Freeing The IP Internet Architecture from Fixed IP Addresses

Spencer Sevilla, J.J. Garcia-Luna-Aceves
{spencer, jj}@soe.ucsc.edu
UC Santa Cruz, Santa Cruz, CA

Abstract—**The IP Internet architecture is such that applications must bind fixed IP addresses and ports before any other operations can be executed. These early bindings cause bottlenecks, reliability issues, and force applications and protocols to manage complex lower-layer issues. This poses a big challenge to the future of the IP Internet, given the large and growing numbers of nomadic Internet users, the shift in Internet usage from centralized servers to peer-to-peer content sharing, and the popularity of service replication and virtualization. To address these issues, we introduce and evaluate HIDRA (Hidden Identifiers for Demultiplexing and Resolution Architecture), a novel architecture that creates indirection between layers of any network stack. HIDRA enables sockets and protocols to evolve with the IP Internet by hiding all mobility, multihoming, and multiplexing issues from applications, does not induce significant overhead in the protocol stack, preserves backwards compatibility with today's Internet and applications, and does not require or preclude any additional identifiers or protocols to be used in the protocol stack.**

I. Introduction

The design of the protocol stack of the IP Internet relies heavily upon applications and protocols early-binding the identifiers used by lower layers. For example, applications must bind a fixed IP address and port, which are identifiers at the network and transport layers, before the application can send or receive a message. In turn, TCP itself binds a pair of fixed IP addresses before it sets up a connection or transmits information. In addition to these *explicit* bindings managed by the operating system, applications typically manage several other implicit bindings, such as mapping a host name to a set of fixed IP addresses through the Domain Name System (DNS), or mapping an application-layer service to its corresponding port number.

Requiring higher layers to bind lower-layer identifiers creates significant problems today, and inhibits further Internet evolution tomorrow. By the very definition of binding, an identifier bound by a higher layer cannot subsequently be modified without breaking said binding. For example, requiring a higher layer to bind a network-layer address fundamentally inhibits network address mobility and multihoming, and also implicitly binds the network-layer protocol specified (e.g., IPv4). The same problem exists at the transport layer, where requiring applications to bind a port number (typically known to the application a priori) inhibits dynamically assigning or changing ports at either end of the connection.

We observe that the root of the problem in the two cases above comes from overloading the meanings of identifiers. Semantically, there is an important difference between an IP address and a host, just as there is a difference between a port number and a service, yet Internet applications today have no way of expressing this difference. These problems are well-understood, and have led to a vast set of prior work, which Section III summarizes. This summary includes works that isolate and target particular problems (e.g., maintaining a TCP connection across address changes), as well as proposals for both dirty- and clean-slate redesigns of the network stack.

Building on this point, we also observe that all approaches proposed or implemented to date rely on a layered design that assumes the internal use of what we call *open identifiers*. Open identifiers are transparent values used by communicating parties to name or locate an entity or resource. In contrast to open identifiers, our own recent prior work [1], [2] introduced the architectural concept of a *hidden identifier*, which is an opaque, meaningless value to be used used internally by a communicating entity instead of an open identifier. Section II provides a discussion of how hidden identifiers differ from and relate to open identifiers.

We have provided simple proofs-of-concept implementations of the concept of hidden identifiers [1], [2]. This paper extends this prior work into a robust, three-part network architecture we call **HIDRA** (Hidden Identifiers for Demultiplexing and Resolution Architecture). In addition to significant architectural enhancements and a complete implementation, we created an ecosystem of peripheral tools and applications, and used these tools to provide a much more extensive set of evaluations on the implemented architecture itself.

Section IV describes the necessary modifications to integrate hidden identifiers into any protocol stack, Section V describes the necessary modifications to network applications and the socket API, and Section VI describes how control processes map hidden identifiers to open identifiers, and use this mapping to express a rich set of semantic bindings and policies. Section VII provides implementation details and experimental results collected on several real-world applications and networks, and Section VIII concludes the paper.

II. Hidden and Open Identifiers

As we survey below, a remarkable similarity of *all* prior work on communication-protocol architectures is that it relies heavily on layered designs that use what we call *open identifiers*. Open identifiers are transparent values that encode meaning, are propagated over a network, and are used to name or address[1] a network entity. Because of these characteristics, open identifiers

- Are visible to other network stack layers as well as the end systems or intermediate systems that employ them.
- Are unique and unambiguous within a scope.
- Do not change once bound.

Examples of open identifiers include MAC and IP addresses; port numbers; DNS hostnames; and service, host, or content identifiers.

We recently introduced the notion of a *hidden identifiers* [1], [2], which are opaque values that explicitly encode no meaning and are not propagated among end systems or intermediate systems (hosts or routers) operating in a network. A hidden identifier is mapped to one or more hidden or open identifiers via a table maintained in the operating system, masks the true value(s) and format(s) of the mapped identifier, and is used internally by a layer or application in place of an open identifier.

Figure 1 provides a simple comparison between open and hidden identifiers. Figure 1(a) illustrates the Internet protocol stack today, in which TCP must work directly with open identifiers (the values ip1 and ip2). In contrast, Figure 1(b) shows how TCP would operate using hidden identifiers (the values hidX and hidY) without any knowledge of the corresponding open identifiers.

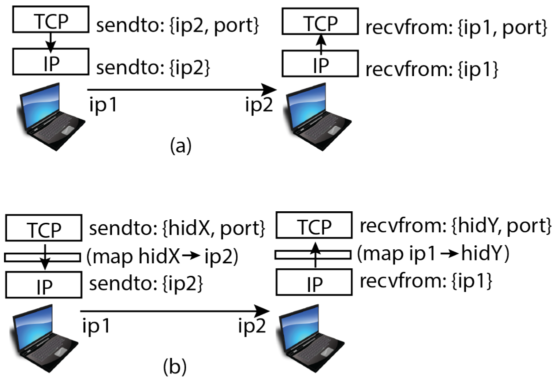


Fig. 1. Open and hidden identifiers

Given that hidden identifiers are not propagated over a network, they must be translated to open identifiers through some process before protocols that rely on them can exchange any messages. For this mapping process to work correctly and support communication, the state of the higher layer must be maintained across lower-layer changes. Thus, hidden identifiers may only multiplex lower-layer open identifiers that are equivalent from the perspective of the higher layer. To this point, we note that the state of the higher layer *includes* any identifiers that it binds. As a result, this requirement ensures that the identifiers used by the higher layer are scoped correctly, and can therefore be resolved unambiguously by the higher layer.

[1]In this context, we use "name" and "address" interchangeably, since as [3] points out, an "address" is just the "name" of a lower-level entity.

A. File Descriptors

While hidden identifiers have not been used in any previous communication protocol architecture, they are not new to operating system design. In particular, the design of hidden identifiers very closely resembles that of file descriptors, which were originally used in UNIX to provide a standard filesystem interface for applications that did not depend on either the physical location of the file or the underlying addressing scheme.

Before the introduction of file descriptors, applications were written for specific hardware profiles. This was a major roadblock to innovation, because even minor changes in the underlying filesystem (e.g., moving a file or altering the storage hardware) broke every application. This problem is analogous to the state of network programming today, where changes in network addresses disrupt connectivity, and changes in network protocols require applications to be rewritten.

B. Characteristics and Benefits of Hidden Identifiers

It is important to stress that hidden identifiers on their own do *not* introduce any new control messages, layers, or bindings into the stack. Building on this point, we also stress that hidden identifiers are simply a new way for end systems and intermediate systems to internally represent the binding process between layers, and are *never* sent over the wire. By definition, any identifier that propagates among systems over the network is an open identifier.

These points are crucial, because the importance of hidden identifiers is not just what they add into to the stack, but what they explicitly do not add. By leaving existing protocols intact, restricting changes to take place within systems, and leaving the data plane unmodified, hidden identifiers provide a solution whereby tremendous flexibility can be injected into the *existing* communication protocol stack *without* requiring a new layer or protocol to be deployed! In this manner, hidden identifiers provide a counterargument to the work by Balakrishnan et al. [4], who claimed that the only way to break the binding between two layers was to introduce an additional layer of identifiers between them.

By breaking these bindings, hidden identifiers enable solutions to a wide range of problems to emerge and be deployed using the existing communication protocol stack. For example, network address mobility and multihoming can be attained without disrupting transport protocols, because they can operate using what amounts to an unchanging virtual address that is mapped to actual addresses at the network layer. By the same token, applications can enjoy the services of different protocol stacks used sequentially or concurrently, and a protocol can benefit from the services provided by multiple lower-layer protocols. These type of advantages are further discussed in Section VII.

Hidden identifiers enable a seamless approach to the evolution of the Internet. While the TCP/IP stack is here to stay for some time, it does *not* preclude the goals of future network architectures. The goals of the alternate identifier bindings and layers proposed in those future network architectures can be accomplished through a combination of signaling protocols that identify and locate desired network resources, identifier multiplexing at end hosts, and using either standard TCP and UDP or a new transport protocol in the data plane.

Hidden identifiers dramatically reduce the ossification of the current protocol stack. By enforcing a clean separation between layers, hidden identifiers provide a mechanism whereby individual layers of the stack can be modified, replaced, or removed entirely without requiring modifications to other layers. This is crucial for future evolution, because it turns the current stack into a modular, evolvable environment wherein changes to one layer do not disrupt the rest of the stack.

III. Related Work and Motivation

Work on the binding of names, addresses, and routes to one another goes back several decades, and due to space limitations we mention only a small fraction of that work. Watson [5] provides an excellent summary of early work on the subject, and Shoch [6] provided one of the most cited characterizations of these concepts: "the *name* of a resource indicates what we seek, an *address* indicates where it is, and a *route* tells how to get there." Interestingly, although these characterizations of bindings among names, addresses, and routes do not advocate *how* they should be carried out, all prior work that implements these bindings uses open identifiers at both end systems and intermediate relays.

A. The Identifier/Locator Split

The challenge of supporting mobility in IP networks has been a primary research motivator for decades, and is fueled by the observation that IP addresses are used to both *identify* hosts and *locate* them in the network. A vast amount of prior work exists, including several surveys of this work [7], [8], [9], [10], [11], [12] and efforts [7], [13], [14] that compare these solutions with respect to a network architecture.

Certain works [15], [16], [17], [18] support address mobility and multihoming entirely within the network layer by providing "shims" that accomplish the identifier/locator split by mapping one identifier (presented to higher layers) to another identifier used for actual network routing. However, these proposals fragment the address space and often introduce triangle routing.

Transport layer approaches [19], [20], [21], [22], [23] propose adapting TCP to coordinate address handoffs, or propose connection multiplexing above the transport layer [24], [25], [26]. These approaches generally ignore the semantics of identifiers and locators, and simply rely on an end-to-end signaling protocol that enables hosts to update addresses as they move in the network.

Other proposals [20], [23], [25], [27], [28], [29] observe that applications typically identify a host through DNS resolution, rather than its IP address. As a result, these works approach the identifier/locator split by arguing for either a socket API or TCP implementation based on *hostnames*, as opposed to network addresses. A number of proposals [4], [24], [30], [31] advocate a similar model based on cryptographic host identifiers in place of hostnames. Unfortunately, these models generally break backwards compatibility. Additionally, they require agreement on, standardization, and widespread deployment of new identity protocols.

Another popular approach is to deploy an entirely separate set of identifiers on top of the physical network, either directly at the network layer [32], [33], [34], [35], [36] or application layer [37], [38], [39], [40], [41]. However, all of these approaches require some degree of encapsulation, and certain network-layer solutions also require custom hardware support in routers and switches. Meanwhile, application-layer approaches incur the significant overhead of propagating overlay network identifiers, either by flooding reachability information or using some form of distributed hash table (DHT). All of these approaches incur significant control message churn when nodes enter or exit the network.

We argue that the root of the identifier/locator problem lies in the design decision to propagate open identifiers across layers: the IP layer uses addresses exclusively for location, and it is only *higher* layers of the stack that semantically equate an IP address to a host. Additionally, we observe the exact same problem at the transport layer, wherein the application layer semantically equates port numbers (e.g., 80) with particular application services (e.g., HTTP).

These problems are the natural consequence of allowing higher layers to bind an open identifier of a lower layer. When open identifiers are exposed to higher layers, they are inevitably ascribed additional semantic meaning by these higher layers, and these additional semantic bindings restrict the lower layer from properly managing its identifiers. This point is underscored by the ability of the system to dynamically select and modify network components that are *not* bound by higher layers; such components include the interface chosen for transmission and the network-layer route between two hosts.

B. Future Network Architectures

Many of the so called "future network architectures" [4], [28], [30], [41], [42], [43], [44], [45] propose introducing one or more new layers of open identifiers as a way of eliminating the naming and addressing problems in the current Internet architecture. FII [28], [45] and Plutarch [46] propose an Internet framework that allows for incremental deployment through heterogeneity between different network domains, and the layered naming architecture [4] and Serval [43] both propose a Service Identity (SID) layer between the network and transport layers.

Other architectures [14], [47], [48], [49], [50] explore the concept of recursion between layers of the stack. This model views each layer as providing an abstract interprocess communication (IPC) service to the layer directly above it, and thus views the entire stack as a recursive series of services that perform both transport *and* routing tasks, as opposed to a model where the entire stack constitutes one distributed IPC service for applications.

Unfortunately, these future network architectures come at a very large adoption price: they require the redesign of network applications and operating systems, and generally also require the replacement of all intermediate hardware (routers and switches).

IV. The HIDRA Protocol Stack

Figure 2 illustrates how HIDRA can be organized into three closely-related and interworking components, detailed in this and the following two sections.

This section describes the core design of how the protocol stack uses hidden identifiers between protocol layers, and how this process works when sending or receiving datagrams. Section V describes how network applications use *peripheral functions* to create an identifier that exactly represents the network resource they desire, and use this identifier to communicate through a protocol-agnostic socket API. Section VI explains how the mapping of hidden to open identifiers is created, maintained, and updated through control processes to reflect the original semantic binding requested by the application.

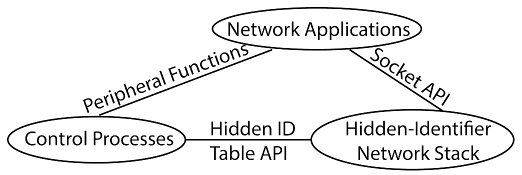
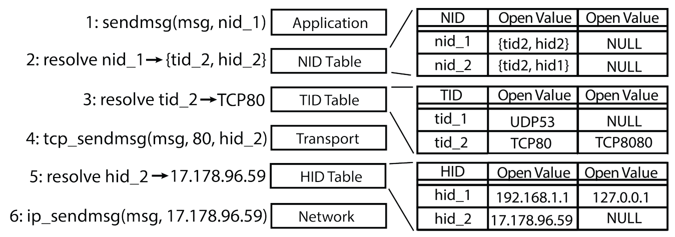


Fig. 2. HIDRA overview

A. HID, TID, and NID Semantics

For successful communication, an end system or intermediate system must be able to map the bound hidden identifiers it uses internally to the open identifiers needed by the protocol stack. In particular, HIDRA employs three sets of hidden identifiers: *Network Identifiers* (NID), *Transport Identifiers* (TID), and *Host Identifiers* (HID). Figure 3 provides examples and illustrates the position of these three identifiers in the protocol stack, and shows how all three hidden identifiers are maintained and multiplexed through tables.


1: sendmsg(msg, nid_1)
Application
2: resolve nid_1→{tid_2, hid_2}
NID Table
3: resolve tid_2→TCP80
TID Table
4: tcp_sendmsg(msg, 80, hid_2)
Transport
5: resolve hid_2→17.178.96.59
HID Table
6: ip_sendmsg(msg, 17.178.96.59)
Network

NID	Open Value	Open Value
nid_1	{tid2, hid2}	NULL
nid_2	{tid2, hid1}	NULL

TID	Open Value	Open Value
tid_1	UDP53	NULL
tid_2	TCP80	TCP8080

HID	Open Value	Open Value
hid_1	192.168.1.1	127.0.0.1
hid_2	17.178.96.59	NULL

Fig. 3. HIDRA protocol stack

A HID is a hidden identifier that sits between the transport and network layers, and maps to one or more network-layer open identifiers (i.e., IP addresses). Since the HID must preserve the state and scope of the transport layer, a single HID may multiplex across different network identifiers owned by the same host, but may *not* multiplex across different hosts. In this context, a "host" can refer to a physical computer, a virtual machine, or any such entity that maintains a transport-layer state. Additionally, HID multiplexing *across* hosts can still work if the transport-layer state is correspondingly migrated from one host to another. However, we leave such an approach to future work.

A TID is a hidden identifier that sits above the transport layer and maps to one or more transport layer open identifiers (i.e., ports). Given that transport-layer identifiers are scoped to a particular host, TIDs are scoped to a particular HID for table storage and multiplexing. As it is the case with an HID, a single TID may multiplex across open identifiers just as long as the corresponding application-layer state is preserved. This enables application-layer services (i.e., a HTTP server) to dynamically bind and migrate ports.

From the perspective of the socket API, replacing a network address with a HID and a port with a TID masks the open values of these identifiers from the application using them. However, simply allowing applications to bind a {TID, HID} tuple as we have proposed in our prior work [2] is still problematic, because such a tuple still implies and requires certain restrictions of the underlying networking stack implementation. These restrictions are: (a) the existence of a transport and a network layer that use open identifiers; (b) the *lack* of any other such identifying layers (e.g., layers that identify services, hosts, or content); and (c) the need by the underlying network stack to use exactly two hidden identifiers. Furthermore, binding a socket to a {TID, HID} tuple ensures that the application is bound to exactly one TID and HID.

We address the above restrictions with the use of NIDs. A NID is a hidden identifier used by applications with the socket API. The NID is agnostic to any specific protocol stack or protocol, and is designed to mask *all* network stack logistics from the application. Thus, the NID can be multiplexed to one or more {TID, HID} tuples, a traditional {IP, port} tuple, a Bluetooth identifier, another NID, or any other such value, including but not limited to a set of one or more identifiers used by a future network architecture. How applications acquire and interact with NIDs is the subject of Section V.

For organizational simplicity, in the remainder of this section we explicitly assume that the application has already obtained a NID that multiplexes to a valid TID and HID, and that the TID and HID correspond to valid open identifiers. The following two sections elaborate on how both of these points are achieved and maintained.

B. Connecting, Sending, and Receiving Messages

Steps 1-6 of Figure 3 illustrate how an application sends a message or connects to a NID.

First, the application passes a NID to the socket API instead of the traditional {IP, port} tuple (Step 1). The system multiplexes the NID to a {TID, HID} tuple (Step 2), translates the TID an open identifier (Step 3), then passes the message to the appropriate transport protocol. The transport protocol processes the message and creates a datagram addressed to the HID (Step 4). When the transport protocol is finished, the HID is translated to a open network address (Step 5), and the network layer processes the packet normally (Step 6).

The same steps are taken whenever data are sent to the socket, regardless of whether the application calls sendmsg() to send a datagram to a NID, connect() to open a stream, or send() to send data to an established stream.

Applications bind a local identifier and receive messages through the inverse of the above steps. After the network layer is done processing a packet destined for the host, the source network address is multiplexed to a HID. If no entry exists in the HID table, as can be the case for an incoming connection to a server, a new HID is generated. The transport layer then processes the packet and multiplexes the port to a TID. The resulting {TID, HID} tuple is mapped to a NID, and then the message is queued for delivery to the appropriate socket.

C. Transport-Layer Changes

As illustrated in Figure 3, the transport layer still uses its own open identifier, but replaces the open network identifier with a HID. Thus, transport-layer protocols must be modified to index connections using HIDs instead of open identifiers. This modification takes place in two different ways. First, the foreign network address is replaced by a HID when storing or looking up connections. Second, the local network address is effectively removed from the lookup tuple. This is needed because, by definition, all packets received by the transport layer are destined to the local host, and a HID referring to the local host would necessarily be the same across local network addresses. When exposed to different layers or bound to NIDs, the local host is denoted as HID 0.

These changes are all that is necessary to ensure successful protocol operation and datagram delivery. However, if the HID is multiplexing across multiple network addresses and routes, datagrams from the same HID may arrive out of order; this is known to negatively affect the performance of certain transport protocols, such as TCP. We address this problem through the inclusion of a small buffer that sits between the HID table and TCP to re-order packets when necessary.[2]

V. Application-Layer Interface

A key goal of HIDRA is that network applications be made as simple as possible. Instead of managing several implicit and explicit identifier bindings, as is the case today, HIDRA applications interact with network resources through two cleanly defined steps. First, they use a *peripheral function* to identify the network resource they desire and map it to a NID. Second, they use this NID to send and receive data through the socket API as in Section IV. This dramatically reduces application complexity, because all other logistical concerns, including the resolution and binding of hidden and open identifiers, are hidden from applications and are managed by the operating system.

[2]We acknowledge that this problem is significant, and can be addressed in several different ways. For the sake of scope, we leave further discussion of this problem to future work that compares and contrasts these different approaches with respect to a hidden-identifier architecture.

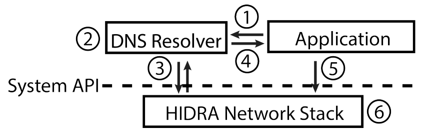


Fig. 4. Peripheral function for DNS resolution

A. Acquiring a Network Identifier

For successful communication to occur in this paradigm, applications must acquire a NID that is bound to their desired network resource. HIDRA applications accomplish this through the use of *peripheral functions*, which embody the interface between the application and the hidden identifier control processes.

Peripheral functions take as input one or more identifiers that unambiguously indicate the network resource desired by the application. The peripheral function then resolves this set of identifiers as necessary, populates and binds the appropriate hidden-identifier tables, and returns a NID to the application.

```
struct sockaddr_hidra sa;
sa.nid = http_resolver("host.com");
sock = socket(AF_HIDRA, SOCK_STREAM, 0);
connect(sock, sa, sizeof(sa));
/* regular socket communication here */
```

Fig. 5. HIDRA example application

Figures 4 and 5 provides an example of how a basic peripheral function could support an HTTP client. The application provides a DNS hostname as input (Step 1), and in turn the peripheral function resolves the hostname to a set of addresses (Step 2), populates the appropriate hidden-identifier tables (Step 3), and returns a corresponding NID to the application (Step 4). Subsequently, the application connects a socket to the NID (Step 5), and the network stack uses the previously populated tables to multiplex hidden to open identifiers as necessary (Step 6).

This design represents a departure from traditional network applications, in that while peripheral functions are often employed by applications to resolve network identifiers, HIDRA *requires* such functions as there exists no other way to obtain a NID. However, this requirement frees the application from managing any other network-related concerns, such as storing and parsing IP addresses returned by a DNS resolver.

Importantly, this design separates the process of binding meaning to identifiers from the process of using these identifiers in the stack. This separation enables the same datapath and protocols (i.e., TCP/IP) to support a rich set of identifiers bound to different semantic meanings! Though Figure 4 illustrates how a peripheral function can bind a NID to a DNS hostname, different peripheral functions could bind a NID to a service identifier, cryptographic identity, or even a content ID!

B. Existing Semantic Mappings

Semantically, the simplest way to assign meaning to a hidden identifier is to create a one-to-one mapping with an open identifier. Table I outlines a set of peripheral functions that provide this basic service, which enables applications to semantically bind raw IP addresses and ports, just as in the current TCP/IP stack.

These helper functions highlight the important difference between an application binding an open identifier because it is exactly what the application desires semantically (i.e., a network utility that explicitly wishes to test the reachability of a particular IPv4 address) or binding an open identifier because the architecture provides no other way for the application to express what is *actually* desired.

Function	Comments
generate_tid_tcp(portno)	Creates a TCP TID
generate_tid_udp(portno)	Creates a UDP TID
generate_hid_ipv4(ip_addr)	Creates an IPv4 HID
generate_hid_ipv6(ip6_addr)	Creates an IPv6 HID

TABLE I
PERIPHERAL FUNCTIONS

C. Future Semantic Mappings

In addition to the semantic bindings that exist today, hidden identifiers can also be semantically bound to a wide range of identifiers proposed by the future Internet architectures referenced in Section III.

Endpoint-centric architectures that support host mobility across network addresses map very well to the HID table. HIDRA can support such an architecture by implementing a directory service or discovery protocol that maps the identifier to a set of network addresses and binds them to an HID.

Alternately, service-centric architectures focus on *application* mobility and replication across multiple hosts. While these architectures generally call for the introduction of one or more new naming layers to uniquely identify these services as they move, we note that the primary function of these layers is not to add end-to-end or intermediate functionality, but rather to mask mobility through the use of an unchanging identifier. This distinction is crucial, because hidden identifiers achieve the same goal by using a peripheral function to locate the service initially, and then sending control messages as the service migrates.

Because multiple peripheral functions may coexist with each other, HIDRA can support several diverse approaches to endpoint- and service-centricity simultaneously! This enables different approaches to evolve over time, without requiring significant modifications to applications or requiring agreement or consensus on a particular protocol or identifier format. Furthermore, it also enables endpoint-centric applications to use an endpoint-centric architecture, and service-centric applications a service-centric architecture, in the same system!

D. Using a Network Identifier

As mentioned in Section IV-A, using NIDs also provides a layer of abstraction that masks the underlying network stack implementation from the application. Thus, depending on the particular network resource requested, as well as the current state of network connectivity, a NID could be multiplexed to a wide range of network identifiers.

This design provides two important benefits for HIDRA-based applications: First, it enables the *system* to manage network identifiers in a way that completely masks them from the application. Second, this support enables the deployment of future identifiers, protocols, and stacks in a way that does not require modification or updates to existing network applications. For example, a new network-layer protocol could be implemented simply by updating the peripheral function in Figure 4 to support it. After this update, the application in Figure 5 would immediately start taking advantage of this protocol *without* the need for any modification.

In addition to these points, the use of NIDs provides a layer of indirection that can multiplex across TID and HID tuples. This provides architectural support for application and service mobility as mentioned in Section V-C, whereby changing the {TID, HID} tuple enables an application to persist a communication session across multiple hosts. Moreover, this allows the same application to be reached at all available network address and port tuples, regardless of the protocols or identifiers used.

VI. HIDRA CONTROL PROCESSES

HIDRA intentionally splits the multiplexing of hidden identifiers in the data path (described in Section IV) from the tasks of populating and maintaining these values in their respective tables. This architectural split enables two key benefits.

First, diverse control processes that create and modify the bindings between hidden and open identifiers can coexist and even work together to aggregate many different forms of information. Second, these control processes can coordinate with the peripheral functions mentioned in Section V to support and maintain the semantic bindings requested by the application.

A. Primitive Table Interface

Control processes interact with the NID, TID, and HID tables through the table-management interface illustrated in Table II. While these functions and their implementation are largely self-evident, the purpose of the bottommost function, set_policy, is more abstract. In those cases in which a hidden identifier maps to more than one open identifier, control processes use set_policy to specify how the system should select an open identifier when sending data. Such policies include round-robin, always choosing a particular address or subnet when available, or weighting certain addresses more than others.

ID Function	Comments
create_id(family, addr)	returns the hidden ID
delete_id(id)	delete a hidden ID
add_oid(id, open_id)	add open ID to a hidden ID's set
remove_oid(id, open_id)	remove open ID from a hidden ID's set
set_policy(id, policy)	set ID muxing policy

TABLE II
HIDDEN-IDENTIFIER TABLE FUNCTIONS

B. Mechanism and Policy

The table management interface is intentionally kept primitive; this choice stems from the system design principle of separating *mechanism* from *policy*. In addition to being good engineering practice, this split keeps the table-interface operations simple and fast, and enables control policies to swiftly be designed, deployed, and automatically integrated into the existing data path.

This roadmap for deployment provides an attractive "third way" when contrasted with the two standard approaches of (a) breaking compatibility by injecting a new layer into the network stack, or (b) injecting additional complexity within a layer by overloading open identifiers or encoding a mapping between them. Rather, with HIDRA, complex and diverse semantic bindings and policies can be simply represented using the functions in Table II. For example, current proposals for identifier mobility or multiplexing generally employ either an end-to-end or a publish-subscribe architecture, yet either architecture can be adapted to HIDRA by modifying them to exist as separate application processes that create and receive control messages, and then express the meaning of these messages through the functions in Table II.

Implementing control signaling this way enables different approaches to coexist and even integrate with each other! For example, publish-subscribe architectures [29], [30], [51], [52] must generally provide some mechanism to ensure that already-established connections are updated as identifiers move. However, given that hidden-identifier tables provide a unifying point for different control processes, such a goal could be accomplished through an entirely separate end-to-end signaling protocol.

VII. EVALUATION AND CASE STUDIES

We implemented HIDRA as a Loadable Kernel Module (LKM) for Linux 3.13.x.[3] Our kernel module consists of a basic HIDRA socket API, NID, TID, and HID tables, as well as the table-management interface described in Section VI.

Additionally, we implemented several different peripheral functions to provide robust functionality for HIDRA applications; these functions include one that maps DNS host names to HIDs, one that maps service-protocol names to TIDs, and one that maps a cryptographic "shared secret" known by an application to a particular TID and HID. Finally, we used these tools to run a series of "case studies" that underscore and evaluate the flexibility, performance, and modularity of the HIDRA protocol stack, in terms of (a) writing HIDRA applications and control processes, (b) porting existing network applications, and (c) supporting new networking paradigms.

[3]Linux 3.13.x was chosen because it is the base distribution for Ubuntu 14.04 LTS, Mint 17, and the current distribution of Raspbian

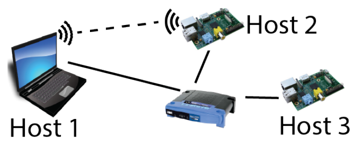
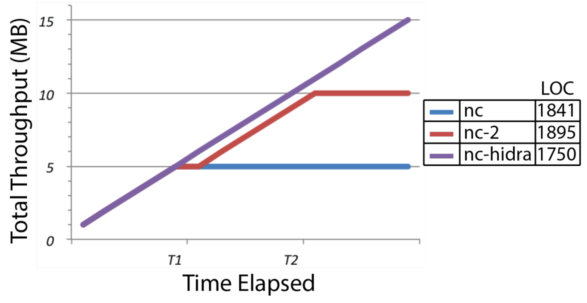


Fig. 6. Testbed topology



	LOC
nc	1841
nc-2	1895
nc-hidra	1750

Fig. 7. Netcat comparison

A. Data-Plane Address and Host Multiplexing

For our first case study, we wrote a HIDRA netcat application, called nc-hidra, which supports both stream- and datagram-based communication. We deployed this application across one laptop and two Raspberry Pis as shown in Figure 6, and registered the set of network addresses of each computer at a local DNS server. We then configured the hidden-identifier tables at Host 1, such that an individual NID (used by nc-hidra) indexed two HIDs (referring to Hosts 2 and 3, respectively), and the HID referring to Host 2 indexed both of its network addresses. Finally, we connected a webcam to Host 1, and used nc-hidra to send datagrams from this webcam to this NID. At time T1, we disconnected Host 2 from the 802.11 ad-hoc network, and at time T2 we disconnected Host 2 from the ethernet network.

With this configuration in place, we compared the performance of nc-hidra to unmodified nc, as well as nc-2, which we modified to be more resilient in the face of disruptions by storing all resolved network addresses for a host and reconnecting if possible. Figure 7 illustrates the performance of all three versions of nc, measured both in throughput received and total lines of application code.

At time T1, standard nc fails, but nc-2 shows that extra application code can mitigate this failure with minimal disruption. However, because each host has a different local DNS entry, even nc-2 is unable to multiplex across hosts and mitigate the complete disconnect seen at time T2. In contrast, nc-hidra uses HID multiplexing to mitigate the first disconnection without any loss in throughput, and NID multiplexing to mitigate the second disconnection just as easily.

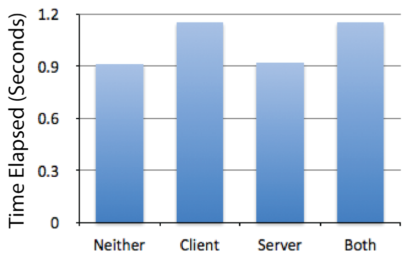


Fig. 8. Hidratunnel overhead

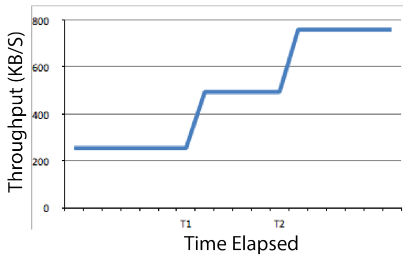


Fig. 9. Link bundling

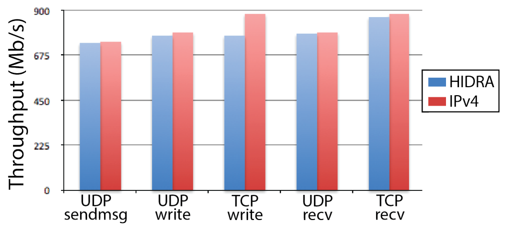


Fig. 10. Multiplexing overhead

B. Adapting Non-HIDRA Applications

In addition to being the only version of the application to persist across all forms of identifier changes, Figure 7 also reveals that nc-hidra requires the least lines of code! This is because all network-related handling is baked into the system itself, as opposed to the network application.

Exploring this point further, we adapted several traditional network applications to use HIDRA and measured the lines of code changed and the total number of lines of code. Our results are shown in Table III, and show that adapting traditional applications to use HIDRA can be accomplished with minimal changes, which typically required between 45 minutes and one hour. In addition, these results also show that in all cases, the HIDRA application is simpler and requires fewer lines of code overall.

Program	Lines Changed	Time Needed	Total Difference
nc	135	0:45	-91
iperf	333	1:15	-288
tftp	119	0:55	-73

TABLE III
LINES OF CODE

C. Legacy Application Support

Building on the above study, we also explored what is possible when the source code of an application cannot be made HIDRA-aware. This may be the case for many proprietary applications, especially those that are not frequently updated or those that have been completely abandoned.

To support these applications, we wrote a simple tunneling proxy application, which we call hidratunnel. hidratunnel supports datagram- and stream-based communication, both client- and server-mode, and works by tunneling a locally-bound INET socket to a foreign-bound HIDRA socket. Thus, by redirecting the unmodified traffic from the application through hidratunnel, the local connection is mapped to a HIDRA NID, and correspondingly receives all the benefits of the HIDRA protocol stack.

After developing hidratunnel, we deployed it with unmodified Firefox on Host 1, unmodified Apache on Host 2, and then timed a 1MB HTTP file transfer 4 separate times: once over regular IP, once with hidratunnel at either side, and once with hidratunnel at both sides. Figure 8 provides these results, which show that hidratunnel does not incur significant overhead when compared to an un-tunneled connection.

D. Virtual Link Bundling

We evaluated the case in which a HID multiplexes to multiple network addresses by cycling through each address in turn, similar to round-robin link-bundling. Notably, because each foreign host has a separate HID and set of network addresses, we found that this enables a unique form of virtual link-bundling wherein different connections can take advantage of different sets of links and addresses.

We explored the performance of this link-bundling effect by connecting Host 1 and Host 2 with three separate links, Ethernet, WiFi, and USB, and using hidra-iperf to measure UDP throughput between the two hosts. To isolate and examine the effect of the bundling itself, we used wondershaper to restrict bandwidth on each link to 256Kbps.

Our throughput test started only using WiFi, and we connected the machines by Ethernet and USB at times T1 and T2, respectively. The results, shown in Figure 9, illustrate that our approach to link bundling takes full advantage of all links, does not introduce significant overhead, and effectively triples the throughput in this experiment.

E. Multiplexing Overhead

The per-datagram identifier multiplexing in HIDRA naturally incurs some performance overhead. To measure this overhead, we ran hidra-iperf over the loopback interface - this test effectively measures the performance and speed of the network stack itself. We tested three different socket API calls: write() requires the socket to have already been connected, sendmsg() requires an unconnected socket (therefore TCP does not support it), and recv() supports both states.

The results of our tests are summarized in Figure 10, and show that across all experiments the difference between HIDRA and IPv4 was consistently small, typically within 10 percent of the base stack. More importantly, the speed of the HIDRA protocol stack is still much higher than most network links, so it does not constitute a bottleneck when compared to other parts of the network.

F. Network Address Mobility

The control processes in Sections VII-A to VII-E populate all necessary hidden identifier mappings at the beginning of the

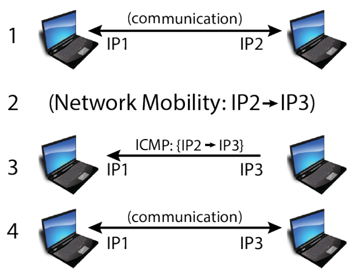


Fig. 11. hid_update ICMP signaling

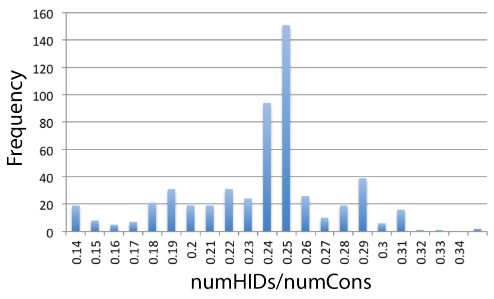


Fig. 12. HIDs/Cons PDF

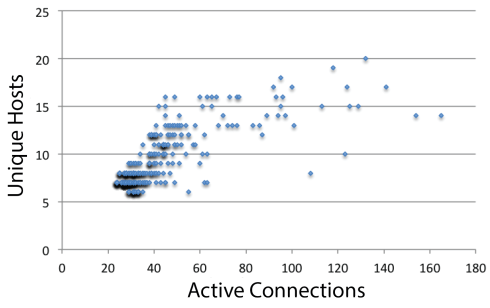


Fig. 13. Connections vs hosts

experiment, and do not subsequently modify these mappings except to delete invalid entries in Section VII-A. However, HIDRA control processes can *also* support mobility by adding, removing, or updating hidden identifier mappings as a host moves throughout the network.

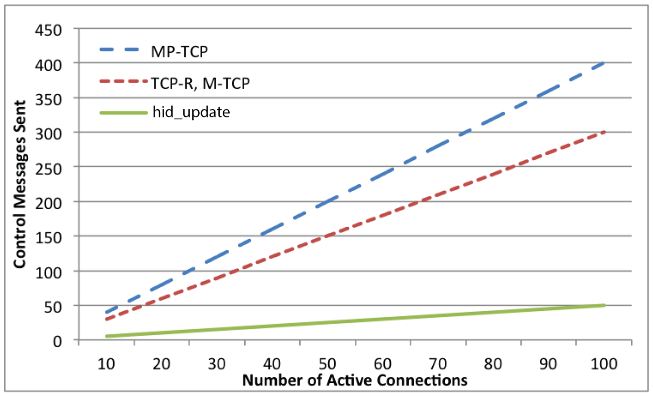


Fig. 14. Scalability comparison

To support network mobility, we created a simple control process called hid_update that runs in a host and monitors the state of the network. Whenever the host adds, removes, or changes network addresses, hid_update sends an ICMP message to all foreign hosts currently communicating with the local host in order to alert them to the change; this is illustrated in Steps 2 and 3 of Figure 11. In turn, when a host receives such a message as in Step 3, it updates its local hidden identifier tables to reflect this change, and this update is reflected in all subsequent communication (Step 4). Finally, any retransmission of misdelivered in-flight datagrams is left to the transport protocol. Having written and deployed hid_update, we then analyzed and evaluated it with respect to several mobility protocols, which we loosely divide into Transport Layer or Network Layer approaches.

1) Connections and Hosts: Most end-to-end mobility solutions today are implemented in TCP, and as a result are executed on a per-connection basis. This introduces a scalability factor, because a mobile host with N active TCP connections must repeat the same migration process N times. In contrast, given that HID tables are kept at each host, HIDRA mobility signaling can be done on a per-host basis.

To explore the relationship between hosts and connections, we wrote a small traffic analyzing tool that logs both the number of active connections and number of unique foreign addresses every five minutes. We ran this tool across several different network clients as they performed normal network activity. With these results, Figure 12 provides a histogram showing that the average value of $\frac{numHIDs}{numCons}$ is roughly $\frac{1}{4}$ (with mean $\mu = 0.242$, and variance $\sigma = 0.04$).

Figure 13 provides a scatter-plot of the collected data points themselves. This plot reveals that, while the observed mean numbers are 37.2 connections and 8.5 hosts, the number of hosts grows much more slowly than the number of connections. At the rightmost part of the plot, we find 165 connections across only 14 hosts! From these plots, we conclude that $numHIDs$ can be approximated by $\frac{1}{4}numCons$, but for purposes of scalability, this comparison should really be considered as an upper-bound on $numHIDs$.

2) Transport Layer Mobility: Most TCP mobility solutions signal network address mobility with a three way handshake that either migrates the initial connection (MP-TCP, TCP-R) or establishes a new connection bound to the same socket (M-TCP). Equations 1-3 quantify the number of end-to-end messages sent or received by an end host when it migrates from one network address to another.

$$Cost_{MP-TCP} = numCons + (3 * numCons) \quad (1)$$

$$Cost_{TCP-R,M-TCP} = (3 * numCons) \quad (2)$$

$$Cost_{hid\_update} = (2 * numHIDs) \quad (3)$$

Figure 14 compares the scalability of Equations 1-3 as measured against the number of active connections at the mobile host. This comparison clearly shows that hid_update incurs less control-message overhead per handoff and scales much better than all other TCP-centric mobility solutions, saving approximately 90-100 messages per handoff if the average number of connections is assumed!

3) Network-Layer Mobility: While pure network-layer solutions have the benefit of masking mobility to all higher layers, they typically introduce issues such as triangle-routing, address space fragmentation, routing table growth, and additional points of failure. In addition to these issues, the need to deploy solutions at intermediate routers, switches, and middleboxes has resulted in such solutions being deemed untenable, and prompted a subsequent move towards end-host solutions enacted at or above the transport layer [7].

hid_update follows this trend of coordinating mobility at end hosts, rather than network entities. However, while hid_update is an end-*host* solution, it is not necessarily end-*to-end*: fundamentally, ICMP messages are still just network-layer communication, and this point enables us to achieve some of the advantages of network-layer proposals. For example, one of the top challenges facing any TCP-based mobility solution is middlebox traversal. While hid_update has no problems traversing middleboxes it may not even have to: if mobility occurs behind a hid_update-aware middlebox performing NAT, the middlebox may simply process the ICMP packet and update its own tables accordingly!

This example illustrates just one of the benefits that are possible by coordinating mobility out-of-band instead of baking mobility into TCP. For the sake of scope, we leave an extensive evaluation of hid_update and comparison with other mobility protocols to future work. However, we do point out that hid_update can achieve the same goals and benefits of Mobile IP *without* incurring triangle-routing in the data plane, splitting the IP address space, or requiring changes to network routers or switches. Additionally, we note that hid_update is just one piece of HIDRA, and does not preclude integration with other protocols or proposals to manage, populate, and update identifiers.

VIII. Conclusions and Future Work

We introduced HIDRA, the first network architecture that splits the semantics and syntax of identifiers used to denote resources or locations. We discussed the theoretical and practical benefits that hidden identifiers provide when used at the interface between layers of the protocol stack, and also explained the restrictions placed on them, including where they can be multiplexed and how they must be maintained at end-hosts. We have shown how HIDRA enables the TCP/IP stack to support alternate semantic identifiers when possible, and how it enables alternate network architectures to emerge to address those cases where TCP/IP is unable to support such identifiers.

We implemented HIDRA as a Linux kernel module, and evaluated it along several performance metrics. We showed how HIDRA divorces network applications from necessary control processes, and how this split enables simple network applications to take advantage of a wide range of network features. Moreover, we showed how this split enables control processes to be standardized and integrated into the system without requiring application integration. Finally, we confirmed that none of these processes or multiplexing incur significant system overhead, and showed how in many cases HIDRA out-performs existing approaches.

Through the use of indirection, HIDRA represents an important first step towards breaking the current reliance on the TCP/IP stack by enabling more diverse identifier meanings as well as incremental evolution between layers of the stack. Moving forward, HIDRA lays the foundation for a wide range of future work on topics such as mobility and multihoming in hidden-identifier architectures, hidden-identifier-centric transport layer protocols, and targeted works on how to support specific future network architectures through a hidden identifier paradigm.

IX. Acknowledgments

This work was supported in part by the Baskin Chair of Computer Engineering at UC Santa Cruz, and by the U.S. Naval Air Systems Command under STTR Project N10A-T006.

References

[1] S. Sevilla and J.J. Garcia-Luna-Aceves. Allowing applications to evolve with the internet: The case for internet resource descriptors. *Proc. IEEE ICC*, pages 3130–3135, 2014.

[2] S. Sevilla and J.J. Garcia-Luna-Aceves. HIDRA: Hiding mobility, multiplexing, and multi-homing from internet applications. *Proc. 17th IEEE Global Internet Symposium*, 2014.

[3] J. Saltzer. On The Naming and Binding of Network Destinations. *IETF RFC 1498*, August 1993.

[4] H. Balakrishnan et. al. A Layered Naming Architecture for The Internet. *Proc. ACM SIGCOMM*, pages 343–352, 2004.

[5] R.W. Watson. Identifiers (Naming) in Distributed Systems. *Distributed Systems–Architecture and Implementation (LCN 105)*, Chapter 9:191–210, 1981.

[6] J. Shoch. Inter-Network Naming, Addressing, and Routing. *Proc. 17th IEEE Computer Society Conference (COMPCON)*, 1978.

[7] W.M. Eddy. At what layer does mobility belong? *IEEE Communications Magazine*, 42(10):155–159, 2004.

[8] E. Perera, V. Sivaraman, and A. Seneviratne. Survey on network mobility support. *Proc. ACM SIGMOBILE*, 2004.

[9] P. Bhagwat and C. Perkins. Network layer mobility: an architecture and survey. *Personal Communications*, 1996.

[10] D. Le, X. Fu, and D. Hogrefe. A review of mobility support paradigms for the internet. *IEEE Communications Surveys & Tutorials*, 8(1):38–51, 2006.

[11] D. Saha, A. Mukherjee, I.S. Misra, and M. Chakraborty. Mobility support in IP: a survey of related protocols. *IEEE Network*, 18(6):34–40, 2004.

[12] C. Perkins and D.B. Johnson. Mobility support in IPv6. *Proc. 2nd International Conference on Mobile Computing and Networking*, pages 27–37, 1996.

[13] Z. Gao, A. Venkataramani, and J.F. Kurose. Towards a quantitative comparison of location-independent network architectures. In *ACM SIGCOMM Computer Communication Review*, 2014.

[14] V. Ishakian, I. Matta, and J. Akinwumi. On the cost of supporting mobility and multihoming. *Proc. GLOBECOM Workshops*, 2010.

[15] E. Nordmark and M. Bagnulo. Shim6: Level 3 multihoming shim protocol for IPv6. *IETF RFC 5533*, 2009.

[16] R. Atkinson, S. Bhatti, and S. Hailes. ILNP: mobility, multi-homing, localised addressing and security through naming. *Telecommunication Systems*, 42(3-4):273–291, 2009.

[17] C. Perkins. Mobile IP. *IEEE Communications Magazine*, 35(5):84–99, 1997.

[18] M. Kunishi, M. Ishiyama, K. Uehara, and H. Esaki. LIN6: A new approach to mobility support in IPv6. *Proc. International Symposium on Wireless Personal Multimedia Communications*, 2000.

[19] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP mobility support for continuous operation. *Proc. International Conference on Network Protocols*, pages 229–236, 1997.

[20] A.C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. *Proc. 6th International Conference on Mobile Computing and Networking*, pages 155–166, 2000.

[21] A. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. *Proc. International Conference on Distributed Computing Systems*, pages 136–143, 1995.

[22] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available internet services using connection migration. *Proc. International Conference on Distributed Computing Systems*, pages 17–26, 2002.

[23] S. Freire and A. Zúquete. A tcp-layer name service for tcp ports. *Proc. USENIX Annual Technical Conference*, pages 275–280, 2008.

[24] D.A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. *Proc. INFOCOM*, pages 1037–1045, 1998.

[25] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath TCP. *Proc. USENIX NSDI*, pages 29–29, 2012.

[26] H. Schulzrinne and E. Wedlund. Application-layer mobility using SIP. *Mobile Computing and Communications Review*, 2000.

[27] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. *IEEE/ACM Transactions on Networking*, 10(5):589–603, 2002.

[28] A. Ghodsi et al. Intelligent Design Enables Architectural Evolution. *Proc. ACM HotNets*, page 3, 2011.

[29] J. Ubillos et al. Name-based sockets architecture. *IETF Draft*, 2010.

[30] R. Moskowitz et. al. Host identity protocol. *RFC 5201*, April 2008.

[31] B.Y.L. Kimura and H.C. Guardia. TIPS: wrapping the sockets API for seamless IP mobility. *Proc. ACM Symposium on Applied Computing*, 2008.

[32] S. HomChaudhuri and M. Foschiano. Cisco Systems' private VLANs: scalable security in a multi-client environment. 2010.

[33] M. et al. Mahalingam. VXLAN: A framework for overlaying virtualized layer 2 networks over layer 3 networks. *IETF Draft*, 2014.

[34] M. et al. Sridharan. NVGRE: Network virtualization using generic routing encapsulation. *IETF Draft*, 2011.

[35] X. Xu and L. Yong. NVGRE and VXLAN encapsulation extension for L3 overlay. *IETF Draft*, 2013.

[36] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009.

[37] E.K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 7(2):72–93, 2005.

[38] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. *Proc. International Conference on Peer-to-Peer Computing*, pages 99–100, 2001.

[39] B. Cohen. The bittorrent protocol specification, 2008.

[40] S. Cheshire and D. Steinberg. *Zero configuration networking: The definitive guide*. O'Reilly Media, Inc., 2005.

[41] I. Stoica et al. Internet Indirection Infrastructure. *Proc. ACM SIGCOMM*, 2002.

[42] B. Ford. Breaking Up The Transport Logjam. *Proc. ACM HotNets*, 2008.

[43] E. Nordstrom et al. Serval: An end-host stack for service-centric networking. *Proc. USENIX NSDI*, 2012.

[44] D. et al. Han. XIA: Efficient Support for Evolvable Internetworking. *Proc. USENIX NSDI*, 2012.

[45] T. Koponen et al. Architecting for innovation. *ACM SIGCOMM Computer Communication Review*, 41(3):24–36, 2011.

[46] J. Crowcroft et al. Plutarch: an argument for network pluralism. *Proc. ACM FDNA '03*, 2003.

[47] J.D. Touch and V.K. Pingali. The RNA metaprotocol. *Proc. International Conference on Computer Communications and Networks*, pages 1–6, 2008.

[48] J. Day, I. Matta, and K. Mattar. Networking is IPC: a guiding principle to a better internet. *Proc. ACM CoNEXT*, 2008.

[49] E. Trouva, E. Grasa, J. Day, and I. Matta. Transport over heterogeneous networks using the RINA architecture. *Wired/Wireless Internet Communications*, 6649(Chapter 25):297–308, 2011.

[50] J. Touch, I. Baldine, R. Dutta, G.G. Finn, and B. Ford. A dynamic recursive unified internet design (DRUID). *Computer Networks*, 2011.

[51] P. Vixie et. al. Dynamic Updates in the Domain Name System. *IETF RFC 2136*, March 2002.

[52] S. Sevilla, P. Mahadevan, and J.J. Garcia-Luna-Aceves. FERN: A unifying framework for name resolution across heterogeneous architectures. *Proc. IFIP NETWORKING*, 2013.