Seamless Overlays for Application Use

Graduate Independent Study, 600.810.12

Edmund Duhaime

Advisors: Dr. Yair Amir, Amy Babay

May 18, 2017

1. Overview

This project focuses on methods for allowing unmodified applications to communicate through overlay networks. The applications can access the benefits of overlay networks, without the need to change the original application in any way. Three programs (a local whole TCP/IP interceptor, a router whole TCP/IP interceptor, and a local payload only interceptor) were developed in the process, using existing tools, to facilitate application communication over an overlay. These programs will be available as part of version 5.3 of the Spines overlay network platform [2].

Section 1 focuses on the motivation, goal, and general approach of this project. Section 2 details the exact methods and tools used. Section 3 includes implementation considerations. Section 4 evaluates the payload only interceptors. Section 5 deals with future work.

1.1 Motivation

TCP/IP and UDP/IP are the building blocks of modern Internet communication. To allow the Internet to scale, the interior of the network is comprised mostly of routers that do little more than forward packets appropriately, while most of the complex logic is located on the edges of the Internet, on host systems. The benefit of this design is that the Internet is able to scale; the downside is that if one desires certain communication guarantees, they cannot get them directly from the existing infrastructure. This is where overlay networks come into play. By setting up a relatively small number of nodes (e.g. tens of nodes), in strategic locations, one can create an overlay network to send traffic through. This is essentially adding intelligence to the middle of the network at the overlay level. An overlay network can offer timeliness, reliability, multicast, instruction tolerance, and other potential benefits, by using techniques such as hop-by-hop recovery, multi-homing, and sending on redundant paths that are not available on the native Internet.

One potential use case for overlay networks is large file transfer. Imagine you have some large file that takes five hours to transfer using a TCP application, even though along the path from source to destination the bandwidth exists for the transfer to take one hour. The most likely cause of the slowdown is TCP's congestion control and TCP's fairness policy. It is also possible that network disruptions, such as dead or high loss links between autonomous systems, also contribute to the slowdown. However, for a service provider that arranges for enough bandwidth between its overlay nodes, considerations such as congestion control should not play a role and should not impose limits on the traffic between its nodes. With such limits removed, the file transfers can complete much faster, normally using some predetermined constant bit rate dissemination. An overlay network set up along the path can make use of a large portion of the purchased bandwidth and reduce the impact of network disruptions. Using this overlay network, perhaps the transfer would only take one hour instead of five. Modification of the application is necessary to use the overlay network for communication, instead of TCP. However, one may not be able to modify the

application - maybe it is a proprietary application whose source code you do not have or maybe you just do not have the resources to make the necessary changes. Because it is not always simple or possible to modify an application to use an overlay network, being able to seamlessly have an unmodified application communicate through an overlay network would be greatly beneficial.

1.2 Project Goal

This independent study seeks to expand on an initial, basic proof-of-concept that allows unmodified applications to use overlay networks. The goal is simple: given a TCP application that achieves a certain throughput over the wide area, have this application use an overlay network for communication and achieve a substantially higher throughput without modification of the application. This would have the benefit of being able to increase application throughput without any changes and in numerous scenarios.

1.3 General Approach

The general idea for accomplishing unmodified application communication over overlays is the capture and forwarding of TCP packets or their payload. The development focuses on Linux, and thus several types of iptables rules are used for the interception of packets. Several programs, hereafter called interceptors, were developed to facilitate the capture, forwarding, and delivery of application packets. In the simplest case, there are two applications, on different machines, that want to communicate with each other. Two interceptors are run, one on each machine with an application. When one application wants to send to the other application, it behaves as normal, sending out TCP packets. The interceptor, with the help of iptables rules, captures the TCP packets and sends the packets, or their payloads, to the other interceptor on the destination machine through an overlay network. On the other side the destination interceptor receives the transferred data from the overlay network, and delivers it to the local application, where it was destined. Three different types of interceptors were created, which mainly differ by capture method, as will be explained in depth in section 2.

2. Methods

All of the interceptors presented here accomplish the goal of allowing unmodified applications to communicate using an overlay network. As mentioned in section 1.3, iptables rules are the main tool used for interception of packets. Capture of packets is limited to those that match specific destination port and destination IP combinations. The techniques used assume that one of the running applications will be listening on a known port for the connection from the other application. The overlay technology used for this project is the Spines [2] overlay infrastructure. Interceptors each connect to a Spines daemon, also called a node, to interact with the overlay network. Configuration files are used for setting up the various rules and resources used by the interceptors. All interceptor programs were written in C, designed for Linux, and tested on CentOS 6.

2.1 Local Whole TCP/IP Packet Interception

The first interceptor design takes the entire TCP/IP packet and transports it across the overlay network. This means that the TCP header, IP header, and payload are all transported from the source interceptor to the destination interceptor through a Spines overlay network. Raw sockets are used to send the entire packet from the destination interceptor to the destination application on the same machine. The raw sockets are necessary to send packets that already have TCP and IP headers. The NFQUEUE chain of iptables and libnetfilter_queue library are used for capture and processing of the packets. Two sets of iptables rules are created: those for packets going to a specific port and specific IP, and those for packets originating from a specific port and going to a specific IP.

As an example, perhaps one wants to SSH into a machine with the traffic going through the interceptors, and thus a Spines network. The destination machine, where SSH is listening on port 22, has IP 203.0.113.10 and the machine that originates the connection has IP 203.0.113.20. On machine 203.0.113.20 an iptables rule of the following form exists:

```
iptables -A OUTPUT -j NFQUEUE -p tcp --dport 22 -m iprange -d
203.0.113.10 --queue-num 1 --queue-bypass
```

This rule says to take TCP packets that are destined to port 22 and IP 203.0.113.10, and deliver them instead to the local netfilterqueue. The rule lies on the OUTPUT chain of the filter table within iptables. The interceptor on that machine receives the packets from the queue, uses the destination IP to determine the Spines daemon to send the packets to (from information in the configuration file), and forwards them through Spines to the destination interceptor. As mentioned before, the destination interceptor receives the packet and sends it locally via a raw socket.

The other type of iptables rule that is put in place would be on the machine with IP 203.0.113.20 of the form:

```
iptables -A OUTPUT -j NFQUEUE -p tcp --sport 22 -s 203.0.113.20 -
d 203.0.113.10 --queue-num 1 --queue-bypass
```

This rule means that any packets that originate from port 22 destined to IP 203.0.113.10 should be sent to the netfilterqueue. The same process as described above happens once the interceptor reads the packet from the queue, sending it to the appropriate interceptor to be delivered to the other side of the application.

It should be noted that the interceptors require root level privileges. This type of interceptor requires this for two reasons. The first is to create the iptables rules as determined from the configuration file. The second is to access raw sockets, which are necessary to send packets that include TCP headers, IP headers, and payload.

2.2 Router Whole TCP/IP Packet Interception

A natural improvement to the previous method is to try and remove the need for root access on every single machine. If you have many machines in the same location that all want to use an overlay to communicate, it would be better if only one machine had to run the interceptor with root permissions. To accomplish this, some changes need to be made. First, all the machines with applications that want to communicate via the overlay network need their traffic to go through a single router machine. This router machine can be any normal Linux machine, setup to be the

gateway of the other machines. When forwarding packets from a router interceptor, the packets are forwarded to the router interceptor associated with the destination IP. The final change is a slight modification to the iptables rules, which are now only put on the router machine. The outbound rules for a specific destination port and IP are now of the form:

```
iptables -A POSTROUTING -t mangle -j NFQUEUE -p tcp --dport 22 -d
203.0.113.20 --queue-num 1 --queue-bypass
```

The difference is that this rule lies on the mangle table in the POSTROUTING chain of iptables, allowing packets that are routed through the machine to be captured. The other iptables rule type is modified in a similar way:

```
iptables -A POSTROUTING -t mangle -j NFQUEUE -p tcp --sport 22 -s
203.0.113.20 -d 203.0.113.10 --queue-num 1 --queue-bypass
```

Entire TCP/IP packets are still transferred through the overlay, and raw sockets are still used to deliver the packet to the correct machine once they get to the interceptor program on the router machine. However, as was the goal with this approach, the only machines that require root access are the router machines.

2.3 TCP/IP Payload Only Interception

The previous two methods can reduce the amount of jitter seen by the applications, through the use of hop-by-hop recovery, as well as allow for faster rerouting when a network disruption occurs, through fast detection of down links between Spines nodes. The issue is that TCP still ends up limiting the amount of throughput used by applications, likely because it is still limited by end-to-end semantics.

One solution to this problem is to break up the TCP connection. The basic idea is as follows. When an application initiates a TCP connection, the connection is instead directed to the local interceptor program. The interceptor accepts the TCP connection as if it were the destination application and receives the payload of the TCP/IP packets. That data is forwarded through the overlay network to the appropriate destination interceptor, determined via lookups similar to before from the configuration file. Some additional information is sent with the data to allow the destination interceptor to give the data to the correct local TCP connection. The interceptor on the destination machine initiates the necessary TCP connection to the destination application, and delivers the forwarded data. From that point on whenever either interceptor receives data from the established TCP connections, it forwards the data to the other interceptor, which delivers it to corresponding local TCP connection.

This solution relaxes the TCP semantics. Unlike the previous interceptors described in section 2.1 and 2.2, this type of interceptor no longer maintains the end-to-end TCP guarantees. This is because receiving an ACK on the TCP level no longer indicates that the packet(s) actually made it to the destination machine. If the overlay network or interceptors go down, data can be lost. However, breaking these semantics is what allows applications using this solution to access higher throughput. Therefore, there is a tradeoff to be had: higher throughput of applications or strong end-to-end data delivery guarantees.

By making these local connections between application and interceptor, the application should, as long as there are no other bottlenecks, be able to make full use of the bandwidth available to the

overlay network. As long as the interceptor is able to read from the TCP connection quickly enough and send the data through the overlay, TCP will keep providing data because it sees no loss and miniscule latency. The only difficulty is redirecting the initial connection to go to the interceptor instead of its true destination. This is accomplished with iptables rules of the form:

```
iptables -A OUTPUT -t nat -j DNAT -p tcp --dport 22 -d 203.0.113.10
--to-destination localhost:8701
```

These rules use the destination network address translation (DNAT) rules of iptables, to redirect packets going to a certain port and IP combination to a local port instead. For example, the above rule means that a TCP connection going to IP 203.0.113.10 and port 22 is instead directed to the localhost on port 8701. The interceptor is listening on that local port, 8701, and has it mapped to the appropriate destination interceptor.

Since any data lost between interceptors would disrupt the connection, due to relaxed TCP semantics, the overlay network must ensure end-to-end reliability and in order delivery between interceptors. To achieve this in Spines the reliable session protocol is used, which provides these guarantees.

One additional note is that the payload only interceptor has slightly weaker requirements on the need for root level privileges. While root privileges are still necessary for creating the iptables rules, the rest of the program does not require any elevated privileges. Therefore, one could break up this interceptor into a part that requires root access to create the iptables rules, and another portion that handles all connections without root privileges. This may be preferred, simply to limit the amount of elevated privileges given.

3. Implementation Considerations

While proof of concept versions exist for the interceptors described in sections 2.1 and 2.2, the interceptor in 2.3 is the most well developed. This section will discuss some details of the payload only interceptor's implementation.

The Lex [1] and Yacc [4] tools are used for parsing configuration files within the payload only interceptor. The use of these tools helps to ensure that additions to the configuration are easier to implement, since they are in an existing framework. The payload only interceptor uses the event handling system from the Spread Toolkit [3], also used by Spines, for handling all connections. The interceptor also uses other components, such as a hash table, included with the Spread Toolkit. Using the tools provided by Spread also makes the payload interceptor more readable, and therefore more extensible.

The Spines implementation also differs slightly from the Spines version 5.1 released (the latest available at the time of writing). The first change to Spines is the ability to disable the TCP congestion control/avoidance system that exists in the reliable session, which is the majority of the flow control in the reliable session. Disabling this system allows for Spines to make use of more bandwidth, as it is no longer bound by TCP-like congestion control. This is acceptable for our use cases as discussed in section 1.2. In addition, the use cases envisioned, such as large file transfer, would have a relatively constant rate of sending, meaning that flow control should not be necessary. Additional tuning is also done in terms of window sizes and other constants in the system, to further improve the bandwidth available to the reliable session.

4. Evaluation

4.1 In-Lab Testing with Emulated Loss

Setup

The first testing done was within a cluster to compare normal TCP, without sending data through an overlay network, to using the interceptors and transporting the data through Spines. Four machines were used in the setup, each running CentOS 6 with 16 GB of ram and Intel Xeon E3-1270 CPUs clocked at 3.5GHz. Between the machines was a 10 Gigabit switch. On two machines only Spines nodes were run. On the other two machines Spines nodes, applications, and interceptors were run. To test normal TCP an emulated latency of 30 ms and uniform loss of 0.5% was placed between the machines running the applications, using netem. To compare against this, an overlay network was created using Spines with a simple path topology consisting of three links, or hops. This Spines network connected the two application machines. An emulated latency of 10 ms was placed on each hop. On one hop emulated uniform loss of 0.5% was placed. All netem rules were specified for outgoing traffic. Two applications were used in testing: scp, secure copy which provides file transfer using SSH, and t_flooder, a simple program included with Spines that sends specific amounts of packets for testing. When testing normal TCP neither interceptors nor Spines programs were run.

Results

Program	AverageThroughput for Normal TCP in Mb/s	Average Throughput for Interceptor and Spines in Mb/s
t_flooder	18	76
t_flooder (rate limited to 200 Mb/s)	18	200
scp	8	193

Table 1. Comparison of average throughput for in-lab testing

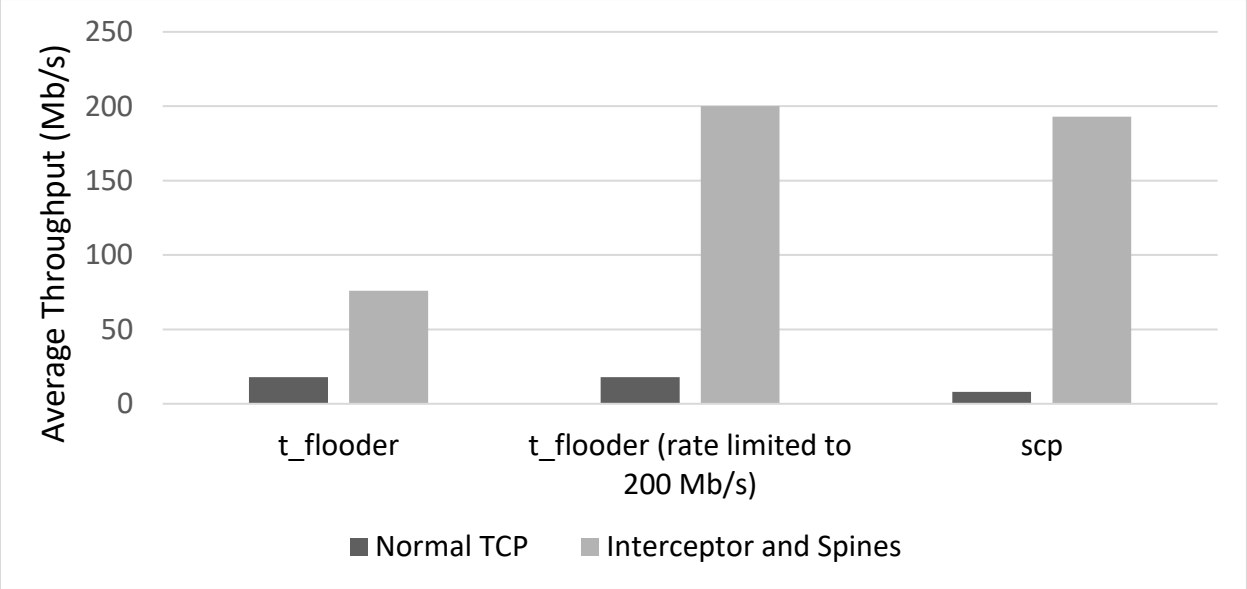

Figure 1. Graphical comparison of average throughput for in-lab testing

As can be seen in Table 1 and Figure 1, using the interceptor and Spines compared with just using TCP performed much better in terms of throughput with emulated 0.5% uniform loss and 30 ms of latency. The worst that the interceptor and Spines did was an average of 76 Mb/s throughput, with a 4.2 times throughput increase from normal TCP, when using the default t_flooder program. The reason for this is that the t_flooder program by default sends as fast as possible. Combined with the fact that the flow control was removed from Spines, this meant that too many packets were sent which overwhelmed the Spines nodes. This made the nodes believe that the links between them had died for a period of time. The nodes had to perform a handshake procedures to reestablish the link. Note that this is not the intended use case envisioned. Either it would be known that the overlay network could handle the throughput of the application or simple flow control protocol could be added. However, even with this additional overhead, the overlay network performs better because it is able to use hop-by-hop recovery. In the other scenarios, rate limiting t_flooder so that it does not kill the Spines links and scp, the interceptor and Spines perform an order of magnitude better than the normal TCP.

This is likely the best-case scenario for the interceptor and Spines setup, since loss is known to greatly reduce the throughput of TCP. However, this does suggest that if there are issues on the wide area that using the interceptors and Spines can minimize the impact of that issue, if the nodes are positioned well, relative to the throughput the application is able to use.

4.2 Wide Area Testing

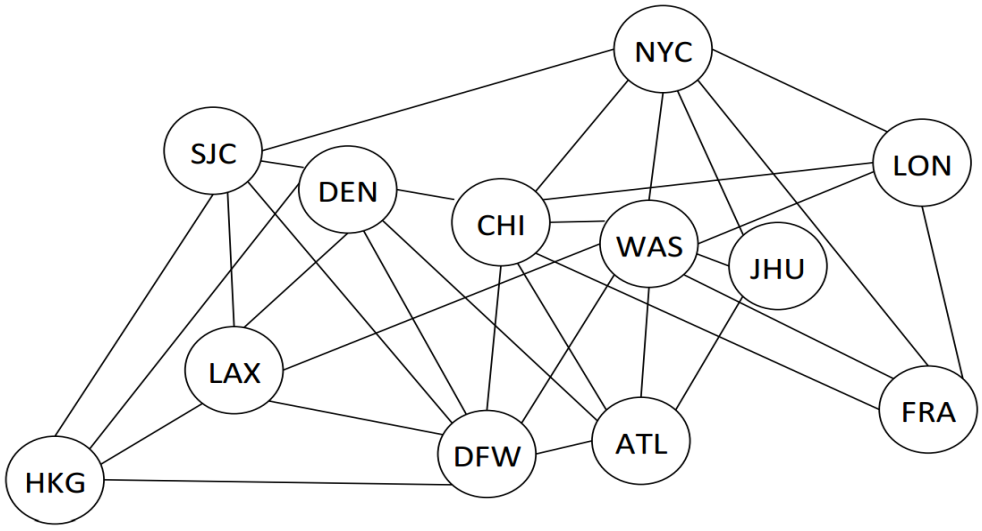

Figure 2. Overlay topology used in wide area testing. Each circle represents a Spines overlay node running on a machine in a data center.

Setup

The second set of tests involved actual wide area communication. The t_flooder and scp applications were again used. One machine was located in the lab at Johns Hopkins University (JHU), and the other machine was in Amazon AWS US West (N. California) Region, also called us-west-1. The machine in the lab had the same specifications as mentioned in the previous tests. The AWS machine was an EC2 r4.large instance using the dedicated hardware option. The locations of the overlay network used are shown in Figure 2, with Spines nodes running on

machines, that the lab has access to through LTN Global Communications, in each location. For this testing, Spines was limited to a maximum of 500 Mb/s that it could use.

Due to firewall rules on the LTN machines, slight modifications had to be made compared to the preferred setup. Ideally the EC2 instance would have connected directly to the Spines node running on the San Jose (SJC) machine. However, firewall rules prevented the establishment of incoming connections to the Spines node. Instead, an additional program, which was called the connector, was made. The interceptor on the EC2 instance connected to a dummy Spines node also running on the EC2 instance. The connector ran on the San Jose machine and made Spines connections to both the San Jose Spines node and to the EC2 instance Spines node. With some configuration modifications to the interceptor on the EC2 instance, when the interceptor needed to send to the interceptor at JHU it sent it to the connector via the dummy Spines node. The connector received the data from the EC2 instance interceptor and sent it through the wide area overlay topology, displayed in Figure 2, to the interceptor at JHU. A similar process happened for the reverse direction. This connector did end up being an unexpected bottleneck, though primarily when sending from JHU to the EC2 instance. Because of this, all testing that was done was mainly from EC2 to JHU. Tests were done as close together time wise as possible to limit differences due to changing network conditions.

Results

Program	Throughput for Normal TCP in Mb/s	Throughput for Interceptor and Spines in Mb/s
t_flooder	163	371
scp	110	182

Table 2. Comparison of average throughput for wide area testing

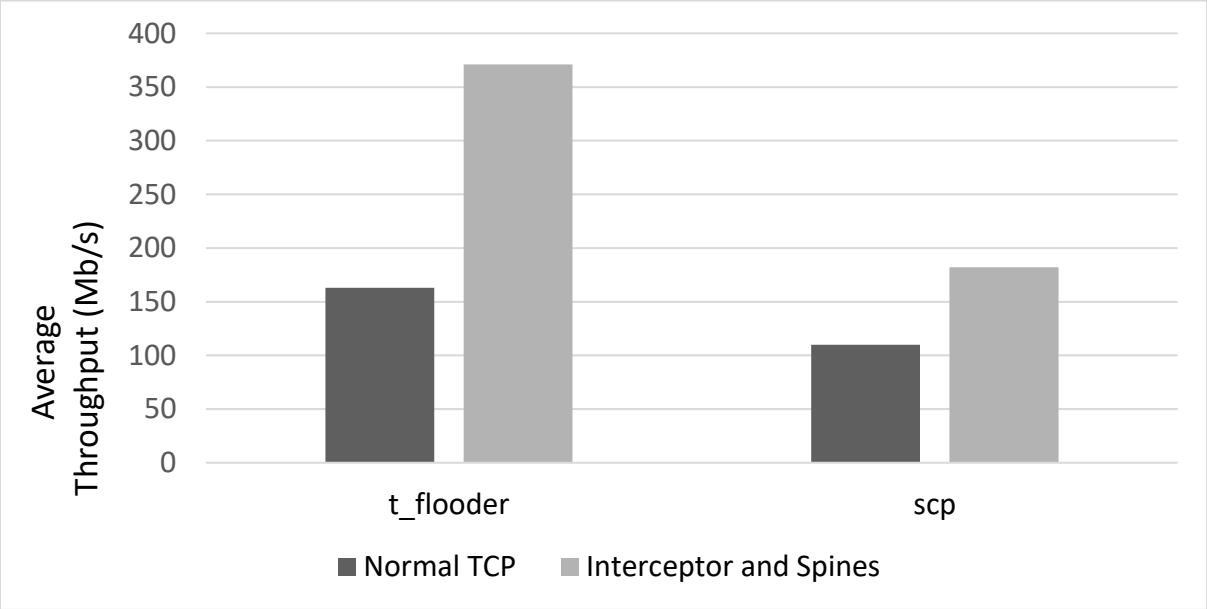

Figure 3. Graphical comparison of average throughput for wide area testing

The scp application is able to see an increase in average throughput, of about 65%, yet is still being limited. It is possible that this is due to other bottlenecks, such as IO on the EC2 instance. The more promising result is that t_flooder is able to see an increase of about 127% to average

throughput, over twice the original throughput that t_flooder was using with normal TCP over the wide area. This result demonstrates that the throughput that a program uses can be increased, even over a real wide-area network, by using an overlay network and without modifying the original application. This shows that the system can truly work. An application previously only bound by network traffic that took five hours before could now take two and a half hours without changing the application. It is also the case that if there are network disruptions on the wide area then, as long as the overlay nodes are well positioned, using the interceptors to go over that overlay network should see a large increase in throughput, similar to the in-lab testing.

5. Future Work

Potential for throughput increases on the wide area

As mentioned in section 3.2 there were several potential bottlenecks that could be removed to further increase the throughput available to applications. The obvious one is that Spines was capped at 500 Mb/s. This was originally done to limit the impact of testing on both the LTN network and on the Johns Hopkins University network. Removing this barrier and further tuning Spines for higher bandwidth, say 1 Gb/s, has the potential for more increases to the throughput an application can use. The other obvious bottleneck, also mentioned in section 3.2, was the connector program. If the interceptor on the EC2 instance was able to communicate directly with the Spines node running on the San Jose machine there would be no need for the connector, and thus eliminate this bottleneck. How much exactly would removing these bottlenecks increase the throughput that applications can use on the wide area is unknown, but it is worth looking into.

Security Concerns

As already discussed in section 2.3, if one is concerned about a program running for long periods of time that has root level access, the data only interceptor can be broken into two programs: one that requires root access for a short time to place the iptables rules, and another for handling all of the connections, which does not need root access.

Another security concern is that at the moment there is no authentication between interceptors. This is of particular concern for the payload only interceptors, which use information included in the stream received from Spines to send the data to the appropriate local TCP connection. Authentication would help to mitigate the risk of a malicious party wishing to crash the interceptor, or using the interceptor to bypass a firewall. More work would need to be done in order to determine the best way to add authentication to this system.

6. Conclusion

Overall this project shows that applications, without modification, can communicate using an overlay network. Furthermore, those applications using TCP can increase their available bandwidth, still without modification, by using an overlay network. These techniques allow for a two times throughput increase on a wide area network, with up to an order of magnitude throughput increase during network disruptions. There are numerous use cases for this technology, and the potential for larger increases in application throughput as well. Now that these methods have been developed they can continue to be worked on and expanded.

References

[1] Lex – A Lexical Analyzer Generator. M. E. Lesk and E. Schmidt. (www.dinosaur.compilertools.net).

[2] The Spines overlay network platform. Y. Amir, C. Danilov, J. Schultz, D. Obenshain and T. Tantillo. (www.spines.org).

[3] Spread – A Wide and Local Area Message Bus and Group Communication Toolkit. Y. Amir, M. Miskin-Amir, J. Stanton and J. Schultz. (www.spread.org).

[4] Yacc: Yet Another Compiler-Compiler. Stephen C. Johnson. (www.dinosaur.compilertools.net).