OCALA: An Architecture for Supporting Legacy Applications over Overlays

Dilip Joseph
Univ. of California at Berkeley

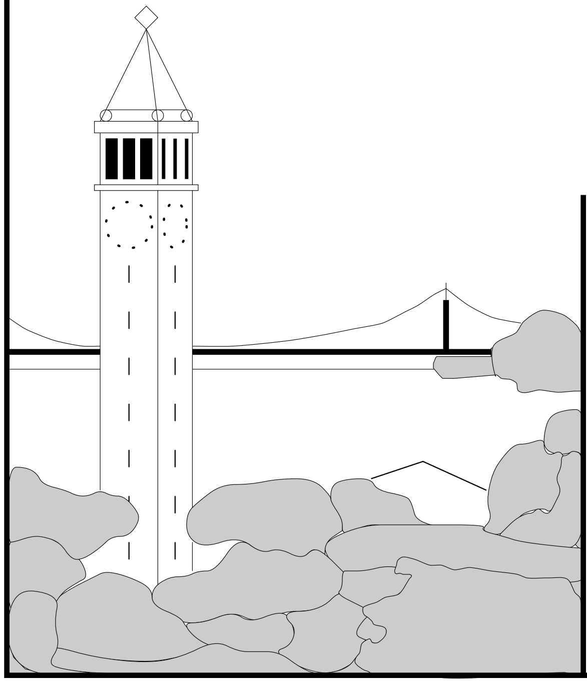
Jayanthkumar Kannan
Univ. of California at Berkeley

Ayumu Kubota
KDDI Labs

Karthik Lakshminarayanan
Univ. of California at Berkeley

Ion Stoica
Univ. of California at Berkeley

Klaus Wehrle
Univ. of Tübingen



Report No. UCB/CSD-005-1397

Computer Science Division (EECS)
University of California
Berkeley, California 94720

OCALA: An Architecture for Supporting Legacy Applications over Overlays

Dilip Joseph
Univ. of California at Berkeley

Jayanthkumar Kannan
Univ. of California at Berkeley

Ayumu Kubota
KDDI Labs

Karthik Lakshminarayanan
Univ. of California at Berkeley

Ion Stoica
Univ. of California at Berkeley

Klaus Wehrle
Univ. of Tübingen

Abstract

The ever increasing demand of new applications coupled with the increasing rigidity of the Internet has led researchers to propose overlay networks as a means of introducing new functionality in the Internet. However, despite sustained efforts, few overlays are used widely. Providing support for legacy Internet applications to access such overlays would significantly expand the user base of the overlays, as the users can instantly benefit from the overlay functionality.

We present the design and implementation of OCALA, an Overlay Convergence Architecture for Legacy Applications. Unlike previous efforts, OCALA allows users to access different overlays simultaneously, as well as hosts in different overlays to communicate with each other. In addition, OCALA reduces the implementation burden on the overlay developers, by factoring out the functions commonly required to support legacy applications, such as tapping legacy traffic, authentication and encryption. Our implementation of OCALA as a proxy requires no changes to the applications or operating systems. We currently support two overlays, $i3$ and RON, on Linux and Windows XP/2000 platforms. We (and a few other research groups and end-users) have used the proxy over a eleven-month period with many legacy applications ranging from web browsers to remote desktop applications.

1 Introduction

Over the past two decades, to meet the demands that new applications are posing on the Internet, several architectural changes that enable new functionality (such as mobility support, multicast, and quality of service) have been proposed. However, despite years of sustained effort, most of these proposals have failed to materialize on a large scale. One needs to look no further than mobile IP, IP multicast, and Intserv, for some oft-cited examples. Worse yet, as the Internet grows in size and usage, it becomes even more resistant to change.

To circumvent the rigidity of the Internet, overlay networks have emerged as the solution of choice for adding new functionality without changing the existing IP infrastructure. To list just a few examples, overlays have been used to provide better performance and resilience [2, 36], QoS [41], mobility [26, 49, 50], multicast [4, 7, 17, 19, 35], content distribution [1, 13], denial-of-service protection [3, 22], support for middleboxes [38, 46], and bridging multiple address spaces [5, 11, 29].

However, despite much research and the advent of PlanetLab [32]—a large-scale testbed for experimenting and deploying overlays—few overlays have gained widespread user acceptance. For end-users to reap the benefits of overlays, one needs to write new applications or port existing ones since many of these overlays come with their own API. Though there have been successful overlay-specific applications (such as vic/vat [25, 43] for the MBONE [9], and more recently, file sharing applications such as KaZaa [21]), these applications represent only a small fraction of all applications used by end-users. Porting legacy applications such as *ssh* and web browsers is not only a painstaking task, but may not be even an option if the source code is unavailable.

A complementary approach to developing applications is to provide support for *unmodified legacy* applications by using a substrate that bridges the legacy applications and the overlay. Allowing legacy applications to instantly take advantage of the overlay functionality (*e.g.*, mobility, resilience) would significantly expand the user base of the overlays. While there are several existing solutions that take this approach [2, 26, 40, 50], unfortunately, they have some important limitations. First, they do not provide *inter-operability*, that is, they do not enable communication across overlays. In a world where many overlays coexist, inter-operability across overlays is not only desirable but arguably necessary [31]. Finally, each overlay has to implement its own substrate, a non-trivial and time-consuming task.

We present the design and implementation of OCALA, an Overlay Convergence Architecture for Legacy Applications, which addresses these two limitations. OCALA allows users to access different overlays simultaneously, and provide inter-operability by allowing users to communicate across multiple overlays. In addition, OCALA reduces the implementation burden on the overlay developers by factoring out the functions commonly required for supporting legacy applications, such as capturing traffic, authentication and encryption. Our design is centered around four main goals:

- *Transparency:* Legacy applications should not break despite the fact that their traffic is relayed over an overlay instead of over IP.
- *Inter-operability:* Hosts in different overlays should be able to communicate with one another, and users should be able to form paths that span many overlays. Hosts that do not participate in any overlay should be accessible.
- *Expose Overlay Functionality:* Users should have control in choosing the overlay used to send their traffic, and should be able to leverage the overlay functions despite using overlay-unaware (legacy) applications.
- *Security:* Instead of relying on the security provided by overlays, the architecture should provide basic security features such as host authentication and encryption.

In a nutshell, OCALA consists of an *Overlay Convergence* (OC) layer, positioned below the transport layer in the IP stack, that bridges legacy applications and overlays. The OC layer is decomposed into the overlay-independent (OC-I) sublayer, which interacts with the legacy applications by presenting an IP-like interface, and the overlay-dependent sublayer (OC-D), which tunnels the traffic of applications over overlays. Splitting the OC layer allows us inter-operability across multiple overlays. To realize our design, we borrow many techniques and protocols from the literature, such as address virtualization [15, 26, 40, 42, 49, 50], DNS capture and rewriting [13, 29, 33, 49] and SSL [14].

The current design of OCALA focuses on *routing overlays* [2, 26, 36, 38, 41, 46], that is, overlays that offer an end-to-end packet delivery service similar to IP. While OCALA enables legacy applications to take advantage of most routing overlay functions such as anycast, mobility, QoS, route optimizations and middleboxes, currently it does not support some functions such as multicast. With multicast, hiding the fact that there are multiple receivers to the sender may require modifications to transport layer functions such as congestion control. OCALA is placed below the transport layer and thus cannot provide such functionality.

Our implementation of OCALA as a proxy requires no changes to applications or operating systems. We have implemented the OC-D sublayer for two overlays, $i3$ and RON, on Linux and Windows XP/2000 platforms. In our experience, we found the OC-D sublayer for both $i3$ and RON to be easy to implement. To illustrate the utility of our design, we have built and deployed a variety of applications, such as an intrusion-detection middlebox, overlay composition, secure Intranet access, and traversal of Network Address Translation (NAT) boxes. The ultimate goal of our implementation is, of course, to reach the end-users. The feedback we have received so far is encouraging especially in the case of the NAT traversal application.

The main contribution of this paper is the overall architecture and the implementation of this architecture as a proxy. To the best of our knowledge, this is the first solution that allows legacy applications running on the same machine to use different overlays at the same time, and allows hosts in different overlays to communicate with each other.

The rest of the paper is organized as follows. We present an overview of the architecture in Section 2 and a detailed goal-driven design in Section 3. The overlay-specific modules are presented in Section 4. We discuss the applications we developed in Section 5. We present implementation details in Section 6 and evaluation in Section 7. We finally present related work in Section 8, lessons from our initial deployment in Section 9, and conclude in Section 10.

2 Design Overview

We present the overlay model and a brief overview of the overlay convergence architecture.

2.1 Overlay Model

In this paper, we focus on *routing overlays*, *i.e.,* overlays that offer a service model of end-to-end packet delivery similar to IP, as opposed to overlays that store data (*e.g.,* Oceanstore [23]). Examples of routing overlays include those that aim to improve Internet routing such as RON [2], Detour [36], OverQoS [41], overlays that provide diverse functionality such as mobility [26, 49, 50], multicast [17, 19] or bridging multiple IP address spaces [5, 29], as well as recent network architectures such as Delay Tolerant Networks [10], $i3$ [38] and Delegation Oriented Architecture (DOA) [46].

Each end-host E in an overlay has an overlay-specific identifier (ID), which is used by other end-hosts to contact E through the overlay. While in the simplest case an overlay ID can be the host's IP address (*e.g.,* RON), many overlays use other forms of identifiers (*e.g.,* $i3$ and DOA use flat IDs). Since overlays IDs may not be human-readable, end-hosts may also be assigned names for convenience.

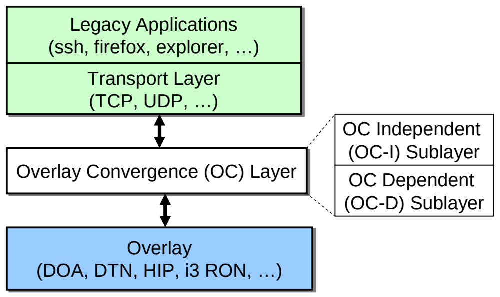


Figure 1: The overlay convergence (OC) layer.

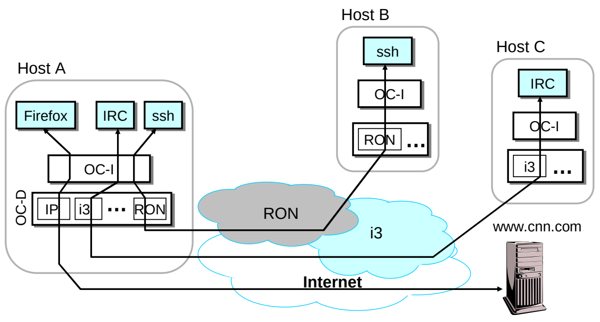


Figure 2: Three applications on host (A) which establish connections via IP and two overlays: RON and $i3$.

2.2 Overlay Convergence Layer

Conceptually our solution interposes a layer, called the *overlay convergence (OC) layer*, between the transport layer and the overlay network layer (see Figure 1). The OC layer replaces the IP layer in the Internet's protocol stack. It consists of two sublayers: an overlay independent (OC-I) sublayer, and an overlay dependent (OC-D) sublayer.

The main functions of the OC-I sublayer are to present a consistent IP-like interface to legacy applications and to multiplex/demultiplex traffic between such applications and various overlays. Functions common to all overlays, such as authentication and encryption, are implemented in the OC-I sublayer.

The OC-D sublayer consists of modules for various overlays, each of which is responsible for setting up overlay-specific state and for sending/receiving packets over the particular overlay. For example, in $i3$, the OC-D sublayer inserts and maintains private triggers at both end-points, while in OverQoS, it performs resource reservation. Note that IP can be viewed as a "default" overlay module.

Such a design allows a single host simultaneous access to various overlays. Figure 2 shows an example in which three applications on host A open connections via IP and two overlays: a web browser (Firefox) uses IP to connect to a CNN server, a chat client (IRC) uses $i3$ to preserve its anonymity, and `ssh` uses RON for improved resilience.

The OC-I layer also allows hosts in different overlays to communicate with each other. Figure 3 shows how two hosts on different overlays can communicate using an intermediate host (B), called a *gateway*, that resides on both networks.

We refer to the communication channel between two end-hosts at the OC-I layer as a *path*, and to the communication channel between two end-hosts at the OC-D layer as a *tunnel*. In the example in Figure 3 the path between the two end-hosts is (A, B, C), and this path consists of two tunnels at the OC-D layer, (A, B) and (B, C), respectively.

Analogous to the network layer that bridges multiple link layer domains in an IP network, the OC-I layer in our architecture bridges multiple OC-D layers together. Indeed, the use of OC-I gateways to connect hosts belonging to different overlays is similar to the way IP routers (gateways) connect hosts belonging to different link layer domains.

3 Detailed Architecture

In this section, we present a goal-driven description of OCALA, by showing how our design achieves our four goals: (1) transparency, (2) inter-operability, (3) exposing overlay functionality, and (4) security. Achieving these design goals is challenging as they have conflicting requirements. For instance, on one hand, we want to expose the rich functionality provided by overlays to users, while on the other, we have to preserve the narrow IP interface exposed to the legacy applications. In our design, we aim to find a sweet spot in achieving these opposing goals.

3.1 Goal 1: Transparency

The basic goal in our system, as with any system that provides support for legacy applications, is to ensure that legacy applications are oblivious to the existence of overlays. Ideally, such applications should work without any changes or re-configuration when the IP layer is replaced by the OC layer.

Our design is fundamentally constrained by how a legacy application interacts with the external world. Most legacy applications make a DNS request, and then send/receive IP packets to/from the IP address returned in the DNS reply. Hence, there are two possibilities for specifying which overlay should handle application traffic, if any: (a) using the fields in the IP headers such as IP addresses and port numbers, or (b) DNS-like names.

In the first approach, a user can specify rules on how packets should be processed using the fields in the IP header. For example, the user can employ a configuration file to specify that packets sent to address `64.236.24.4` and port number `80` should

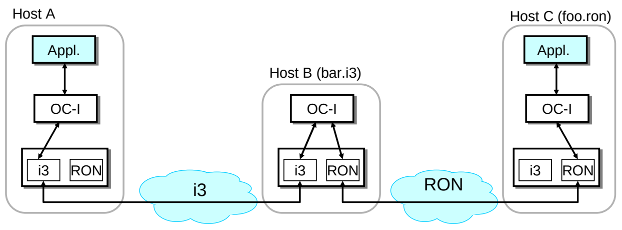


Figure 3: Bridging multiple overlays.

be forwarded through RON, while packets sent to `207.188.7.x` should be forwarded through OverQoS. RON is an example of overlay that uses this approach.

In the second approach, users can encode which overlay should handle the application's traffic in the DNS requests. We assume that each overlay host has a unique name, called *overlay name*, of the form *foo.ov*, where *ov* specifies the overlay, and *foo* is a name unique to that overlay. On receiving a DNS request from the application for an overlay name, the OC layer sets up state which allows it to intercept and forward all the subsequent packets from the application to host *foo.ov* through overlay *ov*.

The main advantage of relying solely on the information in the IP headers is that it works with *all* Internet applications, since at the very least, any application sends and receives IP packets. On the other hand, using DNS names has several advantages. First, DNS names can be used to identify hosts without a routable IP address such as NATed hosts. This property is fundamental to overlays that bridge multiple address spaces [8, 29]. Second, names are human-readable and hence easier to remember and use. Third, the user does not need to know the IP address of the destination in *advance*, which is not feasible in some cases. Indeed, when an overlay provides support for content replication, the IP address of the server that ultimately serves the content may not be known to the users before they actually run the application.

Since the vast majority of popular applications use DNS names anyway, in our implementation, we specify how the packets should be handled at the OC layer using *DNS names*.[1]

3.1.1 *Control Plane: Path Setup*

In this section, we describe the operations performed by the OC layer when it receives a DNS request. The final result of these operations is to establish an end-to-end *path* at the OC-I layer and to set up the state required to handle the application's traffic. While in general a path

[1]Another solution that we plan to investigate in the future is to intercept the first packet of an application and then prompt the user to specify how the application's traffic should be handled.

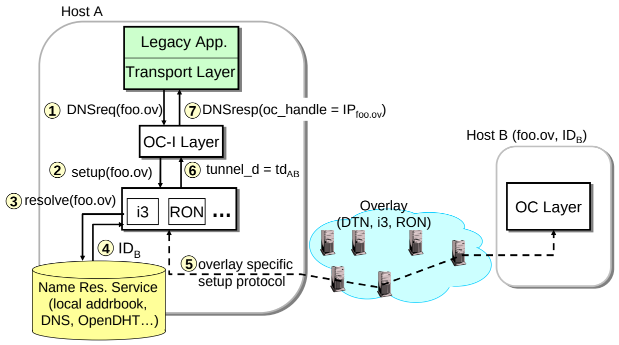


Figure 4: Tunnel setup protocol.

consists of several *tunnels* at the OC-D layer, in this section we consider a single-tunnel path. We generalize the description to multi-tunnel paths in Section 3.2.

Consider a legacy application on host A that wants to communicate with a remote legacy application at host B, called *foo.ov* (see Figure 4). The application first issues a DNS request for *foo.ov*, which is intercepted by the OC-I sublayer. On receiving such a request, the OC-I layer associates a locally unique *path descriptor*, pd_{AB}, and store the mapping between the name and the descriptor $(foo.ov \rightarrow pd_{AB})$.

The OC-I sublayer then invokes the corresponding module in the OC-D sublayer to setup a tunnel to *foo.ov* through overlay *ov*. This tunnel can be then used to relay the traffic of all local applications that communicate with *foo.ov*.

In turn, the OC-D sublayer invokes a resolution service to obtain the overlay ID (ID_B) of *foo.ov*. Examples of resolution services are DNS (used in RON), OpenDHT [20] (used in DOA), or using a local address book (used in $i3$). After the OC-D sublayer resolves the name, it instantiates the necessary state for communicating with *foo.ov*, and returns a pointer to this state, the *tunnel descriptor*, td_{BA}, to OC-I. For example, in $i3$, the setup phase involves negotiating a pair of private triggers with the remote end-host, and instantiating the mapping state between *foo.ov* and the private trigger IDs. The path and tunnel descriptors represent the state handles at the OC-I and OC-D layers respectively; path descriptors are needed since multiple paths might share the same tunnels.

On receiving the tunnel descriptor td_{AB} from OC-D, the OC-I sublayer stores the mapping $(pd_{AB} \rightarrow td_{AB})$, and returns an *OC handle* (oc_handle) to the legacy application in the form of a local scope IP address, IP_{AB}. To maintain compatibility with IP, IP_{AB} belongs to an unallocated address space (*e.g.*, `1.x.x.x` [18]).

Similarly, remote host B allocates a descriptor for the tunnel at the OC-D sublayer (td_{BA}), and a descriptor (pd_{BA}) and an OC handle (IP_{BA}). Figure 5 shows the

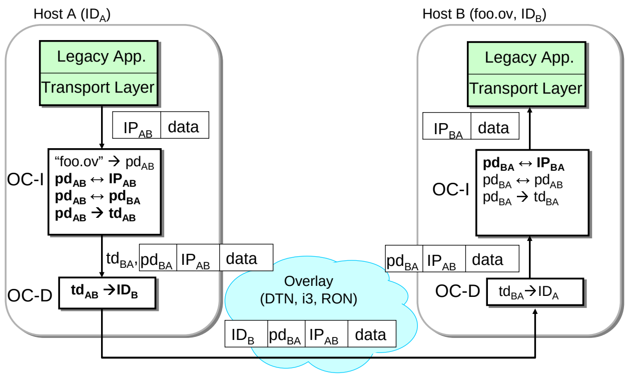


Figure 5: Forwarding a data packet from host A to B. The mappings used to modify the packet are in bold.

state instantiated at both hosts A and B during the setup protocol.

3.1.2 Data Plane: Packet Forwarding

The application at host A addresses packets destined to *foo.ov* to IP_{AB}, the OC handle returned by the OC-I sublayer (see Figure 5). The OC-I sublayer retrieves the state associated with this handle, and appends the path descriptor of the destination pd_{BA} to the packet, before handing it off to the OC-D layer to be sent over tunnel td_{AB}. The OC-D sublayer, using its tunnel state, sends the packets to *foo.ov* using the overlay identifier, ID_B. At the destination, the packet is eventually handed to the OC-I sublayer, which uses the path descriptor in the header to demultiplex the packet. Before sending the packet to the application, the OC-I sublayer rewrites the destination address of the packet to IP_{BA}, the local OC handle corresponding to the path from A to B.

As evident from this description, the constraint imposed by supporting unmodified legacy applications leaves us with little choice but to overload the semantics of application-level names and IP addresses. We discuss the limitations of overloading names and addresses on transparency in Section 3.5.

3.2 Goal 2: Inter-operability

When multiple routing overlays are deployed, a potential undesirable side-effect is that hosts in different overlays may not be able to reach one another. For example, $i3$ allows NATed hosts to act as servers, but such servers will be unreachable through RON. Even in the Internet today, hosts in different IP address spaces cannot communicate with one another [29]. Moreover, it is likely that some of the Internet hosts will not participate in routing overlays. For instance, it might be very hard to convince CNN to join some routing overlay or to deploy the OC-I layer on their servers. Inter-operability with such legacy hosts is also important.

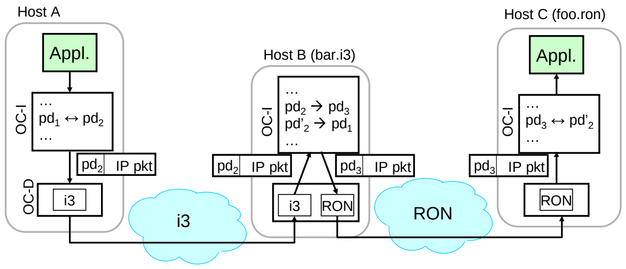


Figure 6: Forwarding a data packet from host A to C via gateway B (at the OC-I layer).

Our architecture can naturally provide inter-operability at the OC-I layer. Any node implementing the OC-I layer can act as a *gateway* between different overlays in the same way an IP router acts as a gateway between multiple links. In addition, we can provide inter-operability between overlay and legacy hosts by designing special OC-D modules that send and receive legacy IP traffic.

3.2.1 Bridging Multiple Overlays

Consider a host *A* in the $i3$ overlay that wishes to contact a host C in the RON overlay (See Figure 3). We can achieve this by deploying a host (gateway) B that resides in both $i3$ and RON, and which runs the OC-D modules for both overlays. Host *A* can then setup a two-hop path to *C* by using the gateway as an intermediate hop. In the case of a multi-hop path, the setup protocol creates tunnels between consecutive hops, and sets up the routing state at the OC-I layer of the intermediate hops. We give the details of the protocol next.

Assume that the overlay name of host C is *foo.ron*, while the overlay name of gateway B in $i3$ is *bar.i3*. To communicate with host C, an application at host A issues a DNS request for *foo.ron_bar.i3*. Note that we use the underscore character "_" to separate the host names along the path[2] , and list these names in the reverse order. On receiving the DNS request, the OC-D layer at host A resolves the last name that appear in the DNS name, *bar.i3*, to host B, and opens a tunnel to host B. This operation is identical to the tunnel setup in Section 3.1.1. Once this tunnel is setup, the OC-I at A asks its peer at B to setup the rest of the path to the destination C recursively. Hence, B will setup its own tunnel to C.

At the end of the setup protocol, hosts A and C maintain the path descriptors pd_1 and pd_3 respectively, while the gateway C maintains *two* descriptors pd_2, pd'_2, one for each direction of the path. The corresponding mappings are as follows: host A maintains the

[2]According to the DNS syntax "_" cannot appear in the host names; it can only appear in the domain names.

mapping $(pd_1 \rightarrow pd_2)$, host B maintains the mappings $(pd_2 \rightarrow pd_3, pd_2' \rightarrow pd_3)$, and host C maintains the mapping $(pd_3 \rightarrow pd_1)$ (see Figure 6).

On intercepting a packet sent to pd_1, the OC-I layer at host A appends pd_2 to the packet's header (see Figure 6). When host B receives this packet, it replaces pd_2 with pd_3 and forwards it to host C using the tunnel between these two hosts. Thus, the routing at the OC-I layer is similar to the label-switching protocol used in MPLS [34]. Also, note that a tunnel can belong to more than one path. For instance, two paths (A,C,B) and (A,C,D) can share the same tunnel from A to C.

3.2.2 *Legacy Gateways*

Legacy gateways are similar to overlay gateways except that one of the tunnels is over IP to a legacy host that does not participate in any overlay natively and does not run the OC-I layer. Thus, overlay functionality, such as improved routing, will be available only on the tunnel established over the overlay (between an overlay host and the gateway).

Legacy server gateway. The legacy server (LS) gateway allows an overlay-enabled client to contact a legacy server (see Figure 7(a)). Functionally, the LS gateway runs a OC-I layer over an OC-D module (say $i3$) and a special OC-D module called *LegacyServerIP* (or LSIP). The setup protocol is similar to that for an overlay gateway. Consider a overlay host connecting to *cnn.com* through the LS gateway. The OC-I layer at the LS gateway forwards such setup requests to the LSIP module. The LSIP module now behaves like a NAT box with respect to the server. It first resolves the name *cnn.com* through DNS and allocates a local port for this tunnel. Packets sent to the server are rewritten by changing the source address to that of the LS gateway, and altering the source port to be the allocated local port. The local port is then used to multiplex incoming packets, which are then sent to the OC-I layer with the appropriate handle.

Legacy client gateway. The legacy client (LC) gateway allows overlay servers to offer their services to legacy clients (see Figure 7(b)). Functionally, the LC gateway runs a OC-I layer over an OC-D module (say $i3$) and a special OC-D module called *LegacyClientIP* (or LCIP). In addition, the client is configured to use the LC gateway as its DNS server. The LCIP module thus intercepts DNS queries from the client, and dispatches them to the OC-I layer which initiates a tunnel over the overlay. The LCIP module also sends a DNS reply with a Internet *routable* address to the client, captures packets sent by the legacy client to that address, and sends them over the overlay. Any client can now contact the machine *foo.i3* from any machine provided her DNS server is set to the address of the LC gateway. The design of our LC gateway is similar to that of AVES [29].

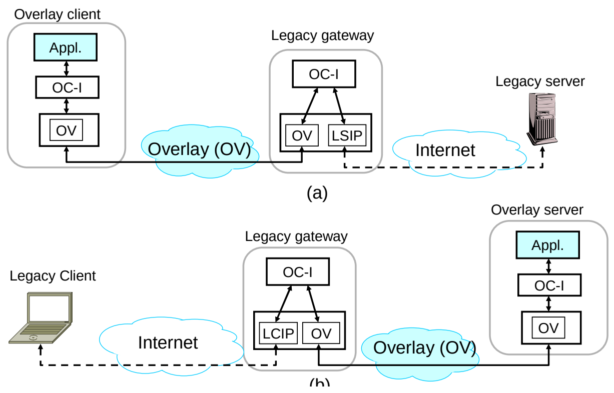


Figure 7: (a) An overlay client connecting to a legacy server. (b) A legacy client connecting to an overlay server.

Note that legacy gateways have two different OC-D modules LSIP and LCIP to interface with legacy servers and clients, unlike an overlay gateway where a single module can be used to contact both overlay servers and clients. In the case of the legacy client gateway, the fact that the addresses returned by the gateway should be routable considerably limits the number of clients that can connect simultaneously [29].[3]

3.3 Goal 3: Customize Overlay Functionality

One of the main goals of overlays is to allow users to customize the functions that they provide. For example, RON allows users to choose the metric based on which the paths are optimized, OverQoS allows users to specify QoS parameters, and architectures like $i3$ and DOA allow users to explicitly interpose middleboxes on the path. These examples illustrate that for better flexibility, users should be able to customize the preferences for each tunnel along a path. Preferences of interest include both overlay-specific options (*e.g.*, use latency optimized paths for RON or use a specific middlebox) and overlay-independent options (*e.g.*, identity of gateways, perform end-to-end authentication).

Given the limited options that a legacy application has to communicate its preferences to the OC layer, we have little choice but to encode them in the DNS name as well. In particular, an overlay name of a host can also encode the preferences of the tunnel to that host. For instance, with OverQoS, a user can make an *ssh* connection to *foo.delay50ms.overqos* instead of *foo.overqos*.

The encoding of tunnel preferences are overlay-specific: OC-I looks only at the name's suffix, *overqos*, to identify the OC-D module to which it needs to forward the setup request; it is the role of the OverQoS

[3]If we were to support only HTTP traffi c, we could remove this limitation by having gateways use the DNS names in the HTTP requests to demultiplex the clients' traffi c.

OC-D module to parse the prefix, *foo.delay50ms*. An overlay is free to encode its preferences any way it wants as long as it does not include "_" (recall this character is used to separate different overlay names). Furthermore, there is an one-to-one mapping between an overlay name and a tunnel. If a user makes a connection to *foo.delay50ms.overqos* and another one to *foo.bandwidth1Mbps.overqos*, the OC-D layer will create two tunnels to *foo*, one for each name.

To encode overlay-independent preferences, we use a special domain name *oci*. Consider the example in Figure 3. A user on host A who wishes to connect to host C using the *telnet* application can use DNS name *bar.shortcut.i3_foo.mindelay.ron_encrypt.oci*. Options *shortcut* and *mindelay* are specific to $i3$ and RON respectively, and are used to optimize latency in each overlay. The option *encrypt.oci* requests the OC-I layer to provide a encrypted channel to the end-host (*foo.i3*), thus making *telnet* secure.

While using DNS names to encode preferences allows users to choose preferences at run-time, this flexibility does not come for free. The DNS names are limited to 255 characters, and typing a long list of preferences may be inconvenient. Furthermore, the user may use the same preferences for many applications, in which case it makes little sense to type the same preferences over and over again. A configuration file that contains preferences for names based on regular expressions is an alternative we allow that trades off flexibility for ease of use. For example, a user that wishes to send all traffic to the domain *company1.com* through a middle-box $i3$ M can specify the rule: **.company1.com* ⇒ **.company1.com_M.i3* .

Another advantage that configuration files have over DNS names is that some legacy applications may perform application-specific interpretation of DNS names which may interfere with our mechanism of encoding preferences. For example, the HTTP protocol interprets DNS names as part of a URL and thus includes domain names in its "GET" request. Consider the case when a browser connects to *cnn.com.ip_legacygateway.i3* to contact CNN over $i3$. The domain name in the URL of the "GET" request does not match *cnn.com*, so the http server would return a error. In this case, a configuration rule that specifies that all DNS names of the form **.cnn.com* should be sent via the legacy gateway, would avoid this problem. This also ensures that various elements, such as images, that are specified as URLs in the html page, will also be downloaded through the same tunnel used to retrieve the web page.

On receiving a setup request for an overlay name, the OC-D sublayer reads the preferences associated with the name (if any) from the configuration file, before proceeding with the setup operation. In processing overlay-specific preferences for setting up a middlebox, the OC-D module layer may need to communicate through the OC-I layer to a middlebox application. We now describe how this is achieved.

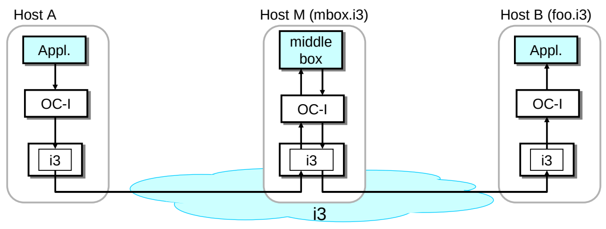


Figure 8: Interfacing a middlebox.

3.3.1 *Interfacing Middlebox Applications*

Several new network architectures [38, 46] provide support for middleboxes, by allowing both the sender and the receiver to *explicitly* insert middleboxes on the data path. We briefly describe how we support this functionality.

Consider the case of a sender-imposed middlebox where a host A wishes to contact a host B through a middlebox M (see Figure 8). The only difference from the operation of a gateway is that the middlebox module (say, a transcoder) running at M should be allowed to perform arbitrary transformations on the data sent by one end-point before forwarding it to the other. In our design, we achieve this by requiring the middle-box module to implement a routing overlay interface (similar to a OC-D). This interface is used by the OC-I layer to send and receive packets from the middlebox module. We use a configuration file at M to specify to the OC-I layer that communication to B ($foo.i3$) should be routed through the middlebox module. The protocol when the middlebox is imposed by the receiver is similar.

3.4 Goal 4: Security

Since security is important for many applications, we provide basic security mechanisms at the OC-I sublayer, rather than leaving each overlay to implement these mechanisms in their OC-D module. In particular, the OC-I sublayer offers the options of *authentication* and *encryption*, both of which operate agnostic of the overlay used for the traffic. The OC-I layer's security mechanism is based on human-readable names and operates independently of the resolution mechanism employed by the overlay. Note that since overlay names are OC-D specific, the OC-I has to request the overlay names it needs to authenticate from the OC-D layer.

Our security model assumes the existence of certification and name allocation authority from whom users can obtain certificates associating their overlay name to their public key. Note that such a centralized authority is necessary for any human-readable and secure naming scheme [48]. It is easy to extend our model to hierarchical name allocation schemes.

The security protocol for authenticating a host by name is very similar to the Secure Sockets Layer protocol (SSL) [14] which relies on certificate authorities like VeriSign. Any host that knows the public key of the certification authority can verify these certificates. The protocol first verifies the purported certificate of the receiver and then sets up symmetric keys. If the user has requested for encryption, all data packets are encrypted using these symmetric keys. Our security protocol supports both sender and receiver authentication.

The main constraint in supporting end-to-end authentication through a middlebox is that the middlebox application needs to operate on unencrypted data. Protocols like SSL cannot be used since the end-hosts only trust each other and do not trust the middlebox; hence, we implemented our custom security protocol instead of reusing SSL. The end-point A that interposes the middlebox certifies the public key of the middlebox, thus delegating its authority to the middlebox. This process is repeated over all intermediate hops and thus the other end-point B obtains a *chain of certificates*. It then verifies this chain by the name of A, and then uses the public key of its next hop, the middlebox, to setup symmetric keys. Thus, different sets of symmetric keys are used for the tunnels from each point to the middlebox. The OC-I layer at the middlebox performs decryption/encryption using these symmetric keys before passing packets to/from the middlebox application. This ensures that the middlebox application only operates on unencrypted data.

3.5 Limitations

The primary goal of our design is to achieve transparency for legacy applications while providing complete access to overlay functions. We review how well our design meets this goal.

3.5.1 *Access to Overlay Functions*

While the OC layer enables legacy applications to take advantage of most overlay functions such as anycast, mobility, QoS, route optimizations and middleboxes, currently it does not support some functions such as multicast. The main problem with multicast is that hiding the fact that there are multiple receivers to the sender requires modifications to transport layer functions such as congestion control. OCALA is placed below the transport layer and thus cannot provide such functionality.

3.5.2 *Transparency*

The OC-I layer overloads IP addresses and DNS names in two ways that may break the assumptions made by legacy applications. In contrast to current IP, the scope of IP addresses returned by the OC-I layer to applications is local. DNS names, in our design, can specify preferences and are not resolved using the global DNS infrastructure. We now discuss the implications that these modifications have on transparency.

First, the use of local scope addresses implies that addresses returned to legacy applications may not be valid at other hosts. Second, applications like *ftp* that encode addresses in data packets will potentially not work since the OC-I layer performs IP header rewriting before delivering packets to the application. Our implementation avoids address rewriting to some extent by negotiating the local addresses at the OC-I layer, a technique borrowed from [49]. However, for legacy gateways, address rewriting cannot be avoided. Finally, applications that do not use DNS requests are not supported. While technically, such applications can be handled by triggering control plane operation when the *first* data packet is sent, this solution would only work with overlays in which hosts are assigned IP routable addresses.

Local-scope addresses have been used before in several contexts—supporting mobility [40, 42, 49], redirection [15], process migration [39, 40] and availability [39]—and their limitations and workarounds are well-known [49]. In supporting overlays where end-hosts may not even have routable IP addresses, we are left with little choice but to work around the limitations of local-scope addresses.

4 The Overlay Dependent Layer

In this section, we describe the implementation of the OC-D module for two routing overlays: $i3$ [38] and RON [2]. This description serves not only as a validation of our architecture but also as a blueprint for implementing OC-D modules for other overlays. We begin by presenting the interface that has to be exported by a OC-D module to the OC-I sublayer and then discuss the $i3$ and RON modules.

4.1 OC-D Sublayer API

Table 1 shows the basic API functions that every OC-D module needs to implement and expose to the OC-I sublayer. For the simplicity of exposition, we omit the error related and the overlay name related functions here.

Function calls: OC-I → OC-D	
`setup`(*path_info, path_d*)	setup path (*path_d*) using names/preferences in *path_info*
`close`(*tunnel_d*)	close tunnel
`send`(*tunnel_d, IP_pkt*)	send IP packet via tunnel
Callbacks: OC-D → OC-I	
`setup_done`(*path_d, tunnel_d*)	callback invoked when tunnel ($tunnel\_d$) was established
`recv`(*path_d, IP_pkt*)	receive IP packet from tunnel

Table 1: OC-D sublayer API.

The basic API consists of three functions and two callbacks. The `setup` function sets up a tunnel between the local host and a remote host according to the user's preferences. The user preferences and the overlay name of the remote host are contained in the *path_info* field. In general, this field can contain a path (source route), where intermediate hops can be middleboxes or gateways bridging two overlay networks. The *path_d* field represents the path descriptor at the OC-I sublayer. Once the OC-D sublayer creates the tunnel it returns the tunnel descriptor (*tunnel_d*) to the OC-I layer using callback *setup_done*. The `close` function call is invoked by the OC-I sublayer to close the specified tunnel. This function is usually called when the path's state at the OC-I sublayer expires. We discuss the timeout values for this state in the context of our implementation in Section 6.1.1.

The `send` function call, invoked by the OC-I sublayer, includes a handle to the OC-D's state for that tunnel and the packet itself. The `recv` call, invoked by a OC-D module to the OC-I sublayer, has the OC-I path handle for the path and the received packet itself.

4.2 The RON Module

RON aims to improve the resilience of the Internet by using alternate routes in the overlay [2]. RON offers an interface similar to IP, and not surprisingly, it requires little effort to implement the OC-D module for RON. RON uses IP addresses and DNS names as overlay IDs and overlay names, respectively. The name resolution is thus implemented by simply querying the DNS infrastructure.

When the OC-I sublayer asks the RON module to setup a connection to a RON host (identified by a name such as *foo.ron*), the RON module first decodes the name to obtain the DNS name of the end-host and the preferences expressed in the name. This DNS name is then resolved using DNS to obtain an IP address. The RON module then sets up state associating the preferences and the destination IP address with the tunnel and passes its handle to the OC-I sublayer.

Data plane operations involve simple encapsulation and decapsulation. When sending packets, the RON module retrieves the tunnel preferences using the handle passed by the OC-I sublayer, encapsulates the packet, and sends it over RON. On receiving a packet, the RON module decapsulates the packet and passes it to the OC-I sublayer.

Our solution offers some advantages over the RON-specific solution for supporting legacy applications. First, our solution is more flexible as it allows users to specify overlay names and preferences using DNS names. In contrast, in the RON specific solution, the IP addresses of all RON machines have to be statically configured. Second, our solution allows stitching together multiple RON overlays.

4.3 The $i3$ Module

$i3$ is a new network architecture, implemented as an overlay. In a nutshell, $i3$ uses a rendezvous-based communication abstraction to support services like mobility, multicast, anycast and service composition using middleboxes. We now describe how the $i3$ module works when host A contacts host B through $i3$. For a detailed description of $i3$ the reader is referred to [38].

On receiving the `setup` request for *B.i3* from the OC-I sublayer, the $i3$ module at A first resolves the name to a $256-$bit $i3$ identifier by using implicit mapping: the identifier of a host is derived by simply hashing its name. The identifier obtained by hashing *B.i3* corresponds to B's public trigger identifier id_B. Thus, $i3$ does not require any resolution infrastructure; only a name allocation and certification authority is required, and the security options provided by the OC-I sublayer can provide authentication and security on top of the implicit mapping.

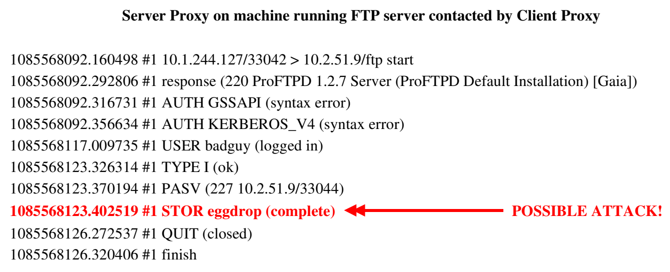
After the name is resolved, the $i3$ module at A initiates private trigger negotiation by contacting host B through its public trigger $[id_B|B]$. Both hosts exchange a pair of private triggers $[id_{AB}|A]$ and $[id_{BA}|B]$, respectively, after when they communicate exclusively through these triggers: host A sends packets to host B using ID id_{BA}, and host B sends packets to A using ID id_{AB}. Once the control protocol sets up the state, the $i3$ module sends packets captured from the application by encapsulating the payload with $i3$ headers that include private triggers identifying the flow. Received packets are decapsulated before delivering to the OC-I module.

The $i3$ OC-D module also allows receiver-imposed middleboxes by using $i3$'s stack of IDs. A $i3$ host B that wishes to impose the middlebox on all connecting hosts inserts a public trigger of the form $[id_B|(id_M, B)]$. When a client A sends a trigger negotiation request via this public trigger id_B, the $i3$ overlay delivers it to M along with the stack (id_M, B). The $i3$ OC-D module thus obtains the identity of the next hop and proceeds to setup the tunnel to B through its OC-I sublayer.

5 Applications

In this section, we discuss some interesting application scenarios in which legacy applications can benefit from $i3$ and RON functionality. We have implemented these applications by leveraging the $i3$ and RON OC-D modules.

- *Middlebox applications*: The OC layer explicitly supports middleboxes in the path, and as an example, we show how a user can redirect legacy traffic through a intrusion detection system running on a remote machine.
- *Overlay Composition:* We used the ability of the OC-I layer to bridge multiple overlays to improve



Server Proxy on machine running FTP server contacted by Client Proxy

```
1085568092.160498 #1 10.1.244.127/33042 > 10.2.51.9/ftp start
1085568092.292806 #1 response (220 ProFTPD 1.2.7 Server (ProFTPD Default Installation) [Gaia])
1085568092.316731 #1 AUTH GSSAPI (syntax error)
1085568092.356634 #1 AUTH KERBEROS_V4 (syntax error)
1085568117.009735 #1 USER badguy (logged in)
1085568123.326314 #1 TYPE I (ok)
1085568123.370194 #1 PASV (227 10.2.51.9/33044)
1085568123.402519 #1 STOR eggdrop (complete) <----------- POSSIBLE ATTACK!
1085568126.272537 #1 QUIT (closed)
1085568126.320406 #1 finish
```

Figure 9: Analysis performed by Bro

wide-area performance for wireless clients.

- *Secure Intranet Access*: We also implemented a more flexible and secure version of Virtual Private Networks (VPNs) [44] by using the OC-I layer to contact legacy hosts over a overlay.
- *NAT Traversal*: Since $i3$ allows access to hosts behind NATs, our implementation of the OC-I layer along with the $i3$ OC-D module allows legacy applications to traverse NATs.

5.1 Middlebox applications

In conjunction with $i3$, our design gives complete control to hosts and applications over the middleboxes on their path. As an example of how this functionality might be used, we implemented an intrusion detection middlebox which can be used as a firewall by any overlay-enabled host. This setting allows users to redirect all their incoming and outgoing traffic through the firewall *independent* of where they are located. For instance, users could use their company firewall from an Internet cafe.

Implementing such a middlebox took less than 200 lines in our system. We wrote a *shim* middlebox module that uses Bro [30], a popular intrusion detection program. The OC-I layer is configured to relay packets through this shim layer, which in turns sends them to Bro. Note that Bro is itself a legacy application, and thus these packets should have valid IP headers. For this reason, the shim layer assigns virtual addresses (which can be considered as middlebox handles) to both end points, rewrites the IP header of packets appropriately, and then sends them to Bro. Thus, to Bro, communication between the remote hosts looks like a conversation between two virtual hosts, and it can perform stateful analysis (*e.g.,* TCP analysis by matching the data packets of a TCP connection with the corresponding acknowledgments).

Figure 9 shows an example of the analysis performed by Bro. Bro uses FTP traffic analysis to identify an attempt by `badguy` to upload a file called `eggdrop`, a well-known backdoor. This example also illustrates the local scope addresses in use (in this case, from the range $1.0.0.0/8$).

Although our solution allows commonly-used techniques such as signature-based attack detection, it does not permit Bro to use certain advanced detection techniques. For example, Bro has the ability to detect address-scanning by looking for several unsuccessful connection attempts to multiple hosts in a single network. This analysis uses IP addresses to identify hosts belong to the same network. However, the addresses seen by Bro are local-scope virtual addresses and give no information about the end-host's networks.

5.2 Overlay Composition

Overlay composition allows an application to explicitly stitch together different overlays. Apart from allowing inter-operability, this allows an user to merge the functionalities of multiple overlays in interesting ways. For example, a user who connects to the Internet through a wireless hop, may use $i3$ for uninterrupted communication while switching between various wireless networks. In addition, the user may also wish to optimize wide-area performance using RON. We achieve this by using $i3$ to connect to a close-by $i3$-to-RON gateway, which will then relay packets over a RON-optimized path.

5.3 Secure Intranet Access

Our implementation can be used without any change to achieve VPN-like functionality by offering a secure way of accessing corporate Intranets. The legacy server gateway runs inside the organization and hence has unrestricted access to all intranet hosts. External end-hosts relay packets through the legacy gateway in order to access Intranet machines. Authentication and encryption are important requirements in this scenario, and we can simply leverage the OC-I layer's security mechanisms for this purpose. Any routing overlay, including vanilla IP, can be used for communicating between the user's machine and the legacy gateway. Our solution has two advantages over VPN-based solutions:

Multiple intranets: In our solution, a client can access multiple intranets at the same time by specifying preferences based on DNS names, *e.g.,* connections to **.company1.com* can be relayed to the company1's gateway, those to **.company2.com* can be relayed to company2's gateway. With traditional VPNs, this may be impossible if both Intranets use the same address range.

Security: In traditional VPNs, an external client that is connected to an Intranet is assigned an IP address from the Intranet address space. As a result, a client infected by a scanning worm can potentially infect Intranet machines. Our solution makes such infection harder because an infected client cannot access any Intranet host directly using its real IP address. Only virtual addresses that have been allocated by the OC-I layer for ongoing connections are vulnerable. This is equivalent to having a white-list of allowed connections, which can be used to detect scans.

5.4 NAT Traversal

$i3$ allows access to machines behind NATs by its ability to bridge different address spaces. Hence, using the $i3$ module in conjunction with the OC-I layer, an user can run legacy servers behind NATs. This allows home users to access their machines from anywhere, and they can do so securely by simply remembering the human-readable name of their home machine. When persuading users to deploy our software, we found NAT traversal to be a very attractive feature from the users' point of view. Unlike other mechanisms proposed today for this purpose (*e.g.,* AVES [29], Hopster [16], DOA [46]), the $i3$ module allows *incoming* access *without* modifying NATs. It is also cleaner and simpler compared to mechanisms based on heuristics to predict NAT behavior such as NATBlaster [6].

6 Implementation

We have implemented the OC-I layer using C++. We have also implemented RON and $i3$ modules for the OC-D layer, along with all the applications described in Section 5 based on these modules. We plan to make our implementation publicly available soon in both source and binary format.

6.1 OC-I Implementation

We implemented the OC-I layer as a *user-level proxy* in order to avoid modifying the operating system stack. This user-level proxy uses a packet capture device, specifically `tun` [12, 45, 47], to interpose itself into the stack. DNS packets and packets sent to local-scope IP addresses are redirected to the `tun` device using *iptables* and *iproute*. The OC-I layer simply reads from the `tun` device to captures packets from the application, and writes packets to it in order to deliver them to the application.

We note that the `tun` device is available on other operating systems, such as Windows, FreeBSD, Solaris, and we have already ported our OC-I layer to Windows. However, our approach has the limitations that administrative privileges are required to use the `tun` device and that all users on the same machine need to share the same configuration; these can be avoided by a dynamic library based implementation.

6.1.1 Control Plane: State Maintenance

On intercepting a DNS request, the OC-I layer initializes state, such as path descriptors and security options, and communicates with its peer OC-I layer(s) to set up the path requested by the application. The setup protocol may also involve local-scope address negotiation and security negotiations. The former requires one roundtrip, while our security protocol requires two roundtrips due to exchange of certificates and nonces. The public keys associated with the certificates are 1024-bit RSA keys and 256-bit symmetric keys are exchanged during security negotiations. After negotiations are complete, the OC-I layer sends the DNS reply to the application. In order to prevent caching of such replies, our implementation sets the Time To Live (TTL) option to zero in the DNS reply sent to applications.

The OC-I layer refreshes the state associated to a DNS name N every time one of the following two events occur: a packet is forwarded using the state associated with N, or a DNS request for N is invoked. If none of these events happen, the OC-I layer removes the mapping state after a predefined interval of time TO. We use a timeout period of $TO = 7200s$ to deal with applications that cache replies beyond the TTL, *e.g.,* Internet Explorer. This timeout seems adequate in our deployment experience. The OC-I layer also periodically checkpoints its state on disk to enable failure recovery during an application session.

6.1.2 Data Plane: Packet Forwarding

Packets sent by the application are addressed to local-scope addresses returned by the OC-I layer. These addresses are allocated from the unused address range *1.0.0.0/8* for this purpose. The OC-I layer may also encrypt the packet before dispatching to the OC-D layer. The headers added by the various layers could lead to packet fragmentation. An application can avoid packet fragmentation by performing end-to-end MTU discovery. At the receiving end, the OC-I layer rewrites IP addresses before sending packets to the application and incrementally updates the IP and transport layer checksums [24]. We have implemented address rewriting for TCP, UDP and ICMP. Note that this rewriting occurs only if address negotiation fails.

6.1.3 Gateways

An OC-I layer can be explicitly configured to behave as a gateway, in which case it is willing to act as a relay for other hosts. Since we have implemented $i3$ and RON modules, the OC-I layer can bridge these two overlays. We also implemented the *LegacyServerIP* (LSIP) and the *LegacyClientIP* (LCIP) modules (see Section 3.2.2).

The LSIP module is based on a Linux software NAT implementation for the legacy server gateway, which includes packet-rewriting support for several applications such as `FTP`, `H.323`, `PPTP` and `SNMP`. The legacy server gateway does not support ICMP since there is no information in an ICMP packet (such as port numbers) to permit multiplexing of a single IP address among multiple hosts. The LCIP implementation is very similar to AVES [29], and thus we do not describe it here.

6.2 OC-D Implementation

An OC-D module is implemented as a C++ class which provides the required OC-I API. Implementing the $i3$ and

RON modules mainly involved supporting this API using the $i3$ and the MIT RON libraries. The RON OC-D module performs little more than translating between the OC-I API calls and the RON library calls. The $i3$ module, apart from this translation, supports a few optimizations to reduce control plane and data plane latency. We removed the roundtrip requirement for private trigger negotiation by piggybacking data packets on control plane setup protocol. On the data plane, the $i3$ OC-D module can request the $i3$ overlay to set up a *shortcut*, which enables a direct path to the remote end-host. Once a shortcut is established, the packets are no longer relied through the $i3$ infrastructure. Typically, shortcuts reduce the roundtrip latency and increase the throughput.

7 Evaluation

We first present micro-benchmarking results to evaluate different costs involved in the data and control path of the proxy. The benchmarking results indicate that the overhead of using the proxy is minimal. We then present wide-area experiments in some simple scenarios. Of course, the purpose of these experiments is only to show that the cost of performing packet capture and tunneling is not large, and that wide-area performance is still acceptable. The real benefit of our architecture and implementation should be gauged by the applications it enables, and eventually, the user acceptance it gains.

7.1 Micro-benchmarks

All our micro-benchmarks were conducted on a 2.4 GHz Pentium IV machine with 512 MB RAM running Linux $2.4.20$. Timing was done using `gettimeofday` at the user level. We report the timing numbers as a median of 100 runs. We used a simple in-house tool that sends data of various sizes and rates as the legacy client for the proxy. For conducting micro-benchmarks, we instrumented the proxy and the tool reporting the time at relevant checkpoints. We do not report on the microbenchmarking experiments from our Windows port of the proxy since cygwin does not have fine-grained timer implementation. Moreover, the cygwin Linux emulation layer also introduces additional overhead that is not fundamental to our proxy implementation.

Data Path Overhead. In comparison to a legacy application running over the host IP stack, the use of the proxy adds two memory copies of the data: from kernel to the user space and back, for both sending and receiving packets. Table 2 reports results for the send and receive times of a single packet of size 1200 bytes[4] for $i3$ and RON with and without encryption. We split up the total send and receive times into three phases: (a) time taken to move a packet between application and proxy (using tun), (b) overhead at the OC-I layer, and (c) overhead at the OC-D layer.

[4]We used this packet size in order to avoid fragmentation due to addition of headers.

Send	$i3$		RON	
(μs)	No-Encr	Encr	No-Encr	Encr
OC-I	19	93	18	91
OC-D	20	20	28	28
tun	24	25	24	24

Recv	$i3$		RON	
(μs)	No-Encr	Encr	No-Encr	Encr
OC-I	8	84	6	82
OC-D	44	43	36	35
`tun`	16	20	15	16

Table 2: Split-up of per-packet overhead of send (above) and recv (below) with the proxy. All numbers are in microseconds.

As expected, the processing time of the OC-I layer is independent of whether we use $i3$ or RON, and without security, the percentage of time spent at OC-I layer is not large (25% for send and 11% for receive) and rest of the overhead is for transferring the packet from the application to the proxy and OC-D processing. The processing time at the OC-D module for both $i3$ and RON are in the same ballpark, and as expected, almost independent of whether security is used or not. A dynamic library implementation can reduce the overhead of transferring the packet from the application to the proxy as mentioned before. The total processing time indicates that the raw throughput that can be sustained is about 15000 and 7000 packets per second (for 1200-byte packets) without and with encryption respectively.

Lookup Overhead. We now quantify the overhead incurred by the proxy in the control path (*i.e.*, name resolution). We distinguish between two cases when an application makes a DNS request: either the DNS name has already been resolved, or it is being resolved for the first time. In the first case, the proxy immediately returns the name with a minimal processing overhead of 15 microseconds. In the second case, the proxy performs additional operations to setup the state and hence takes longer (169 microseconds).

7.2 LAN Experiments

In order to study the effect of the proxy overhead on end-to-end behavior, we measured the latency and TCP throughput between two proxies communicating over a LAN in Table 3. In a LAN environment, the overhead of the proxy can be localized without wide-area artifacts affecting the measurements. We run the experiments between two clients communicating with each other across $i3$, RON and IP. In addition, we also run the experiments for $i3$ with *shortcut* enabled. Shortcut is an optimization that $i3$ developers have added to eliminate the in-efficiency of relaying packets through $i3$ servers in the data path.

Latency (ms)	$i3$	$i3$-shortcut	RON	IP
No-Encr	1.42	0.788	0.762	0.488
Encr	1.74	1.13	1.06	NA

Throughput (kbps)	$i3$	$i3$-shortcut	RON	IP
No-Encr	9589	10504	10022	11749
Encr	5415	5615	5445	NA

Table 3: LAN experiments for latency and throughput.

The latency results are consistent with the earlier micro-benchmarking results. Latencies in the case of $i3$-shortcut and RON are similar, and are about a couple of hundred microseconds larger than IP latency. However, since the LAN latencies are themselves very small, even a single waypoint causes significant relative increase in latency.

The throughput results also indicate that the performance hit using the overlays ($i3$-shortcuts and RON) is only about 10%. The throughput and latency of RON is not better than IP since the setup of the experiment was meant to study overhead of the implementation and hence there weren't many diverse paths with better throughput than IP. Since the $i3$ servers were also located on the same LAN, sending the packets through an $i3$ server also did not cause a significant throughput degradation.

7.3 Wide-area Experiments

For completeness, we performed wide-area experiments with the proxy running on three machines at different locations A, B, and C. The fact that the OCALA proxy requires administrator privileges made it difficult for us to procure more machines. We measured the latency and throughput of the wide-area paths between these three machines. The latency is measured using `ping`, and the throughput using `ttcp`, a popular tool for measuring TCP throughput.

Figure 10 shows the latency and throughput results using the proxy over wide-area for the following configurations of the overlay: (a) $i3$, (b) $i3$-shortcuts, where hosts circumvent $i3$ for data packets, (c) RON (d) direct IP. The RON and $i3$ overlays were run on PlanetLab. In all the experiments, the OC-D $i3$ module on the end-host was configured to use the closest $i3$ server, while for RON, the end-host itself acts as a RON node.

Each latency value shown in Figure 10(a) represents the median of 100 measurements. As expected, the latencies in the case of $i3$-shortcuts and IP are virtually identical, as in both cases the packets follow the direct IP path between the end-points. In contrast, the latency in

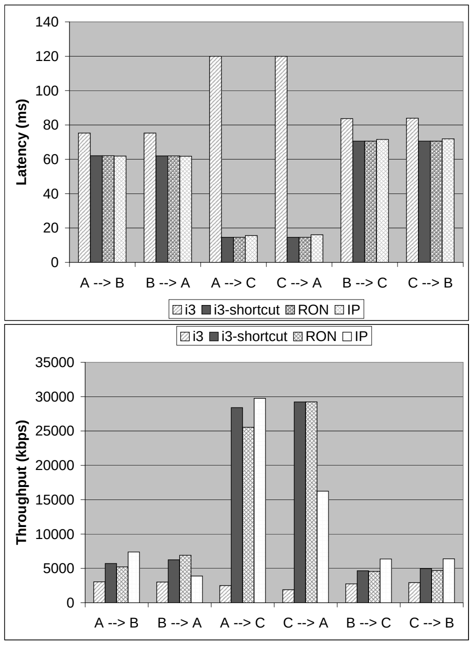


Figure 10: Wide-area experiments: (a) latency (b) throughput.

the case of $i3$ is much larger as in this case the packets are forwarded via an intermediate $i3$ hop. Finally, although we configured RON to choose latency-optimized paths, we observed no significant improvements in latency as compared to the direct IP path.

Figure 10(b) shows the throughput for the same scenarios; each value represents the median over 10 measurements. Again, $i3$ performs consistently worse than the direct IP path. In particular, in scenarios $A \rightarrow C$ and $C \rightarrow A$, the $i3$ throughput is over one order of magnitude lower than the direct IP path. We speculate that these large differences are because of the throughput limitations of the PlanetLab nodes at the time of our measurements. Such effects are also reflected (although in a much smaller degree) in the RON results despite configuring RON to use bandwidth-optimzed paths. Finally, in two cases where the destination was host A, the IP throughput was lower than that of RON and $i3$-shortcuts. We attribute this unexpected phenomenon to be due to rate-limiting of incoming TCP connections at A. (Recall that the proxy tunnels TCP connections over UDP.)

8 Related Work

The problem of supporting legacy applications over non-IP or IP-modified communication infrastructures has been addressed in a variety of contexts. Examples include overlay networks and new network architectures (*e.g.*, RON [2], ROAM [50], HIP [26], DOA [46], WRAP [5]), end-host support for mobility [40, 42, 49]), and mechanisms to allow end-hosts to use overlays without participating in them [28]. However, all these proposals are domain-specific, and as a result they do not support inter-operability across different overlays. In contrast, our architecture not only allows a user to access different overlays simultaneously, but also allows hosts in different overlays to communicate with each other.

Architecturally, many of the previous solutions (*e.g.,* HIP [26], WRAP [5]) interpose a shim layer, similar to the OC layer, between the transport and the network layer. Our architecture is different from these proposals in that it explicitly splits this layer into an overlay independent sublayer that acts as traditional network layer, and an overlay dependent that acts as a traditional link layer. This division is the key which allows us to provide inter-operability across multiple different overlays.

Our goal of stitching together multiple overlays resembles the goal of AVES [29] and TRIAD[5] [8] to stitch together multiple IP networks, such as NATed realms. Another proposal, UIP [11] goes one step farther by providing uniform connectivity not only across IP networks, but also across non-IP networks, such as ad-hoc networks. All these proposals focus on providing universal connectivity. In contrast, we focus equally on exposing the overlay functionality to users, functionality that often goes beyond connectivity. Our work can be viewed as a generalization of AVES and TRIAD, as in our architecture an IP network is a particular instance of an overlay. Compared with UIP, we use names to uniquely identify overlay hosts instead of globally unique identifiers.

In realizing our architecture, we rely on techniques and protocols previously proposed in a variety of contexts. The technique of intercepting DNS requests for the purpose of interposing a proxy has been used in AVES [29], Coral [13], and for improving web browsing performance over wireless networks [33]. Local-scope addresses has been proposed in the context of supporting mobility [26,40,42,49], redirection [15], process migration [39,40] and server availability [39]. Our address-negotiation protocol is similar to that in Yalagandula *et. al.* [49], while the key-exchange protocol is a simple generalization of the SSL protocol [14].

9 Discussion

In this section, we summarize our experiences with the proxy deployment. We (and other groups) have used various versions of the proxy since March 2004. Over this time interval, the proxy has attracted interest from both overlay developers and end-users. Developers of various routing overlays and network architectures, such as Delay Tolerant Networks [10], Host Identity Protocol [26], OverQoS [41], Tetherless Computing [37], QoS Middleware project [27], have expressed interest in leveraging the proxy for their own overlays. One group has been already successful in reusing the proxy to provide support for legacy applications over HIP, without changing the operating system.

[5]The main goal of TRIAD is to provide content routing, but this is out of the scope of this discussion.

The proxy has been used for supporting a variety of applications including *ssh*, *ftp*, web browsing, and virtual network computing (VNC) applications. Most end-users have typically used the proxy for accessing their home machines to get around NAT boxes and dynamic IP address allocation by their ISPs.

Based on our own experience and based on the feedback from other end-users and developers, we have learned a few lessons, some of which are obvious in retrospect. These lessons emphasize what is arguable the main benefit of the proxy: the ability to "open" the overlays to real users and real applications. The feedback received from these users has been invaluable in improving the proxy design, and in some cases, the overlay design.

Efficiency matters. When using legacy applications, the users expect this applications to perform the same "way" no matter whether they run directly on top of IP or on top of an overlay. In particular, more often than not, we found the users unwilling to trade the performance for more functionality. This feedback lead not only to proxy optimizations, but also to overlay optimizations. For example, the developers of $i3$ have added shortcuts to improve the end-to-end latency, and added the ability to share a private trigger among multiple tunnels to decrease the setup cost.

Security matters. Security was not part of our original design agenda. However, we found that the users expected at least the same level of security from the OC-D name resolution mechanism as they get from today's DNS (where impersonation while possible, is not trivial). In the area of mobility, the users and developers argued for even much stronger security guarantees such as authentication and encryption. In the end, this feedback led us to make the security a first order goal of our design.

Usage is unexpected. Initially, we expected mobility to be the most popular application. However, this was not the case. Instead the users were more interested in using the proxy for such "mundane" tasks as accessing home machines behind NATs or firewalls, and getting around various connectivity constraints. In one instance, users leveraged the fact that the proxy communicates with $i3$ via UDP to browse the web through an access point that was configured to block TCP web traffic! The unexpected usage lead us to provide better support for applications over NATs. In particular, we have implemented an OC handle negotiation mechanism that preserves the addresses in the IP headers. This allows us to support some applications that otherwise do not work over NATs (*e.g., ftp*).

10 Conclusion

Overlay networks have been the focus of much research in recent years due to their promise of introducing new functionality *without* changing the Internet infrastructure. Surprisingly little attention has been devoted to achieving the same desirable property at the end-host: provide new functionality without any changes to legacy software such as operating systems, network applications and middlebox applications.

Our work is a preliminary step in this direction and aims to improve the inter-operability between legacy applications and routing overlays, and between different routing overlays. It is our hope that this can help accelerate the deployment and adoption of overlay networks that aim to serve the typical Internet user.

Currently, we (and others) are in the process of extending the proxy implementation to support other overlay networks. Ultimately, we plan to enlarge our user base and gather more feedback to improve the proxy. As our experience showed, users often find unexpected uses to the system, which can push the design in new directions.

References

[1] Akamai Technologies. `http://www.akamai.com`.

[2] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. of SOSP*, 2001.

[3] D. G. Andersen. Mayday: Distributed Filtering for Internet Services. In *USITS*, Seattle, WA, 2003.

[4] J. Angel. Realmedia complete. `http://www.angel.org/Book/chapter2_pdf`.

[5] K. Argyraki and D. Cheriton. Loose Source Routing as a Mechanism for Traffi c Policies. In *Proc. of FDNA*, 2004.

[6] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig. NATBlaster: Establishing TCP Connections Between Hosts Behind NATs, Aug 2004. In Submission.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SCRIBE: a large-scale and decentralised application-level multicast infrastructure. In *NGC'2001*, London, UK, November 2001.

[8] D. R. Cheriton and M. Gritter. TRIAD: A New Next Generation Internet Architecture, Mar. 2001. `http://www-dsg.stanford.edu/triad/triad.ps.gz`.

[9] H. Eriksson. MBONE: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, 1994.

[10] K. Fall. A delay tolerant network architecture for challenged internets. In *Proc. SIGCOMM*, 2003.

[11] B. Ford. Unmanaged Internet Protocol: Taming the edge network management crisis. *SIGCOMM Comput. Commun. Rev.*, 34(1):93–98, 2004.

[12] FreeBSD. `www.freebsd.org`.

[13] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. In *Proc. of NSDI*, 2004.

[14] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet Draft, November 1996. `http://wp.netscape.com/eng/ssl3/`.

[15] S. Gupta and A. L. M. Reddy. A Client Oriented, IP Level Redirection Mechansism. In *Proc. IEEE INFOCOM*, 1999.

[16] Hopster: Bypass fi rewall, Bypass proxy software. `www.hopster.com`.

[17] Y. hua Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. of ACM SIGMETRICS*, 2000.

[18] Internet protocol v4 adress space. `http://www.iana.org/assignments/ipv4-address-space`.

[19] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. of OSDI*, Oct 2000.

[20] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In *Proc. of IPTPS*, 2004.

[21] KaZaa. `http://www.kazaa.com/`.

[22] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proc. of ACM SIGCOMM*, Aug. 2002.

[23] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[24] T. Mallory and A. Kullberg. Incremental Updating of the Internet Checksum. RFC 1141, January 1990.

[25] S. McCanne and V. Jacobson. vic: A Flexible Framework Framework for Packet Video. In *ACM Multimedia*, 1995.

[26] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol, 2003. `http://www.hip4inter.net/documentation/drafts/draft-moskowitz-hip-08.ht%ml`.

[27] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications Magazine*, 2001.

[28] A. Nakao, L. Peterson, and M. Wawrzoniak. A Divert Mechanism for Service Overlays. Technical Report TR-668-03, Computer Science Department, Princeton, Feb 2003.

[29] T. S. E. Ng, I. Stoica, and H. Zhang. A Waypoint Service Approach to Connect Heterogeneous Internet Address Spaces. In *Proc. of USENIX Technical Conference*, 2001.

[30] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[31] L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. In *Proc. of the Third Workshop on Hot Topics in Networking (HotNets-III)*, San Diego, CA, November 2004.

[32] Planet Lab. `http://www.planet-lab.org`.

[33] P. Rodriguez, S. Mukherjee, and S. Rangarajan. Session level techniques for improving web browsing performance on wireless links. In *Proc. of the 13th international conference on World Wide Web*, pages 121–130, 2004.

[34] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, Jan. 2001.

[35] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Splitstream: High-bandwidth multicast in a cooperative environmen. In *SOSP'03*, Lake Bolton, NY, October 2003.

[36] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A Case for Informed Internet Routing and Transport. Technical Report TR-98-10-05, 1998.

[37] A. Seth, P. Darragh, and S. Keshav. A Generalized Architecture for Tetherless Computing in Disconnected Networks. `http://mindstream.watsmore.net/`.

[38] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *SIGCOMM*, 2002.

[39] G. Su. *MOVE: Mobility with Persistent Network Connections*. PhD thesis, Columbia University, Oct 2004.

[40] G. Su and J. Nieh. Mobile Communication with Virtual Network Address Translation. Technical Report CUCS-003-02, Columbia University, Feb 2002.

[41] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *Proc. of NSDI*, 2004.

[42] F. Teraoka, Y. Yokote, and M. Tokoro. A Network Architecture Providing Host Migration Transparency. In *Proc. ACM SIGCOMM*, 1991.

[43] vat - LBNL Audio Conferencing Tool. `http://www-nrg.ee.lbl.gov/vat`.

[44] Virtual private network consortium. `http://www.vpnc.org/`.

[45] Virtual tunnel. `http://vtun.sourceforge.net/`.

[46] M. Walfi sh, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *Proc. of OSDI*, 2004.

[47] K. Wehrle, F. Pahlke, D. Muller, et al. Linux Networking Architecture: Design and Implementation of Networking Protcols in the Linux Kernel, 2004. Prentice-Hall.

[48] B. Wilcox-O'Hearn. Names: Decentralized, Secure, Human-Meaningful: Choose Two. `http://zooko.com/distnames.html`.

[49] P. Yalagandula, A. Garg, M. Dahlin, L. Alvisi, and H. Vin. Transparent Mobility with Minimal Infrastructure. Technical Report TR-01-30, UT Austin, June 2001.

[50] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host Mobility Using an Internet Indirection Infrastructure. In *Proc. of MOBISYS*, 2003.