Linux XIA: An Interoperable Meta Network Architecture to Crowdsource the Future Internet

Michel Machado
Boston University
michel@bu.edu

Cody Doucette
Boston University
doucette@cs.bu.edu

John W. Byers
Boston University
byers@cs.bu.edu

ABSTRACT

With the growing number of proposed clean-slate redesigns of the Internet, the need for a medium that enables all stakeholders to participate in the realization, evaluation, and selection of these designs is increasing. We believe that the missing catalyst is a meta network architecture that welcomes most, if not all, clean-state designs on a level playing field, lowers deployment barriers, and leaves the final evaluation to the broader community. This paper presents Linux XIA, a native implementation of XIA [13] in the Linux kernel, as a candidate. We first describe Linux XIA in terms of its architectural realizations and algorithmic contributions. We then demonstrate how to port several distinct and unrelated network architectures onto Linux XIA. Finally, we provide a hybrid evaluation of Linux XIA at three levels of abstraction in terms of its ability to: evolve and foster interoperation of new architectures, embed disparate architectures inside the implementation's framework, and maintain a comparable forwarding performance to that of the legacy TCP/IP implementation. Given this evaluation, we substantiate a previously unsupported claim of XIA: that it readily supports and enables network evolution, collaboration, and interoperability — traits we view as central to the success of any future Internet architecture.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications

Keywords

Meta network architecture; network evolution; future Internet architecture; XIA; Serval; zFilter; Linux

1. INTRODUCTION

The growing number of clean-slate redesigns of the Internet reflects the sensible effort of the network community to address the need of finding successors to the long reign of TCP/IP. However, this effort lacks a medium to bring all stakeholders to participate in the realization, evaluation, and selection of the next Internet architecture. On the one hand, most existing clean-slate designs are siloed and elevate a few network use cases above others, which fails to facilitate a collaborative environment for the myriad of Internet stakeholders, whose goals are not generally aligned. Further evidence comes from the fact that there are few, if any, examples of cross-pollination of running code across clean-slate proposals. On the other hand, and in the community's defense, designers have justifiably found it difficult to bring a new design into fruition, demonstrate its merits, and have the community at large experiment with it, due to the lack of a suitable comparative evaluation platform on which to do so.

In this work, we advocate a meta network architecture that nurtures coexistence of clean-slate designs, letting stakeholders experiment with and choose the designs that best suit their needs. We envisioned the eXpressive Internet Architecture (XIA) [13] as a candidate, and implemented it as a native network stack in the Linux kernel. XIA is briefly reviewed in Section 2. Then, we ported several rather different architectures to this platform to demonstrate coexistence, and furthermore, built an exemplifying application to show interoperability. As a side effect, we substantiated XIA's claim of supporting architectural evolution. To demonstrate that Linux XIA is indeed a realistic platform for other clean-slate designers, we benchmarked its forwarding performance against the Linux IP routing algorithm.

The three architectures that we ported to Linux XIA are IP, Serval [25], and zFilter [17]. In addition, we give a brief description of how one could realize Named Data Networking (NDN) [16] and Active Networks (ANTS) [31] in XIA. Each of these ports adds a unique perspective to Linux XIA. First, any successful clean-slate architecture will need to coexist with IP indefinitely. Our port of IP to XIA shows both how XIA can emulate IP and how it can progressively wean itself off of IP by removing dependencies in a staged deprecation. Serval is a service-centric architecture that complements Linux XIA with a mobile, multipath, reliable transport; moreover, it aligns well with our goals because, like us, its authors pursued a realistic implementation in the Linux kernel. zFilter, a key component behind the European PURSUIT project, is a multicast architecture that does not require state related to multicast groups on routers. Finally, NDN is a content-centric architecture and ANTS is an early meta architecture with a high degree of generality, both of which provide interesting insights for Linux XIA.

The embedding of various alien designs is only one aspect of the evolvability story, however. Linux XIA is capable of a higher form of evolvability because it encourages *interoperability* between ported architectures as well. In this way, Linux XIA can combine functionality from different architectures to enable a more powerful, componentized network protocol. In an exemplifying demonstration of this idea, we have built a reliable multicast application that is capable of pushing video content across a heterogeneous network by combining XIA, IP, zFilter, and erasure codes [3]. These technologies were never intended to interoperate, but work together under Linux XIA.

The contributions of this paper are fourfold:

1. providing clarifying interpretations of network architecture terminology, as well as providing a taxonomy of meta network architectures (§3);

2. realizing a native implementation of XIA (§4), a meta architecture that empowers the networking research community at large to crowdsource the future Internet;
3. substantiating the expressiveness of XIA by porting three alien designs: IP, Serval, and zFilter (§5); and
4. demonstrating the viability of Linux XIA through a multi-tiered evaluation consisting of evolvability considerations and by testing the forwarding performance against the mature Linux implementation of TCP/IP (§6).

2. XIA IN A NUTSHELL

The central premise underlying XIA is to design for evolvability. By evolvable, we mean having an explicit, well-defined, incremental path to introduce changes to the XIA network protocol, which is called the eXpressive Internet Protocol (XIP). As with today's IP protocol, the XIP protocol facilitates hop-by-hop, best-effort forwarding of datagrams, but does so with a considerably more complex address space than IP. Moreover, the meaning of addresses within this address space is intended and designed to dynamically evolve over time. As XIP is the central focus of this paper, we focus on support for evolvability in XIP and the composition of XIP addresses. Readers interested in other aspects of XIA can refer to the original XIA paper [13].

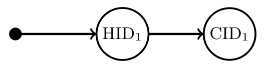
The unit of evolvability within XIA is an XIA principal, each of which must introduce its own class of identifiers, called eXpressive IDentifiers (XIDs), that name objects defined by a given principal. Each XID is the pairing of a principal type (32 bits) and a name or ID (160 bits). Examples of principals and corresponding XIDs are the Autonomous Domain (AD) principal, which names XIA networks, the Host (HID) principal, which names any physical or virtual machine with an XIA stack, and the Content (CID) principal, which names immutable content. A further recommendation associated with XIA identifiers is intrinsic security: each XID is expected to draw from a namespace that provide a cryptographic linkage between that XID and a security property. For example, in XIA, AD XIDs are the hash of public keys of the networks they name, HID XIDs are the hash of public keys of the machines they name, and CID XIDs are the hash of the contents of the file they name.

The primary connection between XIA principals and XIP is the manner in which XIP addresses are built from multiple XIDs. XIP addresses express an application-level intent through a connected single-source, single-sink directed acyclic graph (DAG) of XIDs. The ultimate intent of a packet is expressed in the XID of the sink node of the destination address. The entry node of an address, represented by a dot (•), has the sole purpose of pointing to where the navigation of the DAG begins, and thus the simplest nonempty XIP address is $\bullet \rightarrow XID_1$. While destination addresses must be nonempty, source addresses can be empty; this behavior is important to support architectures that do not have source addresses, such as NDN. All other (internal) nodes of an address represent XIDs, and each node is associated with between one and four strictly prioritized outgoing edges; four being the maximum fanout supported in XIP addresses.

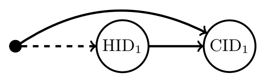
Routers are required to forward packets according to the intent expressed in each DAG destination address. Therefore, a valid set of packet forwarding decisions at routers must correspond to a successful traversal of the DAG from entry node to sink to achieve the final intent. How is this accomplished? First, the XIP header stores the DAG as a collection of nodes and their prioritized edges. Additionally, the XIP header records a dynamic `LastNode` pointer to one of the nodes in the DAG. This pointer, initially set to the entry node, reflects the portion of the DAG in this packet that has been traversed by forwarding decisions so far. When the packet reaches the intended destination, the `LastNode` will point to the sink.

To forward a packet, a router first inspects the `LastNode` field to identify the progress made through the DAG so far. For each of the outgoing edges from the referenced node, in priority order, the router attempts to forward on the corresponding XID. If that XID is local to that router (for example, the XID is an AD and the router is in that domain), the router updates the `LastNode` field of the packet and either recurses on the forwarding decision, or, when `LastNode` points to the sink, delivers the packet to the corresponding principal of the sink node. Otherwise, if the XID is non-local, the router forwards the packet toward the designated XID. Finally, if the router cannot forward along any of the outgoing edges of the DAG, the address is not reachable and the packet is dropped.

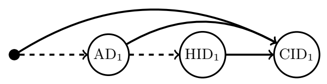
Among the many address structures that DAGs support, three addressing patterns are commonly used to date: scoping, fallback, and iterative refinement. *Scoping* a CID to a given host can be accomplished with an address that requires packets first be forwarded to host HID_1, and from there, on to CID_1:



When a new XIA principal is being deployed, chances are that many routers in the network do not know it, this can be addressed with the *fallback* pattern, which uses a lower priority edge to route to a well-known principal in case the new principal is not known by the router making the routing decision. For example, assuming that the CID principal is not widely deployed, one can still reach CID_1 even if HID_1 is the only host aware of the CID principal (dashed edges reflect lower priority):



Finally, the *iterative refinement* pattern combines scoping and fallback patterns. In the event host addresses such as HID_1 are not globally routable, we can have CID_1 fall back to an AD XID (AD_1) where HID_1 is presumed to reside:



3. META ARCHITECTURE TAXONOMY

3.1 Definitions

To provide context for Linux XIA's place in the literature and to enable architectural comparisons, we have developed a vocabulary for architectural components of networks. We found it necessary to provide clarifying interpretations of basic terms like network architecture, as the literature did not offer consistent definitions that apply to and encompass a diverse array of works and authors like our definitions do. In so doing, we make use of a new building block, network factor, that underlies an architecture, as well as clarify what we believe to be the role of a meta network architecture. We believe that the lack of a suitable meta architecture definition in the literature may explain why prior work has focused on pointwise comparisons instead of contrasting meta architecture designs directly. Our definitions proceed bottom-up.

Definition 1. *A* **network factor** *is a data plane component that specifies abstractions, data formats, procedures, protocols, and at least one class of identifiers that, together, enable the instantiation of functional network configurations of data processors.*

IP, TCP, and UDP are all examples of factors. Although there are many criteria in the factor definition, furnishing a class of identifiers is a central requirement. IP fulfills this requirement with IP addresses, and TCP and UDP do so with port numbers. These examples contrast with XIP, which does not define identifiers of its own, and thus is not a factor. Instead, each XIA principal defined within XIP constitutes an example of a network factor, since every principal must define an addressing scheme as an XID format.

Except for the identifier requirement, the definition of network factor does not impose any restrictions on the size of a factor. We refer to an important side-effect of this flexibility as *factor multiplicity*, that is, combining any number of factors leads to a single factor, analogous to how numeric expressions are defined. Therefore, the TCP/IP suite – TCP, UDP, and IP combined – is also a factor.

Definition 2. *A* **network architecture** *is a self-sufficient factor.*

Continuing our earlier examples, IP accomplishes its existential goal of sending packets between hosts on its own, so IP is a self-sufficient factor and therefore IP is also an architecture. Since TCP/IP is also a factor and IP is self-sufficient, TCP/IP is an architecture as well. The same is not true for TCP and UDP, which depend either on IP or an alternative internetwork factor to function.

Definition 3. *A* **meta network architecture** *is a framework that harmonizes a broad spectrum of factors within its framework without imposing any static dependencies among factors.*

In order to harmonize factors, a meta architecture framework must define an interface that enables factors to be embedded. This definition also mandates support for a broad spectrum of factors, since we view designs that narrowly support a limited amount of architectural diversity as insufficiently general to warrant the meta architecture designation. XIA's framework for harmonizing factors is defined by the XIP protocol. §3.2 presents a taxonomy of meta architectures spanning numerous examples and how they harmonize their supported factors.

Finally, this definition requires that there must be no *static dependencies* between factors. We define a static dependency to be present whenever removing one factor from the set of deployed factors also requires removing or recoding additional factors. For example, if IP were dropped from TCP/IP, TCP and UDP would need to be recoded to work with another internetwork factor. Therefore, despite its support for many applications, TCP/IP cannot be considered a meta architecture because all supported factors statically depend on IP. Importantly, our meta architecture definition does not forbid *runtime* dependencies among factors (a technique we support and whose benefits we articulate later). Interested readers can find an extended presentation of our definitions in [23, Section 1.1].

3.2 Related work on meta architectures

For decades, researchers have been enticed by the notion of a meta network architecture that evolves to accommodate unforeseen network use cases. Softnet [34], developed in the early 1980s, is the first meta architecture of which we are aware. The degree to which meta architectures isolate their factors is enlightening because it highlights the value that a meta architecture offers to its applications, which ultimately reflects the utility functions of the end users of the meta architecture. This section classifies meta architectures according to the degree of isolation between their factors from highest isolation to lowest, and distinguishes Linux XIA's place in this taxonomy.

Network virtualization [1, 28] and SDN [10, 4] are natural meta architectures; they do not limit the number of supported factors, nor do they impose static dependencies among the supported factors. At a high level, these meta architectures slice network infrastructure into independent, isolated resources that support their factors; we call this group *slicing meta architectures.* The degree of factor isolation, however, is sufficiently high that applications are solely responsible for the necessary work to leverage multiple factors, and supported factors must be self-sufficient to properly operate their slices. As a result, these meta architectures only support full-blown architectures.

The next group of meta architectures, the *translating meta architectures*, encompasses Plutarch [8], FII [18], OPAE [12], Omega [26], and SDIA [27]. These meta architectures segment the network into independent regions, map each region to supported factors, and promote bridges between regions to translate the protocols in both directions. Similar to slicing meta architectures, supported factors must be organized into network architectures, but applications are not solely responsible for interoperability between regions. The troubling aspect of this group is that facilities for translation between these pluralistic architectures are not provided, and may not always be possible. For example, there is no clear mapping between a host-centric architecture, such as IP, and a content-centric one, such as NDN [16]; §6.2 returns to this point.

The third group, *active meta architectures*, is centered on active networks [29], and most notably ANTS [31], the meta architecture that pursued programmable networks as the standard-bearer for active networks. ANTS does not slice or segment a network; its factors share the whole network. Factor designers build factors with mobile code that is shipped through the network from applications to routers with the help of a code distribution protocol. While applications can interoperate with multiple factors at the same time, the runtime environment of mobile code intentionally isolates factors to address security issues. Due to isolation in the runtime environment, factors still have to be self-sufficient, as with the previous groups of meta architectures.

XIA distinguishes itself from other meta architectures by promoting *interoperability* among all of its supported factors in the form of XIA principals. This interoperability is made possible by (1) XIA factors sharing the whole network, as in ANTS, (2) network addresses enabling factor composition in every address (§2), and (3) factor designers postponing dependencies among factors until runtime through routing redirects, an extension of XIA's routing algorithm that we introduce in Linux XIA (§4). Thanks to these mechanisms, XIA factors can delegate functions and responsibilities to other factors, which in turn enables XIA factors to specialize. A key novelty is that XIA factors do not have to be self-sufficient, unlike the factors of all the meta architectures cited above.

Because XIA constitutes a meta architecture, stakeholders must choose a set of factors that, together, instantiate XIA as an architecture in order for it to be useful to applications. One plausible baseline XIA architecture, according to our terminology, would consist of the AD, HID, and CID principals. This set would provide basic inter-domain routing, host-to-host communication, and a form of information-centric networking.

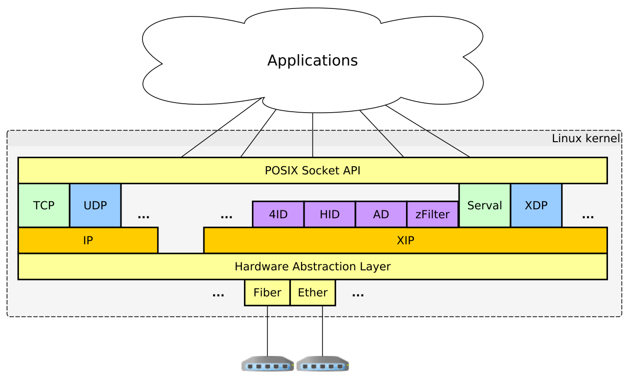


Figure 1: Overview of the TCP/IP and XIA stacks in the Linux kernel. Only the Serval and XDP principals touch the POSIX Socket API, and thus can be used to create sockets; all other current principals can only be used to compose XIP addresses.

4. LINUX XIA

In order to substantiate the claim that XIA is a viable meta architecture, we needed a full-blown network stack to accommodate the implementations of other architectures. Furthermore, a native implementation providing competitive routing performance is a necessary step to show that XIA is deployable in production network environments.

At a block-diagram level, Figure 1 depicts TCP/IP and XIA as parallel stacks in the Linux kernel. In the figure, the matching colors guide the analogy between the stacks: IP maps to XIP, TCP to Serval, and UDP to XDP. Although the figure suggests that TCP, UDP, and IP are individual kernel modules, in practice, the TCP/IP stack is implemented as a single module in Linux. In contrast, each block on the XIA stack corresponds to a distinct Linux kernel module that we implemented. In this section, we focus on the realization of XIP and the implementation of core XIA principals like HID and AD. Later, when we integrate alien designs with XIA, we describe the realization of 4ID to provide interoperability with IP, and describe our implementations of Serval and zFilter.

We next discuss the impact that our experience developing Linux XIA had on our architectural design (§4.1), and cover the algorithmic details of how Linux XIA forwards packets and keeps its forwarding cache synchronized (§4.2).

4.1 Architectural realization of evolvability

Building Linux XIA from scratch gave us many opportunities to explore the kernel design space and to avoid barriers to future evolution of our network stack. Since evolution is the central architectural premise of our work, we developed a radically different design than that of the native TCP/IP implementation. We report on experiences that reinforce the power of modular design and clarify how new principals must themselves be architected for evolvability.

The first influential decision we made was to map each principal to a kernel module. This decision, premised on modular design, ended up interacting with XIA in unexpected ways: it led us to (1) make XIP a truly standalone protocol, in contrast to IP, which needs ARP and ICMP to operate; and (2) conceive of routing redirects, a facility for supporting runtime dependencies between principals.

Mapping each principal to a kernel module may seem to be primarily an implementation decision. In truth, the fact that such a clear mapping exists emphasizes that the principal is the the key abstraction that enables modularity within XIA. As a reference point, the abstraction of layering underlies the modular design of TCP/IP and, more generally, underlies the architectural notion of a network stack. While a network designer could conceivably implement ARP and ICMP as kernel modules in the TCP/IP stack, the designer could only evolve ARP or ICMP by changing the specification of IP, as the latter relies on ARP and ICMP internals. In contrast, the analogues of ARP and ICMP in Linux XIA are not bound to XIP; they are only bound to the principals that require them. The contrast between layers and principals is further developed in [23, Section 6.2].

As long as there are no static dependencies among kernel modules of principals, each XIA router or host can load any set of principals into the stack. This feature avoids biasing stakeholders toward principals that the implementation arbitrarily requires. Any non-self-sufficient set of loaded principals is a valid configuration in Linux XIA, although obviously an impractical one, since not all addresses are reachable in these configurations.

Although static dependencies are problematic, *runtime* dependencies among principals are a useful technique to avoid duplicating code. For example, since both the AD and HID principals need to forward packets to neighbors, writers of AD may be tempted to call functions from HID instead of reimplementing the code. This would effectively link the AD principal's implementation to a specific version of the HID principal. Avoiding these undesirable static dependencies among principals required routing redirects, an enhancement of XIA's routing algorithm that lets stakeholders express dependencies among principals at runtime. Details of this approach, as embodied in Linux XIA's routing mechanism, are described next.

4.2 Algorithmic realization of evolvability

Linux XIA must accomodate more advanced routing table lookups and updates in order for evolution to be practical, since addressing and routing is maintained per-principal. Because these challenges are not present in the legacy architecture, special packet forwarding and routing dependency algorithms are needed. This section describes the algorithmic contributions that help efficiently realize evolution; subsequently, in §6.3, we present our performance evaluation of this machinery against the Linux IP implementation.

4.2.1 Fast packet forwarding

Since the maximum fanout in an XIP address is four edges, an XIA router may have to inspect up to four XIDs to make a routing decision. However, even without hardware optimizations, Linux XIA can efficiently instantiate XIA's forwarding mechanism by not serializing these lookups.

A diagram of the XIA routing algorithm is presented in Figure 2. When a packet arrives on an incoming interface, Linux XIA's routing fast path references the `LastNode` field of the packet to obtain the sequence (in priority order) of outbound edges from that node in the destination address. It then looks up this sequence of XIDs in its routing cache, which is called a DST table. If there is a cache hit, the DST table returns a DST entry, a data structure that holds pointers to functions that forward packets with the same signature sequence of edges.

Otherwise, on a DST cache miss, Linux XIA falls back to its routing slow path. Now, Linux XIA iteratively looks up the candidate edges in the routing table. A software-only solution can do these lookups in parallel, but synchronizing these results can be costly; a solution that also uses hardware support would take greater advantage of this parallelization. The iterator terminates either when all edges of the `LastNode` generate misses in the routing

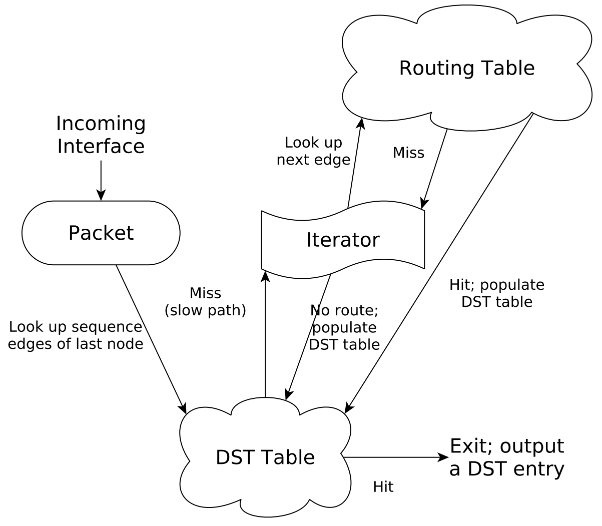


Figure 2: Linux XIA's routing algorithm.

table (unreachable destination), or when one of the edges generates a hit. In both cases, a new DST entry for this sequence of edges is created, added to the DST table, and returned.

Our routing algorithm still has three challenges to overcome: (1) working efficiently under the memory limit of the DST table, (2) reusing principals' code without introducing static dependencies between them, and (3) keeping the routing cache synchronized with the routing table.

The first challenge is that of working under stress or under attack. For example, an adversary could mount a denial-of-service attack in which undeliverable packets with unique edge sequence signatures are sent to an XIA router. These useless entries would require lookups on the slow path and put pressure on the DST table to potentially drop useful entries. A proper solution for this problem is still to be investigated. The current implementation employs simple heuristics to reclaim entries based on frequency and recency in order to maximize the usefulness of entries.

We provide a solution to the second challenge next, and address the third challenge in §4.2.3.

4.2.2 Routing redirects

While the machinery described to this point is sufficient to implement the XIA routing algorithm as originally specified [13], it does not yet achieve our goal of principal independence. To allow principals to work together but remain independent at compile-time, we introduced the concept of *routing redirects* in Linux XIA. A routing redirect takes place when the iterator in Figure 2 looks up an XID and the routing table returns a "redirect," in which case another XID needs to be looked up in order to satisfy the original lookup.

For every XID inserted into the routing table for a principal that employs a routing redirect, an XID from a different principal must be specified as the "gateway" for the redirection. Then, when an XID from a redirecting principal is retrieved from the routing table, the return value of the lookup specifies a redirect to the gateway XID instead of a hit or miss. The routing iterator then recursively looks up the gateway XID to decide how to forward the packet.

Note that the gateway principal may not be loaded at routing time, in which case, XIDs that redirect to this gateway are not routable. We consider this an acceptable behavior in exchange for the benefit of allowing network administrators to choose which principals to load. Moreover, routing redirects solve a more important conceptual problem for evolution of principals: they replace static dependencies between principals with runtime dependencies, which permits principal independence at compile-time. This can be useful in situations such as the example in §4.1, in which the AD principal was tempted to call the HID principal's code for XIDs that simply need to be forwarded to a neighbor. For these XIDs, the AD principal can use routing redirects to delegate work to the HID principal, therefore avoiding code duplication.

Routing redirects have matured beyond our goal of breaking static dependencies. For example, they could be used in the context of routing protocols to map HIDs to XIDs that represent link layer addresses, and can be used to split functionality into separate principals. However, further details regarding these other uses of routing redirects are outside the scope of this paper.

4.2.3 Routing dependencies

To enable the efficiency gains of DST caching, the DST table must be kept in close synchronization with the routing table. But changes to the routing table can affect the DST table in non-obvious ways. For example, when an XID is added to the routing table, finding the DST entries that become invalid could potentially require a full scan of the DST table. This is because a new XID invalidates an arbitrarily large number of DST entries, since those entries prioritize the new XID at a higher priority than their currently chosen edge.

Reviewing the shortcomings of naive solutions to this dependency problem is instructive. Flushing the cache for each routing table update is one possibility, but it causes routing hiccups; suddenly the cache is empty and all packets have to be analyzed on the slow path, one edge at a time. Principals that have XIDs associated with sockets or other user space objects (e.g. Serval) may see corresponding update rates high enough to render the cache a useless burden. Another possibility is to have routing table updates trigger a corresponding scan of the cache to remove stale XIDs. But full table scans are prohibitively expensive, since they put pressure on a machine's memory cache subsystem. Moreover, unless the number of stale cache entries is large, the scan itself is inherently inefficient. Finally, testing the freshness of a cache entry can itself be an involved procedure – routing redirects and default routes are two issues which introduce challenges.

Naive solutions fail to efficiently identify affected cache entries. Linux XIA does so by building dependency chains and *anchoring* these chains on the routing table entries that keep them up-to-date. Although developed independently, our solution is a variant of the Data Update Propagation algorithm [7], originally devised for keeping caches of dynamic web pages consistent with underlying data.

More specifically, our solution incrementally builds dependency chains for each investigated edge of the DAG during the execution of the routing slow path (i.e., by the iterator in Figure 2). These chains thus enumerate the routing table entries on which cache entries depend. Since some chains eventually merge with other chains, the final data structure is a dependency forest. We refer to each internal node in this forest as an *anchor*. When an anchor is updated, its rooted subtree in the forest is stale and the corresponding cache entries must be flushed.

The performance of the routing dependency algorithm is evaluated in the routing update rate experiment we present in §6.3.2. To demonstrate how routing dependencies work in concert with routing redirects, we next give an example of a DST table that must invalidate redirected entries.

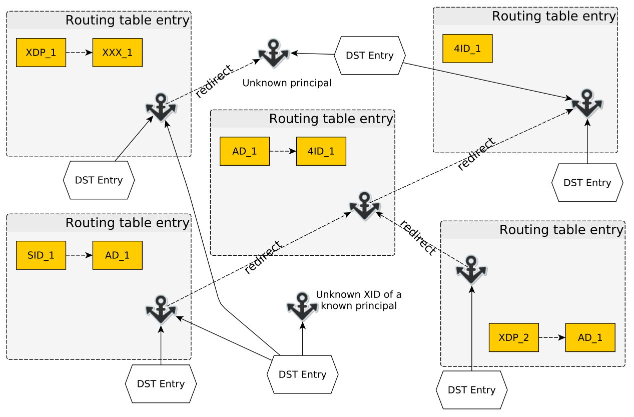


Figure 3: DST dependencies form a rooted forest anchored at routing table entries.

4.2.4 *Putting it all together*

Figure 3 illustrates a small dependency forest and its related routing table and DST entries. In this example, SID_1 redirects to AD_1, which in turn redirects to $4ID_1$. If the routing table entry for $4ID_1$ were removed, then the corresponding cache entries must be updated by flushing the anchor associated with $4ID_1$. There are two DST entries that directly depend on this anchor, and three DST entries that indirectly depend on this anchor through routing redirects. Only these five entries will be removed from the DST table and irrelevant entries will not be inspected. In this case, the only the DST entry remaining is the upper-left one, which is associated with the routing table entry for $XDP_1 \rightarrow XXX_1$.

5. PORTING ALIEN DESIGNS TO XIA

We now describe how three distinct network designs never intended for compatibility with XIA can be adapted to work and coexist in XIA. This is meant to demonstrate the value of an interoperable meta architecture and to provide other researchers with instructive examples on how to adapt their own designs, or design for XIA from scratch.

In porting a network design Y to XIA, we have found that two architectural questions drive most of the work: how can Y's visible identifiers be mapped to XIDs, and how can Y be broken into XIA principals? Whereas the first question is simple and inevitable, it is not always obvious a priori which of the many possible options will prove to be best. Experience with promising options may be required to make a meaningful selection, noting that one implemented interpretation of a network design in XIA does not inhibit other interpretations of the same design. Since many interpretations of a single design can naturally coexist in Linux XIA, value judgments are ultimately left to the stakeholders in an XIA network.

The second question adds more subtlety, as it focuses on exploring interpretations of a given design typically not considered in the original design, where the focus is self-sufficiency, not interoperability. XIA's composition of different principals to form a single address affords principals the opportunity of specializing their behavior, and delegating functions to other principals.

§5.1 provides a starting point, by showing how XIA can support a legacy technology, specifically focusing on how XIP and IP can coexist and interoperate. Then, §5.2 describes our port of Serval [25], a service-centric architecture, and §5.3 presents our port of zFilter [17], a multicast architecture, to Linux XIA.

5.1 Case study #1: IP

Any new Internet architecture has to furnish a friendly coexistence with IP networks in order to be deployable. A widely used approach for introducing new functionality onto legacy networks, which we also adopt, makes use of encapsulation. In this section, we describe a set of XIA principals, called 4IDs (respectively, 6IDs), that allow XIA-enabled hosts to communicate over a legacy IPv4 (respectively, IPv6) network. A key finding, not obvious to us a priori, is that different degrees of integration and interaction with IP networks are suited to multiple 4ID and 6ID principal types, designed to satisfy different stakeholders' needs.

Given that there will be no support for XIA in the open Internet during the early deployment of XIA, any integration must focus on retroactive compatibility; this motivates the design of the U4ID principal. Names of U4ID XIDs are the tuple (IP address, port number) followed by 14 zeros to make up the required 20-byte names. To forward to a U4ID in an address, XIA encapsulates its XIP packet into the payload of an IP/UDP packet whose destination IP address and UDP port number are copied from the XID; from there, the TCP/IP stack delivers this new packet. The XIA stack of the destination host must have the U4ID principal running and listening at the UDP port number, which is provided by the network administrator using a user space management tool, in order to receive the packet. After IP and UDP header decapsulation, the payload XIP packet is transferred to the XIA stack.

Stakeholders operating controlled environments (e.g. data centers, campuses, corporations) may prefer to trade in some compatibility for performance. The I4ID principal achieves this by encapsulating XIP packets into the payload of an IP packet and writing XIP into the *protocol* field of the IP header. In the open Internet, middleboxes are likely to drop these IP/XIP packets, but, in controlled environments, I4ID would avoid UDP's checksum, UDP's 8-byte header, and port demultiplexing. Names of I4ID XIDs are the destination IP addresses followed by 16 zeros.

The U4ID and I4ID principals build on-the-fly tunnels through IP networks. Although these tunnels are necessary for retroactive compatibility with IP, they are limiting because the edges of the destination address of the encapsulated XIP packet are only evaluated at the end of the tunnel, even when all intermediate routers are XIA-enabled. In a later stage of XIA deployment, when a large number of hosts have TCP/IP and XIA stacks to support both legacy IP applications and XIA applications, the limitations of tunnels become salient.

Designed for a dual-stack environment, the X4ID principal leverages the IP routing table to forward its XIDs, thereby avoiding tunnelling. X4IDs have the same trailing-zero format as I4IDs; the distinction between these principals is the forwarding mechanism. Routers forward X4IDs by directly looking up the corresponding IP address in the routing table of the TCP/IP stack. Therefore, the X4ID principal is the tightest integration between TCP/IP and XIA, and can bridge the deployment of XIA at Internet scale because it leverages the current BGP sessions to populate XIA routing tables and globally forward XIP packets.

To go further, we recommend the introduction of a general-purpose Longest Prefix Matching (LPM) principal, which we will also use in Serval. In contrast to the opaque cryptographic identifiers typically associated with XIA, IP (and Serval) identifiers have a hierarchical structure, and, in particular, are designed to support longest prefix matching. To support this functionality in XIA, we designed the LPM principal to support longest prefix matching on the underlying identifiers. An LPM XID is a typed string of bits that XIA routers match against a prefix tree at each hop, analogous to CIDR.

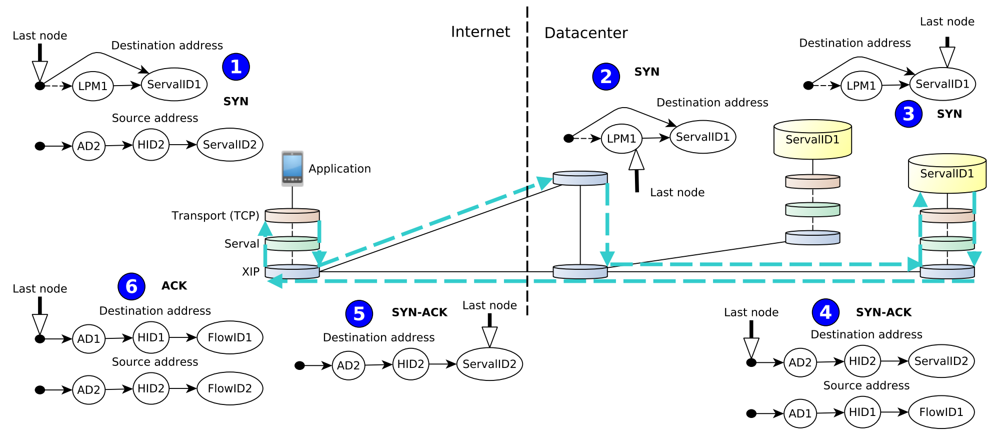


Figure 4: Example of XIA Serval's three-way connection handshake between a client and a service instance.

Returning to 4IDs, while the X4ID principal is a perfect fit for dual stack hosts, it nevertheless perpetuates an undesirable dependency on IP routing tables for XIA-only hosts that simply want to tap into the BGP sessions. These stakeholders can drop the dependence on IP routing tables by using the LPM principal and directly populating the LPM forwarding table with routes from BGP sessions. This achieves the same behavior as the X4ID principal, but without the need for the TCP/IP stack. This solution is similar in nature to how MPLS uses BGP to populate its forwarding tables. We expect that a protocol that leverages intrinsically secure identifiers would eventually replace the use of X4ID and LPM for this purpose; nevertheless, X4ID and LPM offer an easy deployment path to bootstrap global interoperability for XIA.

A natural deployment plan for Linux XIA is to run natively wherever possible and to interconnect XIA networks through an IP-only Internet with the help of 4IDs, 6IDs, and LPM principals as long as necessary. We have implemented the first step of our migration plan, namely, the U4ID principal in Linux XIA. Mukerjee *et al.* [24] have also explored the advantages of incrementally deploying XIA with the help of U4ID or I4ID principals. Our principals X4ID and LPM complement their work in the scenario they call "merged clouds," in which dual-stack hosts are commonplace.

5.2 Case study #2: Serval

Serval, a service-centric architecture, promotes services as first-class entities, as described in detail in prior work [25, 2]. The main goals of Serval are threefold: support replicated instances of a single service, support multihomed access to services, and allow for mobility at the connection endpoints. These goals are implemented through three respective methods: host-agnostic late binding to servers, tightly integrated support for multiple flows per connections, and a formally-verified migration protocol. Given that Serval's connection handshake exposes much of its internals and provides a good working view of its design, we present Serval and its realization in XIA from this angle.

A key enabling technology is Serval's use of two distinct types of identifiers, ServiceID and FlowID, both deployed in a shim layer called the Service Access Layer (SAL), which resides between the network layer and the transport layer in the protocol stack. ServiceIDs logically represent a distinct service, such as an HTTP connection to `www.example.com`, but due to service replication, do not necessarily refer to a unique location. ServiceIDs have a hierarchical meaning in Serval, and thus they are routed using longest prefix matching, like IP addresses. Unlike ServiceIDs, which are used for end-to-end connection establishment and management, FlowIDs are used for established flows *within* a service instance. These FlowIDs are flat identifiers and are only unique with respect to the host that generated it.

Serval uses either the tuple (protocol, destination ServalID), or (protocol, destination FlowID) to multiplex connections. The protocol field identifies the transport protocol above Serval; currently, UDP and TCP are supported. These tuples simplify process migration because they do not bind to remote identifiers, in contrast with TCP and UDP, which explicitly use source and destination IP addresses and port numbers to multiplex connections.

We briefly remark on security considerations in Serval. In the Serval design, ServiceIDs lack intrinsic security, that is, connection endpoints cannot verify each other's identity without a third party. To improve security, Serval uses a nonce field that serves as a shared password between connection endpoints to mitigate off-path attacks, but this cannot prevent on-path attacks.

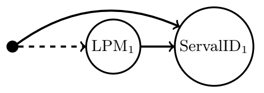
5.2.1 *Mapping Serval to XIA*

Recall that a central challenge of mapping an alien technology onto XIA is an appropriate principal decomposition. We start with naming of services. In Serval, the use of ServiceID corresponds naturally to the role of a new XID type in XIA: it is globally scoped, has a well-defined meaning that corresponds to a user intent, and is routable. While Serval's designers chose to make ServiceID a hierarchical identifier to ensure global routing, our preferred specification is to use flat identifiers imbued with cryptographic meaning. Our disentangling of the two distinct roles of ServiceIDs, naming services and facilitating routing, makes the analogous XIA identifiers, which we call ServalIDs, intrinsically secure in XIA. Thus, in our XIA interpretation of Serval, ServalIDs are the hash of their public key, and scoping is delegated to other, potentially more appropriate principals. For example, the address $\bullet \rightarrow AD_1 \rightarrow HID_1 \rightarrow ServalID_1$ scopes $ServalID_1$ to host HID_1 located in the autonomous domain AD_1.

As for FlowIDs, these local identifiers are not germane to XIP forwarding, and thus comprise a local XID principal type that is used exclusively at endpoints. We retain the semantics of these identifiers as used in the original Serval and thus, when in use, it specifies the primary intent in a destination address, but is *always* scoped to a given host. Together, ServalIDs and FlowIDs define the XIA Serval principal.

Our preference toward intrinsically secure ServalIDs does not preclude a fallback to hierarchical identifiers. One solution would be to have another interpretation of Serval in XIA that preserves hierarchical ServiceIDs. A more modular solution – one in tune with the question of how to break Serval into principals – would use the LPM principal defined earlier to do longest prefix matching.

We can now review Serval connection establishment in the context of XIA Serval. Figure 4 diagrams the packet sequencing of the 3-way handshake between an application at left and XIA Serval service instances replicated in the data center at right. The SYN packet depicted in steps 1-3, destined to an arbitrary service instance, is shown having a destination address with a ServalID as the sink and with optional fallback to an LPM XID:



The packet's source address also has a ServalID as the sink, but with scoping using ADs and HIDs, chosen by the application. Once this packet arrives at a service instance, this source address becomes the destination address in the SYN-ACK, depicted in steps 4-5. Both the SYN and the SYN-ACK in XIA Serval correspond to Serval SYN and SYN-ACK packets with ServiceIDs as the destination addresses. Finally, the ACK packet depicted in 6, as well as all subsequent packets for this flow, have destination addressses with XIA FlowIDs as the sink, typically preceded by host-level scoping, here by AD_2 and HID_2. These correspond to Serval packets which are addressed by Serval FlowIDs in the SAL layer.

Our Serval addressing scheme in XIA mimics the Serval implementation faithfully with one exception. In the SYN packet, Serval uses a FlowID as the source, but in XIA Serval we require instantiation of a ServalID at both endpoints to perform connection setup, and thus defer relaying the proposed FlowID until the third (ACK) packet of the 3-way handshake. This small change preserves end-to-end security guarantees, as ServalIDs have a cryptographic meaning in XIA, whereas FlowIDs do not.

5.2.2 *Discussion*

Integrating Serval with XIA affords several key advantages compared to the original implementation over IP. First, elevating Serval addresses stored in the SAL shim layer to first-class XIA addresses provides much better visibility with respect to user intent. Whereas Serval carries its identifiers in extension headers above IP, in XIA's realization, those identifiers are carried in XIA's network addresses. Thus, XIA Serval enables every router to make decisions purely based on network protocol information. As a result, placing ServalIDs as the primary intent of connection establishment enables routers to seamlessly realize service anycast, More importantly, hardening ServalIDs with the intrinsic security afforded by XIA eliminates the possibility of spoofed services, and renders Serval's (weaker) methods for prevention of off-path attacks unnecessary. Note that this bootstrapping procedure could also be used to harden all subsequent transmissions with cryptographic signing, including the FlowIDs themselves.

5.3 Case study #3: zFilter

zFilter [17] is a multicast architecture that, in contrast to IP multicast (RFC 1112), requires no router state related to multicast groups. The zFilter architecture avoids this state information by having destination addresses encode the physical links comprising a given multicast tree in a compact Bloom filter data structure. This approach contrasts with IP Multicast, whose packets carry specially designated IP destination addresses that serve as labels, matched on each router against a list of active multicast groups. The following paragraphs give more details on how zFilter works, how we mapped zFilter onto XIA, and briefly discuss positive side effects of having the zFilter principal type in XIA.

The key to understanding how zFilter works is the process that the network uses to derive Bloom filter network addresses from multicast trees. The first step of this process starts with each router independently associating a fixed-size Link ID to each of its network interfaces. Link IDs have a size of m bits, of which exactly k bits are one, and all other bits are zero; destination addresses are also m bits long. The parameters m and k are fixed for a given realization of zFilter; for example, zFilter's authors chose $m = 248, k = 5$ for their implementation. Routers choose the k-one bits of their Link IDs at random. By construction, Link IDs are unidirectional, that is, each physical link has two Link IDs. Given the topology of a network, including all Link IDs, the destination address corresponding to a given multicast tree consists of a logical-OR union of all Link IDs in that multicast tree.

zFilter routers forward packets simply by checking the Link IDs of their network interfaces against the Bloom filter addresses in the packets. A router checks the presence of the k-one bits of a Link ID in an address by logical-ANDing the Link ID and the address, and comparing the result with the Link ID. Whenever the comparison is true, the Link ID is in the address, and the packet is transmitted across that interface. False positives arise when all the bits associated with a Link ID that is not in the tree happen to be set to 1 by other Link IDs which are in the tree. The rate of false positives can be tuned by varying m and k. Hardware can check all Link IDs of a router against an address in parallel, so the test is extremely fast.

As in the previous case studies, the porting questions drive the work to bring zFilter into XIA. The choice of XID format for zFilter is straightforward: zFilter XIDs are the Bloom filter addresses. Whereas the decision of how to delegate responsibilities to other principals leads to an elegant solution, Link IDs do not necessarily map to network interfaces in XIA's realization of zFilter, instead, Link IDs use *routing redirects* from §4.2.2 to map a zFilter XID to any XID. Our zFilter principal employs routing redirects whenever a Link ID matches a zFilter XID. Therefore, the zFilter principal does not itself provide code for forwarding packets, instead it delegates this job to an appropriate principal such as AD, HID, and 4IDs via redirection. zFilter's authors have considered cases in which Link IDs represent entities other than network interfaces, but XIA zFilter's use of routing redirects generalizes this behavior. §6.1 showcases this feature exemplifying how a single-zFilter address can emulate a three-XID address just using routing redirects to the other XIDs.

Thanks to having directly mapped the Bloom filter addresses of zFilter onto the XIDs of our zFilter principal, any feature that the original zFilter supports, XIA zFilter supports using the same implementation solution. Among these features are "Link ID Tags" and "Virtual Links" which zFilter authors originally considered, as well as zFormation [11], which was designed later.

Besides enabling zFilter to delegate the final action on the packets to other principals, XIA brings two other important advantages to zFilter. XIP addresses can have multiple zFilter XIDs, which one can leverage to reduce the number of false positive matches of the Bloom filter. Also, XIA zFilter cleanly interoperates with TCP/IP, as §6.1 demonstrates.

6. EVALUATION

To complete the case for Linux XIA as a platform that supports crowdsourced innovation, we evaluate it against three criteria:

1. how do its architectural principles address the networking needs of today and the future?
2. how effectively does the implementation realize those architectural principles?
3. how efficient is the implementation in terms of packet forwarding performance?

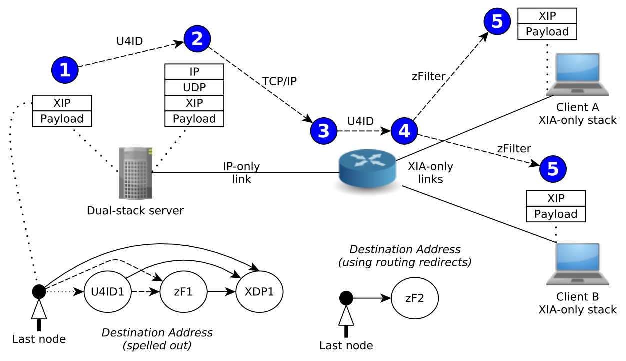


Figure 5: Implemented example that showcases network evolvability and principal interoperability in Linux XIA.

These thrusts test the design's viability at a theoretical, implementation, and performance level, respectively; a three-level evaluation of a network architecture similar to that advocated by Wroclawski [33].

The first criterion is necessary because we view the flaws of the legacy Internet as architectural in nature, so we must evaluate the design principles of clean-slate architectures in order to ensure that the same shortcomings are not realized in the future Internet. §6.1 evaluates the design principles of Linux XIA.

As for the second and third criteria, we evaluate the implementation of Linux XIA in both a qualitative and quantitative manner. An architecture that cannot be adequately instantiated is of no practical use, and an inefficient implementation does not incentivize Internet stakeholders to adopt it. We assess Linux XIA's ability to instantiate its architectural principles in §6.2 and to achieve data plane efficiency in §6.3.

6.1 Architectural evaluation

Developers of clean-slate Internet architectures generally agree that the host-centric legacy Internet is indeed a mismatch for the needs of modern networking; most advocate elevating a new use-case to be a first-class consideration in their architectures. In contrast, the distinguishing feature of XIA is an agnostic approach, refusing to choose one principal type of communication to elevate above all others. Instead, XIA advocates for evolution to be the central principle in the future Internet.

The fact that XIA is welcoming of many, if not all, foreign architectures was not immediately clear from its conception [13]. Providing an avenue for introducing new functionality at the network protocol in a incremental fashion ensures that today's types of communication can be effectively realized, and that as-of-yet unforeseen paradigms can be integrated. Furthermore, the Linux implementation of XIA pushes this architectural notion of evolvability even further, by additionally enabling interoperation of architectures. It is this aspect of the design that distinguishes Linux XIA from other clean-slate designs, including the classes of meta architectures defined in §3.2.

We built a reliable multicast application that combines three principals to deliver content across a heterogeneous network, to demonstrate the value of principal interoperability. This application employs the U4ID principal to cross an IP-only link, the zFilter principal to duplicate packets in the network, the XDP principal to deliver packets to sockets, and erasure codes to make the transmission reliable. Figure 5 illustrates this application in action and shows how these principals can compose an XIP address. We have not yet developed the control plane of Linux XIA, so we manually constructed the addresses for this demo.

The three-node destination address depicted at bottom left in Figure 5 can be understood as expressing the following intent: (1) traverse an IP-only part of the network by encapsulating XIA packets in UDP/IP payloads; (2) multicast the content to multiple hosts; and (3) deliver the content to listening datagram sockets. Alternatively, the depicted single-node destination address can be used in tandem with routing redirects in the network to supply the same functionality. In both cases, this allows the TCP/IP, zFilter, and XIA architectures to interoperate by composing their individual strengths, despite the fact that these architectures were never intended to work together.

Each step in Figure 5 captures a transition in the life of an XIP packet being sent from the server to the clients. Step 1 shows the XIP packet that the XIA stack creates once the application writes a block of data. While routing the packet, XIP discovers that it can only forward the packet following the edge $U4ID_1$ of the address, because the link between the dual-stack server and the router is IP-only. XIP transfers control to the U4ID principal, which encapsulates the XIP packet into the payload of a UDP/IP packet (Step 2), and hands this new packet to the TCP/IP stack. Once the packet arrives at the router (Step 3), the TCP/IP stack hands the packet back to the U4ID principal running at the router. The U4ID principal at the router decapsulates the packet, and hands the new packet to the XIA stack to route the new packet. XIP decides on following the edge zF_1, which leads to duplicating the packet (Step 4), and sending the copied packets toward the two clients. Once the packets arrive at the clients (Step 5), the XDP principal identifies listening datagram sockets to which the data must be delivered. In order to make the multicast transmission reliable without backchannel feedback, our application employs erasure codes as advocated in the Digital Fountain work [5].

This application serves as a proof of concept that XIA has a strong notion of evolvability and that Linux XIA extends this idea to allow interoperation and collaboration. Thus we assert that Linux XIA can act as a meta architecture that incubates networking ideas, old and new, and encourages cooperation, thereby enabling crowd-sourced innovation.

6.2 Implementation evaluation

To assess the efficacy of Linux XIA as an implementation, this section offers a qualitative evaluation in terms of its capability to realize the desired architectural features espoused in §2: deployability and evolvability.

In order for a new network architecture to be a feasible replacement for TCP/IP, it must be deployable. For a practical multi-step deployment plan, we refer the reader to §5.1. Since the first step of this plan has been implemented, Linux XIA already interoperates with the legacy Internet architecture. Furthermore, since Linux is widely used in a variety of network appliances, routers, and end hosts, Linux XIA can be broadly deployed through updates of the Linux kernel.

Once Linux XIA is initially deployed, new functionality should be added incrementally such that hosts with new principals are still able to communicate with hosts that have not yet been updated. Linux XIA supports incremental deployment in three ways. First, DAG addresses were implemented with compatibility fallbacks in mind, whereby multiple edges can be considered simultaneously using the fast routing algorithm described in §4.2.1. Second, principals can be loaded and unloaded on-the-fly because they are implemented as kernel modules; consequently, principals can be introduced and deprecated with a minimum of, or no, downtime. Third, the routing dependency forest efficiently flushes stale routing cache entries as described in §4.2.3. This is especially useful in network settings with constant churn, for example, where the XIDs of a principal are updated on a rolling basis.

However, this deployment plan is only a specfic case of a more general and powerful principle at work, which we call *architectural embedding.* This is the mechanism through which Linux XIA fulfills the promise of evolvability that is at the heart of the design. We have successfully embedded three distinct architectures into Linux XIA (§5), and whitepaper-ported NDN [16], a content-centric architecture, and ANTS [31], an early meta architecture. These latter ports are documented in [23, Sections 4.3 and 4.4].

NDN embodies a design that is difficult to instantiate on a translating meta architecture without losing some of its characteristics, but ultimately proved amenable to XIA. For example, there are anonymity implications associated with the absence of source addresses in NDN that are unlike other architectures we have ported to XIA. Moreover, NDN abstracts the network as an infrastructure that stores and retrieves content within a hierarchical naming structure, which is not inherently built into XIA. Still, these features can be ported to Linux XIA by taking advantage of the fact that XIA does not require source addresses, and by leveraging each principal's ability to define its own addressing scheme. The hierarchical naming scheme is supported by hashing content names to XIDs, and leaving names in NDN headers for the original NDN routing algorithm to interpret.

As meta network architectures, ANTS and XIA share some common ground in terms of their motivation and solutions. Like XIA, ANTS aims to lower barriers to evolve the network protocol, but does so by supporting mobile code to define new factors. Mobile code requires a significant amount of effort to design a code distribution protocol and a run-time environment to deal with security issues. Still, ANTS can be ported to XIA by choosing the ANTS XID to be the type field of an ANTS packet, which is the cryptographic hash of the forwarding code that should be applied to packets of that type. This turns out to be a natural fit in Linux XIA: it defines a single class of intrinsically secure identifiers, and seamlessly interoperates with scoping principals such as ADs, HIDs, and 4IDs. ANTS is much more expressive than the other architectures that have been ported to Linux XIA, and yet it still fits within XIA's framework.

In spite of the strong evidence that Linux XIA can embed many different architectures, a proof that it can do so universally, i.e., embed *any* architecture, is elusive. Advancing the theory of network architecture, wherein such statements could be rigorously formulated and potentially proven, is part of our future work.

Implementing XIA in the Linux kernel afforded us the ability to leverage code that has been developed over decades to lower the development barrier. For example, the three principals used in our exemplifying demo (§6.1) required only a modest engineering effort: the U4ID principal took 480 lines of source code, the zFilter principal 437 lines, and the XDP principal 724 lines according to SLOCCount [32]. We also reused the full POSIX API, the kernel module mechanism, and the hardware abstraction layer available in the kernel to achieve three goals: (1) provide XIA applications a rich API, (2) clearly scope principals code to avoid static dependencies, and (3) be fully independent of TCP/IP. None of these features are available in the XIA prototype [13].

The lessons learned here are that Linux XIA is deployable today, there is strong evidence that Linux XIA can embed many different architectures, and the Linux kernel affords principal developers a rich development environment.

6.3 Performance evaluation

Our forwarding performance evaluation consists of simulating an environment comparable to that seen by a core router, and measuring the impact of various key parameters: Internet users' preferences over destinations, packet sizes, different addresses, and update rates of the routing table on forwarding performance. We benchmark all these measurements against the mature Linux implementation of IP.

6.3.1 The testbed

This section covers three aspects of our experiments: how we simulated the conditions a core router sees, the software and hardware infrastructure we used, and the conditions under which we took the measurements.

Our experiments simulate a core router in which input ports are abstracted as packet writers (PWs) and output ports keep counters of successfully routed packets. We take as possible destinations the 462,150 CIDR blocks obtained from a recent Route Views snapshot [30]. All PWs choose destinations according to a Zipf distribution over these CIDR blocks to account for the popularity of the destinations. The output port for each CIDR block is chosen uniformly at random. All IP experiments reference IP addresses within these CIDR blocks; XIA experiments pessimistically map each CIDR block to a distinct AD XID, and reference these as destinations. We use ADs in the XIA evaluation because core routers are most likely to forward AD instances, so we view AD routing as representative of lookup cost. Before each run, we populate the IP and XIP routing tables with appropriate forwarding entries for each CIDR block.

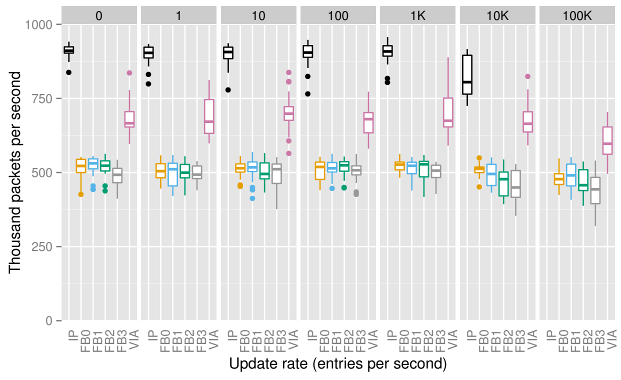
Our experiments use a variety of representative XIP address formats to assess the overhead of XIP packet processing. The VIA address format considers the simple case of one-level AD-based scoping, as described in §2, and the formats FB0 through FB3 consider the case of AD fallback with addresses using from 0 to 3 fallback edges, respectively.

Our experiments ran on a single machine with the router and PWs isolated by Linux Containers (LXC) [20], a lightweight virtualization technology. LXC has been used by others to simplify experiments and make them reproducible [19, 15, 6], and has helped us to focus on the cost of routing instead of dealing with distracting I/O overload and hardware features that we have not leveraged. Our brawny evaluation server has two Intel Xeon Processors E5-2690. Each processor has 8 cores plus Hyperthreading running at 2.90GHz that share 20MB of cache on chip, and a memory bank of 192GB registered DDR3 at 1333 MHz with ECC. All experiments ran with our custom kernel, which is a fork of Linux 3.11.0-rc7. Our kernel and our evaluation code are publicly available on GitHub [21, 22].

Our evaluation metric is the rate of succesfully forwarded packets. All evaluation graphics adopt the unit packets per second (pps) instead of throughput or goodput units, such as bytes per second, to reflect the fact that TCP/IP and XIA headers have different lengths. Each box in the graphics represents 20 runs of that same experiment. Further details of the evaluation setup, and more detailed performance results than described below, are available in the first author's Ph.D. thesis [23, Chapter 5].

6.3.2 The results

In spite of adding dynamically loaded principals, routing redirects, and routing dependencies on top of XIA's already flexible network addresses, Linux XIA sports performance results comparable to those of IP in our simulations of a core router. The results hold even while accounting for different packet sizes, more com-



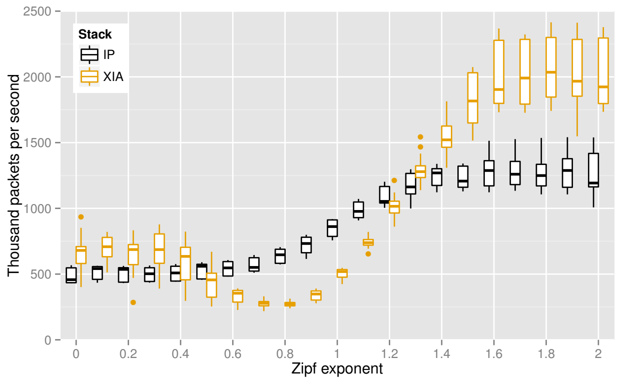


Figure 6: Comparing XIA to IP. Fixed parameters: 256-byte packets, 4 ports. (left) Effects of varying routing table update rate and destination address type. (right) Varying the distribution of destination addresses.

plex addresses used in XIA, and high update rates of the routing table. In addition, the results show that Linux XIA's solution to routing dependencies, the dependency forest, withstands very high update rates.

The distribution of packet addresses is the factor that shows the most pronounced effect between the XIA and IP routing algorithms in our experiments (Figure 6 (right)). Although XIA only approximates IP's forwarding performance in the most realistic range $[0.5, 1.2]$ of the Zipf exponent, it outperforms IP below 0.5 and above 1.2. Linux XIA's worst performance against IP happens when the Zipf exponent is 0.8, in which case XIA's median pps is only 42% of that of IP. We believe that the this performance gap can shrink significantly with more research, since Linux XIA is at version 1.0, and many improvements have not been explored. We do not have a proper understanding, at the time of this writing, why Linux XIA shows a non-monotonic behavior as the Zipf exponent varies. In the remainder of this section, we conservatively fix the Zipf exponent at 1.0.

Beyond the distribution of addresses, we conduct sensitivity analyses to quantify the impact of other variables on forwarding performance. These experiments largely demonstrate that packet sizes, complex addresses, and update rates (Figure 6 (left)) have a small impact on performance for both stacks. However, the IP stack, presumably unoptimized for this case, could not keep up with very high update rates, and as such has no box in the 100K column of the left panel of Figure 6. The advantage of IP over XIA is in accordance with Figure 6 (right) when the Zipf exponent is 1.0.

We do not yet have an adequate justification for the surprising performance of the VIA forwarding results shown in the left panel of Figure 6, as it seems no easier to forward VIA packets than FB0 packets. Although our missing justification for this better performance is vexing, this shows that there is room for improving Linux XIA's overall performance, which we will investigate in future work.

While our experiments are preliminary, they do make the case that Linux XIA is already a viable platform for exploration of network principals. In addition, we do not see a fundamental barrier holding XIA back from matching the performance of IP. We believe that the performance gap is rooted in the fact that Linux XIA is much less polished than Linux IP at this stage. Therefore, closing this gap could be a matter of time, implementing solutions available in the literature, e.g., [9, 35, 14].

7. CONCLUSIONS

Through a Linux implementation, the porting of diverse alien designs, a demonstration of interoperability, and performance benchmarks, we have tested the previously unsupported claim of the evolvability of the XIA framework, which we re-classify as a meta architecture. We view this evaluation of XIA as successful based on the fact that it has met our imposed challenges while remaining largely faithful to its original description [13]. This experience has provided us with deeper insight into XIA and has corroborated our view of an interoperable meta architecture being a catalyst to bring future Internet architectures closer to fruition.

Believing that the community at large is well-placed to both crowdsource and evaluate emerging efforts, we have made a consistent effort not to favor any one principal above another in the implementation of Linux XIA, but to have a level playing field for all principals. This lack of bias has guided all of our implementation choices, for example, Linux XIA does not require any principal to be loaded into the kernel, which leaves principal selection entirely to stakeholders. Ultimately, we expect that the aggregate of utility functions of stakeholders would select and evolve the set of principals deployed in large scale in an XIA Internet.

Those awaiting the arrival of a clean-slate replacement architecture may wish to consider reining in their expectations. Our view is that a winning architecture arriving in a single-focus form as TCP/IP did for host abstractions, or as NDN proposes to do for content, is implausible. As we amass experience with Linux XIA, we have come across a number of interesting ideas for principals that would benefit only a small subset of stakeholders. These narrow principals have led us to the idea that Linux XIA could end up becoming home to a collection of minimal-form principals (e.g., the LPM principal) that rely on each other to properly work, and, therefore, maximize value to stakeholders when considered *in toto*.

Not only do principals add value by themselves, they also increase the value of other principals. For example, 4ID principals bridge principals NDN and Serval to IPv4 networks; similarly, NDN and Serval motivate the use of 4IDs in the first place. These network effects could turn out to be the greatest source of value of Linux XIA since they can even increase the value of already deployed principals.

Finally, the major innovation of Linux XIA may not be XIA itself, but the broad set of new principals collectively and iteratively designed by others that Linux XIA hopes to enable.

8. ACKNOWLEDGMENTS

We would like to thank Sharon Goldberg, Orran Krieger, Peter Steenkiste, and the rest of the XIA team for feedback on our research and drafts of this paper, Erik Nordström for helping us with Serval's internals, Ilija Hadzic for shepherding the final version of this paper, and the anonymous reviewers for their useful comments. This research was supported by the National Science Foundation under awards CNS-1040800, CNS-1345307, and CNS-1347525.

9. REFERENCES

[1] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4), 2005.

[2] M. Arye, E. Nordström, R. Kiefer, J. Rexford, and M. J. Freedman. A formally-verified migration protocol for mobile, multi-homed hosts. In *IEEE ICNP*, 2012.

[3] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An xor-based erasure-resilient coding scheme. Technical Report TR-95-48, ICSI, 1995.

[4] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, 2013.

[5] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM*, 1998.

[6] C. M. S. Cabral, C. E. Rothenberg, and M. F. Magalhães. Reproducing real NDN experiments using mini-CCNx. In *ACM SIGCOMM Workshop on Information-Centric Networking (ICN)*, 2013.

[7] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *IEEE INFOCOM*, 1999.

[8] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An argument for network pluralism. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, 2003.

[9] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *ACM SIGOPS SOSP*, 2009.

[10] N. Feamster, J. Rexford, and E. Zegura. The road to SDN: An intellectual history of programmable networks. *ACM Queue*, 11(12), 2013.

[11] A. H. Ghani and P. Nikander. Secure in-packet Bloom filter forwarding on the NetFPGA. In *European NetFPGA Developers Workshop*, 2010.

[12] A. Ghodsi, T. Koponen, B. Raghavan, S. Shenker, A. Singla, and J. Wilcox. Intelligent design enables architectural evolution. In *ACM HotNets*, 2011.

[13] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient support for evolvable internetworking. In *USENIX NSDI*, 2012.

[14] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*, 2010.

[15] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *ACM CoNEXT*, 2012.

[16] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *ACM CoNEXT*, 2009.

[17] P. Jokela, A. Zahemszky, C. E. Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: Line speed publish/subscribe inter-networking. In *ACM SIGCOMM*, 2009.

[18] T. Koponen, S. Shenker, H. Balakrishnan, N. Feamster, I. Ganichev, A. Ghodsi, P. B. Godfrey, N. McKeown, G. Parulkar, B. Raghavan, J. Rexford, S. Arianfar, and D. Kuptsov. Architecting for innovation. *ACM SIGCOMM CCR*, 41(3), 2011.

[19] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *ACM HotNets*, 2010.

[20] Linux Community. LXC (linux containers) 0.9.0. `http://linuxcontainers.org/`, 2013.

[21] M. Machado. Linux XIA. `https://github.com/AltraMayor/XIA-for-Linux`, 2013.

[22] M. Machado. Network evaluation environment. `https://github.com/AltraMayor/net-eval`, 2013.

[23] M. Machado. *Linux XIA: An Interoperable Meta Network Architecture*. PhD thesis, Boston University, May 2014.

[24] M. K. Mukerjee, D. Han, S. Seshan, and P. Steenkiste. Understanding tradeoffs in incremental deployment of new network architectures. In *ACM CoNEXT*, 2013.

[25] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An end-host stack for service-centric networking. In *USENIX NSDI*, 2012.

[26] B. Raghavan, T. Koponen, A. Ghodsi, V. Brajkovic, and S. Shenker. Making the internet more evolvable. Technical Report TR-12-011, ICSI, 2012.

[27] B. Raghavan, T. Koponen, A. Ghodsi, M. Casado, S. Ratnasamy, and S. Shenker. Software-defined Internet architecture: decoupling architecture from infrastructure. In *ACM HotNets*, 2012.

[28] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar. Can the production network be the testbed? In *USENIX OSDI*, 2010.

[29] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1), 1997.

[30] University of Oregon. Route views project. `http://www.routeviews.org/`, 2013.

[31] D. J. Wetherall. *Service Introduction in an Active Network*. PhD thesis, Massachusetts Institute of Technology, 1999.

[32] D. A. Wheeler. SLOCCount 2.26. `http://www.dwheeler.com/sloccount/`, 2004.

[33] J. Wroclawski. All hat, no answers: Some issues related to the evaluation of architecture. Talk at the Spring '13 NSF FIA PI meeting. Slides at `http://www.nets-fia.net/Meetings/Spring13/FIA-Arch-Eval-JTW.pptx`.

[34] J. Zander and R. Forchheimer. Softnet – an approach to high level packet communication. In *ARRL Computer Networking Conference*, 1983.

[35] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance Ethernet forwarding with CuckooSwitch. In *ACM CoNEXT*, 2013.