Supporting Legacy Applications over $i3$

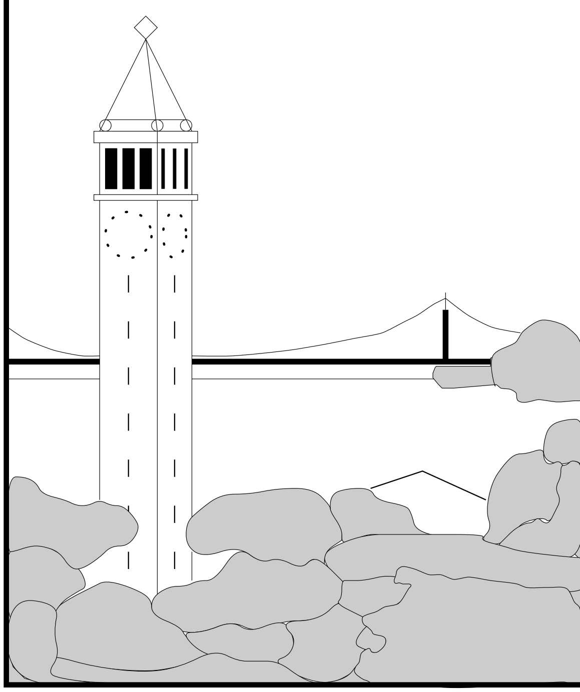
Jayanthkumar Kannan
Univ. of California, Berkeley

Ayumu Kubota
KDDI Labs

Karthik Lakshminarayanan
Univ. of California, Berkeley

Ion Stoica
Univ. of California, Berkeley

Klaus Wehrle
Univ. of Tübingen



Report No. UCB/CSD-04-1342

June 2004

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

Supporting Legacy Applications over $i3$

Jayanthkumar Kannan
Univ. of California, Berkeley

Ayumu Kubota
KDDI Labs

Karthik Lakshminarayanan
Univ. of California, Berkeley

Ion Stoica
Univ. of California, Berkeley

Klaus Wehrle
Univ. of Tübingen

June 2004

Abstract

Providing support for legacy applications is a crucial component of many overlay networks, as it allows end-users to instantly benefit from the functionality introduced by these overlays. This paper presents the design and implementation of a proxy-based solution to support legacy applications in the context of the $i3$ overlay [24]. The proxy design relies on an *address virtualization* technique which allows the proxy to tunnel the legacy traffic over the overlay transparently. Our solution can preserve IP packet headers on an end-to-end basis, even when end-host IP addresses change, or when end-hosts live in different address spaces (*e.g.,* behind NATs). In addition, our solution allows the use of human-readable names to refer to hosts or services, and requires no changes to applications or operating systems.

To illustrate how the proxy enables legacy applications to take advantage of the overlay (*i.e.,* $i3$) functionality, we present four examples: enabling access to machines behind NAT boxes, secure Intranet access, routing legacy traffic through Bro, an intrusion detection system, and anonymous web download. We have implemented the proxy on Linux and Windows XP/2000 platforms, and used it over the $i3$ service on PlanetLab over a three month period with a variety of legacy applications ranging from web browsers to operating system-specific file sharing.

1 Introduction

In recent years, many researchers have focused on using overlay networks as a way of introducing new functionality in the Internet. However, the key issue which will ultimately determine the success of these proposals is the ease with which typical users can take advantage of the new functionality. Some common approaches of addressing this issue are porting popular applications or building new ones on top of these overlays. Examples in this category include vic/vat [5, 14] for the MBONE [8], and more recently peer-to-peer file sharing applications (such as KaZaa, Gnutella and Overnet). An alternative approach is providing support for *existing legacy* applications to use the overlay. This approach would allow typical users to instantly reap the benefits of many overlay proposals that provide mobility [29], composable services [9], quality of service and resilience [6, 22, 26].

Despite the obvious importance of supporting legacy applications, there has been relatively little effort on developing comprehensive solutions. The solutions proposed so far have various limitations such as assuming that overlay nodes are still identified by IP addresses (RON [6] and ROAM [29]), requiring changes to DNS servers or NAT boxes (AVES [16]), or requiring modifications to the name lookup mechanism at the end-host (TRIAD [7] and HIP [15]).

In this paper, we address the limitations of these proposals by designing a proxy-based solution to support legacy applications in the context of the Internet Indirection Infrastructure ($i3$) [24]. To the best of our knowledge, this is the first solution that provides the following desirable properties simultaneously:

- Require no change to the applications, operating systems, DNS servers, or NAT boxes.
- Allow preservation of original IP headers, while providing support for mobility and transparent access to hosts behind NATs and firewalls. Preserving IP headers not only allows the use of many legacy applications (such as `ftp`, `H.323`), but also allows the deployment of network-layer middle-boxes that use IP headers.
- Allow users to identify overlay entities such as end-hosts or services using human-readable names.

This paper describes the design and implementation of the proxy and our experience with deploying applications and middleboxes with the proxy. While our solution is implemented in the context of $i3$, we present much of the design independent of $i3$ and discuss how it can be easily extended to work with other overlay networks.

The central component of the proxy is a simple *address virtualization* technique, which allows the traffic between legacy applications to be tunneled over the overlay. In a nutshell, the proxy traps the DNS requests of the legacy applications, resolves them to virtual IP addresses that have local scope. Then, the proxy intercepts the packets sent by the legacy applications to these virtual addresses and tunnels them over the overlay.

For realizing all the aforementioned properties, the proxy needs to be colocated (*i.e.,* running on the same host) with the application that wishes to use the overlay. We also present alternate remote deployment alternatives for the proxy at the expense of some of the properties. For instance, if the proxy is not colocated with the application that initiates the connection, then we need to modify a DNS server, and if the proxy is not colocated with either the initiator or the receiver, then IP headers would be re-written.

We have fully implemented our proxy on both Linux and Windows XP/2000 platforms and deployed it over an $i3$ network running on PlanetLab [19]. To illustrate how the proxy allows legacy applications to take advantage of $i3$'s functionality, we present four examples: enabling access to machines behind NATs, secure Intranet access, routing legacy traffic through Bro [18], a middle box that performs intrusion detection, and anonymous web download.

Our experience with these applications led us to revisit some aspects of the $i3$ design. We have modified the control path operation in $i3$ to allow transparent access of machines and services behind NATs. We have introduced a *shortcut* option that hosts can use to improve the wide-area performance on the data path.

The rest of the paper is organized as follows. In the next section, we give a brief overview of $i3$. In Section 3, we present the address virtualization technique, the core component of the proxy design, largely agnostic of the details of $i3$. We present how we realize this proxy design over $i3$ in Section 4. In Section 5, we describe the two modifications that we made to $i3$ for supporting NATs and improving the wide-area performance. We elaborate on various application scenarios the proxy can be used for in Section 6. After a description of the implementation in Section 7, we present the evaluation of the proxy in Section 8. We survey related work in Section 9 and conclude after summarizing the lessons from our experience with the proxy and using legacy applications with it.

2 Overview of *i3*

In this section, we provide a brief overview of $i3$. At its roots, $i3$ [24] provides indirection, that is, it decouples the act of sending a packet from the act of receiving it. There are two basic operations in $i3$: sources send packets to a logical *identifier* (ID) and receivers express interest in packets by inserting a *trigger* into the network (Figure 1(a)). Triggers can be thought of as routing entries that point to receivers or to other triggers. Packets are of the form $(id, data)$ and triggers are of the form $(id, addr)$, where $addr$ is either an ID or an IP address. Given a packet $(id, data)$, $i3$ finds the trigger $(id, addr)$ and then forwards $data$ to $addr$. Receivers refresh the triggers that they insert as long as they desire to receive packets sent to the ID that the trigger corresponds

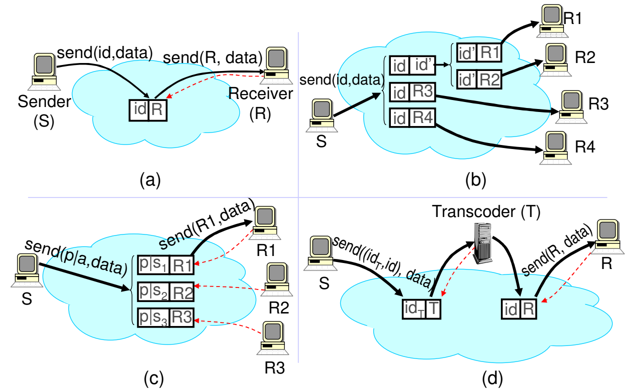


Figure 1: Basic communication primitives in $i3$: (a) unicast, (b) multicast, (c) anycast, and (d) service composition.

to (soft-state approach). In the implementation, a trigger expires at an $i3$ node if it is not refreshed for 30 seconds. In addition, $i3$ supports an operation to remove triggers.

Identifiers in $i3$ are 256 bits long. IDs in packets are matched with those in triggers using longest prefix matching. To reduce the probability of accidental collision, two IDs match only if they share a prefix with a length of at least 128 bits. $i3$ is implemented as an overlay network of nodes that store triggers and forward packets. Identifiers are mapped to $i3$ nodes using a distributed lookup service such as Chord [25]. A trigger is stored at the node that is responsible for its identifier in accordance with the Chord lookup protocol. Similarly, packets are routed to the appropriate node by Chord. The mapping procedure ensures that all IDs which share the same 128-bit prefix are mapped on the same node; thus, the longest prefix matching operation is performed locally.

2.1 Communication Abstractions

$i3$ provides support for a variety of communication abstractions including mobility, multicast, anycast and service composition.

Mobility. A mobile host that changes its address from R to R' can preserve the end-to-end connectivity by updating its trigger from (id, R) to (id, R').

Multicast. Creating a multicast group is equivalent to having all members of the group register triggers with the same ID. There is no difference between unicast and multicast in $i3$, and an application can switch between the two on the fly. Figure 1(b) shows a two-level multicast hierarchy.

Anycast. All hosts in an anycast group maintain triggers that have identical 128-bit prefixes (Figure 1(c)), but different 128-bit suffixes. Packets are delivered to the group member that has the trigger with the longest matching identifier. This scheme can be used for implementing applications such as server selection.

Service composition. $i3$ allows either the sender or the receiver to forward packets through intermediate points in the network. One way to achieve this is to replace the packet ID with a stack of IDs. Forwarding such a packet is similar to source routing in IP. Figure 1(d) shows how a sender S can use a stack of IDs, $[id_T, id]$, to forward the packet through a transcoder T. A receiver can control packet forwarding by replacing the second field of its trigger with a stack that describes the forwarding path.

2.2 Additional Operations

Operation of end-hosts. End-hosts (servers) that expect connections from arbitrary end-hosts (clients) maintain triggers whose IDs are well-known. These triggers are called *public* triggers. Once a client contacts a server through its public trigger, they can exchange a pair of IDs which they use for the remainder of the communication. Triggers corresponding to these IDs are referred to as *private* triggers.[1] The use of public triggers as initial rendezvous points gives end-hosts complete freedom in picking the IDs of their private triggers. Finally, end-hosts keep their private IDs secret.

Caching To improve performance, $i3$ performs aggressive caching. In Figure 1(b), when the $i3$ node storing triggers $(id, *)$, call it M, first sends a packet with ID id', the packet is forwarded by Chord to the node storing triggers $(id', *)$, call it P. To improve the performance, M caches the address of P, and sends all subsequent packets directly via IP. The cache is periodically refreshed (soft-state), with a period of 30 seconds in the $i3$ implementation.

3 Address Virtualization

In this section, we present a simple technique, called *address virtualization*, that the proxy uses to transparently tunnel the traffic of legacy applications over an overlay network. The essence of this technique is to use an address with local significance, called virtual address, to impersonate a remote host. The virtual address can abstract away many of the details concerning the remote host such as its location, identity, or address, and thus allows one to interpose virtually any functionality between two end-hosts. Address virtualization is similar in spirit to other previous proposals that aim to provide support for legacy applications [7, 15, 16], but is different from these proposals in design details and the properties it achieves. We discuss these differences in Section 9.

Address virtualization provides support for unmodified legacy applications that use human-friendly, DNS-like names. Moreover, the technique can preserve the IP headers of the packets, which allows the proxy to support even NAT-unfriendly applications such `ftp`. In an attempt to decouple the proxy basics from the implementation over $i3$, we present this section agnostic of the details of $i3$.

[1] End-hosts that do not need to be contacted by arbitrary end-hosts don't need to maintain public triggers.

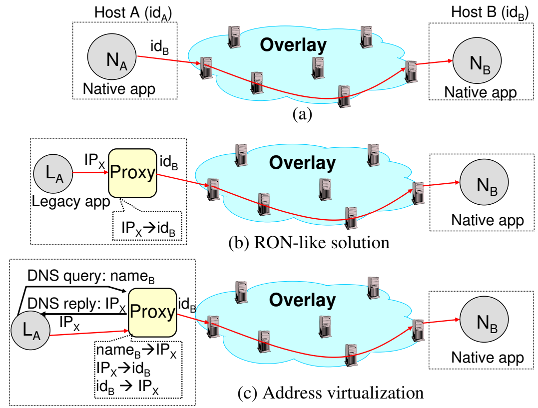


Figure 2: (a) Two native applications communicating over an overlay network. (b) A RON-like Solution (c) Address-virtualization based solution

3.1 Emulating a Native Client

The basic idea behind interfacing a legacy application to an overlay is to emulate a native client. Consider a general overlay network where end-points are identified by unique overlay identifiers. An overlay ID can be an IP address [6], a DNS-like name, a source route [7, 21], an $i3$ ID [24], or any unique string of bits. While in general, an overlay ID can be associated with a service or a connection, for simplicity, in this section, we assume that IDs are associated only to hosts. In the remainder of this section, we use this overlay network model to motivate and present the address virtualization technique.

Figure 2(a) shows a native application[2] N_A running on host A that sends packets to another native application N_B on host B. The packets sent by N_A are destined to id_B, the identifier of the end-host running application N_B. Let us replace native application N_A with a legacy application L_A. Since L_A is oblivious to the existence of the overlay network, it knows nothing about the overlay ID of N_B. A traditional technique to address this problem is to use a proxy to intercept the IP packets sent by legacy applications, and forward them to id_B (see Figure 2(b)). An example of overlay network that employs this technique is RON [6].

However, in a general overlay network that supports mobility or enables the access of hosts behind NATs, the legacy applications cannot use the IP address of the remote host. Our solution to this problem is to use a *virtual address* to identify the other end host. A virtual address has local scope only and it is used by the proxy to intercept the legacy application packets and forward them to the corresponding overlay ID.

The final piece of our solution is to leverage the DNS lookup mechanism to hand over virtual addresses to applications. In particular, the proxy intercepts the DNS request of an application (locally) and resolves it to a virtual address and an

[2] A native application directly talks to the overlay network. For *e.g.,* it sends and receives packets using the overlay packet format.

overlay ID. Figure 2(c) illustrates this process. Let $name_B$ be the name of the end-host running N_B. The proxy intercepts the DNS request of L_A for $name_B$ and resolves it to a *virtual* address IP_X which has local scope only.

Note that IP_X need not have any relationship with the address of B, and thus need not change when the address of B changes. This allows A and B to live in different address spaces (*e.g.,* A can be on an IPv4 network, while B can be on an IPv6 network) or/and be mobile. For now, assume that the proxy knows id_B, the overlay identifier of the machine named $name_B$. The proxy then stores the mapping $IP_X \rightarrow id_B$, and returns IP_X as the DNS reply to L_A. The legacy application sends the subsequent packets to address IP_X, which are encapsulated and sent over the overlay. Packets in the other direction are sent to id_A and are received by the proxy over the overlay. The proxy decapsulates these packets, rewrites the source address to IP_X and delivers them to the application. Thus, to the legacy application, the proxy emulates a remote host with address IP_X. We refer to this technique as *address virtualization*.

Intuitively, we can imagine that the proxy captures packets that the legacy applications send/receive by faking a virtual interface. Addresses for both the virtual interface and for emulating the remote host can be allocated from a pool of reserved IP addresses (such as 10.0.0.0/8). At the sender, whenever a new DNS request is intercepted, the proxy chooses a free address randomly from this pool.

Using DNS-like names to intercept legacy traffic has two main advantages: convenience and flexibility. Firstly, users of the application can use human-friendly DNS-like names as they are used to. Secondly, this approach allows users to independently define a new namespace specific to an overlay. Application preferences can be encoded in the name used, thus allowing users to decide a policy at "run-time". This would be useful if the proxy is used with multiple overlays simultaneously; the packets can be routed over a particular overlay based on the name suffix.

In the next few sections, we address the following questions: (a) how is an overlay DNS name resolved to an overlay identifier? and (b) how can we preserve the original IP headers without re-writing addresses?

3.2 Resolving Names

We now describes how the proxy resolves a name to an overlay identifier. These names need not belong to a registered DNS namespace, since resolution of these names is usually done by a proxy and not a legacy DNS server. For clarity, we refer to names that are not resolved using DNS as overlay DNS names. In our implementation, we use names with a suffix `.i3` to refer to *i*3 DNS names. We consider three design alternatives for resolving an overlay DNS name to an overlay ID.

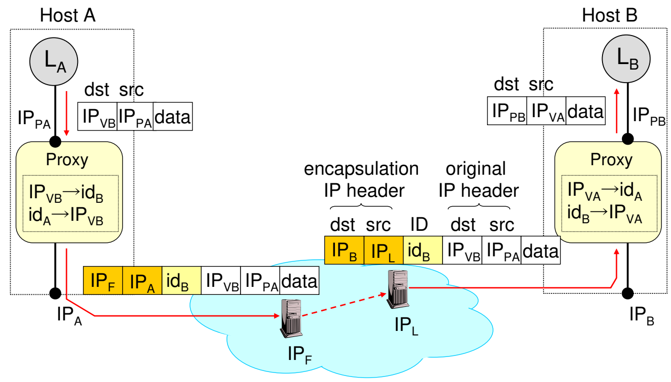


Figure 3: The headers of a packet forwarded from L_A to L_B. Packet headers contain destination address followed by source address.

- *Global Resolution.* Use a modified DNS or a DHT such as OpenHash [11] to store the mapping between overlay DNS names and overlay IDs.
- *Local Resolution, Global Scope.* Use an implicit method such as applying a well-known hash function on the name to obtain the ID.
- *Local Resolution, Local Scope.* Use a local address book to explicitly store the name-to-ID mappings.

Global resolution provides a convenient way for users to manage names. However, this level of names would be a space of conflict much like DNS names are (as argued by SFR [27]). In addition, a global resolution scheme usually requires a central authority and a separate infrastructure to store the mapping.

Local resolution of names with global scope also has the convenience of global resolution without requiring an infrastructure. However, it also has the problem of name conflicts, apart from being susceptible to impersonation and other security attacks described in [3].

On the other hand, explicit storage of name-to-ID mappings in a local address book provides better security. An address book is similar to a phone book where person names are replaced by overlay names and phone numbers by overlay IDs. Also, like the names in a phone book, the names in the address book have only local scope. The downside of using address books is that they are not easy to maintain and operate with.

In fact, the tradeoffs between decentralized operation, security, and human-friendliness in the name-to-ID mapping that we see here have been articulated before in [3]. In our implementation, we use the local resolution mechanism using an address book, as we place greater emphasis on security.

3.3 Preserving IP Headers

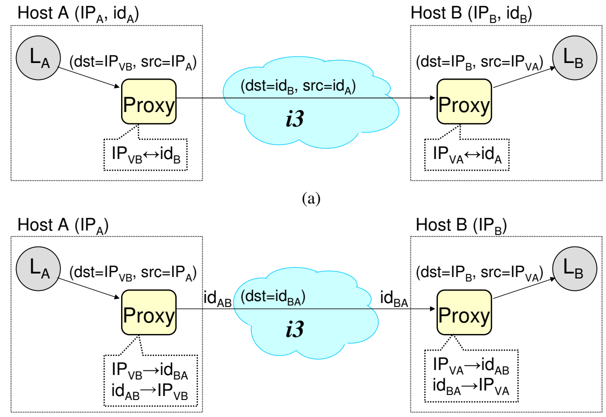
In our solution, the proxy emulates a remote host to the local legacy application by returning a virtual address. With a random choice of virtual addresses at the proxy at both end hosts, source address re-writing is necessary, even if both the sender and receiver live in the same address space.

There are two reasons to avoid IP header rewriting. First, it allows us to support popular applications (*e.g.,* `ftp` and `H.323`) that encode addresses in the packet payload. Second, it allows us to deploy network layer middle-boxes, such as intrusion detection systems, on the overlay data path. We describe such an application in Section 6.4.1.

In this section, we present a simple solution that avoids rewriting IP headers. The main idea behind our solution is to assign virtual interface addresses to the proxies in such a way that the need for address re-writing is obviated. To understand our solution better, consider the example in Figure 3 that shows the headers of a packet as it is forwarded from L_A to L_B. The crucial point to note is that proxy at host A is configured to fake a virtual interface with address IP_{PA} that is different from the host's Internet-routable address IP_A. From the point of view of a legacy application, this is equivalent to a host with two interfaces: IP_{PA} and IP_A. In order to intercept the packets of the legacy applications, host A is configured such that all legacy applications send and receive packets on interface IP_{PA}. As a result, the source address of an IP packet sent by L_A is IP_{PA}. The destination address of the packet is IP_{VB}, which represents the virtual address returned by the proxy to L_A in response to its DNS request. Similarly, the source and destination addresses of the packet received by L_B are IP_{VA} and IP_{PB}.

Hence, the only way to avoid address rewriting is to let $IP_{PB} = IP_{VB}$ and $IP_{VA} = IP_{PA}$. While these constraints can be enforced by exchanging the virtual interface addresses during the control protocol, our implementation avoids the overhead of an additional round-trip by choosing the virtual interface address of the proxy as a hash of its public ID. In our example, we have $IP_{PB} = IP_{VB} = 10.H(id_B)$ and $IP_{VA} = IP_{PA} = 10.H(id_A)$, where $H()$ is a well-known hash function that maps 256 bit IDs to 24 bits.

In our implementation, we use an address block of size 2^{24} to allocate the virtual addresses, and so the probability of collision is not negligible. In case of collision, the proxy reverts to choosing an available virtual address at random. In such cases, we no longer preserve the IP header of the packets, but would continue to support the vast majority of legacy applications which do not need this requirement. We note that this technique assumes that applications do not specifically bind to a particular interface but can receive packets sent to any interface on the host (*e.g.,* by the use of `INADDR_ANY`). This assumption holds in practice as most applications support multiple interfaces anyway.



(b)

Figure 4: Two design choices for the $i3$ proxy. (a) Per-host Identifiers (b) Each host maintains an ID for every other host it communicate with.

4 Realization in $i3$

In this section, we present the realization of the address virtualization technique in $i3$. We start with a simple scenario in which both the proxy and the legacy applications run on the same machine. We remove this assumption in Section 4.4.

We now describe the three main aspects of the $i3$ proxy design in detail: how $i3$ identifiers are mapped to underlying connections, how packets are tunneled over $i3$ (data plane), and how this mapping state is installed (control plane).

4.1 Identifier Allocation

In the description of the general proxy, IDs were associated with hosts. In $i3$, this would mean that each host would maintain only one trigger in $i3$, and each $i3$ encapsulated packet carries both the destination and the source IDs. An alternate realization of ID mapping that $i3$ allows is using the notion of public and private triggers (refer Section 2).

Figure 4 illustrates these two possible realizations when a legacy application L_A running on host A contacts application L_B running on host B. In the first scenario, proxies at A and B maintain only one ID each, id_A and id_B respectively. In the second scenario, in addition to the ID used for initial communication (public IDs), hosts maintain a separate private trigger for every end-host it communicates with. Hosts A and B maintain IDs id_{AB} and id_{BA} for communicating with each other, along with the mappings required to perform overlay↔IP translations.

Using private triggers has the following two advantages. First, in the case of $i3$, an end-host can protect itself against DoS attacks by just removing the private trigger corresponding to the malicious host [12]. Hence, all the the packets from the malicious host will be dropped in $i3$. Second, the fact that an overlay packet does not need to contain a source ID translates to a non-trivial saving of 32 bytes per packet. However, the downside of the second realization is that the number of

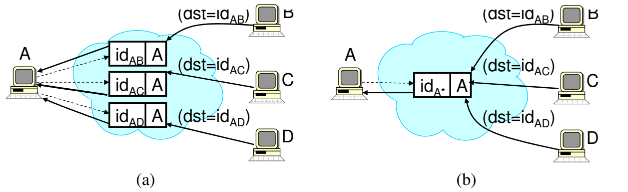


Figure 5: (a) Host A maintains a trigger into $i3$ for every host it communicate with. (b) Host A maintains a single trigger with a prefix shared by IDs id_{AB}, id_{AC}, and id_{AD}.

triggers the end-host needs to maintain in $i3$ can be quite large. We now discuss how we use the anycast primitive in $i3$ to achieve the best of both these realizations.

Efficient State Maintenance. At one extreme, a proxy A can choose the IDs of all its triggers such that they share the same prefix p, and then insert only one trigger with prefix p in the infrastructure (see Figure 5). Since packets forwarded by $i3$ to the end-hosts contain the IDs that they were sent to, the proxy can determine which connection the packet was sent to.

While this approach does not undermine any host-specific policies that can be implemented purely at the end-hosts, this anycast optimization would prevent the proxy from exercising DoS protection features of $i3$ [12]. If anycast is employed, an end-host can stop a DoS attack only by dropping the shared trigger, which, in this case, would sever all existing connections. An end-host can strike a middle ground by associating distinct triggers to the more important connections, and triggers with shared IDs to the less important ones.

4.2 Data Plane: Packet Tunneling

As shown in Figure 4(b), the proxies need to maintan the following mappings to perform packet tunneling: (a) local virtual address to remote private and public triggers, and (b) local private trigger to local virtual address.

Packets sent by local applications destined to local virtual addresses are encapsulated and sent to the remote private ID. Packets received on the local private ID are decapsulated, and sent to the application after the source address is re-written. For instance, in Figure 4(b), the proxy at A uses the mapping $IP_{VB}{\rightarrow}id_{BA}$ to tunnel packets sent by L_A to B, and the mapping $id_{AB}{\rightarrow}IP_{VB}$ to re-write the source address of the packets from B.

A proxy can reduce the latency on the data path by choosing its private IDs to reside on nearby $i3$ nodes. One such mechanism is described in [24]: the basic idea is for each proxy to independently sample the ID space and determine the latency to a particular ID. This mechanism can be implemented without knowledge of all the $i3$ servers and the IDs each server is responsible for.

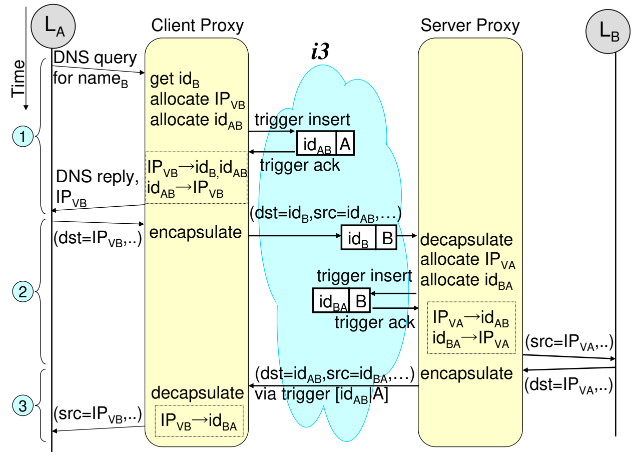


Figure 6: The time diagram of setting up the state at proxies running on hosts A and B in Figure 4(b).

4.3 Control Plane: Installing Mapping State

In this section, we present a protocol for installing the mapping state at proxies. We use the terms *client proxy* and *server proxy* to refer to the proxy when it performs the operations invoked at the initiator (*e.g.,* ssh client) and target (*e.g.,* ssh server) of a connection respectively. This separation is useful while describing the different proxy instantiations in Section 4.4.

Figure 6 shows the time diagram of the protocol when client L_A initiates a connection to server L_B. We assume that host B maintains a trigger with a well-known ID (id_B) at which host A can initiate the connection. In addition, host A inserts trigger $[id_{AB}|A]$ to receive packets from B, and host B inserts trigger $[id_{BA}|B]$ to receive packets from A. Using the terminology in [24], we refer to $[id_B|B]$ as the public trigger of B, and to $[id_{AB}|A]$ and $[id_{BA}|B]$ as private triggers. The main component of the protocol is how proxies exchange the private identifiers id_{AB} and id_{BA}.

As shown in Figure 6, the protocol can be divided in three steps: (1) the client proxy resolves the DNS request, inserts a private trigger for the server, and starts installing the mapping state; (2) the server proxy installs the mapping state, and inserts a private trigger for the client; (3) the client proxy completes the installation of the mapping state. We describe the details of the protocol next.

In step 1, the client proxy at A intercepts the DNS request of L_A for $name_B$, and resolves it to the identifier of the public trigger inserted by host B, id_B. The proxy allocates a virtual IP address IP_{VB} that is used to impersonate host B, inserts a private trigger $[id_{AB}|A]$ (to be used by the remote host B to send packets to A) in $i3$, and stores the mappings $IP_{VB}{\rightarrow}id_B$ and $id_{AB}{\rightarrow}IP_{VB}$. At this point, the client proxy returns IP_{VB} as the DNS reply to L_A.

In step 2, client L_A sends packets using the address IP_{VB} which the client proxy intercepts and, using the mapping it stores, encapsulates as an $i3$ packet destined to id_B. The client proxy piggybacks the private ID id_{AB} specific to host B along with this packet.

When the server proxy at B receives the packet, it decapsulates the packet and performs a procedure similar to the client proxy. In particular, the server proxy allocates a virtual address, IP_{VA} to impersonate host A, inserts private trigger $[id_{BA}|B]$ to receive packets from host A and stores mappings $id_{BA} \rightarrow IP_{VA}$ and $IP_{VA} \rightarrow id_{AB}$. The server proxy rewrites the source address of the IP packet to IP_{VA} before sending it to L_B.

In step 3, when the data packet is sent from B to A, the server proxy piggybacks its private ID id_{BA} to the client proxy. As an optimization, instead of waiting for a data packet, the server proxy can send a control packet to L_A with its private ID. On receiving this private ID, the client proxy updates its mapping to $IP_{VB} \rightarrow id_B, id_{BA}$.

This protocol is both efficient and robust. Since the private triggers are inserted at nearby servers, the additional delay when opening a connection to a new machine is minimal. We can avoid the delay of trigger insertion by using the anycast optimization discussed in Section 4.1. Furthermore, subsequent connections to the same machine simply re-use the mapping state and the private triggers.

The protocol is robust in the presence of packet losses. If either the first packet from A to B or the first packet from B to A is lost, the client proxy will simply continue to piggyback the private ID id_{AB} in the subsequent packets sent by L_A and forward these packets to B via its public identifier id_B. Once the client proxy obtains the private identifier id_{BA}, it sends packets to id_{BA}.

4.4 Deployment Alternatives: Remote Proxies

The basic $i3$ proxy presented in Section 4 assumes that both ends of the communication run the proxy. However, there are practical scenarios where $i3$-enabled hosts might wish to contact (or be contacted by) hosts that do not run $i3$ proxy. For example, $i3$-enabled clients may still want to contact public servers such as `cnn.com` to take advantage of mobility or service composition. In another example, an organization might wish to run one proxy for all its machines instead of installing an $i3$ proxy on each of them. To support these scenarios, we have developed an $i3$-to-IP proxy, which acts as a proxy for a set of hosts that are not $i3$-enabled. We refer to such hosts as legacy hosts.

4.4.1 $i3$-to-IP Proxy

In this section, we consider an $i3$-to-IP proxy deployed to facilitate an $i3$-enabled client to communicate with a legacy server over $i3$. Since the functionality of this proxy is very similar to that of the server proxy, we present only the differences from the server proxy here (see Figure 7).

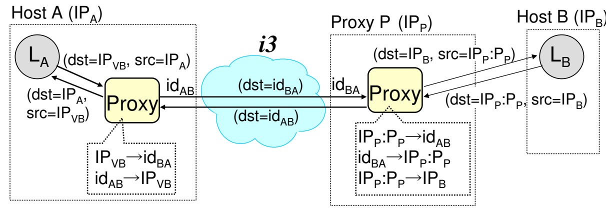


Figure 7: A remote proxy for legacy application L_B.

Consider an $i3$-to-IP proxy P that acts on the behalf of a set of legacy hosts H_p, and let id_P be the public identifier of P. An $i3$-enabled client that wishes to contact a legacy server B from the set H_p sends the DNS name of the legacy server along with its own private trigger to id_P. When P receives the DNS request it resolves the request, inserts a private trigger id_{BA} to set up its end of the $i3$ connection, and sends id_{BA} to the client.

In addition, the $i3$-to-IP proxy plays the role of a NAT box for the set of servers H_P. The $i3$-to-IP proxy allocates a port number P_P that is used, together with its address IP_P, to impersonate host A to host B. In particular, the $i3$-to-IP proxy forwards the packets from host A to B using the source address and port number $(IP_P{:}P_P)$, and it forwards the packets received at $(IP_P{:}P_P)$ to host A using ID id_{AB}.

We allow users to specify routing policies for a legacy DNS name, which indicates whether packets to that host should be sent directly over IP or through an $i3$-to-IP proxy (identified by its public trigger).

4.4.2 IP-to-$i3$ Proxy

In this section, we consider the case where the client does not run a proxy. Instead this functionality is implemented by a special proxy, called IP-to-$i3$ proxy, that sits between the client and either an $i3$-enabled server or an $i3$-to-IP proxy. The functionality of IP-to-$i3$ proxy is almost identical to the client proxy. The key differences between the client proxy and the IP-to-$i3$ proxy are that the DNS request sent out by the legacy client has to reach a valid name server which the proxy should have control over, and that the address that the name server returns has to be a real Internet-routable address.

Our solution is very similar to the scheme implemented in AVES to offer transparent connectivity to machines behind NATs [16]. As a result, our solution shares the same fundamental limitations with AVES. Since DNS queries can be recursive, the proxy (acting as the name server) is not guaranteed to know the IP address of the legacy client that made the DNS request. Hence, it has to use a heuristic to correlate a DNS request and the first data packet. We discuss how we partially address this issue in Appendix 11.

4.5 Tradeoffs in Functionality

Our solutions for the three deployment scenarios involve tradeoff between several factors that we outline in Table 1.

Table 1: Deployment Scenario Vs Properties

	$i3$ proxy	$i3$-to-IP proxy	IP-to-$i3$ proxy
Access Required	End-host	End-host	DNS server
Additional Infrastructure Required	None	Dedicated Machine	Waypoint Machines
Preserving IP headers	Yes	No	No
Mobility	Yes	Limited (Only Client)	Limited (Only Server)
NATed Hosts Supported	Yes	Limited (Only Client)	Limited (Only Server)
Name Resolution	Local Resolution	Remote DNS Resolution	Allocation Authority

Firstly, while the remote proxies provide an alternative when running the basic $i3$ proxy on the end-host is not possible, they are limited in their support for mobility, NAT and preservation of IP headers. Secondly, the IP-to-$i3$ proxy also requires control over a DNS server for a domain name, and in general, the remote proxies need dedicated machines to run on. Finally, the name resolution scheme also differs in these deployment scenarios: an $i3$-to-IP proxy resolves a DNS name on the behalf of $i3$-enabled hosts and an IP-to-$i3$ proxy uses its private address book to resolve $i3$ names. This implies that the IP-to-$i3$ proxy has to manage allocation of the $i3$ DNS namespace seen by legacy IP clients.

5 $i3$ Revisited

Deploying real applications with the proxy has helped us identify certain functionalities desired by users. While some of these functionalities (such as connecting to legacy servers through $i3$) can be handled by proxy and purely at end-hosts, others require modification to the infrastructure/overlay. In this section, we discuss two such issues and discuss how we address them. First, while the very large ID space of $i3$ allows us to uniquely identify every computer and device in the world, the standard $i3$ implementation does not allow one to access machines behind NATs or firewalls. Second, some users were concerned about the fact that every packet was relayed through at least one $i3$ node; they wanted the flexibility of circumventing $i3$ for the data path.

5.1 Accommodating NAT Boxes

In this section, we present a simple modification to $i3$ in order to accommodate end-hosts behind NATs. The modification happens only on the control plane; data forwarding is unmodified. The main idea is to rely on the control plane protocol initiated by the host for inserting triggers in $i3$. During trigger insertion, the NAT allocates a globally routable address through which external hosts can contact the NATed host. Our solution ensures that other hosts only use this NAT-allocated address to contact the host behind the NAT. This solution requires no changes or configuration of the NAT box as trigger insertion is initiated by the end-host.

Figure 8 illustrates our solution. Let IP_A be the (locally valid) IP address of a host behind a NAT, and let P_{proxy} be the port number of the proxy. Assume the proxy inserts a

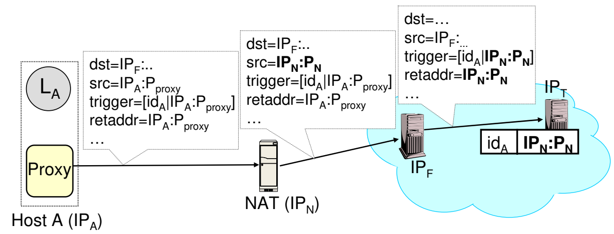


Figure 8: Supporting NATs in $i3$.

trigger $[id_A|IP_A{:}P_{proxy}]$. The fields dst and src represents the IP destination and source addresses of the packet, and the field *retaddr* represents the address of the proxy that has generated the message. Note that the *dst* field is set to the address of an $i3$ node known by the end-host (IP_F, in this case). This node is not necessarily the node where the trigger is ultimately inserted. Upon receiving the trigger insertion packet, the NAT box translates the source address and port number of the packet, $IP_A{:}P_{proxy}$, to $IP_N{:}P_N$ where IP_N is the IP address of the NAT box, and P_N represents the port number used by the NAT box to identify $IP_A{:}P_{proxy}$.

When the packet arrives at the first $i3$ node, IP_F, the node checks whether the packet's *dst* and *retaddr* fields are the same. If not, the node concludes that the packet has traversed a NAT and overwrites *retaddr* with the packet's source address. This enables cache messages to be sent to the NAT. In addition, if the trigger points to an IP address (which is the case in this example), the $i3$ node also overwrites the second field of the trigger. If the trigger points to another ID, the $i3$ node does not modify the trigger. The packet is then routed to the $i3$ node IP_T that is responsible for id_A. Upon receiving the packet, node IP_T inserts, challenges, or refreshes the trigger. Thus, the control plane protocol ensures that triggers point to only globally valid IP addresses. This allows the data forwarding to work seamlessly.

The protocol we have described however cannot support certain types of restrictive NATs [20] such as restricted cone NATs and symmetric NATs which allocate address and filter packets based on source and destination addresses. In our example, this means that IP_T cannot send packets using the address $IP_N{:}PN$, since this address is allocated only for the source-destination pair (IP_A, IP_F). To get around this problem, we force the proxy to contact IP_T directly during trigger insertion, so that an address is allocated by the NAT for the $i3$ server IP_T.

5.2 Shortcuts

In the original $i3$ design, every packet is relayed through at least one $i3$ node. While this indirection is necessary for useful functionalities such as mobility, anonymity, and access to machines behind NATs, it is, in general, less efficient than direct IP communication. To alleviate the efficiency problem, we propose a straightforward modification to $i3$: we allow the communication path between an sender and receiver to circumvent $i3$. We refer to this technique as *shortcut*. Shortcuts are used only if both the sender and the receiver allow its use. Shortcuts apply only to triggers of the form (id, R) where R is the IP address of the receiver and the sender sends packets addressed to id.

The shortcut technique allows the sender and the receiver to express their preference for using shortcut in the data and trigger insertion packets respectively. Consider a host A sending a data packet to ID id_{BA}. When the packet reaches the $i3$ node storing trigger $[id_{BA}|B]$, if either the data packet or the trigger do not have the shortcut preference set, the $i3$ node sends a cache reply message to A and forwards the message to B. Otherwise, the $i3$ node just forwards the packet to host B, and lets the proxy on host B reply with a cache message to A. After receiving the cache message from B, proxy A sends all subsequent $i3$ packets directly to the proxy B.

Note that this protocol does not work if the sender and the receiver are behind NATs (of the restricted cone or symmetric kind). This implies that shortcuts should not be used by a a NATed host for sending data or inserting triggers. An end-host might also avoid using shortcuts to preserve its anonymity or to maintain connectivity under mobility.

With shortcuts, the role of $i3$ reduces from a routing infrastructure to a lookup infrastructure. The natural question that follows is: why would an end-host that prefers shortcuts use $i3$ instead of sending packets directly via IP? For some applications, $i3$ provides a better lookup infrastructure than DNS in that it allows end-hosts to quickly update their triggers. In the above example, $A's$ proxy maintains the cache entry of B by soft state. This implies that if B moves during the course of the communication, once the cache entry times out, $A's$ proxy would revert to sending packets addressed to id_{BA} to $i3$, and eventually to the new location of B.

6 Applications

In this section, we discuss four application scenarios supported by the $i3$ proxy that we have experience with: enabling access to home machines behind NATs, secure Intranet access, anonymization, and middle-box applications (*e.g.,* intrusion detection).

6.1 Accessing Machines Behind NATs

Users today use ad-hoc mechanisms to access their home machines behind NATs. This problem can be solved by a straightforward use of $i3$ proxy: the home machine runs a $i3$ proxy and the user can use $i3$ to contact his machine from elsewhere. This solution requires no changes or reconfiguration of the NAT box.

6.2 Secure Intranet Access

For security reasons, organizations typically would like to provide restricted access for connections into their Intranet from outside. To this end, VPN-based (Virtual Private Networks) solutions are currently used in conjunction with firewalls (for more information on VPNs, refer to [1]). We now describe how $i3$-to-IP proxy can be used as an alternative for secure Intranet access, and discuss the potential advantages over VPNs.

In this scenario, the $i3$-to-IP proxy runs inside the organization and hence has unrestricted access to all the intranet machines. External end-hosts relay through the $i3$-to-IP proxy in order to access Intranet machines. Authentication of external end-hosts is a fundamental requirement in this scenario and to this end, we added an authentication wrapper over the $i3$-to-IP proxy.

Our authentication mechanism for the $i3$-to-IP proxy is based on a simple RSA public-key based protocol similar to `ssh`. We assume that there is some out-of-band mechanism for the $i3$-to-IP proxy and a legitimate client proxy to exchange their public keys (*e.g.,* smart cards). Our authentication protocol is a simple challenge-response protocol (which incurs two additional round trips). Only if the authentication mechanism is successful, the client proxy obtains a private trigger through which it can contact the the $i3$-to-IP proxy. Communication from this point onwards proceeds just as before. Unlike `ssh` which encrypts all data packets using session keys negotiated in the authentication phase, we do not perform any encryption operation on the data path to avoid additional overheads. Resistance against a man-in-the-middle adversary who hijacks an authenticated connection can be obtained by application-level end-to-end encryption.

Our $i3$ based solution has two advantages over VPN-based solutions:

- *Multiple intranets:* Using an $i3$-to-IP proxy, a client can access multiple intranets at the same time. In contrast, existing VPN solutions do not usually support this feature. If both the intranets use the same address range, then connecting to both intranets can be impossible.
- *Security:* Since a VPN client gets a virtual interface address on the organization's network, an infected client can potentially infect internal machines to which the client has unrestricted access to. Such an infection is hard to perform in our case. This is because an infected

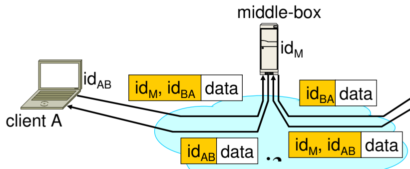


Figure 9: Example of using a middle-box with ID id_M. The private IDs of the end-hosts are id_{AB}, and id_{BA}, respectively.

client cannot access any Intranet host directly using its real IP address. Moreover, only the addresses in the virtual address range that have been allocated by the proxy for ongoing connections are vulnerable.

6.3 Anonymization

In the current Internet, an end-host who wishes to contact other end-hosts anonymously relays its traffic through a trusted third party. Examples of such schemes include remailers that relay email anonymously [10] and web surfing anonymizers [4]. This mechanism is naturally emulated by an $i3$ proxy that relays its traffic through $i3$ (without using shortcuts), as such a proxy reveals only its private triggers to the server proxy. This solution assumes that the $i3$ infrastructure is trusted. One way of strengthening the anonymity guarantee is to choose a random $i3$ server as a relay rather than the closest $i3$ server. The performance experienced by the user is determined by his latency to the $i3$ server he chooses to relay through: clearly, choosing a random $i3$ server incurs an additional performance penalty. In fact, the user can trade between the level of anonymity and performance. Note that preserving IP headers according to the address allocation mechanism in Section 3.3 leaks information about the identity of the client: a server can identify other requests initiated by the same client by using the virtual address exchanged during private trigger exchange. If the client desires anonymity, it can either periodically change the virtual address assigned to its applications (which affects ongoing sessions) or simply allow its IP headers to be modified. Note that our approach also assumes that application-level identifying data (*e.g.,* cookie data) is removed at the client proxy itself.

6.4 Middle-box Applications

We now discuss a more general class of applications enabled by the $i3$ proxy. Several useful functionalities such as Intrusion Detection ([18]), QoS ([26]) can be conceptually viewed as being achieved by imposing a middle-box on the data path. We describe how such a middle-box can be accommodated by the $i3$ proxy.

$i3$'s primitive of stack of IDs enables the $i3$ proxy to support middle-boxes in a seamless fashion. First, we describe how a client proxy can impose a middle-box on its path and then describe how a server proxy can do the same. An $i3$ client proxy A that wishes to communicate with another $i3$ server proxy B through a middle-box M addresses its data packets to the stack $[id_M, id_{BA}]$ where id_M is the public trigger of M, and id_{BA} is the private trigger of B (see Figure 9). During control plane negotiations, A also informs B that packets sent to A should be addressed to the stack $[id_M, id_{AB}]$. This protocol ensures that data packets are sent in both directions through the middle-box. An $i3$ server proxy that wishes all users contact it through a middle-box can use a similar protocol to achieve the same. Note that this solution allows B to address its packets directly to id_{AB} and thus subvert the middle-box: this can be prevented by some protocol between the client proxy and the middle-box (*e.g.,* the middle-box marks packets in a special fashion). Our solution also allows the middle-box to authenticate client proxies, if it desires to do so.

This proxy-based solution has several advantages over currently used solutions. Today, setting up a data path through a middle-box is typically done either in an end-host oblivious fashion, or by application-specific configuration such as using HTTP proxies. The first alternative requires that the middle-box be placed at specific topological locations (*e.g.,* behind the access link of a network) over which end-hosts typically have no direct control. The second alternative requires users to configure every application of interest. In contrast, our proxy-based solution allows any arbitrary host to serve as a middle-box and does not require any application-specific configuration. Furthermore, our solution can use $i3$ anycast to distribute the middle-box functionality over several machines.

6.4.1 *Intrusion Detection*

As a proof of concept, we have implemented an intrusion detection system that can be used by any $i3$ enabled client. Our system is based on Bro [18], a well-known intrusion detection software. This example illustrates not only how legacy applications can use the middle-box functionality, but also how the middle-box functionality can be realized using legacy software. Our implementation involves an $i3$-specific shim layer that runs on the same machine as Bro. This shim layer acts as an $i3$ middle-box in order to capture packets sent between $i3$ proxies and rewrites such packets with virtual source and destination address to emulate an conversation between two virtual hosts. Note that no rewriting is required if the source and destination preserve IP headers, in which case, the middle-box uses the virtual addresses exchanged during trigger negotiation. Bro captures these rewritten packets and performs analysis in order to identify potential attacks. We assume that end-hosts are informed of these attacks in some out-of-band fashion *e.g.,* by sending alerts to a trigger specified by the end-host.

An useful type of analysis performed by Bro is stateful analysis which maintains per-flow state while analyzing packets, *e.g.,* reconstructing TCP sessions in order to detect certain at-

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
```

Web Server at www.nytimes.com contacted by Anonymous Client Proxy via *i3*-to-IP proxy

```
1085567384.9332 %1 start 10.0.0.1 > 10.0.0.2
%1 GET /
%1 GET /js/csssniff.js
%1 GET /js/browserSize.js
%1 GET /css/sft1.css
1085567387.86953 %2 start 10.0.0.3 > 10.0.0.4
%2 GET /adx/images/ADS/08/26/ad.82645/1dog_noloop.gif
1085567387.92174 %3 start 10.0.0.5 > 10.0.0.6
%3 GET /SHC/iview/nwyrkjkp00100017shc/direct/012004.05.26.10.29.44?click=
```

Figure 10: Analysis performed by Bro

tack signatures. This requires the shim layer to re-write the IP headers of all packets belonging to a given session with the same source and destination IP addresses. Thus, the $i3$ shim layer needs to maintain state for each source-destination pair. In order to install this state, we also relay the control messages that negotiate the private triggers between the end-host proxies through the middle-box using ID stacks.

We deployed Bro on a FreeBSD machine in Berkeley along with the $i3$ shim layer, through which users can route their traffic by configuring their proxies. We discuss two examples of possible uses to illustrate the flexibility of our implementation. Figure 10 illustrates a ftp analysis performed by Bro on the traffic to a server proxy running a ftp server that imposes the middle-box on the path. This ftp analysis identifies an attempt by `badguy` to upload a file called `eggdrop`, the name of a well known backdoor. Another example illustrated in the figure is a http analysis of the traffic of a client proxy browsing `www.nytimes.com` through two middle-boxes: the Bro middle-box and $i3$-to-IP proxy. The http analyzer in Bro identifies the multiple connections initiated by the browser and the GET requests in each of these connections. Also note that in the first example, the proxies preserve the IP addresses $(10.1.244.127, 10.2.51.9)$ and thus reveal their identity to the middle-box. In the second example, the proxies contact the middle-box anonymously, and the middle-box assigns its own virtual addresses $(10.0.0.1 - 10.0.0.4)$. The second example illustrates the practical necessity of shortcuts: without shortcuts, an additional overhead of 4 Internet hops would be imposed even for clients that do not desire anonymity.

Our solution has certain fundamental limitations as to the type of analysis that can be performed by Bro. Due to the fact that an end-host can contact the middle-box anonymously, it is not possible for Bro to correlate information gleaned from different communication sessions. Note however that an user of the Bro-middle box can always choose to reveal his identity to Bro in order to allow such analysis. Even under the assumption that the end-host contacts the middle-box anonymously, our implementation allows most analysis mechanisms that are used in currently deployed intrusion detection systems such as signature-based attack detection.

7 Implementation

We have implemented the $i3$ proxy and its variants, the $i3$-to-IP proxy and IP-to-$i3$ proxy, in C/C++ in Linux. Our implementation is available in source and binary format at `i3.cs.berkeley.edu`. We first discuss our implementation of the address virtualization and tunneling techniques that apply to all $i3$ proxy variants. Then, we give details of our implementation of the $i3$ proxy and the $i3$-to-IP proxy (for details about the IP-to-$i3$ proxy, refer to Appendix 11).

7.1 TUN Based Address Virtualization

The address virtualization technique uses `tun` [2,28], a software network loopback interface, to capture packets sent by local applications. The `tun` interface allows user-level programs to capture packets from kernel space. `iptables` and `route` [28] are used to redirect DNS packets and packets sent to virtual address respectively to the `tun` interface. Thus, a legacy application need not be recompiled or linked with new libraries. The `tun` device also supports packet writes: packets written on the interface can be sent to the local application. This feature is used to send the decapsulated $i3$ packets to the local application.

There are three limitations of our implementation of the address virtualization technique. Firstly, the proxies require administrator privileges to use the `tun` device. While this is not a concern for the $i3$-to-IP proxy and the IP-to-$i3$ proxy which will be deployed by few individuals and organizations on dedicated machines, this requirement might be inconvenient for users who wish to run the $i3$ proxy. Secondly, the use of the `tun` device also implies that only per-host policies can be specified; per-user policies cannot be specified. Thirdly, using the `tun` device leads to additional overhead due to copying of packets from kernel to application space and vice-versa.

To avoid these limitations, one can implement a dynamic library that can hijack desired system calls. We leave the implementation of such a library to future work.

7.2 Tunneling

Tunneling involves two main operations: encapsulation at the sender and decapsulation at the receiver followed by address rewriting. Packet encapsulation mechanism adds an overhead of 37 bytes per packet due to $i3$ headers. Addition of such headers can lead to fragmentation due to MTU constraints. An application that wishes to avoid fragmentation can perform end-to-end MTU discovery. The address rewriting mechanism rewrites the addresses and incrementally updates the IP and transport player checksums [13]. We have implemented address rewriting for TCP, UDP and ICMP.

Note that address rewriting is performed only when when the address allocation mechanism described in Section 3.3 is not possible (*e.g.,* the virtual address is already allocated).

Maintaining State Consistency. In Section 4.3, we have described how the proxy's tunneling state is installed; now we discuss when this state is removed. Intuitively, the proxy should maintain the mapping state longer than the legacy application maintains the virtual address returned by the proxy. The proxy refreshes the mapping state every time one of the following two events occur: a packet is forwarded using the mapping state, or a DNS request for that mapping state is invoked. If none of these events happen, the proxy removes the mapping state after a predefined interval of time TO.

We consider two issues when setting the value of TO: the default TCP keep-alive timer which is 7200 sec, and the interaction with the DNS caching. Despite the fact that the proxy always returns a DNS reply with a Time-to-Live (TTL) value of zero, some popular applications (*e.g.,* Internet explorer) ignore the TTL and cache the address. However, in our experience we found that none of the applications we used cache the DNS reply for more than 7200 sec, which is consistent with the findings reported in [16]. As a result, we chose the default value of TO to be 7200 sec. While this is a large value, note that the overhead to maintain the mapping state is minimal when the proxy uses the anycast optimization. In this case, the proxy maintains the mapping state only locally; no trigger is inserted in the infrastructure.

Another scenario in which such incorrect behavior can arise is if the $i3$ proxy is restarted during a application session (gracefully or otherwise). The application would continue to try to connect to virtual addresses which the proxy no longer has state for. To mitigate this problem, the $i3$ proxy logs its state periodically to a file. When the proxy starts up, it can optionally reconstruct its state by reading in the log from the file.

7.3 $i3$ proxy

The $i3$ proxy is implemented as a multi-threaded application with two main threads: the packet capture thread and the overlay interface thread. The packet capture thread reads from the `tun` device to capture application-generated DNS requests and data packets. The overlay interface threads receives packets from the $i3$ infrastructure and sends them to local applications. The $i3$ proxy has been ported to the Windows (XP, 2000) under Cygwin using `vtun` [2] (a Windows port of `tun`). The $i3$ proxy is designed to be highly flexible and allows configuration through an XML configuration file. The configuration file also allows the user to specify which legacy DNS names should be redirected to an $i3$-to-IP proxy. For ease of configuration, these policies are expressed as a sequence of regular expressions for the name, along with a policy for names that match that regular expression (the policy specifies whether to relay through an $i3$-to-IP proxy or directly via IP).

7.4 $i3$-to-IP proxy

The $i3$-to-IP proxy is implemented as two main threads: the $i3$ interface thread and IP interface thread. The $i3$ interface thread interacts with an $i3$ proxy in setting up private triggers etc, while the IP interface thread maintains the mapping between private triggers and ports allocated on behalf of an $i3$ host. The $i3$ interface thread also spawns a DNS helper thread for doing non-blocking DNS lookups: this allows it to scale to multiple users. If the DNS query cannot be resolved, then an error message is returned to the connecting proxy, which then informs the application. The IP interface thread maintains the private trigger to port mapping in two steps: it stores a private trigger to virtual address mapping, and uses a software NAT to maintain the mapping from virtual address to fake port. We used a Linux software NAT implementation for the $i3$-to-IP proxy, which includes packet-rewriting support for several applications (Linux Kernel 2.4.22 has support for FTP, H.323, PPTP, SNMP, TFTP etc.). The $i3$-to-IP proxy 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.

8 Evaluation

In this section, we first present results micro-benchmarking the 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, and it can support high data forwarding rates. We then present wide-area experiments that show that, in practical scenarios, using the proxy along with $i3$ performs almost as well as underlying IP. These experiments also indicate the performance advantages of the shortcut option (especially throughput). Even though we have implemented and deployed the applications in Section 6, we do not present any wide-area benchmarking results for them since the performance does not reflect anything on their instantantion in $i3$ (since the proxy adds minimal overhead, as our microbenchmarking results show).

8.1 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 at least 100 runs (the variation we obtained was minimal).

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 pertinent checkpoints.

$i3$ **Proxy: 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. Figure 11 reports results for the send and receive times of a single packet of size 1200[3] bytes at the sender and receiver. We split up the total send and receive time into three phases: (a) time taken to move a packet between application and proxy, (b) proxy processing overhead, and (c) time taken for proxy to send/receive a packet.

Total send time per packet = 73		
App send to Proxy	Proxy processing	Proxy send
42 (57%)	11 (15%)	20 (28%)
Total recv time per packet = 44		
Proxy recv	Proxy processing	Proxy to App
17 (39%)	4 (9%)	23 (52%)

Figure 11: Split-up of per-packet overhead of send and recv with the proxy. All numbers are in microseconds.

We see that the processing time of the proxy is very little (15% for send and 9% for receive) and much of the overhead is for transferring the packet from the application to the proxy (57% for send and 52% for receive). This overhead can be avoided by a dynamic library implementation as mentioned before. The total send time indicates that the raw throughput that can be sustained is $13,700$ packets/second (or 131 Mbps).

$i3$ **Proxy: Lookup Overhead.** We first quantify the overhead incurred by the proxy in the control path (*i.e.,* resolution of names). 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 (74 microseconds).

8.2 Wide-area Experiments

Over a deployment of $i3$ on PlanetLab, we compare the $i3$ proxy to the underlying IP path based on two metrics, round-trip time (RTT) and TCP throughput. In all our experiments, each proxy was configured to use the closest $i3$ server.

Basic $i3$ **Proxy.** We performed experiments with the proxy running on three machines at different locations: one each at Berkeley, UT-Austin and UIUC. The fact that the basic $i3$ proxy requires administrator privileges made it difficult for us to procure more machines. We measure RTT with an in-house custom tool that uses UDP packets, and throughput with `ttcp`, a popular tool for measuring TCP throughput.

Figure 12 shows the RTT (left) and throughput (right) for the following cases: (a) over $i3$, (b) over $i3$ using shortcut, and (c) direct IP. For $i3$, we consider the first packet RTT in addition because the first packet is routed via Chord. We make this distinction because $i3$ caching ensures that the subsequent packets are sent between $i3$ servers directly via IP.

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

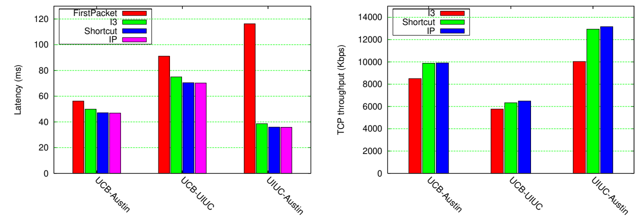


Figure 12: $i3$ proxy wide-area experiments: (a) latency and (b) throughput.

Since Chord routing takes a logarithmic number of hops on average to route to an $i3$ node, the first packet latency is high in one case. The key point to note is that $i3$ latency is only about 7% worse than the IP latency. Furthermore, using shortcuts makes the latency almost equal to the IP latency.

The throughput reduction by choosing an $i3$ path is between $12 - 24\%$. We ascribe this to the fact to the increase in RTT (and hence TCP throughput goes down [17]) and inefficiencies due to additional forwarding through the shared PlanetLab infrastructure. However, using shortcuts achieves a throughput of within 2% of the throughput of the direct path.

$i3$**-to-IP proxy.** Here, we consider the scenario where the $i3$-to-IP proxy is deployed on behalf a set of clients who want to contact legacy Internet servers. Consistent with this scenario, we ran an $i3$ proxy and an $i3$-to-IP proxy on machines in Berkeley, and used this configuration to browse the web.

We quantify the control overhead in resolving a DNS request in three cases. First, we measured the baseline lookup time for a DNS request without proxies to be 1.78 ms, which is nearly the latency between the host and its DNS server. Our second metric was the overhead of resolving a DNS request using an $i3$ proxy configured to route this DNS request via IP. Recall that the $i3$ proxy taps all DNS requests, irrespective of whether they are forwarded over $i3$ or not, and this metric measures the overhead involved. We measured this latency to be 1.82 ms pointing to a processing overhead of 40 microseconds. Finally, we measured the latency of using the $i3$-to-IP proxy for contacting legacy web servers to be 4.24 ms, which is mainly due to the wide-area latency (2.03 ms) between the $i3$ proxy at the client and the $i3$-to-IP proxy via $i3$.

To quantify the cost of $i3$-to-IP proxy on the data path, we consider six Mozilla mirrors (in USA and Europe), and compare the RTT and throughput obtained by using the $i3$-to-IP proxy (with and without shortcuts) against the IP RTT and throughput. For computing RTT, we used `ping`, and for throughput we measured the download time of a $5.5MB$ file.

Figure 13 shows the RTT (left) and throughput (right) re-

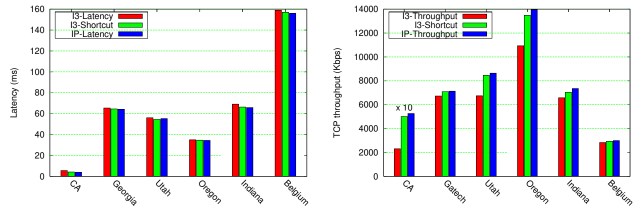


Figure 13: $i3$-to-IP proxy wide-area experiments: (a) latency and (b) throughput. Note that CA mirror's throughput is scaled down by a factor of 10 to fit the scale.

sults. For RTT, the cost of going through the $i3$-to-IP proxy is less than 2% and drops to 0.5% if shortcuts are used. For throughput, the penalty when $i3$ is used is high if the direct throughput is itself very high; in other cases the penalty was less than 20%. For instance, the raw bandwidth to the CA mirror was about 52 Mbps (note that this is scaled down in the figure to fit the scale) whereas the $i3$ throughput was 26 Mbps. However, with shortcuts, the loss in throughput is less than 5%.

9 Related Work

The problem of providing support for legacy applications has been addressed by many proposals that aimed to introduce new functionality in the Internet either in the form of overlays, or new network protocols [6, 7, 15, 16, 29]. One difference between these proposals and our solution is that, to the best of our knowledge, none of these proposals preserve the IP headers in the presence of NATs and mobile hosts. We discuss other differences between these proposals and ours next.

RON [6] uses FreeBSD's divert sockets to capture packets from local applications in a fashion similar to the way our proxy uses the `tun` device. However, RON uses host IP addresses as overlay identifiers, which though suitable for a routing overlay, does not generalize to overlays that support mobility or NATs. ROAM [29] also uses IP addresses as overlay identifiers, but these addresses are different from the end-host IP addresses. This allows ROAM, which is built on top of $i3$, to provide mobility, but no support for NATs.

TRIAD [7] proposes an implementation called WRAP (Wide-Area Relay Addressing Protocol) that modifies the name lookup mechanism and uses a set of Content Resolution Routers for resolving the DNS names. HIP [15] also requires the modification of the name lookup mechanism at end-hosts and relies on Dynamic DNS enabled with DNSSEC or a set of rendezvous servers. Unlike these solutions, our baseline solution[4] requires no changes to the lookup protocols.

AVES [16] uses a set of waypoints to stitch together multiple address spaces through the Internet. Our IP-to-$i3$ proxy uses the waypoint approach, but unlike AVES it does not require the modification of NAT boxes. Instead, the IP-to-$i3$ proxy relies on $i3$ to access machines behind NATs.

We also share the broader objective of providing new functionality to legacy applications with TCP Migrate [23]. However, TCP Migrate [23] modifies the protocol stack, and relies on the DNS infrastructure to provide mobility.

10 Discussion

In this section, we address two natural questions: (a) how can the proxy design be extended to other overlays, (b) what are the fundamental limitations of the proxy design? We then summarize our experience with building the proxy and using it with legacy applications in practical scenarios.

Generalization to Other Overlays. While so far we have only implemented and experimented with the proxy in the context of $i3$, we believe that our design can be easily extended to other overlay networks. Developing the proxy for a new overlay network requires overlay-specific changes to the following: (a) the mechanism to resolve names to overlay IDs (see Section 3.2), (2) the control protocol to set up the mapping state (see Section 4.3), and (3) the data packet encapsulation and decapsulation primitives.

In general, the proxy can be modified to support multiple overlays simultaneously. For instance, one can specify that voice conference application traffic should go over a low-delay RON path, while the traffic to a mobile client should go over $i3$. Supporting multiple overlays is one of our goals for future work.

Limitations. Since the proxy architecturally sits between the network and the transport layers, it might be hard, if not impossible, to enable legacy applications to take advantage of overlay functionalities designed for *native* applications. For example, consider an overlay network that implements multipath routing to improve the end-to-end throughput. In order to provide such functionality to legacy applications, the proxy would need to implement transport layer functionalities such as packet buffering and reordering. In general, the proxy would need to duplicate, at the least, the functionality of the native application in order to provide the same functionality to the legacy applications. Such an approach is neither general nor guaranteed to succeed because of the interactions between the higher layers in the protocol stacks of the native and legacy applications.

Experience. Over three months of using the proxy with $i3$, we learned a few lessons (some of which are obvious in retrospect):

Efficiency matters. In spite of the fact that the $i3$/proxy combination provides many benefits over IP, the users were sometimes not willing to trade performance for these benefits. In particular, they were not willing to use the $i3$-to-IP proxy or redirect their traffic through the intrusion detection

[4] The only exception is the IP-to-$i3$ proxy.

system at the expense of their application performance. This ultimately led us to add the shortcut option to $i3$.

Usage is unexpected. Initially, we expected mobility to be the most popular application of $i3$. However, this was not the case. Instead the users were more interested in using $i3$ 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 proxy communicates with $i3$ via UDP to browse the web through an access point that was configured to block web traffic!

Routing infrastructure helps. The fact that $i3$ is a routing infrastructure, instead of just a lookup infrastructure, allowed us to provide for "free" some very useful functionalities such as access to services and hosts behind NATs/firewalls, and end-host mobility. Among the applications in which users expressed interest, only the intrusion detection application required us to write some code, and even in that case the code was just to interface Bro with $i3$ (see Section 6.4.1).

Different users have different needs. There are several tradeoffs involved in our design and we allow the user to choose his own sweet spot depending on his requirements. For example, preserving IP headers supports applications like `ftp` and middle-boxes like Bro, at the cost of anonymity. Shortcuts provide efficiency at the expense of anonymity and NAT support. An user can control this tradeoff with simple configuration parameters.

11 Conclusion

In this paper, we have described the design and implementation of a proxy-based solution that transparently tunnels the traffic of legacy applications over the $i3$ overlay. This allows end-users to use virtually any existing application to take advantage of the $i3$ functionality. To illustrate this point, we have presented four useful applications: access to machines behind NATs, secure Intranet access, routing user's traffic through and intrusion detection system, and anonymous web download. We have discussed our experience with using the proxy, how this experience led us to revisiting the $i3$ design, and presented several deployment scenarios where users can get partial benefits even when they cannot install the proxy at both end-points.

As future work, we plan to extend the proxy to other overlay networks, and to support multiple overlay networks simultaneously. Another venue of future work is to extend the proxy functionality, for example, to provide protection against DoS attacks. Ultimately, we plan to enlarge our user base and gather more feedback to improve the proxy and the $i3$ design. As our experience showed, users often find unexpected uses to the system, which can push the design in a new direction.

References

[1] `http://www.cse.ohio-state.edu/~jain/refs/refs_vpn.htm`.

[2] `http://vtun.sourceforge.net/`.

[3] Names: Decentralized, Secure, Human-Meaningful: Choose Two. `http://zooko.com/distnames.html`.

[4] The Anonymizer. `www.anonymizer.com`.

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

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

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

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

[9] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services, 2000.

[10] C. Gülcü and G. Tsudik. Mixing E-mail with Babel. In *Proc. of NDSS*, 1996.

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

[12] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica. Taming IP Packet Flooding Attacks. In *Proc. of ACM HotNets-II*, Cambridge, MA, Nov. 2003.

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

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

[15] 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`.

[16] 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.

[17] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *SIGCOMM*, 1998.

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

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

[20] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489, March 2003.

[21] J. Saltzer. On the naming and binding of network destinations. In P. Ravasio et al., editor, *Local Computer Networks*, pages 311–317. North-Holland Publishing Company, 1982.

[22] 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.

[23] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proc. of MobiCom*, 2000.

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

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, 2001.

[26] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *Proc. of NSDI*, 2004.
[27] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *Proc. of NSDI*, 2004.
[28] K. Wehrle, F. Pahlke, D. Muller, et al. Linux Networking Architecture: Design and Implementation of Networking Protcols in the Linux Kernel, 2004. Prentice-Hall.
[29] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host Mobility Using an Internet Indirection Infrastructure. In *Proc. of MOBISYS*, 2003.

APPENDIX: IP-to-$i3$ proxy

This section details the implementation of the IP-to-$i3$ proxy. The IP-to-$i3$ proxy is similar to the $i3$ proxy except that it does not runs on the legacy client itself. This difference leads to two issues: the DNS request made by a legacy client for a $i3$ host should reach a valid name server which the proxy has control over and the address that the name server returns has to be a real Internet-routable address. We address the first issue by registering the IP-to-$i3$ proxy as the authoritative DNS server of a real DNS domain such as `i3.6to4.jp`. For contacting an $i3$ host `foo.i3`, a legacy client now uses `foo.i3.6to4.jp` which is eventually resolved by the IP-to-$i3$ proxy. However, the proxy might not have any information about the client (due to recursive DNS queries). Hence, the proxy returns unique IP addresses to each new request from a legacy client (the DNS response has a TTL of zero).

Similar to AVES [16], the IP address returned in response to DNS queries, is chosen from a set of waypoint machines under control of the proxy. When the proxy returns the IP address of a waypoint machine W to a legacy client C, it informs the waypoint machine W of the destination $i3$ host the client C wishes to connect to. W sets up partial state by negotiating private triggers with the destination $i3$ host. It now awaits a TCP SYN packet, and if one arrives within a period of 1 sec, it assumes that it was sent by the C. It then establishes state associating the IP address of the C with the private trigger IDs. This state is then used to tunnel packets transparently between the legacy client C and the destination $i3$ host.

The IP-to-$i3$ proxy suffers from certain limitations imposed by the requirement of supporting unmodified legacy applications. The main limitation arises due to the nature of the DNS mechanism. The heuristic used by the waypoint to establish state associated with the legacy host limits the rate at which new connections can be initiated through the IP-to-$i3$ proxy. This heuristic is clearly susceptible to DoS attacks where a malicious node simply sends SYN packets to waypoint machines. For the case of HTTP, we have removed this limitation by using HTTP request information in data packets to deduce the destination server the IP client wishes to connect to. The second limitation is that the IP-to-$i3$ proxy has to maintain an address book on behalf of legacy clients.