A DOA Module for OCALA

Evelyn Eastmond, Lev Popov and Daniel Wendel
6.829: Computer Networks
Massachusetts Institute of Technology
{evhan55,levpopov,djwendel}@mit.edu

Abstract

Numerous proposals have been written which attempt to address the problem of topological addressing (IP) on the Internet today. In order to be deployed, however, many of these proposed overlay systems require the impractical task of rewriting of the TCP/IP stack to support the new addressing schemes. The Overlay Convergence Architecture for Legacy Applications (OCALA) is a new architecture that attempts to provide a generic system into which these overlays can be effortlessly installed as modules and tested, without the need to rewrite the network protocol stack. In order to test OCALA's ease of use and performance, we wrote an OCALA module for the Delegation Oriented Architecture (DOA) overlay. We found that with few lines of code written into a provided sample OCALA module, we were able to successfully implement a working DOA implementation that could be tested in the OCALA framework. The performance of the system was not found to be stellar. However, it was suitable enough for OCALA to serve as a flexible test bed for the new Internet overlay proposals which have been needing an easily deployable framework such as OCALA.

1 Introduction

OCALA is a new network architecture that provides a generic framework for new Internet overlay addressing schemes [1,3,4,5,6]. The project is designed to allow any new overlay to be plugged in as a module and to be readily used over networks that also use the OCALA framework. The end goal for OCALA is to provide an effective and easily usable test bed for these new network overlays, by removing the need to rewrite the TCP stack individually for every overlay. Until now, the only module plug-ins that had been developed for OCALA were for the i3 (Internet Indirection Infrastructure) and RON (Resilient Overlay Networks) architectures. However, these modules were developed by the OCALA team members themselves. In order to prove its usefulness and effectiveness, it was important to test the development and deployment of an OCALA module from outside the OCALA team.

The Delegation Oriented Architecture is a proposed addressing scheme which focuses on the delegation to "middleboxes" (e.g. firewalls, NATs, etc) which are prominent in Internet communication today [2]. DOA however, hasn't seen a widespread test because its current implementation requires kernel level changes. As a result, DOA seemed like the perfect system to test as a new module for OCALA. If the module implementation were found to be successful, not only would OCALA benefit by being tested for ease of development and performance, but DOA would also benefit by being implemented in an easily deployable framework, therefore expanding its potential test bed to any host on the OCALA network.

In this project, therefore, our goal was to merge the work done in these two projects to answer the following questions: Is the OCALA framework flexible and generic enough to make it feasible to write modules for it? Is DOA reasonable to use? Are the two projects compatible? Is the performance of a working DOA OCALA module satisfactory?

In order to answer these questions, we had to do the following things:

- Read and understand the OCALA specifications and documentation
- Read and understand the DOA specifications and documentation
- Use the knowledge we gained to design and implement a DOA OCALA plug-in module
- Test the performance of the DOA module (running over OCALA) for DOA-specific measures (i.e. number of delegates).
- Evaluate the ease with which such modules can be developed.

Our findings were positive, indicating that OCALA is indeed a good framework for which to develop a DOA plug-in and that our DOA module, while slow, was able to demonstrate the benefits of DOA's explicit delegation in an easily deployable package which could be tested on many systems.

The rest of this paper is structured as follows: Section 2 gives background on the design and implementation of OCALA and DOA, Section 3 discusses various design decisions that we needed to make for our module, Section 4 outlines the implementation steps needed to build the module, Section 5 evaluates the effectiveness of OCALA and the DOA module and then in Section 6 we present our conclusions.

2 Background

2.1 OCALA

OCALA, at its core, is a proxy that different overlay modules can plug in to. It intercepts packets between applications and the network and gives its overlays a chance to act on the packets before passing them along.

In the control plane, OCALA intercepts DNS packets from applications and checks to see if any of its overlays want to handle the URL. If one does, OCALA asks this overlay to set up a connection, a tunnel over the overlay, to the specified host and returns a fake IP address to the application. It then stores a mapping between this fake IP address and the tunnel that the overlay opened.

From then on OCALA captures any packets sent to the fake IP address and sends them instead through the corresponding overlay tunnel. Essentially, OCALA functions like a NAT that translates overlay-specific addresses to fake IP addresses and vice-versa. This allows OCALA to present an IP like interface to the transport layer for legacy applications while its overlay modules send and receive overlay-specific packets on the network. The layer that does these translations is known as the Overlay Convergence (OC) layer. Figure 1 shows how the OC layer fits into the network stack between the overlays in the transport layer.

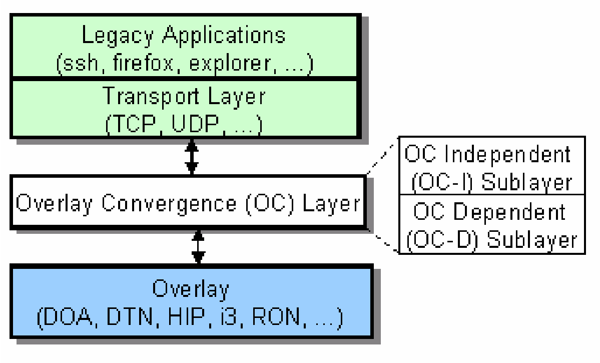

Figure 1. OCALA layer stack. [1]

The OC layer is itself split into two sub-layers - the OC Independent sub-layer, or OCI, and the Overlay Convergence Dependent sub-layer, or OCD. The OCI layer is where OCALA stores a mapping between transport layer connections and overlay tunnel handles. OCIState objects store necessary connection state between applications and the OCD layer. The OCD layer, on the other hand, is where OCALA interfaces with the overlays. The OCD layer stores mappings between tunnel handles and the tunnels themselves.

The original OCALA distribution included the source code for the OCALA proxy service, the OCI and OCD layers, and complete implementation of an I3 overlay module, all for Windows or Linux operating systems. It also included many README files describing how to compile and run the proxy and the OCD plug-ins. In addition, an updated distribution included a SampleOCD overlay module which demonstrated a barebones implementation of the OCD plug-in APIs. This SampleOCD module proved to be an invaluable resource in the development of our own plug-in.

2.2 DOA

DOA, or the Delegation Oriented Architecture, is an addressing scheme developed with middleboxes (firewalls, NATs, etc.) in mind. It is designed to allow and hosts to explicitly control which middleboxes their traffic should be delegated to before reaching them. An important feature of this architecture is that it provides enforcement of this delegation even when the delegate is off-path, not physically interposed between the two hosts.

The central idea behind DOA is to provide unique flat identifiers (eid's) for every host on the Internet that conveys identity but not location. In order for a host to send a packet to another host, it must first resolve this eid to a location. The resolution system is built around a DHT that stores a mapping between eid's and erecords which contain information about the host.

These erecords can contain either an IP address or one or more eid's. This IP address can either be the address of the host or the address of a middlebox. The eid's, on the other hand, by definition refer to delegates for this host. These in turn must be resolved recursively to IP addresses before the host knows where to send its packet. This allows receiving hosts to completely hide their IP address from all but their closest delegate if they desire. Additionally, even without hiding IP addresses, hosts can confirm that packets took their required path by validating the packets if each delegate signs the packets that it processes.

3 Design

3.1 Overview

In this section, we will describe the design decisions that we made during our implementation wherever either of the original two designs/projects provided room for interpretation or lacked detail. We also had to make certain design decisions in order to fit the DOA requirements into the OCALA framework.

3.2 The Control Plane for OCALA: When and How to Do DHT Lookups

One of the challenges in developing our DOA module for OCALA and was determining how and when to do DHT lookups. OCALA has a built-in callback mechanism for OCD tunnels to indicate when they are done setting up. The motivation behind this callback was to allow time for DNS queries. We were able to package our DHT lookups for outgoing connections into this DNS stage. However, OCALA does not have the same waiting period built into incoming connections, because the source of an incoming connection is written in the packet header.

Since DOA allows for a different backward path than forward path, however, our module had to do DHT lookups the first time a packet was sent on a connection the local hosts did not initiate. Additionally since DHT lookups can be recursive in nature, the delay before sending this first packet could be substantial. We designed a queuing mechanism to solve this problem. Each time a packet is sent over a specified tunnel, our module checks to see if that tunnel has finished resolving its outgoing eid stack. If it has not, that packet is added to the tunnel's queue. When the tunnel completes its DHT resolution it sends all of the packets waiting its queue.

Putting the resolution check in the send method also allows us to defer resolution for incoming connections until we want to send something along those connections. This prevents unnecessary work for connections over which we do not expect to send any packets.

3.3 Delegation: Packet Forwarding

Another important aspect of our module was the way in which we designed delegation. Our design starts with eid resolution via DHT lookups. We chose to use the hint field of each host's DOA erecord to store the host's IP address. The delegates portion of the erecord can contain either an IP address or an eid. In the resolution stage, each tunnel builds a stack of intermediate eid's, with the end host on the bottom, based on this possibly recursive resolution structure. In addition, it accesses the hint field from each intermediate erecord and stores the resulting IP addresses in a stack as well. Whenever packet is sent through this tunnel, these stacks of eid's and corresponding IP addresses are added to the header. This means that each middlebox on each packet's route are fully specified in the header.

The motivation for this design is that it allows for stateless middleboxes to correctly forward packets to end hosts without having to do DHT lookups. This is especially important for middleboxes that see many connections but few packets per connection, such as a firewall for a web server. However, the stack of IP addresses can be eliminated from the header for all middleboxes that explicitly store the locations of the end hosts behind them in some internal state.

We built packet forwarding capabilities directly into our module so that each host can also function as a middlebox. Whenever a packet is received whose header stack length is greater than one, a short routine pops the eid and IP address from the top of the stack and immediately sends the packet to the host specified by the new top of the stack. Although our implementation does not filter the packets in any way, it would be trivial to add a call to a filtering routine that either changes the packets in some way or simply determines whether or not to forward the packets. Regardless, forwarded packets are never passed to the legacy transport layer on the local machine.

3.4 Security

Although none of the features are currently implemented, our DOA module is designed to fully support the security features which the DOA design provides. Specifically,

- eid generation,
- erecord signing, and
- erecord verification

can all be added easily by a developer with an understanding of existing security libraries. eid generation is accomplished through a hash of the host's public key. Erecord signing should be implemented in the registerWithDHT() method of DOADHT, the DHT interface component. Erecord verification should be implemented in the receivedDHTReply() method of the DOAStateInfo component.

The interesting aspect of erecord verification is that since the eid of each host is a hash of its public key, hosts can be sure not only that an erecord was not tampered with, but also that the erecord belongs to the eid whose erecord was requested from the DHT.

With the exception of guaranteeing the security of the DHT itself, which is beyond the scope of this paper, the final piece of the design is in verifying that packets came from the proper delegates. With our current IP stack design, each delegate must sign the packets that it processes, and the end host can verify the signature of each delegate that it is interested in.

4 Implementation

4.1 Overview

Our general approach to implementation was to start with the source code of the provided SampleOCD module which already implemented the necessary interfaces for an OCD in the OCALA API.

By having most of the OCALA specific OCD interfaces already implemented, we were able to focus on the sections of the source code which related directly to the DOA specific interfaces. Specifically, this involved adapting the various send and receive methods and implementing the DOA specific control plane and packet structures.

4.2 DOA Specific Components

In this section, we present the DOA specific components that we needed to implement. The headings correspond to class names, and their specifications are provided here as documentation for our code so that other developers can build upon what we've implemented.

DOAOCD

- Overall parent class, implements the OCD API required by OCALA

DOAContext

- Reads a configuration file upon setup, for delegate and port information
- Contains DOA specific information about this host, including: eid, IP and erecord if desired
- ReceiveDOA() method receives packets from the overlay tunnel
    - Forwards DNS/DHT reply packets to DOAStateInfo for processing
    - Creates new DOAStateInfo objects for new incoming connections
    - Passes received data packets upwards to the OCIState object corresponding to the connection
- SendDOA() sends a DOA data packet to the end host specified by a DOAStateInfo
- If not yet resolved, asks DOAStateInfo to queue messages for later, and starts off a DHT resolution cascade

DOAStateInfo

- Contains state info for each connection from this host to another host including end host eid
- Calls DHT lookups to resolve the end host eid to a stack of eid's and IP addresses
- Keeps track if the eid has been fully resolved or not, and keeps queuing outgoing packets while the eid is not resolved (while it waits for more DHT replies)

DOADNS

- Sends DNS registration packet
- Sends DNS resolve request packets
- Hard-coded to send direct packets to a stub DNS server (DOADNSServer)

DOADHT

- Sends DHT registration packet
- Sends DHT resolve request packets.
- Hard-coded to send direct packets to a stub DHT server (DOADHTServer)

4.3 Example: The Lifespan of Ping Packet

The following is a summary of the path a ping packet takes between two DOA end hosts using our module running over OCALA.

Host A ("hosta.doa") pings Host B ("hostb.doa"):

(begin control plane)

1. Ping sends DNS lookup packet for "hostb.doa".
2. OCALA proxy captures DNS lookup, sees that it matches the DOA pattern, and calls the tunnel setup method of the DOA overlay, with the URL being "hostb.doa".
3. DOAStateInfo uses DOADNSServer to resolve "hostb.doa" to eid *b*.
4. When DNS reply is received:
    a. A mapping is added in DOAContext between the destination eid and the corresponding DOAStateInfo
    b. A DHT resolution cascade begins
5. Once the eid stack has resolved to the next-hop IP address, DOAStateInfo notifies OCALA that tunnel setup is complete.
6. OCALA then creates fake IP address *IPf*, and adds a mapping between it and its corresponding DOAStateInfo.
7. OCALA sends a fake DNS reply to ping with fake IP address *IPf.*

(end control plane, begin data plane)

8. Ping sends a ping packet to *IPf*.
9. OCALA captures packet, sees fake IP address, looks up mapping to DOAStateInfo, and calls DOAOCD's send() method .
10. DOAOCD calls DOAContext's sendDOA() method which creates a DOA packet from the original packet and the tunnel's DOAStateInfo, and sends it to the next-hop IP address on the DOA port.
11. Host B's DOAOCD is listening on the DOA port and receives the ping DOA packet.
12. Since Host B's context doesn't have an entry for the incoming end host A, it creates a new DOAStateInfo and then it adds an entry mapping the host to the state info.
13. It then strips the DOA header from the packet and passes the packet up to the OCALA receive() method.
14. OCALA inserts a fake source IP address and sends the packet to the ICMP receive mechanism which immediately sends a response.
15. OCALA captures the packet and passes it back to DOAOCD which tries to send it, but the DOAStateInfo hasn't resolved source A to an IP address yet so it queues the packet and does a DHT resolve for Host A.
16. Upon completing the resolution, it does something like step 11 above (creates a DOA packet and sends it).
17. Host A's DOAOCD is listening on the port, recognizes Host B's eid in the source field, strips the DOA header and sends the packet to OCALA with an indication of which tunnel the packet was received over.
18. OCALA maps the tunnel back to the ping application, and sends the ping reply to ping.

5 Evaluation

In order to evaluate the feasibility of developing modules for OCALA, we both tested the network performance of our DOA plug-in, and examined the amount of effort needed to develop the DOA module from concept to completion.

5.1 Performance

Our network performance tests were run on three computers in a variety of DOA and non-DOA configurations. Two of the computers were laptops running off the same 802.11g wireless router at 54Mb/s. The third computer, which always acted as the ping responder and fileserver, was connected to the network via a 100Mbps Ethernet link. These tests are not meant to be representative of any real usage situations, but rather to illustrate performance differences between the various configurations. We focused our network performance tests specifically on the latency and throughput of our system.

5.1.1 Latency

To test latency we ran three trials of 50 pings each for different network configurations with an increasing number of delegates on the forward path, with the end hosts remaining the same. We recorded and averaged the average latency from each of the trials. Our network configurations were as follows:

1) no OCALA running (no delegates)
2) OCALA+DOA, no delegates
3) OCALA+DOA, one delegate on forward path
4) OCALA+DOA, two delegates on forward path.

The figure below shows our test results, with the number of delegates increasing to the right.

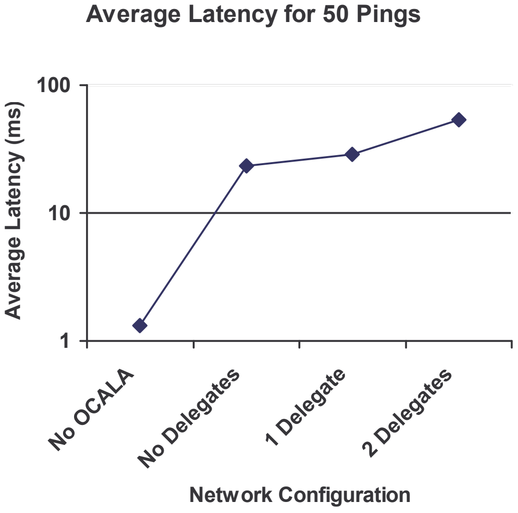

Figure 2. Average Latency for 50 Pings.

The results here were slightly surprising. While it is clear that adding more hops to the network path (i.e. delegates) should increase latency, the difference in latency between a non-OCALA ping and an OCALA ping with no delegates was more than tenfold. The relative slowdown of adding delegates once OCALA was running is much smaller in comparison, although still substantial.

The results of this test indicate that OCALA or our DOA modules itself adds a substantial amount of latency to the system. Additionally, each delegate in our test system added latency as one would expect. However, as our tests were run using a small network with very short link latencies, the relative importance of these factors may not be clear from the tests. For example, delegates spread out around the world would most likely be much more expensive to add than our tests show. In addition, although the largest change in latency in our tests was between not running OCALA and running OCALA, in a network where link latencies are greater than the latency introduced by OCALA, the OCALA latency would cease to be the dominant factor.

5.1.2. Throughput

Our throughput tests were run by using wget on a laptop to download a 1.4MB file from the fileserver. As in the latency tests, we ran different configurations and measured the performance for each. For throughput testing, our configurations were as follows:

1) No OCALA (no delegates)
2) OCALA+DOA, no delegates
3) OCALA+DOA, one delegate on return (download) path
4) OCALA+DOA, two delegates on return (download) path

The results of our tests are shown in Figure 3 below.

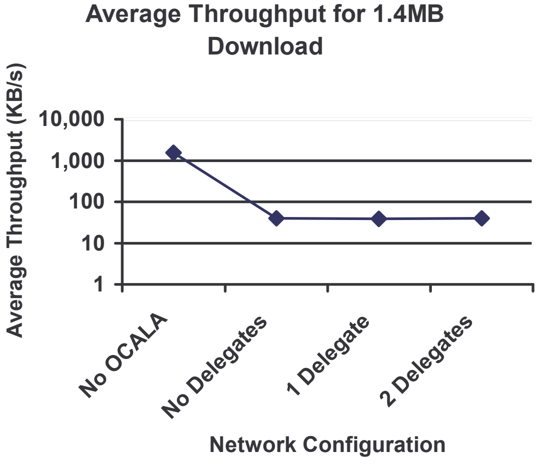

Figure 3. Average Throughput for 1.4 MB Download

The data from our throughput tests indicates that the only variable effecting throughput is whether or the connection is tunneled over OCALA+DOA. Although this was at first surprising, is seems that the bottleneck in the system is the data transmission rate of OCALA+DOA itself, not the network links. We hypothesize that this is either due to the inherent slowness of user-level sockets code, or due to extraneous buffer copies and per-byte data operations. However, even with the performance hit introduced by the OCALA/DOA system, we were able to obtain a throughput better than home internet link upload rates. It is slow, but still useable.

5.2 Ease of Development

The entirety of this project took 2 months from concept to final implementation and testing. The first full month required referencing the OCALA website [1], having email discussions with the OCALA team, reading the DOA papers [2][4], and having discussions with the DOA team. Once we had a deep understanding of the OCALA and the DOA project architectures, the implementation of the DOA OCALA module took approximately two weeks of heavy duty coding. Our final product contained approximately 1,500 lines of code, compared to the approximately 1,000 lines of code for SampleOCD. We found that it was easy to start with the provided SampleOCD implementation and replace relevant methods with code of our own. This saved us from having to learn the specifics of the OCALA API from scratch. Our learning curve also involved learning the details of the existing DOA implementation, but another research group wanting to use OCALA to prototype their own overlay architecture would presumably skip this step. The fact that we were able to go from having no knowledge of either OCALA or DOA to having a working DOA plug-in for OCALA as a class project indicates that the OCALA framework is indeed generic, flexible and easy to use.

6 Conclusion

Our findings indicate that OCALA is indeed a good framework for which to develop a DOA plug-in and that our DOA module, while slow, was able to demonstrate the benefits of DOA's explicit delegation in an easily deployable package which could be tested on many systems.

Acknowledgements

We would like to thank Dilip Anthony Joseph, Professor Ion Stoica and the rest of the OCALA team for their continued support and helpful advice throughout our project. We would also like to thank Jeremy Stribling and Mike Walfish for providing the DOA source code and explaining its implementation to us. Finally, we would like to thank Professor Hari Balakrishnan for posing our original problem and exposing us to the OCALA project.

References

[1] OCALA: An Overlay Convergence Architecture for Legacy Applications. http://ocala.cs.berkeley.edu.

[2] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful.

[3] J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. Supporting Legacy Applications over *i3*, June 2004.

[4] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. *SIGCOMM'04*, August 2004.

[5] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. *SIGCOMM '02*.

[6] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks, *18th ACM Symp. On Operating Systems Principles (SOSP)*, October 2001.