HIDRA: Hiding Mobility, Multiplexing, and Multi-Homing from Internet Applications

Spencer Sevilla*, J.J. Garcia-Luna-Aceves†*
{spencer, jj}@soe.ucsc.edu
*University of California, Santa Cruz, CA 95064
†PARC, Palo Alto, CA 94304

***Abstract*—Today's socket API requires an application to bind a socket to a transport-layer identifier (e.g., TCP80) and network-layer identifier (e.g., an IP address). These early bindings create significant bottlenecks, reliability issues, and force applications to manage complex lower-layer issues. Many approaches have been proposed to address these problems; however, all of them introduce additional identifiers, modify applications, or require additional protocols in the protocol stack. We introduce HIDRA (Hidden Identifiers for Demultiplexing and Resolution Architecture), an approach based on hidden identifiers used internally at end systems and intermediate systems. HIDRA enables sockets to evolve with the 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

Before any application can send or receive messages in today's Internet, it must obtain two identifiers: the destination IP address, which is typically retrieved by resolving a hostname through the DNS, and a tuple consisting of the transport protocol and port, which is typically known a-priori. After binding a socket to these two identifiers, the application may use it to send and receive messages. All other network concerns, such as the interface chosen, or the route between the two hosts, are selected, established and maintained in the system and masked from the application.

The use of identifiers that are *open* (i.e., shared among hosts or routers acting as end points or relays of end-to-end transactions or connections), and the early-binding of a socket to such open identifiers, dates back more than 30 years to the inception of the Internet [1]. However, as the Internet has become ubiquitous and wireless networks and devices have proliferated, new application requirements make the traditional approach to binding untenable. Specifically, supporting multi-homing and mobility of processes, seamlessly multiplexing among multiple network interfaces at each host, and using diverse protocols in wired or wireless networks cannot be accomplished today.

By the very definition of binding, an open identifier specified by the application running in a host cannot subsequently be changed by the network. Thus, requiring the application to denote a resource by means of a network-layer open identifier that is also used by other hosts and routers fundamentally inhibits network-layer mobility and multihoming, and implicitly binds the socket to the network-layer protocol specified (e.g., IPv4). Moreover, it also forces applications to be rewritten to support any new network-layer protocol.

The same arguments apply to the use of transport-layer open identifiers, and to the interface *between* the transport and network layers. Even ignoring issues pertaining to middle boxes, requiring applications to be rewritten to support a new transport-layer protocol is an insurmountable hurdle for the adoption of a new protocol, especially when the deployment of that protocol is not synchronized or coordinated among hosts. Furthermore, the congestion-control algorithm in TCP can adapt to a change in routes or interfaces, yet it cannot accommodate IP address changes, because TCP identifies connections using four open identifiers stated as the tuple {saddr, sport, daddr, dport}.

What is remarkable about the above set of problems is that they do not arise from any intrinsic limitation of the socket API or the protocol stack! There is no fundamental reason why a network application should be rewritten to accommodate a new network-layer protocol, or why TCP cannot migrate across IP addresses: these limitations are simply consequences of the design decision to always use open identifiers to denote resources at hosts and intermediate systems. Surprisingly, *all* prior proposals addressing the naming and addressing problems with today's Internet, which are summarized in Section II, rely on open identifiers as an integral part of the new naming and addressing schemes.

We argue that the current protocol stack is sufficient to handle mobility, multiplexing and multi-homing issues, and that the solution to existing naming and addressing problems in the Internet lies in the interaction between the application, transport, and network layers within an end system or intermediate system. Section III elaborates on the differences between open identifiers that are assigned uniquely to resources and shared among end systems and intermediate systems, and *hidden identifiers* that are known only within a given end system or intermediate system and mapped to open identifiers as needed.

Section IV introduces HIDRA (Hidden Identifiers for Demultiplexing and Resolution Architecture), which is the first

This work was sponsored in part by the Baskin Chair of Computer Engineering at UC Santa Cruz

978-1-4799-3088-3/14/$31.00 ©2014 IEEE

solution that takes advantage of *hidden identifiers* used in hosts. HIDRA uses two protocol-agnostic hidden identifiers: the *Transport Identifier* (TID) and the *Host Identifier* (HID). When an application calls a name-resolution or service-discovery function, it receives a {TID, HID} tuple in lieu of the traditional `sockaddr` structure. The application then uses this tuple with the socket API to send and receive messages. In turn, the operating system demultiplexes the hidden values in the {TID, HID} tuple to process the message. This provides layers of indirection that enable applications to seamlessly migrate across network addresses and entire network protocols.

Section V shows how HIDRA can be integrated with other approaches to provide an effective solution to several existing Internet problems with naming and addressing. Section VI discusses the advantages of HIDRA using an example implementation as a Linux kernel module. The results of experiments based on this prototype show that HIDRA supports mobility and multihoming without requiring major changes to existing applications or introducing any significant overhead in the protocol stack.

II. Related Work

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 [2] provides an excellent summary of early work on the subject. Shoch [3] provided one of the best characterization 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." These primitives were also discussed by Saltzer [4], who pointed out that an address is really just a name of a lower-level entity, and the *binding* process connects a name to a particular address. Interestingly, this characterization of bindings among names, addresses and routes does not advocate how they should be carried out, and assumes that the same identifiers are used by both the end systems and the relays of end-to-end communication.

Many proposals [5], [6], [7], [8], [9], [10] advocate the introduction of new layers of *open identifiers* into the stack as a way of eliminating some of the naming and addressing problems in the current Internet architecture. [6] proposes that applications use a service identifier (SID) provided by the end-user, transport protocols use an endpoint identifier (EID), and network addresses remain unchanged. A similar proposal, Serval [10], identifies the same problems and proposes the introduction of a Service Access Layer (SAL) between the network and transport layers. The SAL redoes the socket API to bind directly to service identifiers (SID) instead of the traditional tuple based on an IP address and a port number.

[9], [11] argue for a network API based on hostnames, as opposed to network addresses, and several object-oriented languages (including Java, Objective-C, and C#) already provide such an API call. These implementations ease software development, but do not keep state or interact with the operating system itself, thus they are unable to support features such as in-flight handoffs or address multihoming.

Several approaches [7], [12], [13], [14] keep network-layer mobility within the network layer by providing "shims" that map one network identifier (typically presented to higher layers) to another used for actual network routing. Unfortunately, all of these proposals fragment the address space, and many introduce triangle-routing.

Raiciu et al [15] provide a multi-path TCP design that focuses on creating a feasible deployment by addressing such issues as backwards-compatibility and middle-box traversal. Their design centers around applications communicating through a "meta-socket," which in turn opens several simultaneous TCP sessions and stripes outgoing data across concurrent connections.

It is apparent from the summary above that all the proposals addressing the name-address binding limitations of the Internet assume that applications and protocols must bind themselves to *open identifiers* (e.g., IP addresses or SIDs) that are known *outside* the end systems or intermediate systems in which the applications and protocols run. Furthermore, assuming the use of open identifiers at hosts and routers, it has been pointed out [6] that the only way to break the early binding between two layers is to introduce an additional layer of identifiers between them. However, this approach still locks the applications, and the socket API or newly proposed network APIs, to particular formats for the open identifiers and the communication protocols using them. This is a big problem for the Internet evolution: just as the designers of the original Internet architecture could not predict today's problems associated with early bindings of names to addresses, it is not possible to predict what problems may result from the use of new open identifiers that must be unambiguous on a network-wide basis. Additionally, requiring applications to use new open identifiers in the API forces application developers to modify applications as the Internet evolves.

III. Open and Hidden Identifiers

Open identifiers are *necessary* for information dissemination to and from end systems (hosts) or intermediate systems (routers, switches, and middle boxes). Destinations must be denoted unambiguously among all the entities involved in any end-to-end information exchange, so that relays can forward the information to their intended destinations. For example, two hosts on the same private network cannot share the IP address `192.168.100.1` or local DNS name `name_1.local`.

While open identifiers are usually thought of as having global meaning, this need not be the case. Multiple types of open identifiers may be needed in the network, because globally unique identifiers may not make sense in certain networks (e.g., Plutarch [16]) and may be considered detrimental in others. For example, a network of things inside a house might prefer to use only local addressing for security, and resource-constrained sensor networks may not be able to afford the overhead of a universal identifying protocol - even the IPv4 header today is considered overly bloated for sensors.

As we pointed out in Section II, starting with the original proposal by Cerf and Kahn [1], all Internet architectures use

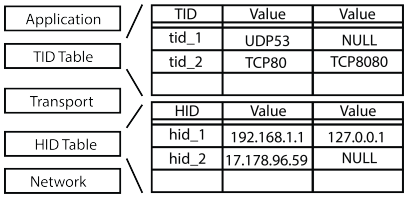

Fig. 1: HIDRA stack at end nodes

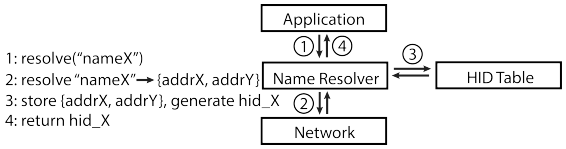

Fig. 2: Name resolution and HID generation

open identifiers exclusively. At first glance, this appears to be a trivial choice, given that they are required to disseminate information across end systems and intermediate systems.

However, the above choice overlooks the fact that the internal management of resources at an end system or intermediate system can be decoupled from the way in which multiple systems collaborate to share information. More importantly, requiring the use of open identifiers for internal purposes at end systems effectively ties applications and higher-level protocols to the specific protocols and identifiers used for information exchange, which significantly inhibits the deployment of any new networking approach based on different types of open identifiers.

The alternative to using open identifiers all the time is to allow applications to use *hidden identifiers* to denote resources and destinations within the systems in which they run. Because open identifiers are needed for communication between systems, the stack of a system in which hidden identifiers are used must translate between them.

Mapping open addresses and ports known within a network to open addresses and ports known outside the network is done today with network address translation (NAT) boxes. However, hidden identifiers have not been used in any previous Internet architecture within a system. Fortunately, hidden identifiers are not new to computing system design. Specifically, file descriptors were originally designed as a part of UNIX to provide a standard 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 had to be written for specific hardware profiles, and this provided a significant roadblock to innovation, given that minor changes in the hardware broke all the applications. 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.

Today, file descriptors allow applications to remain ignorant of lower-level concerns, and this has enabled tremendous innovation in both filesystem and hardware design. Similarly, the use of hidden identifiers in the network stack can provide an architectural solution to the naming and addressing problems in today's Internet by allowing different components of the stack to evolve and change independently of each other. In contrast, an API based on open identifiers is not nearly as modular: by design, an application using a open identifier must specify both the identifier and its format. This implicitly binds the application to whatever values were supplied, and ensures that the application must deal with any change in either value, such as switching addresses or protocols.

IV. HIDRA

HIDRA is based on two main principles. The first is that systems should be allowed to denote Internet resources internally using hidden identifiers. This decouples the applications running in such systems from the network and transport layers, including the open identifiers needed to disseminate information. The second is that systems should provide the mapping from hidden to open identifiers in a way that preserves the existing functionality of the network and transport layers.

By injecting this additional indirection between the network, transport, and application layers, we leave the core TCP/IP stack and intermediate systems unmodified while still achieving strong support for mobility and multihoming. Equally important, HIDRA provides support for incremental evolution and deployment of new networking technologies in layers that were previously considered to be converged-upon and unmodifiable. Figure 1 illustrates the HIDRA network stack at an end system, which uses two tables to manage two separate hidden identifiers. Applications communicate with a socket using a {TID, HID} tuple. The TID table bridges communication between the socket and transport layer, such that the socket uses a TID and the transport layer uses its open identifier. In turn, the HID table bridges communication between the transport and network layers the same way.

A. Populating The TID and HID Tables

Today, applications typically use the DNS to resolve a domain name to a set of IP addresses. HIDRA follows this exact same model, except that it extends the functionality of the name resolver, as illustrated in Figure 2. When `getaddrinfo` resolves a hostname (Steps 1-2), instead of returning the set of IP addresses directly to the application, it stores the set of IP addresses as an entry in the HID table and generates a corresponding HID (Step 3). Lastly, the function returns this HID to the application (Step 4).

Populating the TID table is slightly more challenging, because no such helper methods exist for transport protocols. Thus, the current state-of-the-art can be considered a "magic numbers" approach in that it simply relies on well-known ports corresponding to certain services, such as TCP80 or UDP53. Indeed, this approach has created a new set of problems, such as NAT hole-punching and middlebox traversal.

The most immediate solution for TID table-population is to create a TID that corresponds directly to a transport protocol and identifier, using a simple helper method such as

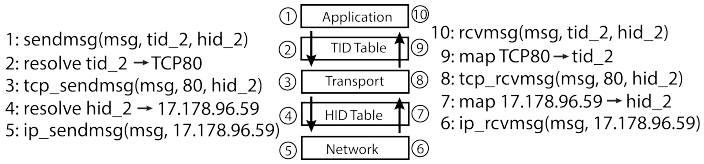

Fig. 3: Sending and receiving messages

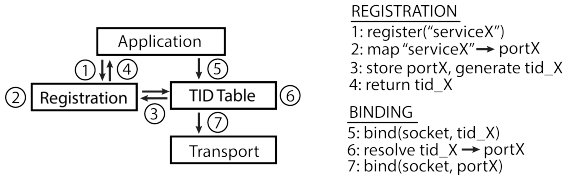

Fig. 4: Registration and binding a socket

`generate_tid(TCP80)`. However, other discovery protocols, such as the mDNS-SD service registry, have been designed to enable applications to reference a service provided on a host by using a string name, such as `_http` or `_printer`. For these protocols, the TID table provides a natural point to aggregate and manage these identifiers.

B. Sending Messages

Once an application acquires a TID and HID, it can then use them to send messages through the standard socket API as shown in Steps 1-5 of Figure 3. When an application sends messages by calling `sendmsg`, the application passes the TID and HID (1) instead of the traditional {IP, port} tuple. The system translates the TID (2), then passes the message to the appropriate transport protocol. The transport protocol processes the message and creates a datagram (3) addressed to the HID. When the transport protocol is finished, the HID is translated to a open network address (4), and the network layer processes the packet normally (5).

C. Binding a Socket

For a server application to receive messages, it must bind a socket to a local IP address, transport protocol, and port. To support the common case where the application wishes to bind across all local IP addresses, the socket API provides the `INADDR_ANY` macro for IPv4, and `INADDR6_ANY` for IPv6.

In addition to binding the socket, an application must somehow publish the identifiers that it has registered before it can receive messages from a foreign host. This step is crucial, because every outgoing connection must already know its destination. However, despite its importance, this step is often overlooked or executed in an ad-hoc manner, such as manually configuring a DNS server or relying on the same a-priori understanding of well-known ports that was first referenced in Section IV-A.

As in Section IV-A, HIDRA provides a solution that abstracts addresses and ports away from the application while simultaneously incorporating this publication process. Any service-discovery protocol must provide a complementary mechanism for service registration, and this function can populate the TID table and generate a TID through either static or dynamic means. In the interim, until such new solutions evolve, an immediate solution exists through creating a simple helper function, such as `register_local_tid(TCP80)`.

In contrast to today's Internet model, binding in HIDRA does not require a local network address. Because one of the goals of HIDRA is to mask network-layer concerns from the application, the default binding behavior assumes that the application intends to bind the socket across all local addresses of all network protocols simultaneously. Section IV-F describes how HIDRA handles the case where this behavior is unwanted.

D. Receiving Messages

Once an application binds a socket, it receives messages through the inverse of the process described in Section IV-B, illustrated in Steps 6-10 of Figure 3. After the network layer is done processing the packet (6), the source network address is multiplexed to a HID (7). If no entry exists in the HID table, as can be the case for a server accepting incoming connections, a new HID is generated. The transport layer then processes the packet (8), multiplexes the port to a TID (9), and finally queues the message for delivery to the appropriate socket (10).

E. Changes To Transport Protocols

For HIDRA to operate, existing transport protocols must be adapted to use HIDs as opposed to open network addresses. This is a simple shift for UDP, given that it maintains no state or connection information. However, this requires important changes to TCP, because HIDRA implicitly changes the {saddr, sport, daddr, dport} tuple used by TCP to identify and lookup connections.

When an application *sends* data using TCP, the socket is already bound to a connection, and thus all that remains is to multiplex the destination HID to an open network identifier, which is a trivial operation. However, *receiving* data requires additional work, since TCP must lookup the corresponding connection to process the packet.

HIDRA changes the TCP tuple for incoming lookups in two key ways: first, the foreign address (saddr for incoming packets) is replaced by a HID by the time TCP receives the packet. Second, the local address is completely removed from the lookup, because HIDRA seeks to mask this concern from the transport layer. Thus, the lookup for established connections consists of a {hid, sport, dport} tuple, and the lookup for listening connections is simply based on the destination port.

This shift is remarkable because it enables address multi-homing and mobility by masking the network address from the transport layer, thus keeping TCP unaware of network-layer changes. Yet, HIDRA is the first approach to accomplish this without introducing any new protocols or layers in the stack.

F. Supporting an Open-Identifier Stack

By design, HIDRA masks the transport and network layer identifiers from the application. Though an application typically has no need to inspect these identifiers, sometimes it is necessary. A tool designed to test the connectivity of a particular

network protocol, for example, will not work successfully if HIDRA masks and changes these identifiers.

HIDRA supports such a requirement in two ways: First, it supports manually creating and editing TID and HID entries through an exposed API. Second, HIDRA sockets are simply created through a new socket family, `AF_HIDRA`, and thus can easily coexist with traditional sockets based on open identifiers.

V. HIDRA and Other New Mechanisms

Remarkably, there already exist many approaches that have been designed to address some of the problems we listed in Section I. Though well-designed, a large number of these approaches cannot be implemented simply because they have no place in the current network stack. These approaches illustrate and prove the strength of HIDRA, because the TID and HID tables provide a natural location to implement and deploy several of them.

There are several proposals [17], [18], [19] to support mobility through in-flight address handovers. These solutions are different architecturally, each has different advantages and disadvantages, and arguably more work will be forthcoming on transport-layer approaches aimed at handling mobility. However, with HIDRA, any of these approaches can be implemented as a method that updates a particular HID as mobility occurs, and this seamlessly integrates mobility support into all higher layers of the stack.

The most prevalent example of a system mapping a host identifier to multiple network addresses is the DNS. By integrating name-resolution into the HID table directly, HIDRA provides integrated support for hostname multihoming, while simultaneously moving this logic *out* of the network application itself, where it typically resides. Other works [7], [15], [11] support multihoming through in-flight methods where hosts exchange additional addresses with each other during or after connection establishment. HIDRA easily integrates such methods into the stack, whereas none of these proposals fully address deployment issues relating to the use of an open network identifier by the application.

There has been very little work on mobility across transport layer protocols and identifiers. It has been assumed that an application wishing to change one of these values would simply create a new socket. Despite the lack of prior work, we believe that dynamically assigning and changing ports can assist with NAT traversal, and even enable process mobility *across* hosts. Moreover, dynamically switching transport protocols gives rise to an entire new set of benefits: for example, a TCP session between two hosts could be seamlessly switched to a multicast transport protocol to accommodate another client requesting the same data. This is a new research area enabled by HIDRA.

Last, it should be noted that the TID *and* the HID can be changed simultaneously. This could enable mobility beyond simply ports or addresses, but across entire network stacks: for example, two nearby laptops communicating over WiFi could switch to Bluetooth or NFC to preserve connectivity when an access point fails.

```
struct sockaddr_hidra sa;
sa.tid = hidra_generate_tid(TCP, 5060);
sa.hid = hidra_getaddrinfo("otherhost.local");
sock = socket(AF_HIDRA, SOCK_DGRAM, 0);
msg = askUserForMessage();
while (msg != "quit") {
     sendto(sock, sa, msg);
     msg = askUserForMessage();
}
```

Fig. 5: HIDRA chat application pseudocode

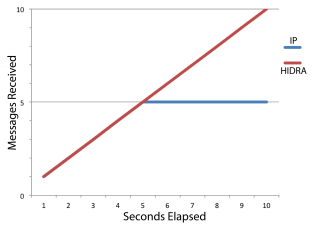

Fig. 6: Chat message-delivery comparison

VI. Prototype Implementation

We developed a prototype implementation of HIDRA as a Linux 3.0.x kernel module. To generate TIDs and HIDs, we wrote simple helper functions to interact with and populate the TID and HID tables.

Our module defines a new socket family, `AF_HIDRA`, and the {TID, HID} tuple as a subtype of the generic `sockaddr` structure. Defining the tuple this way lets us leave the generic socket API fully intact, while still affording us a large address space (14 bytes) for the TID and HID. Our TID and HID table implementations follow a simple policy for address-selection: rank addresses in the same order they are entered into the table. If an error such as `EHOSTUNREACH` is encountered, the offending address is removed and the message is resent using the next address, only returning an error to the application when no more addresses exist. Notably, our module only requires about 600 lines of original code, most of which is devoted to managing the TID and HID tables.

A. Handling Mobility, Multihoming, and Disconnections

To test the functionality of our prototype, we wrote a very simple datagram-oriented chat application, roughly outlined in Figure 5, and deployed it across four computers running Linux Mint 9. The computers are all equipped with WiFi cards and ethernet ports, and the WiFi radios all within broadcast range of each other. As a base-case, we configured the WiFi interfaces into an ad-hoc network with manually-assigned private IP addresses, and ensured that each node could send messages to every other node. With this topology in place, we were able to conduct a series of connectivity experiments highlighting support for network multihoming, failures, and mobility.

In our first experiment, we tested multihoming as well as Internet compatibility by connecting two of the computers to the Internet via ethernet and manually adding their new network addresses to each other's HID table. By manually changing the order of addresses, we were able to seamlessly enable multihoming between these two network addresses, and do so

	UDP sendmsg()		UDP write()		TCP write()	
	IPv4 Client	HIDRA Client	IPv4 Client	HIDRA Client	IPv4 Client	HIDRA Client
IPv4 Server	112.9 MBps	106.4 MBps	135.7 MBps	117.5 MBps	181.7 MBps	145.8 MBps
HIDRA Server	108.6 MBps	104.8 MBps	134.6 MBps	113.8 MBps	175.3 MBps	143.2 MBps

TABLE I: Loopback Throughput

without either (1) changing anything in the application or (2) even alerting the application to the fact that a change in network addresses had occurred.

Building on this, Figure 6 shows the results of our second experiment, in which one application sent messages to another at a rate of one message per second, and at the five-second mark the sender was disconnected from the 802.11 ad-hoc network. Here, the IP-based sender immediately failed to deliver any further messages, yet the HIDRA sender was able to switch network addresses and continue delivery uninterrupted.

Though some network-layer problems can be mitigated by introducing additional application-specific code, this presents the application developer with an additional hurdle and a fundamental tradeoff of effort rendered unnecessary with HIDRA. The pseudocode in Figure 5 shows all that is needed for the HIDRA-based chat application: the code is remarkably simple, and yet it can still support complex network cases that the traditional IP stack cannot.

B. Performance Benchmarks

Any time a message is sent in HIDRA, the two hidden values must be demultiplexed to open identifiers, and this incurs some performance overhead. To measure this overhead in the system, we developed an iperf-style application to measure throughput over the loopback interface, effectively measuring the performance and speed of the network stack itself. We tested two different socket API calls: write() requires the socket to have already been connected to a destination, whereas sendmsg() requires an unconnected socket (thereby TCP does not support it).

The results of our tests are summarized in Table I. Across all experiments we found a consistently small difference between HIDRA and IPv4 in the performance of the receive-path. In the sending path, we observed a growing performance discrepancy between the two stacks as we move left-to-right through the three test cases. This is because the three test-cases are ranked left-to-right in order of the amount of state they maintain for a socket, which also correlates to the amount of fast-path optimization in the kernel. This optimization accounts for the discrepancy, because our implementation is a very non-optimized prototype. Though this discrepancy is substantial for TCP, we believe the same fast-path optimizations for TCP can be applied to HIDRA to reduce this difference: for example, instead of demultiplexing the HID and TID for each individual datagram, the chosen open identifiers could be stored for a socket and simply updated when changed. Addressing such optimizations is fruitful future research.

C. Backwards Compatibility

Because HIDRA works without changing the protocol stack itself, it is fully compatible with existing protocols, including non-HIDRA end hosts. In our tests, we found that HIDRA clients could easily send messages to traditionally bound servers, and HIDRA servers could receive messages from traditionally bound clients. This is a crucial consideration for deployment, because it means that applications and systems can be migrated to HIDRA asynchronously, without fear of breaking compatibility with other endpoints.

VII. Conclusion

We introduced HIDRA, the first proposal to introduce the concept of hidden identifiers used internally by end systems and intermediate systems to support mobility and multihoming, and allow Internet applications and individual layers of the stack to evolve independently of one other. We showed how HIDRA supports multiple network addresses and protocols simultaneously, and that HIDRA provides significant improvements over today's Internet architecture by supporting multihoming and mobility without sacrificing performance.

References

[1] V. Cerf and R. Kahn. A Protocol for Packet Network Interconnection. *IEEE Trans. Commun.*, pages 637–648, 1974.

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

[3] J. Shoch. Inter-Network Naming, Addressing, and Routing. *17th IEEE Computer Society Conference (COMPCON 78)*, 1978.

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

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

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

[7] R. Moskowitz et. al. Host identity protocol. *RFC5201, April*, 2008.

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

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

[10] E. Nordstrom et al. Serval: An end-host stack for service-centric networking. *Proc. 9th USENIX NSDI*, 2012.

[11] J. Ubillos et al. Name-based sockets architecture. *IETF Draft: draft-ubillos-name-based-sockets-03 (work in progress)*, 2010.

[12] Erik Nordmark and Marcelo Bagnulo. IETF RFC 5533: Shim6: Level 3 multihoming shim protocol for IPv6. 2009.

[13] Randall Atkinson, Saleem Bhatti, and Stephen Hailes. Ilnp: mobility, multi-homing, localised addressing and security through naming. *Telecommunication Systems*, 42(3-4):273–291, 2009.

[14] Charles E Perkins. Mobile ip. *Communications Magazine, IEEE*, 35(5):84–99, 1997.

[15] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. 12:29–29, 2012.

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

[17] K. Brown and S. Singh. M-tcp: Tcp for mobile cellular networks. *ACM SIGCOMM Computer Communication Review*, pages 19–43, 1997.

[18] D Funato, K Y., and H. Tokuda. TCP-R: TCP mobility support for continuous operation. pages 229–236, 1997.

[19] A. Bakre and BR. Badrinath. I-TCP: Indirect TCP for mobile hosts. *ICDCS '95*, pages 136–143, 1995.