# UC Santa Cruz

## UC Santa Cruz Previously Published Works


### Title

Allowing applications to evolve with the Internet: The case for Internet Resource Descriptors


### Permalink

https://escholarship.org/uc/item/5ch8f5w8



### Authors

Garcia-Luna-Aceves, J.J.
Sevilla, Spencer



### Publication Date

2014-06-01

Peer reviewed



eScholarship.org Powered by the California Digital Library
University of California

# Allowing Applications To Evolve with The Internet: The Case For Internet Resource Descriptors

Spencer Sevilla*, J.J. Garcia-Luna-Aceves*†
{spencer, jj}@soe.ucsc.edu
* UC Santa Cruz, Santa Cruz, CA
† Palo Alto Research Center, Palo Alto, CA

***Abstract*—Today's socket API requires an application to bind a socket to a network address before it can use the socket to communicate. Early bindings of names to addresses create significant bottlenecks, reliability problems, and force applications to manage complex lower-layer issues. Many approaches have been introduced to address this problem; however, all prior proposals introduce additional identifiers, modify applications, or require additional protocols in the protocol stack. In contrast, we propose a generalized socket API based on *Internet Resource Descriptors* (IRDs), which are opaque identifiers used by applications to refer to network resources and are known only within the hosts in which the applications run. IRDs enable sockets to evolve with the Internet by hiding mobility, multihoming, and multiplexing issues from applications, do not induce significant overhead in the protocol stack, preserve backwards compatibility with today's networks and applications, and do not require additional identifiers or protocols to be used in the protocol stack.**

## I. Introduction

A network application operating over the Internet requires the description of a service, content, or destination to be mapped into a route over which the service or content can be provided or the destination can be reached from the source. Since the inception of the Internet [1], this mapping has been carried out through a number of indirections aimed at separating the description of *what* an application requires (names) from *where* (addresses) it is and *how* (routes) to reach it [2]. Today, a network application wishing to communicate with a remote process over the Internet carries out a two-step resolution process based on the domain name system (DNS), as illustrated in Fig. 1. First, the application contacts its DNS resolver, which is another process located in the same host, to map a user-friendly name into a set of IP addresses. In turn, the DNS resolver interacts with the DNS and eventually resolves the name provided by the application to a set of IP addresses. The application then binds the resolved name to a specific IP address by selecting one of the IP addresses from the set provided by the DNS resolver, and passes it to the socket API to send messages to the remote process. The routes between the host where the application runs and the host where the remote process resides are established and maintained in the system based on IP routing tables.

The existing algorithms for name resolution and name-address binding have served the Internet well over many years.

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

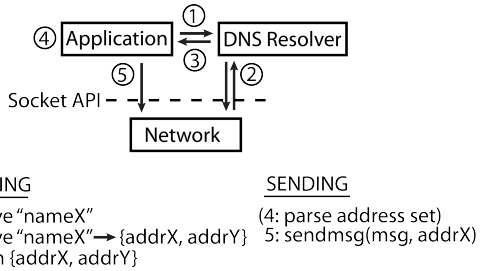


Fig. 1: Traditional name resolution and name-address binding

However, as the Internet has become ubiquitous and wireless networks and devices have proliferated, new application requirements make the traditional approach to name resolution and name-address binding untenable. Specifically, supporting multi-homing and mobility of processes, seamlessly multiplexing among multiple network interfaces at each host, and using diverse protocols in wireless networks cannot be accomplished today.

Not surprisingly, as Section II summarizes, a large body of work has been aimed at making naming and addressing more responsive to the new realities of the Internet. The key problems that these proposals address are the "early binding" established between the name of a process and the address where it can be provided, and the need for applications to monitor and manage this binding. Interestingly, all prior approaches address these problems by introducing additional layers of *transparent identifiers* (i.e., known outside a host) into the protocol stack. Using these identifiers requires new communication protocols to help solve the name-address binding problem, and changes the interface provided to end-user applications. We argue that this is not the right approach to solving the naming and addressing problem of the current Internet architecture, because it creates significant roadblocks to the adoption of any of these new approaches. These roadblocks typically include requiring the support of middle-boxes, requiring changes to the network routing core, or requiring changes to applications (which is arguably the greatest roadblock) by changing the socket API.

Returning to Fig. 1, the dashed line illustrates the socket API between the application and the operating system where it runs. It is apparent from the figure that the application binds itself to a particular name for the service or content it requests, and then *also* binds itself explicitly to a particular

IP address by selecting one from the set provided by the DNS resolver. The first binding is unavoidable, because any application must identify *what* network resource it requires, and the operating system cannot assume this choice. However, any demultiplexing and further binding by the application is unnecessary and detrimental. These additional bindings require applications to be rewritten to take advantage of system innovations, prevent old protocols from being "retired" without breaking backwards compatibility with older applications, and prohibit mobility and multi-homing. Additional bindings also add complexity into applications, including logic which is often duplicated across all applications (such as selecting an IP address from a set).

To address these problems, we introduce a new approach that allows applications to "evolve with the Internet" by freeing them from managing the demultiplexing of addresses and the binding of names to addresses. What is novel about our approach is that we do so without introducing, requiring, or preventing the use of any new identifiers, services, or protocols in the Internet stack. This achieves support for several new network features (such as mobility and multihoming) today while maintaining backwards compatibility, supporting future extensibility, and avoiding the roadblocks listed above. The basis of our approach consists of augmenting the current use of identifiers (names or addresses) that are the same across an entire network, which we call *transparent identifiers*, with identifiers that are known only inside the host where named processes are running, which we call *opaque identifiers*. Section III addresses the benefits of using opaque identifiers.

Section IV describes our approach, which introduces *Internet Resource Descriptors* (IRDs) as an integral part of the existing socket API. IRDs are protocol-agnostic *opaque identifiers* known only inside a host that replace network addresses or other transparent identifiers in the current socket API. When an application calls a name-resolution or service-discovery function, it receives an IRD instead of an address. Once the application has an IRD, it uses the descriptor with the socket API to send and receive messages, and the operating system then demultiplexes the resource descriptor to a network address. This provides a layer of indirection that enables applications to seamlessly migrate across network addresses and entire network protocols.

Section V discusses the advantages of IRDs implemented as a Linux kernel module. We present the results of experiments based on this prototype, which show that IRDs support 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 a small fraction of that work. Watson [3] provides an excellent summary of early work on the subject. Shoch [2] 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." This set of primitives was 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. It is implied that a particular layer in the network maintains and manages its named bindings to the next layer down. Interestingly, early works on the characterization of bindings among names, addresses and routes do not advocate how they should be carried out.

Several proposals have been made on how to evolve the Internet to address its current limitations with naming and addressing. FII [5], [6] and Plutarch [7] highlight the fact that new solutions cannot be deployed incrementally, and must be uniformly adopted simultaneously. To address this problem, these proposals advocate for an Internet framework that allows for heterogeneity between different network domains (referred to as "contexts" in Plutarch). Both Plutarch and FII advocate a new network API, and FII suggests some guidelines for its design, but neither proposal provides a model of what this API should be, its implementation, or a roadmap for migrating applications to use it. Ghodsi et. al. [6] briefly propose that a future network API should be based on hostnames, as opposed to network addresses. However, they describe the network API as something that must be redone from the ground-up, without specifics.

Many proposals [8], [9], [10], [11], [12] advocate the introduction of new layers of transparent identifiers into the stack as a way of eliminating some of the naming and addressing problems in the current Internet architecture. In [9], the authors propose that applications start with a service identifier (SID) provided by the end-user, resolve it to a set of endpoint identifiers (EID), and then choose one to bind to a socket. EIDs are used only by the transport layer, and are translated and bound to network addresses in order for routing and communication to occur. A similar proposal, Serval [12], 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.

As we have stated in Section I, a striking feature of today's Internet architecture and all the proposals addressing its name-address binding limitations is that they all assume that applications must bind themselves to *transparent identifiers* (e.g., IP addresses or SIDs) that are known *outside* the hosts in which the applications run. It has been pointed out in [9] that, if transparent identifiers are used, then the only way to break the early binding between names and addresses is by introducing additional layers of such identifiers and the protocols and interfaces needed to use them. However, this approach still locks the applications, and the socket API or newly proposed network APIs, to particular formats for the transparent 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 identifiers that must be unambiguous on a network-wide basis. Furthermore,

requiring applications to use new identifiers in the API forces application developers to modify applications as the Internet evolves.

## III. The Argument For Opacity

### A. Identifiers and The Network

A resource or destination must be denoted with the same identifier by all network forwarding devices (e.g., middle boxes and routers) in order for data packets to reach their intended destinations. Hence, from the perspective of the network, only transparent identifiers are useful.

The very nature of transparent identifiers requires that they be unique, unambiguous, and supported within the domain in which they are to be used. For example, two hosts on the same network cannot share the private IP address `192.168.100.1` or multicast DNS name `name_1.local`. However, all transparent identifiers need not be global. Multiple types of transparent identifiers may be needed in the network, because globally unique identifiers may not make sense in certain networks (e.g., Plutarch [7]) and may be considered detrimental in others. For example, a network of things inside a house might prefer to only use local addressing for security, and an extremely resource-constrained sensor network may not be able to afford the overhead of a universal identifying protocol - even the IPv4 header today is considered overly bloated for sensors.

### B. Identifiers and End Systems

Transparent identifiers have been used to denote resources in end systems (hosts) in all Internet architectures, starting with the original proposal by Cerf and Khan [1]. At first glance, this appears to be a trivial choice, given that intermediate systems (routers, switches and middle boxes) require the use of transparent identifiers. However, this choice overlooks the fact that end systems manage resources individually, while intermediate systems do so in coordination with other systems. More importantly, it ties the application developers to specific protocols and types of identifiers used in the network, which inhibits the deployment of any new networking approach based on new transparent identifiers.

To allow the applications and the Internet to evolve more freely, a host should be allowed to denote resources by means of opaque identifiers known only within the host, and translate the opaque identifiers it uses to transparent values before packets are sent out over the wire. Given that opaque values are meaningless until translated by the system, they can easily support multiple network stacks and protocols simultaneously, as well as switching between them. Moreover, they can do so without requiring additional overhead, coordination, or a separate identity layer.

Interestingly, while opaque identifiers have not been used in any previous Internet architecture, they 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 their introduction, 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.

By adopting file descriptors, applications remain ignorant of lower-level concerns, and this has enabled tremendous innovation in both filesystem and hardware design. Similarly, the use of opaque identifiers in the network stack provides an architectural solution to the majority of problems in today's networks by allowing different components of the stack to evolve and change independently of each other. In contrast, an API based on transparent identifiers is not nearly as modular: by design, an application using a transparent 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.

From the perspective of the application, opaque identifiers enable simple applications to take advantage of a wide range of network features. More importantly, applications written using opaque identifiers can automatically "opt-in" to new network features without being re-written. Application developers do not have to, and are not able to, make any assumptions about network addresses, protocols, or stacks being used, or the features provided by the host stack.

### C. Identifiers, Locators, and Translation

[10], [13], [14], [15] have each proposed a network stack architecture that translates a *host identifier*, such as a DNS hostname or HIP identity, into a *network locator* (e.g. an IPv4 address). These proposals adapt the socket API and transport layer to bind directly to the host identifier, and then translate the identifier to the network locator between the transport and network layers. Such architectures enable network-layer mobility and multihoming, and are particularly appealing because they accomplish this without injecting an additional naming layer into the stack.

However, in each of these proposals, the host identifier is still transparent, so the constraints of the chosen host identifier still apply. For example, DNS-based sockets cannot support environments where DNS is inappropriate, such as MANETs, and HIP-based sockets cannot support resource-constrained environments, such as sensor networks. Moreover, since all of these proposals translate between the transport and network layers, they are implicitly bound to the TCP/IP stack, and therefore cannot support alternate network stacks such as Bluetooth, Zigbee, or NFC. Thus, these proposals merely substitute one set of identifiers for another, and remain fundamentally unable to support a wide range of future Internet proposals, such as as FII and Plutarch.

## IV. IRDs: Internet Resource Descriptors

We propose to allow applications to evolve freely with the Internet by adopting *Internet Resource Descriptors* (IRDs) into the socket API as an extrapolation of file descriptors.

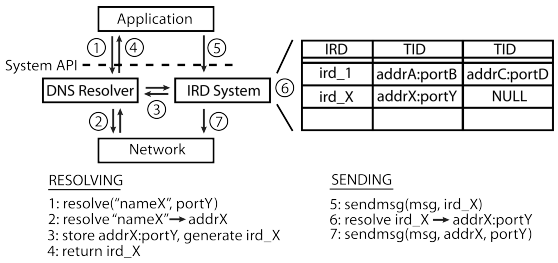


Fig. 2: Name resolution with IRDs and DNS

By design, an IRD is an opaque identifier, protocol-agnostic, known only within the host in which the applications using the identifier run, and meaningless until it is translated by the system to bind it with a transparent identifier associated with one or more places in the Internet. An IRD could represent some content, another host, a peripheral device connected through Bluetooth or NFC, an Internet application distributed across multiple machines, or even a toaster or light switch in the coming Internet-of-Things.

### A. Acquiring a Resource Descriptor

Applications acquire an IRD through any service-discovery or name-resolution function that would traditionally return a transparent identifier (TID) to the application. Figure 2 illustrates this process for the case of DNS. When the application calls the name-resolution function, the function uses whatever resolution protocol is appropriate (Steps 1-2 in the figure). Note that which values are supplied by the application, and which are generated by the resolver, largely depends on the resolution protocol used. For example, the DNS has no understanding of port numbers, so the application must provide a port, whereas the mDNS-SD service registry enables applications to reference a service using its name directly, and other protocols such as Bluetooth coordinate through other discovery methods.

After resolution is complete, instead of returning the TID directly to the application, the function stores the TID as an entry in the *resource descriptor table* (RDT) and generates an IRD corresponding to the RDT entry (Step 3). Last, the function returns this IRD to the application (Step 4).

An important part of IRD acquisition is that it takes place through a peripheral resolution function, and not the socket API itself. This split is crucial, because it supports a much more diverse set of resolution protocols. In addition to DNS hostnames, proposals for resource discovery today include attribute-based querying (e.g. type=printer, loc=lab5), implied scoping (the Bluetooth device currently paired with my computer and labeled as "My Phone"), or cryptographic IDs (such as HIP [10]). It would be exceedingly difficult to design a single, unifying socket function call that could support such a diverse set of resolution protocols, yet each protocol can easily exist as its own specific resolution function with its own parameters, as long as the function populates the RDT and returns a corresponding IRD. Building on this argument, other approaches to IRD acquisition could emerge, such as obtaining an IRD from another application (via an IPC process) or specifically tailored helper functions. These methods allow for intricate relationships between applications to emerge, and could also provide additional security measures, such as obtaining an IRD from a "black box" function without knowing what was resolved or where the IRD points to.

In line with this decision, DNS resolution through the `getaddrinfo` function is currently implemented as a user-space library function, as opposed to a kernel-space syscall. This design keeps the complexity of DNS resolution out of the kernel, which contributes to system speed and stability. By following this model, we maintain these same goals, and achieve another significant benefit of lowering the bar needed to deploy a new resolution protocol. Deploying or updating a function implemented in a user-space library is far less challenging or risky than a kernel-level change, and this allows a vast set of different resolution protocols to be developed and distributed without compromising or affecting the operating system itself.

### B. Sending Messages

Once an application has acquired an IRD, it can use it to communicate with the standard socket API, as shown in Steps 5 to 7 of Figure 2. When an application sends messages with a socket by calling `sendmsg`, instead of passing a network address, the application passes the IRD it acquired. The system uses the RDT to translate the IRD to an appropriate TID and protocol, and completes the operation using this value. Note that the bottom-most box is labeled as "Network" instead of "IP," given that demultiplexing an IRD could result in an IP address and port, a Bluetooth ID, a HIP identity, an EID, or any other transparent identifier, depending on the RDT entry.

Similar to file descriptors, a separate RDT is kept for each process. This alleviates security issues (such as a malicious process poisoning the RDT), simplifies RDT management (e.g. creating/destroying entries), and reduces the work necessary to translate an IRD.

Discovery protocols often return several network addresses for a single request. In this case, the RDT stores the entire set of addresses for the IRD. When an application sends messages to an IRD with multiple address values associated with it in the corresponding RDT entry, the system demultiplexes the IRD and selects a particular address. This helps keep applications simple, yet still enables them to take advantage of integrated support for features such as mobility and multihoming. It also enables policies to quickly be implemented system-wide, such as preferring one interface or protocol to another. Moving the demultiplexing to the system enables greater optimization and decision-making with a more complete perspective of the state of the network, and also enables the system to mask recoverable errors (e.g., routes changing, or one network address going down) from the application entirely.

### C. Binding And Receiving Messages

For an Internet application to receive messages in the traditional Internet architecture, it must accomplish two tasks: First, it must bind a socket to a particular network protocol and protocol-specific identifier, such as an IP-port tuple. Second, it

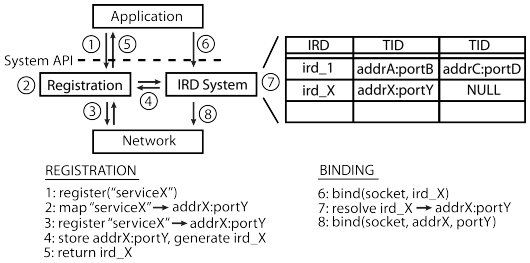


Fig. 3: Binding to an IRD

must announce its presence by registering the identifier with a discovery or resolution service. Despite its importance, this second step is typically overlooked or executed in an ad-hoc manner, such as manually configuring a DNS server or relying on a priori knowledge that certain ports correspond to certain services. The one exception to this is the mDNS API, which requires applications to programmatically announce their services as user-friendly names.

Following this model, we propose the introduction of registration functions that complement the resolution functions described above. Applications wishing to receive connections must first use a function to register the resource or service they wish to provide, and the registration function returns an IRD to the application. This process is illustrated in Steps 1-5 of Figure 3, and is largely similar to the resolution process shown in Figure 2. Once a service is registered and mapped to an IRD, the application may then bind a socket to this IRD the same way it binds a socket today, as shown in Steps 6-8.

This process enables an application to identify itself using only the appropriate registration function, and still receive messages across multiple network addresses and stacks.

### D. Connection-Oriented Protocols

Connection-oriented protocols typically provide guarantees, such as reliable in-order delivery. Dynamically changing the addresses or protocols used can potentially violate these guarantees unless such a handoff is coordinated by the protocol itself. Thus, when a connection-oriented socket (indicated by the `SOCK_STREAM` argument) is bound to an IRD, the system must not dynamically change the TID.

However, connection-oriented protocols still benefit from the use of IRDs, since they ensure that changes to a transport protocol do not propagate up or down the stack. For example, there exist several different proposals for TCP multihoming and mobility, ranging from opening multiple simultaneous TCP sessions [16] to implementing one of many solutions [17], [18], [19] designed for in-flight 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, from the perspective of the application and the rest of the network stack, all of these approaches are identical: the IRD is unchanged, and connectivity is preserved.

## V. Prototype Implementation

We developed a prototype implementation of IRDs as a Linux kernel module, deployable on any distribution based on Linux 2.6.x or 3.0.x. We chose to develop a kernel module to reduce compilation effort and to make our code more portable. We wrote a custom `getaddrinfo` function that interacts with the RDT to resolve DNS hostnames to IRDs instead of IP addresses, and added IPv4 and IPv6 support into the RDT.

```
port = 5060;
ird = getaddrinfo("otherhost.local", port);
sock = socket(AF_IRD, SOCK_DGRAM, 0);
msg = askUserForMessage();
while (msg != "quit") {
     sendto(sock, ird, msg);
     msg = askUserForMessage();
}
```

Fig. 4: IRD Chat App Pseudocode

Our prototype works by implementing a new socket family, `AF_IRD`, and defining IRDs as a subtype of the generic `sockaddr` structure. Defining IRDs this way lets us leave the generic socket API intact, while still affording us a large address space (twelve bytes) for identifying Internet resources. These sockaddrs are translated by our prototype RDT, which follows a simple policy: it stores and uses addresses in the same order they are entered into the RDT by a resolution protocol. If an error is returned when sending a message, 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.

### 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 4, that uses IRDs 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 RDT. By simply changing the order of addresses in the RDT, we were able to seamlessly enable multihoming between these two network addresses, and do so without either (1) having to change anything in the application or (2) even alerting the application to the fact that a change in network addresses had occurred. Building on this experiment, we induced failures into a network path by unplugging the ethernet cable or disconnecting from the 802.11 ad-hoc network, at either the client or server side. When these failures occurred, the RDT removed the invalid network address and switched over to the other one automatically, again without requiring anything from the application. When we ran this same experiment over the traditional IP stack, the application was unable to deal with any of these cases. On the client side, the application was forced to re-

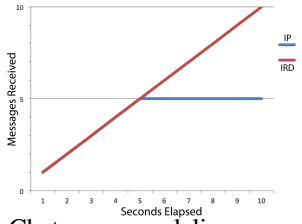


Fig. 5: Chat message-delivery comparison

resolve a hostname or manually manage a list of network addresses in order to continue to send messages after a network disconnection. Conversely, on the server-side, the bound socket was unable to receive messages and had to be restarted after a network handoff.

Figure 5 shows the results of a simple experiment in which one application sent messages to another at a rate of one message per second, and at the 5-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 IRD sender was able to switch network addresses and continue delivery uninterrupted. Clearly, some of these network problems can be mitigated by introducing additional application-specific code to handle network-error cases; however, this presents the application developer with an additional hurdle and a fundamental tradeoff of effort rendered unnecessary with the use of IRDs. The pseudocode in Figure 4 shows what is needed for the IRD-based chat application, and this highlights that the code is remarkably simple and yet it can still support complex network cases that the traditional IP stack cannot.

### B. Performance Evaluation of IRD Translation

When a message is sent to an IRD, it must be translated to a network address, and this necessarily incurs some performance overhead. To measure this overhead in the system, we developed an iperf-style application to measure UDP throughput over the loopback interface, effectively measuring the performance and speed of the network stack itself. The results of this test are summarized in Table I, and show that while IRDs do introduce a small amount of overhead, the difference is well within acceptable bounds and is not likely to introduce performance bottlenecks.

The difference between write() and sendmsg() is that sendmsg() requires the application to pass a sockaddr structure with each message, whereas write() requires the socket to have previously been bound to a sockaddr. Thus, in our implementation, we only translate the IRD when the socket is bound, as opposed to per-datagram. This fundamental difference results in simpler execution, and explains the increased throughput seen in Table 1, both in the traditional IP stack and for the IRD-based approach.

### C. Backwards Compatibility

Because IRDs work without changing the protocol stack itself, they are fully compatible with existing protocols, and this includes application endpoints that do not use resource descriptors. We tested this and found that IRD-enabled clients could easily send messages to applications bound to a traditional address and port combination. The inverse is also true, in that server-side applications adapted to use resource descriptors were equally able to receive messages and communicate with non-IRD clients. This is a crucial consideration for deployment, because it means that applications and systems can be migrated to resource descriptors asynchronously, without fear of breaking compatibility with other endpoints.

| | Native Stack | Resource Descriptors |
|---|---|---|
| sendmsg() | 348.7 MBps | 318.8 MBps |
| write() | 613.2 MBps | 603.1 MBps |

TABLE I: Loopback throughput

## VI. Conclusion

We introduced the concept of Internet resource descriptors (IRD), which extends the concept of file descriptors in UNIX to allow applications to use opaque identifiers known only within a host to bind only once to application-friendly names of Internet resources. This allows Internet applications and the Internet stack to evolve independently of one other. We showed how IRDs can be demultiplexed to support multiple network addresses and protocols simultaneously, and why IRDs provide significant improvements to 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] J. Shoch. Inter-Network Naming, Addressing, and Routing. *17th IEEE Computer Society Conference (COMPCON 78)*, 1978.
[3] R.W. Watson. Identifiers (Naming) in Distributed Systems. *Distributed Systems–Architecture and Implementation (LCN 105)*, Chapter 9:191–210, 1981.
[4] J. Saltzer. On The Naming and Binding of Network Destinations. *RFC 1498*, August 1993.
[5] T. Koponen et. al. Architecting for innovation. *ACM SIGCOMM Computer Communication Review*, 41(3):24–36, 2011.
[6] A. Ghodsi et. al. Intelligent Design Enables Architectural Evolution. *ACM HotNets*, page 3, 2011.
[7] J. Crowcroft et.al. Plutarch: an argument for network pluralism. *ACM FDNA '03*, 2003.
[8] I. Stoica et. al. Internet Indirection Infrastructure. *ACM SIGCOMM*, 2002.
[9] H. Balakrishnan et. al. A Layered Naming Architecture for The Internet. *ACM SIGCOMM*, pages 343–352, 2004.
[10] R. Moskowitz et. al. IETF RFC 5201: Host identity protocol. 2008.
[11] B Ford. Breaking Up The Transport Logjam. *ACM HotNets*, 2008.
[12] E. Nordstrom et. al. Serval: An end-host stack for service-centric networking. *Proc. 9th USENIX NSDI*, 2012.
[13] J. Ubillos et. al. IETF draft: Name-based sockets architecture. 2010.
[14] Erik Nordmark and Marcelo Bagnulo. IETF RFC 5533: Shim6: Level 3 multihoming shim protocol for IPv6. 2009.
[15] D. Farinacci et. al. IETF RFC 6830: The locator/ID separation protocol (LISP). 2013.
[16] Alan Ford, Costin Raiciu, Mark Handley, Sebastien Barre, and Janardhan Iyengar. Architectural guidelines for multipath tcp development. *RFC6182 (March 2011), www. ietf. ort/rfc/6182*, 2011.
[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.