O.K.; Daiqin Yang; , "An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges," Parallel and Distributed Systems, IEEE Transactions on , vol.18, no.4, pp.522-535, April 2007

[22] Tcptrace, http://tcptrace.org. Last accessed Mar. 2012.

[23] GENI: Exploring Networks of the Future, http://www.geni.net/, last accessed in March. 2012.

[24] UDT: UDP based Data Transfer Protocol, http://udt.sourceforge.net/, last accessed in Nov. 2011

[25] E. Kissel, and M. Swany, “Session layer burst switching for high performance data movement,” in Proceedings of PFLDNet, Lancaster, PA, 2010.

[26] HP Switch Software OpenFlow Supplement, Software version K.15.05.5001 User Manual. Nov. 2011.

[27] Protogeni, http://protogeni.net. Last accessed Mar.2012