BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

LINUX XIA: AN INTEROPERABLE META NETWORK ARCHITECTURE

by

MICHEL SILVA MACHADO

B.S., Pontifícia Universidade Católica do Rio de Janeiro, 2004
M.A., Boston University, 2008

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2014

© Copyright by
Michel Silva Machado
2014

Approved by

First Reader

John W. Byers, PhD
Associate Professor of Computer Science
Boston University

Second Reader

Orran Krieger, PhD
Research Professor of Computer Science
Boston University

Third Reader

Peter Steenkiste, PhD
Professor of Computer Science and
Electrical and Computer Engineering
Carnegie Mellon University

*Success is not measured by what you accomplish,
but by the opposition you have encountered,
and the courage with which you have maintained
the struggle against overwhelming odds.*
Orison Swett Marden

To Juliana and Lucas

Acknowledgments

I would like to thank Geraldo Bittencourt, Kauê Linden, and Ramiro Lobo for having believed in my potential at a very early stage, assumed the incurred costs of sending me to the US to pursue my PhD, and never given up on me when things did not go well.

If I have seen further it is by standing on the shoulders of *a giant*. Had John Byers not backed my wildest bets, my work would have been much smaller. I am enormously grateful for having had him as a role model.

Throughout my work, I have counted with the help of a terrier brother, Cody Doucette. I was responsible for teaching and guiding him to implement code that neither of us had written before. As a good team, all our work is better than it would have been without each other's help.

When the time of writing my thesis arrived, I could not find a way to put my work into a coherent story. That was when I serendipitously found David Wetherall's PhD thesis. Not only has his thesis inspired me to write mine, but its content was also a missing piece of my own work.

Finally, I would not have finished this thesis "on time" (if there is such a thing), had my mother, Aidê Viana, not stayed with my wife and I for six months to help with her latest grandson, Lucas.

To some degree, it is selfish to call this document *my* thesis. Thank you all, including those I did not mention here but have been with me throughout this journey, for having been there!

LINUX XIA: AN INTEROPERABLE META NETWORK ARCHITECTURE

(Order No.)

MICHEL SILVA MACHADO

Boston University, Graduate School of Arts and Sciences, 2014

Major Professor: John W. Byers, Associate Professor of Computer Science

ABSTRACT

With the growing number of 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 thesis presents the eXpressive Internet (Meta) Architecture (XIA), itself a clean-slate design, as well as Linux XIA, a native implementation of XIA in the Linux kernel, as a candidate. As a meta network architecture, XIA is highly flexible, leaving stakeholders to choose an expressive set of network principals to instantiate a given network architecture within the XIA framework. Central to XIA is its novel, non-linear network addressing format, from which derive key architectural features such as evolvability, intrinsically secure identifiers, and a low degree of principal isolation. XIP, the network layer protocol of XIA, forwards packets by navigating these structured addresses and delegating the decision-making and packet processing to ap-

propriate principals, accordingly. Taken together, these mechanisms work in tandem to support a broad spectrum of interoperable principals.

We demonstrate how to port four distinct and unrelated network architectures onto Linux XIA, none of which were designed for interoperability with this platform. We then show that, notwithstanding this flexibility, Linux XIA's forwarding performance remains comparable to that of the more mature legacy TCP/IP stack implementation. Moreover, the ported architectures, namely IP, Serval (Nordström et al., 2012), NDN (Jacobson et al., 2009), and ANTS (Wetherall, 1999), empower us to present a deployment plan for XIA, to explore design variations of the ported architectures that were impossible in their original form due to the requirement of self-sufficiency that a standalone network architecture bears, and to substantiate the claim that XIA readily supports and enables network evolution. Our work highlights the benefits of specializing network designs that XIA affords, and comprises instructive examples for the network researcher interested in design and implementation for future interoperability.

Contents

1 Introduction **1**

1.1 Definitions 3

1.2 The need for a meta network architecture 9

1.3 Overview of XIA 12

1.4 Taxonomy of meta architectures 16

1.5 Thesis statement and approach 18

1.6 Contributions 19

1.7 Thesis outline 21

2 eXpressive Internet (Meta) Architecture **23**

2.1 Introduction 23

2.2 Foundational ideas of XIA 27

2.3 eXpressive Internet Protocol 29

2.3.1 Principals 30

2.3.2 XIP addressing 32

2.3.3 XIP header and per-hop processing 36

2.4 XIP addresses in action 41

2.4.1 Bootstrapping addresses 41

2.4.2 Simple application scenarios 44

2.4.3 Support for richer scenarios 46

2.5 Related work 48

2.6 Conclusions 51

3 Linux XIA **53**

3.1 Influences from our experience 54

3.2 Forwarding packets 57

3.3 Routing dependencies 61

4 Porting alien designs to XIA **65**

4.1 Case study #1: IP 66

4.2 (In-depth) case study #2: Serval 69

4.2.1 Mapping Serval to XIA 71

4.2.2 Discussion 73

4.3 Case study #3: NDN 74

4.4 Case study #4: ANTS 77

5 Evaluation **81**

5.1 The testbed 82

5.2 The results 88

6 Conclusions **94**

6.1 Future work 95

6.2 Layers vs. factors 96

6.3 Conclusions 99

Bibliography **100**

Curriculum Vitae **109**

List of Figures

2·1 XIP packet header 36

2·2 Simplified diagram of an XIP router 38

2·3 Bank of the Future example scenario 45

3·1 Overview of the TCP/IP and XIA stacks in the Linux kernel 55

3·2 Linux XIA's routing algorithm 58

3·3 Example of routing-dependency linkage between routing table entries and DST entries in Linux XIA 63

4·1 Example of Serval's three-way connection handshake between a client on host a and a service instance on e 70

4·2 Example of XIA Serval's three-way connection handshake between a client and a service instance 72

5·1 Topology used in our forwarding performance evaluation 83

5·2 Address formats used in the evaluation of forwading performance of Linux XIA 85

5·3 Impact of the number of ports on performance 87

5·4 Impact of the Zipf exponent on performance 89

5·5 Packet size and more complex addresses have a small impact on performance 90

5·6 Impact of updating the routing table on performance 92

List of Abbreviations

4ID	IPv4 principal
6ID	IPv6 principal
AD	Autonomous Domain principal
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
CDN	Content Delivery Network
CID	Content principal
CIDR	Classless Inter-Domain Routing
DAG	Directed Acyclic Graph
DoS	Denial of Service
DNS	Domain Name System
FIA	Future Internet Architecture
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HID	Host principal
HTTP	HyperText Transfer Protocol
I4ID	IP 4ID
ICMP	Internet Control Message Protocol
ICN	Information-Centric Networking
IP	Internet Protocol
LPM	Longest Prefix Matching principal
LXC	LinuX Containers
MPLS	Multiprotocol Label Switching
NDN	Named Data Network
NWP	Neighborhood Watch Protocol
OSPF	Open Shortest Path First
POSIX	Portable Operating System Interface
PW	Packet Writer
SAL	Service Access Layer
TCP	Transmission Control Protocol
U4ID	UDP 4ID
UDP	User Datagram Protocol

X4ID	....	XIP 4ID
XDP	....	eXpressive Datagram Principal
XIA	....	eXpressive Internet (Meta) Architecture
XID	....	eXpressive IDentifier
XIP	....	eXpressive Internet Protocol

Chapter 1

Introduction

How are we, the research community, going to find the next Internet architecture? The research community is split into two groups, purists and pluralists, when it comes to answering this question (Anderson et al., 2005; Roscoe, 2006). With the benefit of hindsight, we can summarize those answers as follows: purists have been busy designing customized network architectures for the specific future scenarios that they each envision, whereas pluralists have focused on designs that both embrace the diversity of architectures and future scenarios in the purists' visions, as well as those yet unforeseen. We are pluralists.

The search for future Internet architectures is motivated by deficiencies of the current Internet. Many proposals have been put forward, but little has been done to broadly evaluate and compare these proposals, which in turn has split the efforts of the research community. We believe that a catalyst to find the next Internet architecture is missing, and that once the community comes to consensus to identify what the next Internet architecture should be, we would promptly work together to realize it.

In this work, we advocate eXpressive Internet (Meta) Architecture (XIA) (Anand et al., 2011a; Anand et al., 2011b; Han et al., 2012), a new kind of meta architecture that nurtures coexistence of clean-slate designs, lets stakeholders experiment with and choose the designs that best suit their needs. The distinctive feature of XIA over

previous meta architectures is that XIA enables coexistence of designs through the use of so-called *principals*, each of which are afforded the opportunity to specialize. These specialized principals enable designers to focus on the key functionality they want to materialize, and promotes composition or reuse of these functionalities by other supported principals. It is this distinctive feature of XIA that, through architectural pluralism, lets stakeholders experiment with and choose the designs that best suit their needs.

The rest of the introduction is organized as follows:

- First, we provide unifying terminology and definitions. The community lacks a consistent vocabulary for enabling architectural comparisons (or even an agreed-upon definition of network architecture itself). Therefore, we begin by presenting and formally defining our unifying terms: network factors, network architecture, and meta architectures in the next section.

- We argue for the need for meta architectures in Section 1.2. The presented discussion centers on a broad motivation of meta architectures. Although it encompasses some of the contexts that have motivated meta architectures in previous works, it does not try to summarize all of them.

- Section 1.3 succinctly introduces XIA. This short presentation gives readers a quick overview of our design deep enough to understand our contributions, and most of the other chapters.

- We introduce a taxonomy of meta architectures in Section 1.4 that classifies meta architectures documented in the literature, and contextualizes our work.

- Our thesis statement as well as a description of the approach we have employed to demonstrate our statement is in Section 1.5.

- Contributions other than those our thesis statement covers are in Section 1.6.
- Section 1.7 describes the organization of the remaining chapters.

1.1 Definitions

In order to clarify the exposition of our work, we have identified the need of crafting definitions for the following terms: network factor, architecture, and meta architecture. We introduce these terms one by one, each followed by examples, provide motivation for the structure of the definition, and an explanation of the vocabulary used in the definition. The section concludes with a justification for not reusing definitions found elsewhere.

The definitions in this section only cover data plane elements. Therefore, control plane elements such as routing protocols are not considered. We adopt Shenker's definitions for data and control planes (Shenker, 2013):

Data plane: process packets with local forwarding state.

Control plane: compute the forwarding state of the data plane.

Our first definition is factors:

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.

We can see that IP, TCP and UDP are all examples of factors. Although a factor is defined through many items, the essential one is furnishing a class of identifiers. IP fulfills this requirement with its IP addresses, while the identifiers TCP and UDP define are their port numbers. While this statement may seem obvious, it will gain contrast when we introduce XIP in Section 1.3, XIA's network layer protocol, which

does not define identifiers of its own, and thus is not a factor. Nevertheless, each of XIA's core principals are examples of a network factor.

The definition of network factor does not impose any size restrictions, reflecting the reality that factor designers, as well as different meta architectures, may optimize for different utilities, and may therefore view any such restrictions as arbitrary. This degree of freedom is expressed in the definition through the lack of quantifiers before every element that a factor defines except its classes of identifiers. An important side-effect of this flexibility in the definition is what we call *factor multiplicity*, that is, combining a fixed number of factors leads to a single factor, analogous to how numeric expressions are defined. Therefore, TCP/IP, that is TCP, UDP, and IP combined, is a factor.

Factor multiplicity copes with the fact that factors are often combined. The examples of this fact can be sophisticated, and may be out of reach of first-time readers. For example, in ANTS, if a coder wants to have the functionality of two or more factors in a single factor, she has to literally combine the mobile code that defines those factors, due to the degree of factor isolation in ANTS (Section 1.4); ANTS is discussed in Sections 1.4 and 4.4. As another example, factors U4ID, I4ID, and X4ID (Section 4.1) could be merged into a single factor if a network architect favors this approach. Finally, the Serval factor (Section 4.2) defines two classes of identifiers that are tightly intertwined.

With factors defined, we can now define architectures:

A network architecture is a self-sufficient factor.

Self-sufficiency is a test. The authors of an architecture describe what the architecture is good for, and their design is self-sufficient if they can describe any working aspect of the data plane of their design without resorting to components external to the design, other than a control plane that populates forwarding state and the hard-

ware necessary to implement the architecture. Self-sufficiency is analogous to proving that a program terminates for the inputs that their designers claim it would. Later, Section 6.2 presents a more advanced view of factors that allows us to fold the hardware necessary to implement an architecture into a factor, leaving only the control plane as an external component. It is worth pointing out that self-sufficiency is an implicit assumption in the literature on network architecture, and is well illustrated by the title and goal of Section 3 of Omega's design (Raghavan et al., 2012a) that reads as "Making Omega Sufficient".

Continuing our earlier examples, IP accomplishes its existential goal of sending packets from a host to another on its own, that is, IP is a self-sufficient factor; therefore, IP is an architecture. Given that TCP/IP is a factor, as established before, and IP is self-sufficient, TCP/IP is an architecture as well. The same is not true for TCP and UDP, which depend either upon IP, or upon an alternative internetwork factor, to function.

Thanks to factor multiplicity, the definition of architecture does not have to explicitly make mention of multiple factors. Having multiplicity in the definition of architectures is intuitively appealing, but having multiplicity in both definitions leads to ambiguous interpretations. For examples, TCP/IP would be an architecture either following the description above, or because the set of factors it includes would be an architecture as well. Finally, the choice of having multiplicity at the factor instead of at the architecture definition is due the prevalence of factor multiplicity as already discussed.

We define meta architectures directly over factors instead of over architectures:

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

The demand for supporting a broad spectrum of factors is meant to match the intuition that designs that support a limited amount of diversity are insufficiently general to warrant the meta architecture designation. Whereas the demand for not imposing static dependencies, or dependencies that arise in the *design* phase, among factors captures the notion that network evolution not only entails the ability to add new factors to a meta architecture, but also to drop deprecated factors. It is worth pointing out that our definition of meta architecture does not forbid *runtime* dependencies among factors (a technique we support and whose benefits we articulate later). Viewing them in a different way, static dependencies are imposed on stakeholders by designers, whereas runtime dependencies are derived from stakeholders' choices.

A framework that defines a meta architecture necessarily describes how factors are embedded in the framework. Thanks to this description, each supported factor has the same capabilities as any other factor in the framework. This description is analogous to how a country constitution folds states in its framework. This description is what *harmonizes*, that is, makes uniform, the interfaces that factors have to fulfill in order to be supported. Fulfilling these interfaces is the essential requirement for mapping factors onto a meta architecture. XIA's framework for harmonizing factors is defined by the XIP protocol, which is outlined in Section 1.3. While Section 1.4 presents other meta architectures, it also gives an overview of how the supported factors are folded into the framework of those meta architectures.

While designing and implementing a network architecture, it is easy to include an arbitrary dependency between factors. Architectural designers and coders find it alluring because it makes designs more concrete and intelligible earlier, and reduces initial coding effort. We define a *static dependency* among factors to be present whenever removing one factor from the set of deployed factors requires the removal of additional factors because they have to be *recoded* in order to work again. For

example, if IP were dropped from TCP/IP, TCP and UDP would need to be re-coded 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. Section 1.4 presents positive examples of meta architectures.

Three additional observations are warranted here. First, if a surviving factor has its functionality reduced due to the new set of deployed factors in which the factor finds itself, it does not characterize a static dependency. For example, consider the last internetwork factor that is removed from a deployed set that also includes the Serval factor (Section 4.2). Serval will no longer be able to establish connections between hosts, but it will still be able to establish connections whose endpoints reside within the same host. Second, network operators may deploy a meta architecture in such a way that they force runtime dependency among factors. Since these dependencies are derived from choices of stakeholders of the network, we do not view them as static dependencies, but rather as runtime dependencies. Finally, our third observation is that the control plane may need to be retooled to adapt to the change of the deployed set. As stated above, our definitions do not cover the control plane.

Our definitions of architectures and meta architectures are related through the factor definition. Factors are defined somewhat more abstractly than architectures. Once self-sufficiency is given up, the size and scope of factors become less clear. This property of the definition is desirable since it captures the intuitions that are present in other works. It is intuitively tempting to define meta architectures over architectures instead of over factors. In fact, Section 1.4 shows that yielding to this temptation has been the norm. The reason for relying on factors to define meta architectures is to reflect that factors, when allowed, can specialize, and increase their chance of being reused in contexts others than those that motivated their conception. In other words,

it reflects that promoting interoperability among factors is attainable. Factor LPM, introduced in Section 4.1 and reused in Section 4.2, materializes this point.

Readers may benefit from an analogy between our definitions and, perhaps more familiar, electric circuits. Factors are equivalent to electronic components; they can do as little as a resistor[1], or as much as an integrated circuit, which can amalgamate huge numbers of other electronic components, or somewhere in between those two extremes. Building upon this analogy, an architecture is a combination of a finite number of electronic components that together does something, for example: TVs, radios, and network switches. While we do not have a clear element in electronics to represent a meta architecture, the map proposed here highlights that defining meta architectures directly over electronic components, instead of over a fully functional combination of them, enables components being shared by the whole meta architecture.

To conclude, why did we not use definitions already available in the literature for factors, architectures, and meta architectures? We have not come across a formal definition equivalent to ours for factors, and the literature on meta architectures does not offer a broad definition that encompasses works of multiple authors as our definition of meta architecture does. We believe that the lack of a meta architecture definition in the literature explains why prior work has not contrasted their designs directly against other meta architectures instead of focusing on pointwise comparisons.

It may surprise some researchers that although the term architecture has long and extensively been used by the network community, few have tried to pin it down. Roscoe (Roscoe, 2006, Section 3) captures well this pattern in the literature: "I can't say what an architecture is, but I know it when I see it". It is not to say that we are the first to try to define architectures, Raghavan *et al.* has proposed the following definition (Raghavan et al., 2012b, Section 1):

[1] An electric component that controls current by providing resistance to the flow of electrons that crosses it.

Architecture: This refers to the current IP protocol or, more generally, any globally agreed upon convention that dictates how packets are handled.

While this definition explicitly agrees with ours that IP is an architecture, it places a high toll on new architectures since they have to become broadly understood before becoming an architecture. An online search for the string "define network architecture" lists a number of definitions. While we have found that these definitions are often reasonable, we have not found a definition that allowed us to relate an architecture and a meta architecture, to say when a network design becomes an architecture, or identify a single property that is common to all architectures. For example, a corollary from our definitions is that an architecture defines at least a class of identifiers.

Although this section extensively explains our definitions, we recognize that more examples and experience with them is required to master them. Many sections throughout this dissertation return to and elaborate more examples of these definitions. Finally, we expect that as other researchers seek to pick up these definitions, they will help to refine them further.

1.2 The need for a meta network architecture

An increasingly frequently debated question is whether or not TCP/IP has entered the end of its life cycle. There is no definitive answer to this question, and both sides of the argument have accumulated evidence to support their position. This section visits oft-cited evidence that TCP/IP is failing to keep up with demand, as well as arguments to dismiss this evidence, and presents a pragmatic view that stands aside of this discussion. This pragmatic view points out that the search for a TCP/IP replacement provides benefits independently of which side of the argument is right.

The contrarians, those who believe that TCP/IP is rapidly approaching the end of its life cycle, point to the growing pressure on aspects that the original design of

TCP/IP does not address, and has not evolved to do so. Among those aspects, the three most often cited are (1) limited node mobility, (2) lack of support for diversity at the network layer such as content routing, and (3) insufficient security. In the last year, a number of companies have reported evidence that the pressure on these aspects are growing: (1) the world surpassed a 1 billion smartphones in use (Koetsier, 2013; Hornyak, 2014), (2) static content takes more than half of all downstream traffic during peak periods (Sandvine, 2013), and (3) denial of service (DoS) attacks have threatened to disrupt the Internet (Prince, 2013).

The conservatives, those who believe that the Internet has not entered the end of its life cycle, point that incremental evolution has been the norm since the early days in 1970s, and argue that incremental evolution is the way to go (Rexford and Dovrolis, 2010). Mark Handley (Handley, 2006, Section 2.1) summarizes the incremental evolution that has kept the Internet growing, such as the development of DNS, congestion control, and Classless Inter-Domain Routing. Conservatives dismiss the evidence of the contrarians to point out that not only is the Internet properly working and still growing, but the pressure on weak aspects of the Internet are being worked out: (1) once deployed, Multipath TCP (Ford et al., 2011) holds the promise of addressing most of the mobility issue, (2) Content Delivery Networks are expected to handle 51% of all Internet traffic in 2017, up from 34% in 2012 (Cisco, 2013); moreover, an incrementally deployable version of Information-Centric Networking (ICN) has already been proposed (Fayazbakhsh et al., 2013), and (3) there already exist companies that provide DoS protection (CloudFlare, Inc., 2014; Arbor Networks, Inc., 2014; VeriSign, Inc., 2014).

Independently from when or whether TCP/IP will be replaced, moderates recognize that there are proposed solutions for (1) node mobility (Nordström et al., 2012), (2) an alternative network layer for ICN (Jokela et al., 2009), and (3) new defenses in

the literature (Yang et al., 2005) that have not been fully explored due to the lack of of a proper medium to prove their value. In addition, given the already large and still growing dependence of the world upon the Internet, it would be prudent to identify a suitable replacement for TCP/IP in a timely fashion, so as to avoid disruption that may take place if the search for this replacement is unfinished when demand becomes urgent. Finally, in case an opportunity for replacing TCP/IP is an elusive dream, valuable findings could nevertheless be back-ported onto the existing architecture.

The growing number of clean-slate redesigns of the Internet reflects the sensible effort of the network community to address the need of timely finding successors to TCP/IP. However, this effort lacks a medium to bring all stakeholders to participate in the realization, evaluation, and selection of future Internet architectures. 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.

Moreover, the current push toward cloud computing has centralized many servers into massive datacenters, which in turn creates opportunities to explore architectures that improve the return on investment made by datacenter operators. The networks of these datacenters are built from, to some degree, programmable devices, thanks to the rise of merchant silicon[2]. A meta architecture reduces the risk of managing these datacenters, because it does not bind datacenter operators to a single architecture,

[2] Merchant silicon are "off the shelf" chips that facilitate the implementation of networking devices such as switches and routers.

or to a small set of architectures that may turn out to be losers in the long run.

We believe that the missing catalyst to bring the exploration of future Internet architectures into fruition is a meta network architecture. In fact, part of the research community has already identified this need given the meta architectures that have been explored (Section 1.4). In the thesis, we advocate a new variety of meta architecture that promotes interoperability among ported factors to bring the community at large to create the next Internet architecture. The next sections present XIA, our meta architecture, and a taxonomy of meta architectures that provides a broad comparison of meta architectures, including XIA.

1.3 Overview of XIA

XIA's central goal is an evolvable and secure Internet architecture. By evolvable, XIA means having an explicit, well-defined, incremental path to introduce changes to its network layer, which is called the eXpressive Internet Protocol (XIP). These changes are introduced and removed in units; each of these units is called an XIA principal. We will show that XIA principals are examples of our more general term, factors. By secure, XIA means providing the capabilities to deliver security guarantees to applications. XIA's main vehicles to carry evolution and enable security guarantees are, its expressive network addresses and the intrinsic security found at its network identifiers, respectively. The remainder of this section presents the key concepts of XIA's design, puts these concepts together to form sample addresses, and concludes with why XIA with an empty set of principals is a meta architecture, but not an architecture, according to our definitions. The content here serves as a quick introduction to XIA, a more complete description is provided in Chapter 2.

In order for XIA principals to influence the forwarding mechanism of XIP, they must introduce their own identifiers. These identifiers are called eXpressive IDentifiers

(XIDs), and name any object or concept that principals define. Each XID is the pairing of a principal type (32 bits) and a name or ID (160 bits). Example of principals and corresponding XIDs are the Autonomous Domain (AD) principal, which names XIA networks, the Host (HID) principal, which names any machine (virtual or not) with an XIA stack, and the Content (CID) principal, which names immutable content.

Intrinsic security cryptographically links each XID's name to some property. For example, 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. When a network delivers to an application the file corresponding to the requested XID CID_1, the application can verify that it received the correct file by hashing the content of the file and comparing the hash against the content name CID_1. The hash of a public key allows an application to obtain the corresponding public key from any source, trusted or not, verify that it is the correct public key, and from there, bootstrap a secure communication to the entity bound to that public key. While it is desirable to have intrinsic security for all XIDs, this is not attainable because some principals do not have security properties to offer on their XIDs; see Section 4.1 for examples. Nevertheless, principal designers are strongly encouraged to imbue their principals' XIDs with intrinsic security wherever it is possible.

XIP addresses amalgamate principals' behaviors to accomplish application-level intents, and are represented as single-component, single-source, single-sink directed acyclic graphs (DAGs) 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 ($\bullet$), 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; see

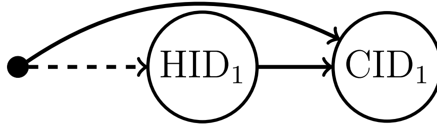
Section 4.3 for an example. All other (internal) nodes of an address represent XIDs, and each node is associated with between one and four strictly prioritized outgoing edges; four being the maximum fanout supported in XIP addresses.

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 that has been realized by this packet by forwarding decisions so far. Thus, when the packet reaches the intended destination, the `LastNode` will point to the sink.

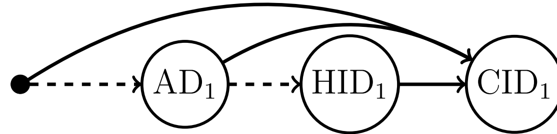
To forward a packet, a router first inspects the `LastNode` field to identify the progress made through the DAG so far. For each of the outgoing edges from the referenced node, in priority order, the router attempts to forward on the corresponding XID. If that XID is local to that router (for example, 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, as normal. 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 afford, 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 like $\bullet \rightarrow HID_1 \rightarrow CID_1$; this address 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:



Recalling our earlier architectural definitions, note that, unlike IP, XIP is not a factor, because it does not define any class of identifiers. Therefore, XIA with an empty set of principals, or equivalently, XIP alone, is not an architecture. On the other hand, each XIA principal, through the requirement that it introduce its own eXpressive IDentifiers (XIDs), is a network factor. Through its use of XIA principals (including those introduced in this section and those in Chapter 4), XIP harmonizes a broad spectrum of network factors without imposing any static dependencies (as discussed in Chapter 3). Therefore XIA constitutes a meta architecture. Stakeholders have to choose a set of factors that, together, instantiate XIA as an architecture. One plausible choice is inclusion of the AD, HID, and CID principals; this can serve as a baseline XIA architecture, according to our terminology.

1.4 Taxonomy of meta architectures

The notion of a meta network architecture that evolves to accommodate unforeseen network use cases has attracted researchers for decades. Researchers in Sweden developed Softnet (Zander and Forchheimer, 1983), the first meta architecture of which we are aware, in the early 1980s. While studying meta architectures, the degree to which they isolate their factors is enlightening because it highlights the value that meta architectures offer to applications, which ultimately reflect the utility functions of their end users. This section groups other meta architectures according to the degree of isolation between their factors; proceeding from the highest degree of factor isolation to the lowest.

Network virtualization (Anderson et al., 2005; Sherwood et al., 2010) and SDN/OpenFlow (McKeown et al., 2008; Bosshart et al., 2013; Feamster et al., 2013) 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 sets of resource that are used to support their factors; we call this group *slicing meta architectures*. The degree of factor isolation, however, is high, so high that applications are solely responsible for all the necessary work to leverage multiple factors, which requires access to multiple slices of the network. Moreover, the high degree of isolation forces the supported factors to be self-sufficient in order 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 (Crowcroft et al., 2003), FII (Koponen et al., 2011), OPAE (Ghodsi et al., 2011), Omega (Raghavan et al., 2012a), and SDIA (Raghavan et al., 2012b). 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 in fact 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 (Jacobson et al., 2009); Section 4.3 returns to this point.

The third group, *active meta architectures*, is centered on active networks (Tennenhouse and Wetherall, 1996; Tennenhouse et al., 1997), and most notably ANTS (Wetherall, 1999), 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. Nevertheless, factors can be combined to compose a single factor. But 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 takes place with (1) XIA factors sharing the whole network, as in ANTS, (2) network addresses enabling factor composition at every address (Sections 1.3 and 2.3.2), 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 (Section 3.2). 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 all of the meta architectures cited above.

The degree of factor isolation has deep effects on meta architectures. A high degree of isolation forces factors to be architectures, as arises in slicing, translating, and active meta architectures. In contrast, the low degree of isolation found in XIA allows factors to specialize and achieve their functionality with minimal form. This effect is explored in Chapter 4, where we leverage this flexibility when porting designs onto the XIA meta architecture.

1.5 Thesis statement and approach

The central hypothesis in this dissertation is that *an interoperable meta network architecture can welcome most, if not all, clean-state designs on a level playing field, yet it leaves the final evaluation of these designs to the broader community.* It is worth pointing out that although we emphasize clean-state designs because they lack a platform for deployment, XIA welcomes existing designs into its framework as well. The following chapters demonstrate this hypothesis by defining XIA, describing our native implementation of XIA, and answering the following questions:

- Is XIA expressive enough to accommodate a broad spectrum of factors? If the expressiveness of XIA were myopic, our thesis hypothesis would have a rather limited scope. To answer this question we ported four distinct and unrelated network architectures onto Linux XIA, namely IP, Serval (Nordström et al., 2012), NDN (Jacobson et al., 2009), and ANTS (Wetherall, 1999). None of the ported architectures were designed for interoperability with XIA. Moreover, the port of ANTS, an early meta architecture with a high degree of generality, is the essential step in our larger demonstration that the evolutionary model of XIA supports a superset of that of ANTS.

- What is necessary to avoid static dependency among factors? While static dependency may be spotted in some high-level descriptions of designs, this is not always the case. Some static dependencies only become apparent when designs are made concrete through an implementation; this was the case with XIA. During the development of Linux XIA, static dependency was creeping into the code due to the need to reuse functionality of factors by other factors. Avoiding this static dependency required a small change on the forwarding algorithm of XIA to include what we named routing redirects.

- What is impact of the flexibility built into Linux XIA on forwarding performance? Given that Linux XIA supports dynamically loaded factors, routing redirects, and dynamic routing dependencies on top of XIA's already flexible addresses, the forwarding performance of Linux XIA is a concern for anyone interested in developing factors, or deploying XIA. Not to mention that failing to realize a reasonable forwarding performance would again limit the scope of our hypothesis. Linux XIA sports performance results comparable to those of Linux IP, a mature implementation of TCP/IP, in our simulations of a core router. The results hold even while accounting for different packet sizes, more complex addresses used in XIA, and high update rates of the routing table.

During the development of the work to substantiate our thesis hypothesis and to answer the questions above, we have made a number of other contributions, which we discuss next.

1.6 Contributions

The central finding of this research is that XIA can instantiate a broad spectrum of factors, and avoid static dependence among factors. The port of IP, Serval (Nordström

et al., 2012), NDN (Jacobson et al., 2009), and ANTS (Wetherall, 1999) provides evidence of the expressiveness of XIA. The forwarding performance measurements of Linux XIA show that the additional flexibility is reasonable when compared to IP. More broadly, we conclude that Linux XIA is a suitable platform for the search of the next Internet architecture.

The contributions of this dissertation also include the following:

- Clarifying interpretations of network factors, network architectures, and meta network architectures;
- A taxonomy of meta architectures;
- An open source, fully functional, native implementation of XIA in Linux (Machado, 2013a);
- An open source environment for evaluation of forwarding performance of network stacks implemented in Linux (Machado, 2013b);
- A deployment plan of XIA that leverages the current Internet as a medium to interconnect XIA networks. Any successful clean-slate architecture will need to coexist with IP for years (or forever). 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;
- A design improvement of Serval that is not possible in its original form; more specifically, ServalID identifiers are intrinsic secure in XIA, what enables resilience against on-path attacks. Serval is a service-centric architecture that complements Linux XIA with a mobile, multipath, reliable transport;
- Two whitepaper ports of NDN to XIA that expose the tradeoff between supporting dynamic content and public-key management in NDN. NDN, a content-

centric architecture, embodies a design that may seem to necessitate a standalone architecture; its port to XIA, in fact, tests aspects of XIA not explored by other architectures, such as the fundamental role of source addresses.

1.7 Thesis outline

The following chapters are organized as follows. The two following chapters define XIA and discuss its concepts at a high level of abstraction (Chapter 2), and present key internals of our Linux implementation of XIA (Chapter 3). While Chapter 2 addresses the first defining property of meta architectures, namely, the support to a broad spectrum of factors, it is only in Chapter 3 that we show how to address the second defining property of meta architectures, that is, how to avoid static dependencies among factors. While one can appreciate the harmful effects of static dependency at a high level, it is during the implementation that the second defining property gains color.

Chapter 4 shows how to map alien designs onto XIA. This chapter accomplishes many things in parallel to the description of the ports, for example, it (1) presents a deployment plan of XIA through the port of IP, (2) shows how Serval can become resilient to on-path attacks, (3) exposes a key tradeoff of NDN as well as a path for componentization of the design of NDN, and (4) substantiates the generality of XIA with the port of ANTS to XIA. All those side goals are accomplished while providing instructive porting examples for other researchers, and fulfilling our view of XIA being the missing catalyst to bring the next Internet architecture.

After being exposed to all the flexibility that is built into Linux XIA, a natural question is on how its forwarding performance compare to IP, more specifically, to Linux's implementation of IP. Chapter 5 evaluates the forwarding performance of Linux XIA against that of Linux IP.

Finally, we present future work directions, a new interpretation of layers in network architectures, and a summary of our work and view of the future in Chapter 6.

Chapter 2

eXpressive Internet (Meta) Architecture

Motivated by limitations in today's host-centric IP architecture, recent studies have proposed clean-slate network architectures centered around alternate first-class principals, such as content, services, or users. However, much like the host-centric IP design, elevating one principal above others hinders communication between other principals and inhibits the network's capability to evolve. This chapter presents the eXpressive Internet (Meta) Architecture (XIA), a meta architecture with native support for multiple principals and the ability to evolve its functionality to accommodate new, as yet unforeseen, principals over time. We describe key design requirements, and demonstrate how XIA's rich addressing and forwarding semantics facilitate flexibility and evolvability, while keeping core network functions simple and efficient. We describe case studies that demonstrate key functionality that XIA enables.

The content of this chapter borrows extensively from our prior work (Anand et al., 2011a; Anand et al., 2011b; Han et al., 2012). Nevertheless, the text here has been harmonized with our new definitions introduced in Section 1.1.

2.1 Introduction

The "narrow waist" design of the Internet has been tremendously successful, helping to create a flourishing ecosystem of applications and protocols above the waist, and diverse media, physical layers, and access technologies below. However, the Internet,

almost by design, does not facilitate a clean, incremental path for the adoption of new capabilities at the waist. This shortcoming is clearly illustrated by the 15+ year deployment history of IPv6 and the difficulty of deploying primitives needed to secure the Internet. Serious barriers to evolvability arise when:

- Protocols, formats, and information must be agreed upon by a large number of independent actors in the architecture; and
- There is no built-in mechanism that supports (and embraces) incremental deployment of new functionality with minimal friction.

IP today faces both of these barriers. First, senders, receivers, and every router in between must agree on the format and meaning of the IP header. It is not possible, therefore, for a destination to switch to IPv6-based addressing and still remain reachable by unmodified senders who use IPv4. Second, today's paths to incremental deployment typically involve tunnels or overlays, which have the drawback that they *hide* the new functionality from the existing network. For example, enabling a single router in a legacy network to support some form of content-centric networking is fruitless if that traffic ends up being tunneled through the network using IPv4[1].

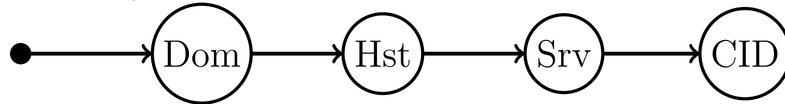
This chapter presents a meta architecture, called the eXpressive Internet (Meta) Architecture or XIA, that addresses these problems from the ground up. XIA maintains some features of the current Internet, such as a narrow waist that networks must support, and packet switching, but it differs from today's Internet in several areas.

The philosophy underlying the design of XIA is, simply, that we do not believe we can predict the usage models for the Internet of 50 years hence. The research community has presented compelling arguments for supporting many types of communication—content-centric architecture (Jacobson et al., 2009; Trossen et al., 2010), service-based communication (Nordström et al., 2012; Saif and Paluska, 2003),

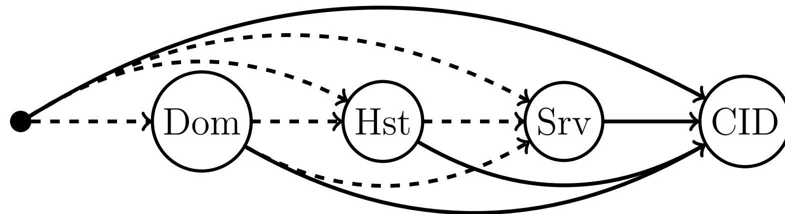
[1]Without resorting to deep packet inspection of various sorts, which itself is typically fragile.

multicast (Deering, 1991), user-centric architecture (Ford, 2008), enhanced support for mobility (Snoeren and Balakrishnan, 2000; Raychaudhuri et al., 2010; Zhuang et al., 2003), and so on. We believe that a new network architecture should facilitate *any or all* of these capabilities, and it must be possible to enable or disable “native” support for them as makes sense in that time and place.

The key architectural element that XIA adds to improve evolvability is one we call expressing *intent*. XIA’s addresses can simultaneously express both a “new” type of address (or addresses), and one or more backwards compatible pathways to reach that address. This notion is best explained by example: Consider the process of retrieving a particular piece of content (CID) using a network *that provides only host-to-host communication*, much like today’s Internet. The source would send a packet destined to a destination network (Dom); the destination network would deliver it to a host; the host would deliver it to a process providing a service (Srv) such as HTTP; and the process would reply with the desired content, as such:



XIA makes this path explicit in addressing, and allows flexibility in expressing it, e.g., “The source really just wants to retrieve this content, and it does not care whether it goes through Dom to get it.” As a result, this process of content retrieval might be expressed in XIA by specifying the destination address as a *directed acyclic graph*, not a single address, like this:



By expressing the destination in this way, senders *give flexibility to the network to satisfy their intent*. Imagine a future network in which the destination domain supported routing directly to services (Nordström et al., 2012) (instead of needing to route to a particular host). Using the address as expressed above, this hypothetical network would already have both the *information* it needs (the service ID) and *permission* to do so (the link from the source directly to the service ID). A router or other network element that does not know how to operate on, e.g., the service ID, would simply route the packet to the furthest-along node that it does know (the domain or the host in this example). Note that the sender can use the same address before and after support for service routing is introduced.

XIA terms the identifiers on each node in an address *eXpressive IDentifiers* (XIDs); the supported principals define the XIDs. Examples of supported principals include hosts, autonomous domains (analogous to today's autonomous systems), services, content IDs, and so on. The set of principals in XIA is not fixed: hosts or applications can define new principals and begin using them *at any time*, without waiting for support from the network. Of course, if they want to get anything done, they must also express a way to get their work done in the current network. We believe that the ability to express not just "how to do it today", but also the application's underlying intent, is key to enabling future evolvability.

The second difference between XIA and today's Internet comes from a design philosophy that encourages creating principals that have *intrinsic security*: the ability for an entity to validate that it is communicating with the correct counterpart without needing access to external databases, information, or configuration. An example of an intrinsically secure address is using the hash of a public key for a host address (Andersen et al., 2008). With this mechanism, the host can prove that it sent a particular packet to any receiver who knows its host address. Intrinsic security is

central to reliable sharing of information between principals and the network and to ensuring correct fulfillment of the contract between them. It can furthermore be used to bootstrap higher level security mechanisms.

We outline novel design ideas for evolution (Section 2.2), and systematically incorporate them into the eXpressive Internet Protocol (XIP) (Section 2.3). Then, through concrete examples, we show how networks, hosts, and applications interact with each other and benefit from XIA's flexibility and evolvability (Section 2.4). Finally, we discuss related work (Section 2.5) and conclude (Section 2.6).

2.2 Foundational ideas of XIA

XIA is based upon three core ideas for designing an evolvable and secure Internet architecture:

1. Principals as the units of evolution. Applications can use one or more principals to directly express their intent to access specific functionality. Each principal defines its own "narrow waist", with an interface for applications and ways in which routers should process packets destined to a particular type of principal.

XIA supports an open-ended set of principals, from the familiar (hosts), to those popular in current research (content, or services), to those that we have yet to formalize. As new principals are introduced, applications may start to use these new principals at any time, *even before the network has been modified to natively support the new function*. This allows incremental deployment of native network support without further change to the network endpoints, as we will explore through examples in Sections 2.4.2 and 2.4.3. Every time that a principal is introduced or removed from the set of deployed principals, the network evolves.

Establishing principals as the units of evolution in XIA clearly distinguishes XIA

from previous meta architectures. Design decisions have motivated previous meta architectures to keep a higher degree of isolation among their supported factors, which has forced these factors to be architectures (Section 1.4). Enabling non-self-sufficient principals in XIA allows principals to specialize their semantic and functionality to what they do best, which, in turn, increases the chance of principals to be reused in contexts not foreseen by their designer[2].

2. Flexible addressing. XIA aims to avoid the "bootstrapping problem": why develop applications or protocols that depend on network functionality that does not yet exist, and why develop network functionality when no applications can use it? XIA provides a built-in mechanism for enabling new functions to be deployed piecewise, e.g., starting from the applications and hosts, then, if popular enough, providing gradual network support. The key challenge is: how should a legacy router in the middle of the network handle a new principal type that it does not recognize? To address this, we introduce the architectural notion of a *fallback*. Fallbacks allow communicating parties to specify alternative action(s) if routers cannot operate upon the primary intent. We provide details in Section 2.3.2.

3. Intrinsically secure identifiers. IP is notoriously hard to secure, as network security was not a first-order consideration in its design. XIA aims to build security into the core architecture as much as possible, without impairing expressiveness. In particular, principals used in XIA source and destination addresses are strongly encouraged to be *intrinsically secure*, i.e., cryptographically derived from the associated communicating entities in a principal-specific fashion. This allows communicating entities to more accurately ascertain the security and integrity of their transfers; for

[2] A good example of the value of specialization is available later, in Chapter 4, with principal LPM. LPM is reused in Section 4.2 for a totally different motivation from the one that led to its introduction in Section 4.1.

example, a publisher can attest that it delivered specific bytes to the intended recipients. Section 2.3.1 describes the intrinsic security of some principals as well as the specification requirements for intrinsic security for new principals. While we expect the principal designers will prefer to support intrinsic security wherever possible, we recognize that some valuable principals may not have such an enabling security property; see Section 4.1 for examples.

2.3 eXpressive Internet Protocol

XIA facilitates communication between a rich set of principals. We therefore split both the design and our discussion of communication within XIA into two components: the XIP protocol, and XIA forwarding behavior. First, the basic building block of per-hop communication is the core eXpressive Internet Protocol, or XIP. XIP is principal-independent, and defines an address format, packet header, and associated packet processing logic. A key element of XIP is a flexible format for specifying multiple paths to a destination principal, allowing for “fallback” or “backwards-compatible” paths that use, e.g., more traditional autonomous system and host-based communication.

The second component is the per-hop processing for each principal type. Principals name entities that they define with uniquely typed, expressive identifiers which we refer to as XIDs. In this chapter, we focus on host, service, content, and administrative domain principals to provide examples of how XIA supports multiple principals. We refer to the above types as HIDs, SIDs, CIDs, and ADs, respectively. The set of principals is extensible, and more examples are found in Chapter 4.

Our goal is for the set of deployed principals to evolve over time. We envision that an initial deployment of XIA would mandate support for ADs and HIDs, which provide global reachability for host-to-host communication—a core building block

today. Support for other principals would be optional and over time, future network architects could mandate or remove the mandate for these or other principals as the needs of the network change. Or, put more colloquially, we do not see the need for ADs and HIDs disappearing any time soon, but our own short-sightedness should not tie the hands of future designers!

2.3.1 Principals

The specification of a principal must define:

1. The semantics of communicating with the entities that the principal defines.

2. One or more unique XID types, methods for allocating XIDs, and, when possible, a definition of the intrinsic security properties of any communication involving these XIDs. We expect that these intrinsically secure XIDs will typically be globally unique, even if, for scalability, they are routed using hierarchical means, and that they will be generated in a distributed and collision-resistant way. Nevertheless, given the diversity of architectures, innovative ways to define XIDs are going to be common as well; Chapter 4 explores some examples.

3. Any principal-specific per-hop processing and routing of packets that must either be coordinated or kept consistent in a distributed fashion.

These three features together define the principal-specific support for a new principal. The following material describes the administrative domain, host, service, and content principals in terms of these features.

Network and *host* principals represent autonomous routing domains and hosts that attach to the network. ADs provide hierarchy or scoping for other principals, that is, they primarily provide control over routing. Hosts have a single identifier that is constant regardless of the interface used or network that a host is attached

to. ADs and HIDs are self-certifying: they are generated by hashing the public key of an autonomous domain or a host, unforgeably binding the key to the address. The format of ADs and HIDs and their intrinsic security properties are similar to those of the network and host identifiers used in AIP (Andersen et al., 2008).

Services represent an application service running on one or more hosts within the network. Examples range from an SSH daemon running on a host, to a Web server, to Akamai's global content distribution service, to Google's search service. Each service will use its own application protocol, such as HTTP, for its interactions. An SID is the hash of the public key of a service. To interact with a service, an application transmits packets with the SID of the service as the destination address. Any entity communicating with an SID can verify that the service has the private key associated with the SID. This allows the communicating entity to verify the destination and bootstrap further encryption or authentication.

In today's Internet, the true endpoints of communication are typically application processes—other than, e.g., ICMP messages, very few packets are sent to an IP destination without specifying application port numbers at a higher layer. In XIA, this notion of processes as the true destination can be made explicit by specifying an SID associated with the application process (e.g., a socket) as the intent. An AD, HID pair can be used as the "legacy path" to ensure global reachability, in which case the AD forwards the packet to the host, and the host then forwards it to the appropriate process (SID). Section 2.4 shows that making the true process-level destination explicit facilitates transparent process migration, which is difficult in today's IP networks because the true destination is hidden as state in the receiving end-host.

Lastly, *the content principal* allows applications to express their intent to retrieve content without regard to its location. Sending a request packet to a CID initiates

retrieval of the content from either a host, an in-network content cache, or other future source. CIDs are the cryptographic hash (e.g., SHA-1, RIPEMD-160) of the associated content. The self-certifying nature of this identifier allows any network element to verify that the content retrieved matches its content identifier.

2.3.2 XIP addressing

Next, we introduce key concepts for XIP addresses that support the long-term evolution of principals, the encoding mechanism for these addresses, and a representative set of addressing "styles" supported in XIP.

Core concepts in addressing

XIA provides native support for multiple principals, allowing senders to express their intent by specifying an XID as part of the XIP destination address. However, XIA's design goal of evolvability implies that a principal used as the intent of an XIP address may not be supported by all routers. Evolvability thus leads us to the architectural notion of **fallback**: intent that may not be globally understood *must* be expressed with an alternative backwards compatible route, such as a globally routable service or a host, that can satisfy the request corresponding to the intent. This fallback is expressed *within* an XIP address since it may be needed to reach the intended destination.

XIP addressing must also deal with the fact that not all XID types will be globally routable, for example, due to scalability issues. This problem is typically addressed through scoping based on network identifiers (Andersen et al., 2008). Since XIA supports multiple principals, we generalize scoping by allowing the use of XID types other than ADs for scoping. For example, scaling global flat routing for CIDs may be prohibitively expensive (Stoica et al., 2002; Balakrishnan et al., 2004), and, thus,

addresses containing *only* a CID may not be routable. Allowing the application to refine its intent using hierarchical **scoping** using ADs, HIDs, or another principal to help specify the CID's location can improve scalability and eliminate the need for XID-level global routing.

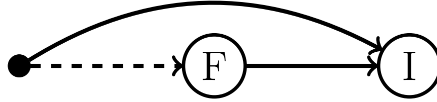
The drawback of scoping intent is that a narrow interpretation could limit the network's flexibility to satisfy the intent in the most efficient manner, e.g., by delivering content from the nearest cache holding a copy of the CID, rather than routing to a specific publisher. We can avoid this limitation by combining fallback and scoping, a concept we call (iterative) **refinement** of intent. When using refinement of intent, we give the XID at each scoping step the opportunity to satisfy the intent directly without having to traverse the remainder of the scoping hierarchy.

Addressing mechanisms

XIA's addressing scheme is a direct realization of these high-level concepts. To implement fallback, scoping, and iterative refinement, XIA uses a restricted directed acyclic graph (DAG) representation of XIDs to specify XIP addresses. A packet contains both the destination DAG and the source DAG to which a reply can be sent. Because of symmetry, we describe only the destination address.

Three basic building blocks are: intent, fallback, and scoping. XIP addresses must have a *single* intent, which can be of any XID type. The simplest XIP address has only an entry node and an intent (I) as the sink: $\bullet \rightarrow I$. The entry node ($\bullet$) appears in all visualizations of XIP addresses to represent the conceptual source of the packet, and to indicate where the navigation of the address begins.

A fallback is represented using an additional XID (F) and a "fallback" edge (dotted line):



The fallback edge can be taken if a direct route to the intent is unavailable; we allow up to four outgoing edges of each node, including the entry node. Section 2.3.3 justifies this fanout limit.

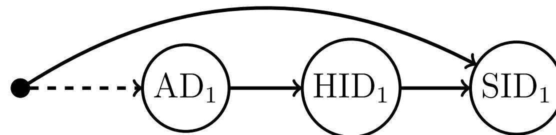
Scoping of intent is represented as: $\bullet \rightarrow S \rightarrow I$. This structure means that the packet must be first routed to a scoping XID S, even if the intent is directly routable.

These building blocks are combined to form more generic DAG addresses that deliver rich semantics, implementing the high-level concepts described above. To forward a packet, routers traverse edges in the address in order and forward using the next routable XID. Detailed behavior of packet processing is specified in Section 2.3.3.

Addressing style examples

XIP's DAG addressing provides considerable flexibility. In this subsection, we present three (non-exhaustive) "styles" of how it might be used to achieve important architectural goals.

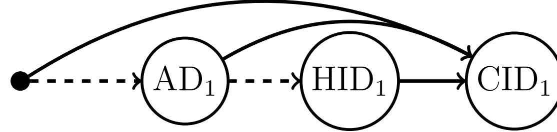
Supporting evolution: The destination address encodes a service XID as the intent, and an autonomous domain and a host are provided as a fallback path, in case routers do not understand the new principal:



This scheme provides both fallback and scalable routing. A router outside of AD_1 that does not know how to route based on intent SID_1 directly will instead route to AD_1.

Iterative refinement: In this example, every node includes a direct edge to the

intent, with fallback to domain and host-based routing. This allows iterative incremental refinement of the intent. If the CID_1 is unknown, the packet is then forwarded to AD_1. If AD_1 cannot route to the CID, it forwards the packet to HID_1.



An example of the flexibility afforded by this addressing is that an on-path content-caching router could directly reply to a CID query without forwarding the query to the content source. We term this *on-path interception*. Moreover, if technology advances to the point that content IDs became globally routable, the network and applications could benefit directly, without changes to applications.

Service binding and more: DAGs also enable application control in various contexts. In the case of legacy HTTP, while the initial packet may go to any host handling the web service, subsequent packets of the same "session" (e.g., HTTP keep-alive) *must* go to the same host. In XIA, we do so by having the initial packet destined for: $\bullet \rightarrow AD_1 \rightarrow SID_1$. A router inside AD_1 routes the request to a host that provides SID_1. The service replies with a source address bound to the host, $\bullet \rightarrow AD_1 \rightarrow HID_1 \rightarrow SID_1$, to which subsequent packets can be sent. We explore an alternative approach to service binding in the context of our port of Serval, as described in Chapter 4.

Other uses of DAG addresses are described in (Anand et al., 2011a). Some potential uses of DAG-based addresses, such as source routing, raise questions of policy and authorization, which have been addressed in the literature (Zhang et al., 2011; Naous et al., 2011; Zhao et al., 2012).

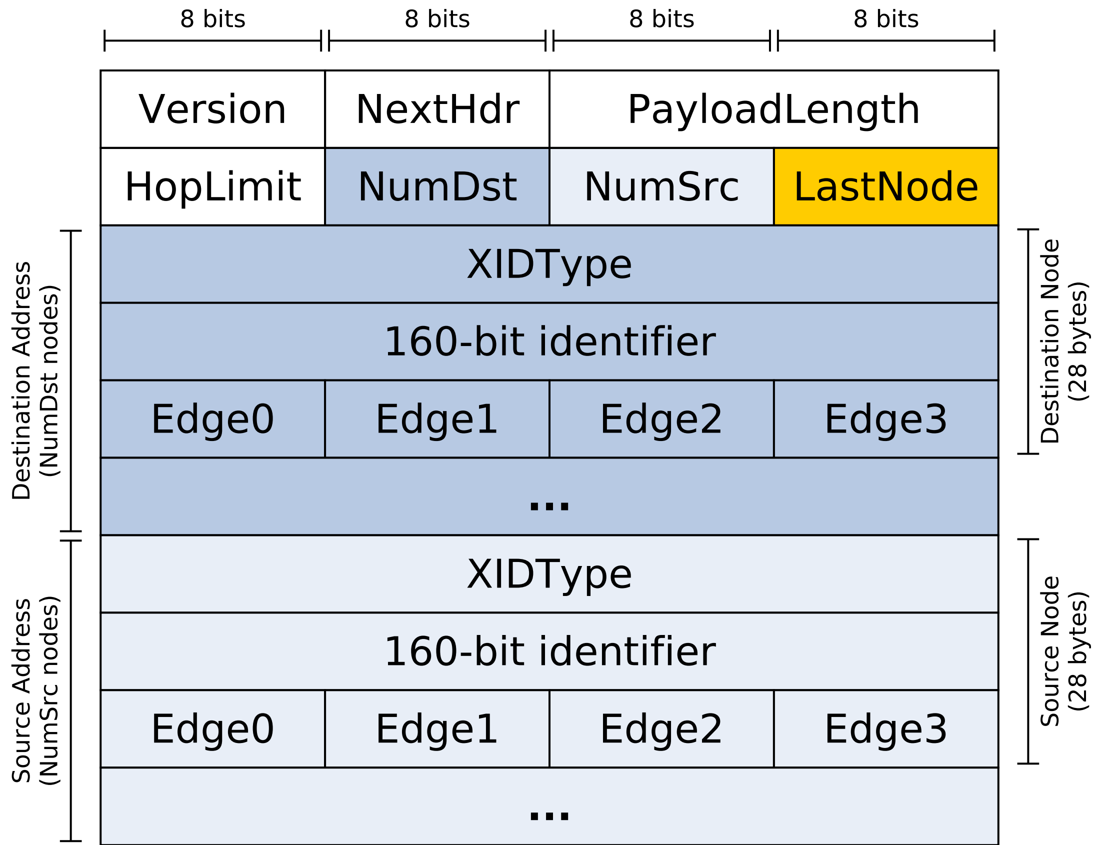


Figure 2·1: XIP packet header.

2.3.3 XIP header and per-hop processing

This section describes the XIP packet format and per-hop processing that routers perform on packets. Later, in Chapter 5, we show that this design satisfies both the requirements for flexibility and efficient router processing.

Header format

Figure 2·1 shows the header format. Our header encodes a source and a destination DAG, and as a result our address is variable-length—`NumDst` and `NumSrc` indicate the

size of the destination and source address, respectively. The header contains fields for version, next header, payload length, and hop limit.

Our header stores a pointer, `LastNode`, to the previously visited node in the destination address, for routers to know where to begin forwarding lookups. This makes per-hop processing more efficient by enabling routers to process a partial DAG instead of a full DAG in general.

DAGs are stored as adjacency lists. Each node in the adjacency list contains three fields: an XID Type; a 160-bit XID; and an array of the node indices that represent the node's outgoing edges in the DAG. The adjacency list format allows at most four outgoing edges per node (`Edge0`...`Edge3`). This choice balances: (a) the per-hop processing cost, overall header size, and simple router implementation; with (b) the desire to flexibly express many styles of addressing. However, we do not limit the degree of expressibility; one can express more outgoing edges by using a special node with a predefined `XIDType` to represent indirection.

Note that our choice of 160-bit XID adds large overhead, which could be unacceptable for bandwidth or power-limited devices. We believe, however, that common header compression techniques (Lilley et al., 2000) can effectively reduce the header size substantially without much computational overhead.

The XIP header packs DAG addresses using topological sorting. Any DAG is guaranteed to have at least one topological ordering, and the ordering can be easily found by applying a typical topological sorting algorithm. The topological ordering brings two good properties: cheap verification that an XIP address does not have cycles, and a compact encoding of the entry node. Due to the topological ordering, a node cannot have edges pointing to nodes that appear before the node; by exploiting this fact, hosts can quickly check if the adjacency list indeed represents a DAG (i.e. has no cycle). Similarly, the sink node cannot precede other nodes, so the sink node

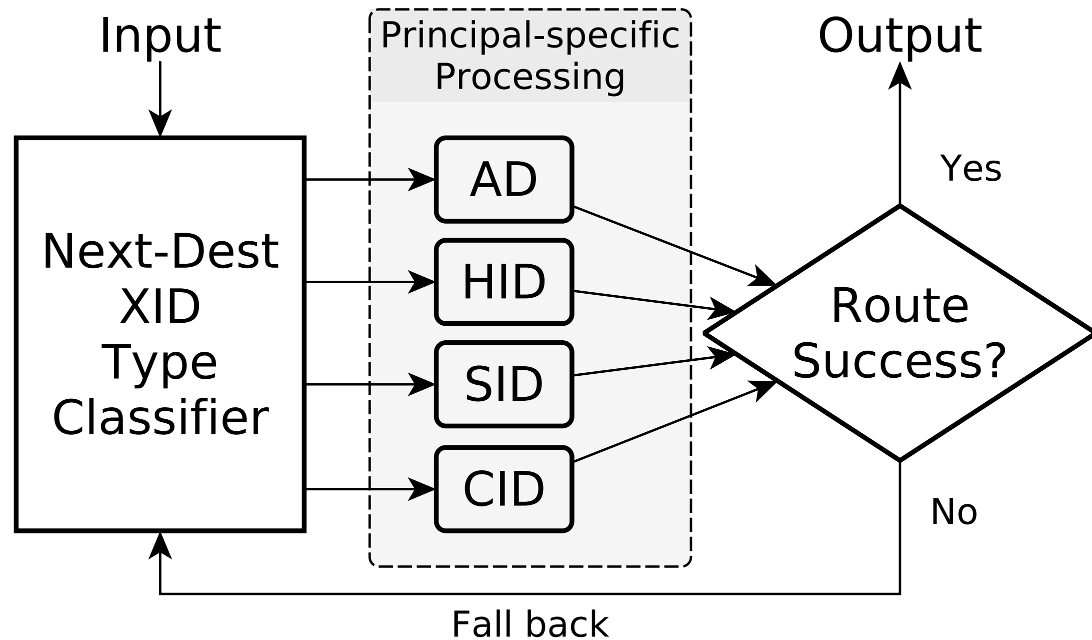


Figure 2·2: Simplified diagram of an XIP router.

always appears at the bottom of the adjacency list. For compact encoding, the entry node's edges reuse the sink node's edge fields, which are always empty.

Per-hop processing

Figure 2·2 depicts a simplified flow diagram for packet processing in an XIP router. The edges represent the flow of packets among processing components. Shaded elements are principal specific, whereas other elements are common to all principals. Our design isolates principal-type specific logic to make it easier to add support for new principals.

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. When a packet arrives, a router iteratively examines the outbound edges of the last-visited node of the DAG in priority order. The `LastNode` field, initially set to the entry node, reflects the portion of the DAG that has been realized by this packet by forwarding decisions so far. Thus, when the packet reaches the intended destination, the `LastNode` will point to the sink. We refer the node pointed by the edge in consideration as the next destination.

To attempt to forward along an adjacency, the router examines the XID type of the next destination. If the router supports that principal, it invokes a principal-specific component based on the XID type; this is what enables principals to customize the forwarding mechanism. If the router does not support the principal or does not have an appropriate forwarding rule, it moves on to the next edge. If no outgoing edge of the last-visited node can be used for forwarding, the destination is considered unreachable and the packet is dropped. Principals decide when `LastNode` field is updated. For example, if the next destination is an AD XID, 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. When an edge is chosen, the router sets the most significant bit of the entry of the edge array used to forward the packet; these marks are useful for probing and debugging.

Optimizations for high performance

The per-hop processing of XIA is more complex than that of IP, which raises obvious concerns about the performance of routers, especially in scenarios using more complex DAGs for addressing. In this subsection, we argue that, despite those concerns, an XIP router can achieve comparable performance to IP routers by taking advantage of well-known optimizations, such as parallel processing and fast-path evaluation.

Packet-level parallelism: By processing multiple packets concurrently, *parallel packet processing* can speed up XIP forwarding. Fortunately for XIP, the packet processing of principals such as AD and HID resembles IP processing in terms of data dependencies; the forwarding path contains no per-packet state changes at all. In addition, the AD and HID lookup tables are the only shared, global data structure, and like IP forwarding tables, their update rate is relatively infrequent (once a second or so). This makes it relatively straightforward to process packets destined for ADs and HIDs in parallel. While SID and CID packet processing may have common data structures shared by pipelines, any data update can be deferred for less synchronization overhead as the processing can be opportunistic and can always fall back into AD and HID packet processing. This makes CID and SID packet processing parallelizable.

Packet-parallel processing may result in out-of-order packet delivery, which disrupts existing congestion control mechanisms in TCP/IP networks (Ludwig and Katz, 2000). One solution is to preserve intra-flow packet ordering by serializing packets from the same flow and executing them in the same pipeline processor, or alternatively by using a reordering buffer (Govind et al., 2007; Wu et al., 2009). An alternative solution is to design to ensure that congestion control and reliability techniques deployed in XIP networks are more tolerant of reordering (Blanton and Allman, 2002).

Intra-packet parallelism: As discussed earlier, a DAG may encode multiple next-hop candidates as a forwarding destination. Since the evaluation of each candidate can be done in parallel, this address structure also enables *intra-packet parallel processing*. While the different next-hops can be evaluated in parallel, the results of these lookups must be combined and only the highest priority next-hop candidate with a successful lookup should be used. Note that this synchronization stage is expensive in software implementations and this type of parallelism is most appropriate

in specialized hardware implementations.

Fast-path evaluation: Finally, the XIP design can use *fast-path* processing to speed commonly observed addresses—either as an optimization to reduce average power consumption, or to construct low cost routers that do not require the robust, worst-case performance of backbone routers. For example, Linux XIA caches routing decisions (i.e. the output port and the new last-node value) for sets of edges pointed by the `LastNode` field. When a packet's destination address matches an entry in the cache, the router simply takes the cached result and skips all other destination lookup processing (Section 3.2). Otherwise, the router pushes the packet into the original slow-path processing path. Since the processing cost of this fast path does not depend on the address used, this performance optimization may also help conceal the impact of packet processing time variability caused by differences in DAG complexity.

Chapter 5 shows that the combination of these optimizations enables XIP routers to forward at speeds comparable to IP routers.

2.4 XIP addresses in action

We now elaborate how the abstractions introduced in previous sections can be put to work to create an XIA network. The following subsections explain how addresses are created and obtained, and show how XIA's architectural components can work together to support rich applications.

2.4.1 Bootstrapping addresses

We assume the existence of autonomous domains and a global inter-domain routing protocol for ADs, e.g., as discussed in (Andersen et al., 2008). We walk through how HIDs, SIDs, and CIDs join a network, and how communication occurs.

Attaching hosts to a network: Each host has a public/private key pair. As a first step, each host listens for a periodic advertisement that its AD sends out. This message contains the public key of the AD, plus possibly "well-known" services that the AD provides such as a name resolution service. Using this information, the host then sends an association packet to the AD, which will be forwarded by the AD routers to a service that can proceed with authentication based on the respective public keys.

Advertising services and content: Applications create XIA sockets through the standard POSIX socket API in Linux XIA. In contrast, XIA's original prototype (XIA Team, 2013), which is implemented in the Click modular router (Kohler et al., 2000), uses a variation of the POSIX socket API. Since the CID principal is only implemented in the XIA prototype and not in Linux XIA, all references below to CID's API refer to the prototype. The prototype's API is described in detail in our technical report (Anand et al., 2011a). We describe here how hosts can advertise services and content using these APIs.

To advertise a service, a process running on a host first calls `bind()` to bind itself to a public key of the service. This binding inserts the SID (the hash of the service's public key) into the host's routing table so that the service is reachable through the host. Likewise, `putContent()` stores the CID (the hash of the content) in the host's routing table. Since at this point, services and content are only locally routable, request packets will have to be scoped using the host's HID, e.g., $\bullet \rightarrow AD_1 \rightarrow HID_1 \rightarrow CID_1$, to reach the intent.

For a service or a content to be reachable more broadly, the routing information must be propagated. For example, an AD can support direct routing to services and content within its domain using an intra-domain routing protocol that propagates the SIDs and CIDs supported by each host or in-network cache to the routers. Global

route propagation can be handled by an inter-domain routing protocol, subject to the AD's policies and business relationships. We leave the exact mechanism, protocol, and policy for principal-specific routing (e.g., content or service routing) as future research.

Obtaining XIP addresses: Now we look into how two parties (hosts, services, or content) can obtain source and destination XIP addresses to communicate. As in today's Internet, obtaining addresses is the application's responsibility. Here, we provide a few example scenarios of how XIP addresses can be created.

Source addresses specify the return address to the specific instance of the principal (i.e., a bound address). Therefore, when a principal generates a packet, the source address is generally of the form $\bullet \rightarrow AD_1 \rightarrow HID_1 \rightarrow XID_1$[3]. The AD-prefix is given by the AD when a host attaches to the network; the HID is known by the host; and the XID is provided by the application, allowing the socket layer to create the source address. XIDs will often be ephemeral SIDs; In this case, an application creates an SID's keys and calls `bind()` to bind the source address before calling `connect()`. The explicit binding ensures that the private key of the SID never leaves the application, and gives applications flexibility to create their source addresses, which could be more sophisticated than the one described here to include, for example, alternative network paths in case the primary path fails. An XIA library available for applications does most of the work applications have to do, so, from the application's point of view, the extra work described above is mostly transparent like TCP/IP's auto-binding of sockets.

Destination addresses can be obtained in many alternative ways. One way is to use a name resolution service to resolve XIP addresses from human readable names.

[3] When communication happens within a domain, the common AD prefix can be omitted from both destination and source addresses.

For example, a lookup of "Google search" in the name resolution service can return a DAG that includes the intent SID along with one or more fallback ADs that host (advertised) instances of the service (similar to today's DNS SRV records). Alternatively, a Web service can embed URLs in its pages that include an intent CID for an image or document, along with the original source ($\bullet \rightarrow AD_1 \rightarrow HID_1$) or content-distribution SIDs as fallbacks. This information can be in the form of a "ready-to-use" DAG, or as separate fields that can be assembled into a destination address (e.g., iterative refinement style) by the client based on local preferences. For example, the client could choose to receive content via a specific CDN based on the network interface it is using.

Note that we intentionally placed the burden of specifying fallbacks to the application layer. This is because a fallback is an authoritative location of an intent that the underlying network may not know about. Name resolution systems and other application-layer systems are more suitable to provide such information in a globally consistent manner. On the other hand, network optimizations can be applied locally in a much more dynamic fashion. Networks may choose to locally optimize for intent by locally replicating the object of intent and dynamically routing the intent.

A final point is that client ADs may have policies for what addresses are allowed. For example, it may want to specify that all packets entering or leaving the AD go through a firewall. This can be achieved by inserting an $HID_{Firewall}$ in the address, e.g., $\bullet \rightarrow AD_1 \rightarrow HID_{Firewall} \rightarrow HID_1 \rightarrow SID_1$, for a source address.

2.4.2 Simple application scenarios

In this section and the next, we use the example of online banking to walk through the lifecycle of an XIA application and its interaction with a client. Our goal is to illustrate how XIA's support for multiple principals as well as its addressing format

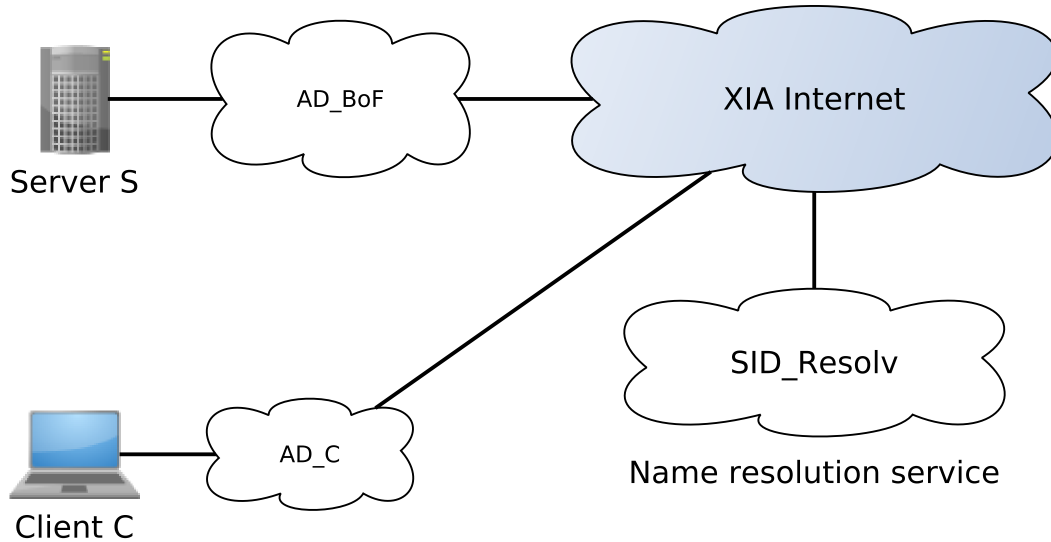


Figure 2·3: Bank of the Future example scenario.

gives significant flexibility and control to applications.

In Figure 2·3, Bank of the Future (BoF) provides a secure online banking service hosted at a large data center on the XIA Internet. The service runs on many BoF servers and has a public key that hashes to SID_{BoF}. We assume that all components in BoF's network natively support service and content principals. We focus on a banking interaction between a particular client host HID_C, and a particular BoF process P_S running on server HID_S.

Publishing the service: When process P_S starts on the server, it binds an SID socket to SID_{BoF} by calling `bind()` with its public/private key pair. This SID binding adds SID_{BoF} to the server S's routing table, and the route to SID_{BoF} is advertised in the BoF network AD_{BoF}. The service also publishes the association between a human readable service name (e.g., "Bank of the Future Online") and $\bullet \rightarrow AD_{BoF} \rightarrow SID_{BoF}$ through a global name resolution service (SID_{Resolv}).

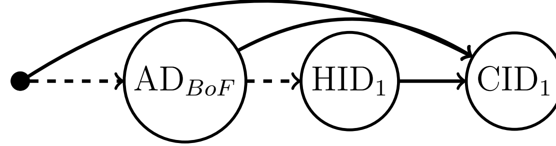
Connection initiation and binding: When a client wants to connect to the service, it first contacts the name resolution service SID_{Resolv} to obtain the service address. It then initiates a connection by sending a packet destined to $\bullet \rightarrow AD_{BoF} \rightarrow SID_{BoF}$ using the socket API. The source address is $\bullet \rightarrow AD_C \rightarrow HID_C \rightarrow SID_C$, where AD_C is the AD of the client, and SID_C is the ephemeral SID. This packet is routed to AD_{BoF} and then to an instance of SID_{BoF}. After the initial exchange, both processes will establish a session, which includes, for example, establishing a symmetric key derived from their public/private key pairs. Because of this session state, the client needs to continue to communicate with the same server, not just any server that supports SID_{BoF}. To ensure this, the client changes the destination address to $\bullet \rightarrow AD_{BoF} \rightarrow HID_S \rightarrow SID_{BoF}$, where HID_S is the server that it is currently talking to.

Content transfer: The client can now download content from the online banking service. For convenience, we assume that the content being transferred is a Web page. Let us consider both static (faq.html) and dynamic content (statement.html), both of which may contain static images. For *static (cacheable) content*, the SID_{BoF} service will provide the client with the CID_{faq} of the static Web page faq.html along with the CIDs of the images contained in it. The client can then issue parallel requests for those content identifiers, e.g., using $\bullet \rightarrow AD_{BoF} \rightarrow CID_{faq}$ as the destination address for the Web page. The request for *dynamic (non-cacheable) content*, e.g., a list of recent bank transactions, is directly sent to SID_{BoF}.

2.4.3 Support for richer scenarios

Using the example above, we now show how XIA's addressing format can support more challenging scenarios such as evolution towards content networking, process migration, and client mobility.

Network evolution for content support: The previous section described how the client can specify static content using scoped intent. Switching to the iterative refinement style (Section 2.3.2):



means that routing through the specified AD_{BoF} or HID_1 is optional, and opens the door for *any set of XIA routers to satisfy the client's request for static content.*

The DAG address format supports incremental deployment of content support in an XIA Internet. As a first step, BoF can deploy support for CIDs internally in its network. Even if no ISPs support CIDs, the above address will allow the delivery of the above request (using the AD_{BoF}) to the BoF network, where the intent CID_1 can be served.

As the next step, some ISPs may incrementally deploy *on-path* caches. The above address allows them to opportunistically serve content, which may allow them to cut costs by reducing their payments to upstream service providers (Agyapong and Sirbu, 2011). Over time, as support for content-centric networking expands, ISPs may make bilateral agreements to enable access to each other's (cached) content. XIA can help leverage such *off-path* caches as well; of course, it would require ISPs to exchange information about cached content and update router forwarding tables appropriately.

Process migration: XIA's addressing can also support seamless process (service) migration through rebinding. Suppose that the server process P_S migrates to another machine, namely HID_T, as part of load balancing or due to a failure; we assume that appropriate OS support for process migration is available. At the start of migration, the route to SID_{BoF} is removed from the old server and added to the new server's routing table. Before migration, the service communication was bound to $\bullet \rightarrow$

$AD_{BoF} \rightarrow HID_S \rightarrow SID_{BoF}$. After migration, the OS notifies the ongoing connections of the new HID, and the binding is changed to $\bullet \rightarrow AD_{BoF} \rightarrow HID_T \rightarrow SID_{BoF}$ at the socket layer. Notification of the binding change propagates to the client via a packet containing the message authentication code (MAC) signed by SID_{BoF} that certifies the binding change. When the client accepts the binding change message, the client updates the socket's destination address. To minimize packet drops, in-flight packets from the client can be forwarded to the new server by putting a redirection entry to SID_{BoF} in the routing table entry of the old server.

Client mobility: The same rebinding mechanism can be used to support client mobility in a way that generalizes approaches such as TCP Migrate (Snoeren and Balakrishnan, 2000). When a client moves and attaches to another AD, AD_{new}, the new source address of the client becomes: $\bullet \rightarrow AD_{new} \rightarrow HID_C \rightarrow SID_C$. When a rebind message arrives at the server, the server updates the binding of the client's address.

Although the locations of both the server and the client have changed in the previous two examples, the two SID end-points did not change. The intrinsic security property remains the same because it relies on the SID. Both the server and the client can verify that they are talking to the services whose public keys hash to SID_{BoF} and SID_C.

2.5 Related work

XIA borrows broadly from purist architectures, discussions on evolving the Internet architecture, and conceptual foundations. The following material highlights how XIA connects to these related works, and how we integrated them into XIA to move toward our goal of finding the next Internet architecture.

Principal blueprints: Substantial prior work has examined the benefits of network architectures tailored for specific principals, that is, purist architectures. We view this work as largely complementary to ours, and we have drawn upon it in the design of individual principals presented in this chapter. The set of relevant architectural work is large and includes proposals for

- Content-centric architectures: NDN (Jacobson et al., 2009), DONA (Koponen et al., 2007), TRIAD (Cheriton and Gritter, 2000);
- Service-centric architectures: SCAFFOLD (Freedman et al., 2010), Serval (Nordström et al., 2012);
- User-centric architectures: MPA (Maniatis et al., 1999), UIA (Ford, 2008);
- Host-centric architecture: IP;
- Multicast architectures: TOMA (Lao et al., 2005), LIPSIN (Jokela et al., 2009);
- Delay-tolerant architecture: DTN (Fall, 2003);

among others. It is worth pointing out that although some works identify themselves as overlays, they are, according to our definitions, network architectures. They define new class of identifiers and specialized forwarding behaviors on those identifiers, and address all questions necessary to make the whole design fully functional (i.e. self-sufficient). We see their choice of shaping their designs into overlays as a pragmatic decision to make their work readily deployable. Thus, including papers more broadly about overlay networks in general, the list above grows even larger to include works such as (Keromytis et al., 2002; Stoica et al., 2002; Subramanian et al., 2004; Dingledine et al., 2004).

Not only do these network architectures offer sources of inspiration to design XIA principals, they are also themselves targets to be ported to XIA. Chapter 4 describes

the port of three purist architectures: IP (Section 4.1), Serval (Section 4.2), and NDN (Section 4.3). The port of purist architectures to XIA substantiates the evolvability claim of XIA as well as provides a level playing field to compare architectures and to specialize their principals.

Evolving the Internet architecture: Besides the research effort on meta architectures, which is covered in Section 1.4 and 4.4, the broad discussion on evolving the Internet architecture has explored network evolution from different angles. For example, Ratnasamy *et al.* propose deployable modifications to IP to enable migration from IPv4 to any other architecture (Ratnasamy et al., 2005). While XIA could be such a candidate architecture, their approach is better suited when there is a consensus on what architecture to adopt next. Others have argued that we should concede that IP (and HTTP) are here to stay, and simply evolve networks atop them (Popa et al., 2010). However, this is not a solution in the long run; EvoArch (Akhshabi and Dovrolis, 2011) points out that merely pushing the narrow waist from layer 3 to layer 5 would result in yet another "ossified" layer.

Self-certifying identifiers were used in many systems (Moskowitz and Nikander, 2006; Mazières et al., 1999). To the best of our knowledge, the first work to establish intrinsic security on network identifiers was the meta architecture ANTS (Wetherall, 1999). Nevertheless, intrinsic security is not pervasive in ANTS; no other class of identifiers in ANTS is defined intrinsically securely. AIP (Andersen et al., 2008) used self-certifying identifiers for both network and host addresses, as XIP does, to simplify network-level security mechanisms. Several content-based networking proposals, such as DONA (Koponen et al., 2007), use them to ensure the authenticity of content. These works demonstrate the substantial power of these intrinsically secure identifiers, which XIA generalizes to an architectural level.

Addressing schemes: The flexibility of DAG addresses has been used elsewhere, notably Slick Packets (Nguyen et al., 2011). Our addressing scheme uses this concept in a new way to provide support for network evolution.

Related concepts: Clark *et al.* present a compelling argument for the need to enable competition at an architectural level (Clark et al., 2002), which is internalized in XIA. We believe that there are substantial benefits to ensuring that applications are free to choose their form of communication yet are still capable of sharing a single Internet instance. Role-Based Architecture (RBA) (Braden et al., 2003) promotes a non-layered design where a role provides a modularized functionality—similar to XIA's principal-specific processing. Although XIA and RBA diverge at the approaches, XIA can be seen as a layerless architecture; Section 6.2 exposes this perspective.

2.6 Conclusions

XIA builds upon the TCP/IP stack's proven ability to accommodate technology evolution at higher and lower network layers by incorporating evolvability directly into the narrow waist of the network. The centerpiece of our design, XIP, is a network-layer substrate that enables network innovation. The principals AD, HID, CID, and SID, which are specifically designed for XIA and introduced in this chapter, and those principals ported to XIA in Chapter 4 demonstrate that XIP supports and amalgamates diverse sets of principals.

Our definition of a meta architecture (Section 1.1) has two requirements, namely, the support of a broad spectrum of principals, and no static dependency among principals. Although this chapter starts to address the first requirement while presenting XIA, static dependency is completely absent in the high-level discussion of this chap-

ter. The reason for this is that the issue of static dependency has only crept in during the implementation of XIA, which is the subject of the next chapter.

Chapter 3

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 3·1 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 largely implemented as a single monolithic module in Linux. In contrast, each block on the XIA side corresponds to a distinct Linux kernel module that we implemented. In this chapter, we focus on the realization of XIP and the implementation of core XIA principals like HID and AD[1]. Later, when we integrate alien designs with XIA, we describe the realization of 4ID to provide interoperability with IP, and describe our implementation of Serval. These appear in Sections 4.1 and 4.2 respectively. Principal XDP (eXpressive Datagram Principal) is not discussed further; it is just a variation of UDP.

The remainder of this chapter is organized as follows: we discuss the impact that

[1] Note that the boxes for these principals are not contiguous to the POSIX Socket API because one cannot create sockets to directly drive them; one can only use these principals to compose XIP addresses.

our experience developing Linux XIA had on our architectural design (Section 3.1), cover the internals of how Linux XIA forwards packets (Section 3.2), and describe how Linux XIA keeps its forwarding cache synchronized with its routing table (Section 3.3).

3.1 Influences from our experience

Designing Linux XIA from scratch gave us many opportunities to explore and to avoid barriers to future evolution of our network stack, a central architectural premise of our work. Working from this first principle led us to a radically different design than that of the native TCP/IP stack. We report on experiences that reinforce the power of modular design, alert us to the dangers of hard-wiring to companion protocols such as ARP and ICMP, and clarify how new principals must themselves be architected for evolvability.

The first influential decision we made during the implementation of Linux XIA was to map each principal to a kernel module. This decision, premised on modular design[2], 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, (2) define a simple path for evolving already-deployed principals, and (3) conceive of routing redirects, a facility for supporting runtime dependencies between principals.

In the influential design of TCP/IP, ARP[3] serves the key role of mapping network

[2]Our original motivation was the fear of losing control of the codebase since it was growing fast, and would need to welcome much more code as other principals joined the stack; according to sloccount (Wheeler, 2004), the kernel part of our codebase alone has 17,669 lines of code. Supporting static independence among principals required considerable extra work, but it gave us a strong grip on the overall codebase.

[3] ARP (Plummer, 1982) was originally designed as an interface between IPv4 and Ethernet, but the protocol has evolved into a companion for IPv4 to interface with other link technologies. This companion status has been endorsed and extended in IPv6 in the form of the Neighbor Discovery protocol (Narten et al., 2007). In spite of the fact that one can redesign IPv4 to externalize ARP,

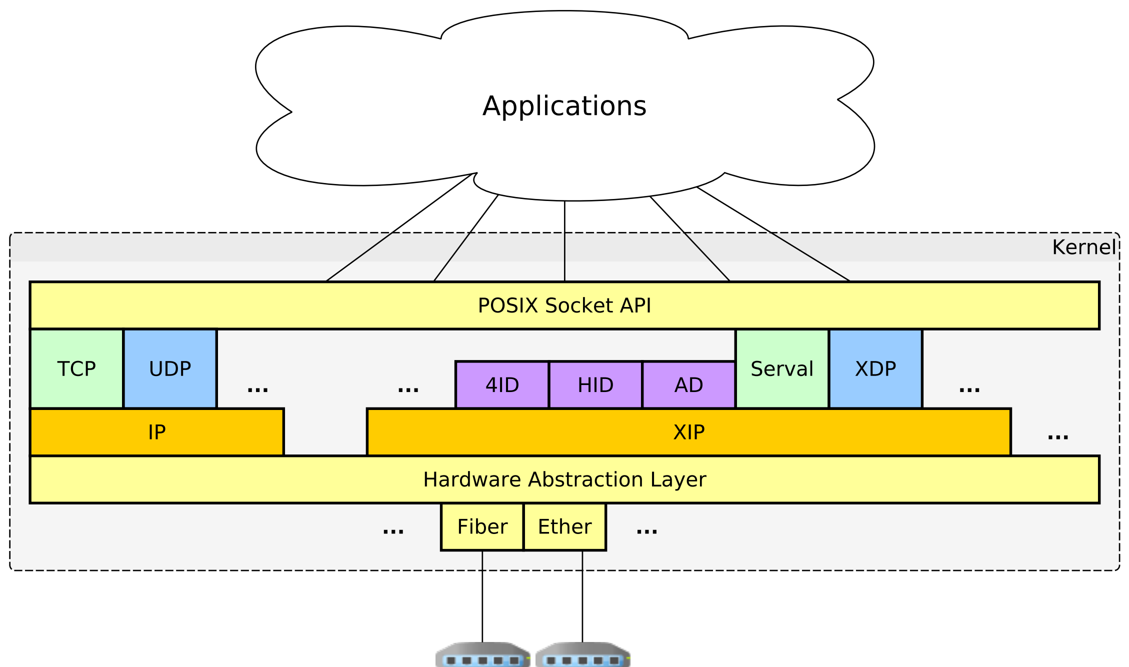


Figure 3·1: Overview of the TCP/IP and XIA stacks in the Linux kernel.

address to hardware addresses, and is a distinct protocol coexisting with IP at the narrow waist of the stack. But as we have seen, such an undesirable static dependency can hinder evolution, and we wondered, does the analogous protocol for XIA necessitate the same design? We found that, in part due to the componentized implementation of Linux XIA, retaining such a dependency (by inertia) is overly restrictive and unnecessary. Indeed, since ARP specifically relates to *hosts*, we identified that the most natural fit was to integrate its XIA replacement with the *HID principal*, and not with XIP itself. To do so, we implemented an analogue of ARP in Linux XIA as part of the HID principal, which we refer to as the Neighborhood Watch Protocol (NWP). Although ARP and NWP serve equivalent purposes, they accomplish them through distinct mechanisms. NWP tracks neighbors and synchronizes state among them like ARP, but also leverages intrinsic security of HID identifiers to avoid problems such as ARP poisoning (a full description of NWP's internals is outside the scope of this dissertation because it digresses from our thesis statement). Ultimately, while NWP and HID must co-evolve, an entanglement which we view as potentially unavoidable, moving ARP functionality out of the network layer frees up additional constraints on XIP. Similarly to the ARP case, we have pushed the analogue of ICMP out of the network layer and into a principal, making XIP truly a standalone protocol.

Having principals as kernel modules helps evolution of already-deployed principals. For example, suppose that a new version of HID principal, namely HIDv2, with an enhanced, but not backwards compatible, version of NWP is released. One could gradually deploy HIDv2 in XIA routers and hosts, make necessary configuration adjustments in the network to move from HIDv1 to HIDv2, and, once HIDv1 is no longer necessary, drop HIDv1 from XIA routers and hosts. In the same vein, but more concretely, we detail a four-step transition plan for XIA to move away from,

ARP effectively acts as an extension of IPv4. Further evidence of this state of affairs is that, even in virtual environments in which the whole network is virtualized and all information that ARP gathers is already available in virtual switches and hypervisors, ARP is still used.

and ultimately off of, IP in Section 4.1.

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. Even an empty set of principals is a valid configuration in Linux XIA, although obviously an impractical one.

Although static dependencies are problematic, *runtime* dependencies among principals emerge as a useful technique to avoid duplicating code. For example, instead of reimplementing much of what the HID principal does in the AD principal, given that the latter also needs to forward packets to neighbors, writers of the AD principal may be tempted to call internal functions of HIDs to deal with it. This in turn, would effectively link AD's implementation to a specific version of HID. Avoiding these undesirable static dependencies among principals required an enhancement of XIA's routing algorithm, namely *routing redirects*, that lets stakeholders express dependencies among principals they select in runtime. Details of this approach, as embodied in Linux XIA's routing mechanism, are described next.

3.2 Forwarding packets

Although an XIA router may have to inspect up to four XIDs, the maximum fanout in an XIP address, to make a routing decision, one does not have to serialize those lookups. This section explains how Linux XIA instantiates XIA's forwarding mechanism without hardware optimizations. Subsequently, in Chapter 5, we present our performance evaluation of this machinery against the Linux IP implementation.

A diagram of the XIA routing algorithm is presented in Figure 3·2. 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

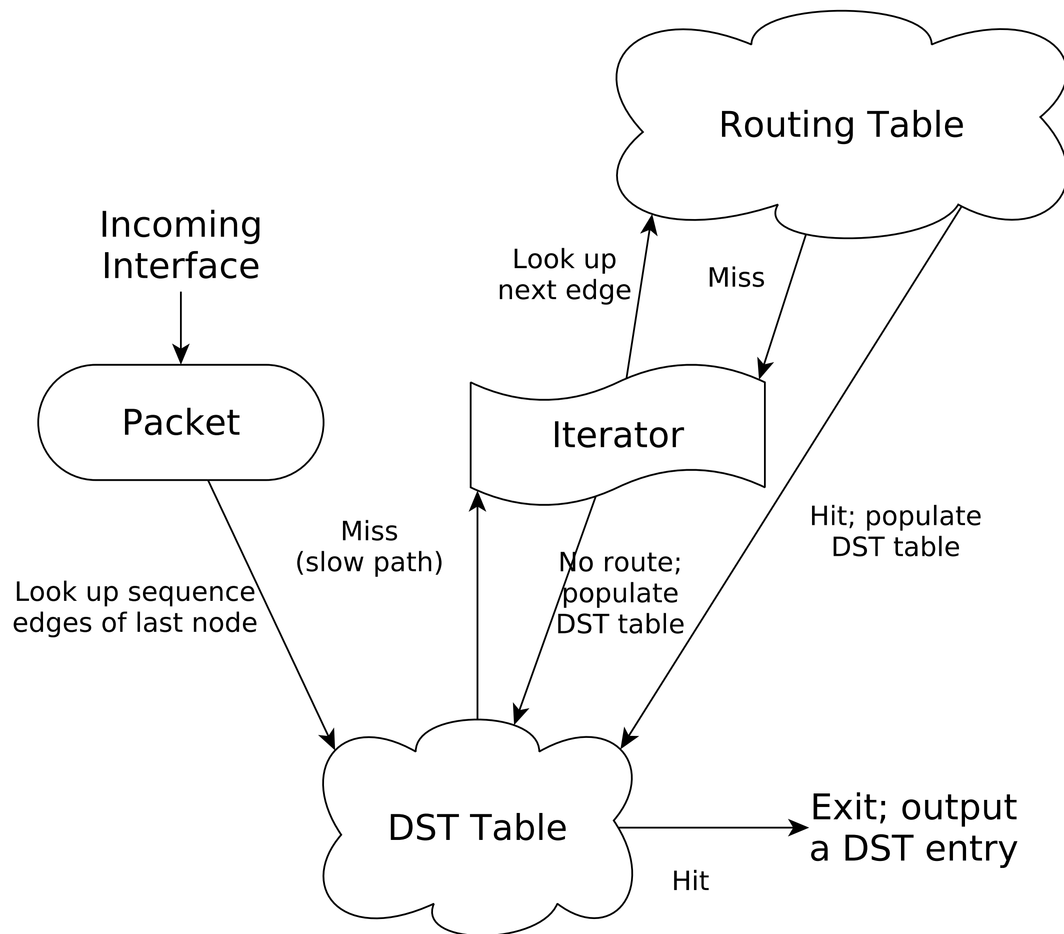


Figure 3·2: Linux XIA's routing algorithm.

edges from that node in the destination address. It then hashes this entire sequence of XIDs to look up in its routing cache, which is called a DST table[4]. If there is a cache hit, the DST table returns a DST entry, a data structure that holds multiple pointers to functions that forward packets with the same signature sequence of edges. The multiple function pointers in a DST entry deal with different moments of the life cycle of a packet.

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. One can do these lookups in parallel, but, without hardware support, a software-only solution cannot take advantage of this opportunity due to the high cost of synchronizing the results. The iterator terminates either with all edges of `LastNode` generating misses in the routing table (unreachable destination), or when one of the edges is hit. In both cases, a new DST entry for this sequence of edges is created, added to the DST table, and returned.

While the machinery described so far is sufficient to implement the XIA routing algorithm as originally specified (Section 2.3.3), does it also achieve our goal of principal independence? As the example below indicates, it does not. Therefore, we introduced the concept of routing redirects. A routing redirect takes place when the iterator in Figure 3·2 looks up an XID and the routing table returns a “miss”, in which case another XID is to be looked up in order to satisfy the first lookup. This simple solution replaces static dependencies among principals with runtime dependencies. Continuing the example of the previous section in which AD was tempted to call HID’s code, instead, the AD principal just redirects its XIDs that need to be forwarded to a neighbor returning HID XIDs to the iterator, which, in turn, will delegate the work to HID principal; Figure 3·3 illustrates some examples of routing

[4]Linux’s DST subsystem is a code infrastructure that binds queues, network devices, and network-layer routing. The name DST is derived from “destination cache”.

redirects. Routing redirects have matured beyond our goal of breaking static dependencies, and have found their way into routing protocols we have been exploring as well as a flexible mechanism to split functionality among principals.

The routing algorithm explained above has two challenges to overcome: (1) keeping the routing cache synchronized to the routing table, and (2) working efficiently under a memory limit at the DST table.

The first challenge deals with the fact the routing table and the DST table are two independent data structures that deal with non-overlapping constraints. For example, the DST table is read and written in atomic context[5], whereas the routing table is written in process context. The challenge is not only limited to an operating system technicality; changes to the routing table can affect the DST table in non-obvious ways. For example, even without accounting for routing redirects, finding the DST entries that become invalid when an XID is added to the routing table may require a full scan of the DST table, since adding an XID to the routing table invalidates all DST entries that prioritize this XID at a higher priority than their currently chosen edge. We provide a solution to this challenge, based on fast data structures for maintaining forests of dependencies, in the next section.

The second 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 are sent to an XIA router each with a unique edge sequence signature. 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.

[5]Execution context that is not associated to a process, so it cannot block. For example, interruptions, bottom halves, and non-preemptable locks.

3.3 Routing dependencies

Only an effective mechanism to keep the routing table and routing cache[6] synchronized realizes the gains that the routing cache brings to Linux XIA's routing algorithm. Our solution consists in incrementally building dependency chains for each edge queried during the routing slow path and *anchoring* these chains on the entries that keep them fresh. Although developed independently, our solution is a variation of the Data Update Propagation algorithm originally devised for keeping caches of dynamic web pages consistent with underlying data (Challenger et al., 1999). First, we argue that naive solutions such as flushing the routing cache whenever the routing table is updated, or periodically scanning the whole routing cache to identify stale entries are not effective. Nevertheless, the shortcomings of naive solutions are instructive for introducing the domain of the problem. Second, we present our solution, and conclude with a root classification that helps reasoning about the data structure.

Flushing the DST table every time the routing table is updated causes routing hiccups; suddenly the DST table is empty and all packets have to be analyzed on the slow path, one edge at a time. XIA principals that have their XIDs associated with sockets or other userland objects (e.g. Serval services) may see an update rate of their part of the routing table high enough to render the DST table a useless burden.

Scanning the DST table for XIDs to be added to or removed from the routing table, to avoid flushing all DST entries, is demanding. Unless the DST table takes a small amount of memory, scanning the DST table puts pressure on a machine's memory cache subsystem due to the rotation of cached memory pages before reusing them. Moreover, unless the number of stale DST entries is a large proportion of the entries in the table, the scan itself is inherently inefficient. On top of that, the freshness test of a DST entry after the addition or removal of a given XID to the

[6]Just another designation for the DST table.

routing table is not a simple string comparison between the edges that the DST entry represents and this XID. For example, consider a principal that supports a default route when all its other routes are not applicable (e.g. principal AD); the default route does not have an XID. A special XID representing the default route does not help because it will not be seen in the edges of a DST entry. Moreover, the cost of the freshness test of DST entries grows rather quickly in the presence of routing redirects.

The problem of the first naive solution is the lack of any identification of the entries that have to be flushed, whereas the second naive solution pays a high cost to identify these entries. Our solution addresses both problems at once leveraging the routing slow path to incrementally build dependency chains for each edge investigated. These chains enumerate everything that keeps a DST entry fresh. Given that these chains eventually merge with each other, one should talk about a dependency forest—every node but the leaves of this forest are called anchors. When an anchor goes away, all of its dependencies, or equivalently, the entire subtree of the forest rooted at this anchor, must be flushed. Figure 3·3 illustrates a small dependency forest and its related routing table and DST entries.

While reasoning about and implementing routing dependencies, we have found it useful to classify root anchors as positive, or negative. Positive root anchors are existing entries in the routing table, whereas negative root anchors[7] are nonexisting entries. Nonexisting entries may come from known or unknown principals; a principal becoming known when the corresponding kernel module for that principal is loaded into memory. Each edge of a DST entry is classified and signed according to the sign of the root anchor it reaches. Combining these classifications and the routing slow path, one derives simple properties that help reasoning about the whole routing mechanism such as:

[7] This expression was inspired by negative cache, a cache of negative responses, i.e. failures.

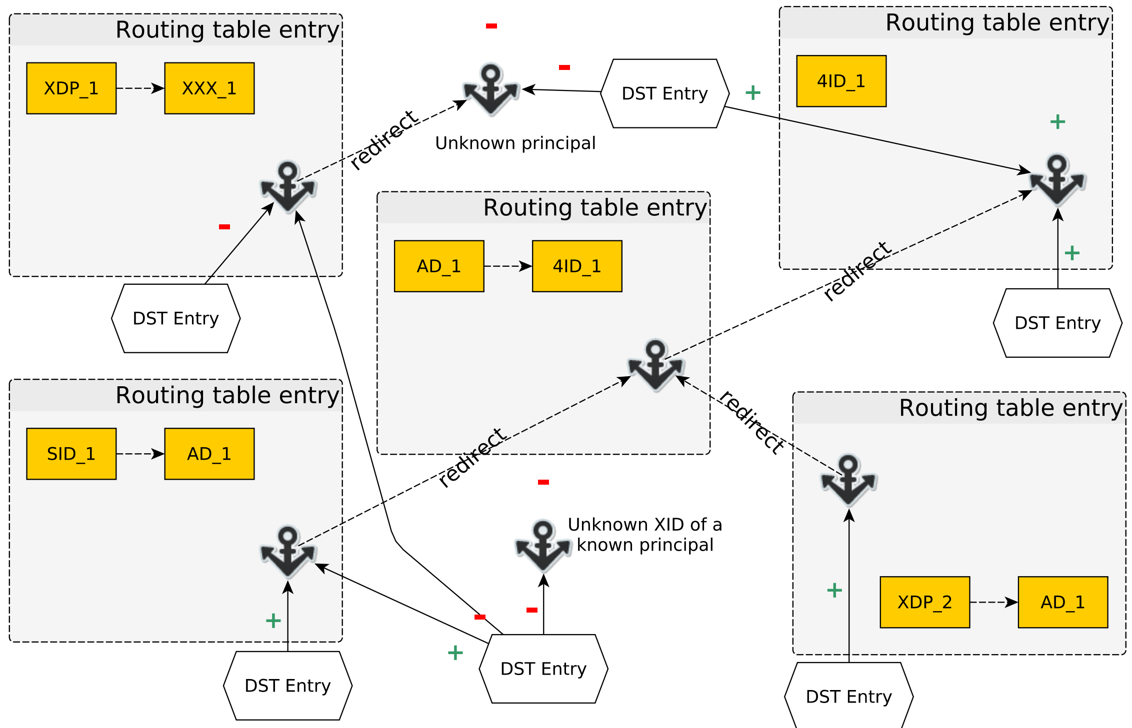


Figure 3·3: Example of routing-dependency linkage between routing table entries and DST entries in Linux XIA. DST dependencies (signed edges) form a rooted forest anchored at routing table entries (positive anchors) and non-entries (negative anchors). Only root anchors' and DST entries' edges are classified as positive, or negative.

- DST entries have either zero or one single positive edge; otherwise there would be more than one chosen edge.
- DST entries have from zero to four negative anchors, as four is the maximum fanout of a node.
- Negative edges are always evaluated before a positive edge if the latter exists.

Chapter 4

Porting alien designs to XIA

This chapter explains how four distinct network designs never intended for compatibility with XIA can be adapted to work and coexist in XIA. It 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 questions drive most of the work: how can Y's visible identifiers be mapped to XIDs, and how can Y be broken into 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 the best one. In fact, 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 in any way. Since many interpretations of a single design can naturally coexist in Linux XIA, value judgments are ultimately left to the stakeholders of 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. The first and second porting questions

respectively derive from our principal and architecture definitions introduced in Section 1.1.

Section 4.1 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, Section 4.2 describes our port of Serval (Nordström et al., 2012), a service-centric architecture. Section 4.3 presents a whitepaper design of NDN (Jacobson et al., 2009), a content-centric architecture, for XIA. Finally, Section 4.4 maps ANTS (Wetherall, 1999), an active meta architecture, onto XIA.

4.1 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 (IP) 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 motivation drives the design of the U4ID principal. U4ID XIDs' names[1] 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

[1] Recall from Section 1.3 that an XID is the pairing of a principal type and a name.

XIA stack of the destination host must have principal U4ID running and listening at the UDP port number 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. datacenters, campuses, corporations) may prefer to trade in some compatibility for performance. Principal I4ID 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. I4ID XIDs' names are the destination IP addresses followed by 16 zeros.

Principals U4ID and I4ID build on-the-fly tunnels through IP networks. Although one needs these tunnels for retroactive compatibility with IP, these tunnels 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 deployment of XIA, where 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, principal X4ID leverages IP's 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 forwards X4IDs by directly looking up the corresponding IP address in the routing table of TCP/IP stack. Principal X4ID is the tightest integration between TCP/IP and XIA stacks, and can bridge the deployment of XIA at Internet scale because it leverages the current BGP sessions to globally forward XIP packets.

To go further, we recommend the introduction of a general-purpose principal, which we will also use in Serval. The motivation is that, in contrast to opaque

cryptographic identifiers typically associated with XIA, identifiers of IP (and Serval) have a hierarchical structure, and, in particular, are designed to support longest prefix matching. To support this functionality in XIA we designed a simple LPM principal to support longest prefix matching on the underlying identifiers. With this, an LPM XID is a typed string of bits that XIA routers match against a prefix tree at each hop, analogous to CIDR.

Returning to 4IDs, while principal X4ID is a perfect fit for dual stack hosts, it nevertheless perpetuates an undesirable dependency on IP's routing tables for XIA-only hosts that simply want to tap into the BGP sessions. These stakeholders can drop the dependence on IP's routing tables by using the LPM principal and directly populating the LPM forwarding table with routes from BGP sessions, achieving the same behavior as principal X4ID, but without the need for a 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, principal U4ID in Linux XIA. Mukerjee *et al.* (Mukerjee et al., 2013) have 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.

4.2 (In-depth) case study #2: Serval

Serval, a service-centric architecture, promotes services as first-class entities, as described in detail in prior work (Nordström et al., 2012; Arye et al., 2012). 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, ServalID[2] 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. ServalIDs 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. ServalIDs have a hierarchical meaning in Serval, and thus they are routed using longest prefix matching, like IP addresses. Unlike ServalIDs, 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 use the 5-tuple (protocol, source IP address, source port

[2]The original Serval paper refers to this as a ServiceID, but we renamed it to avoid ambiguity with XIA's ServiceID.

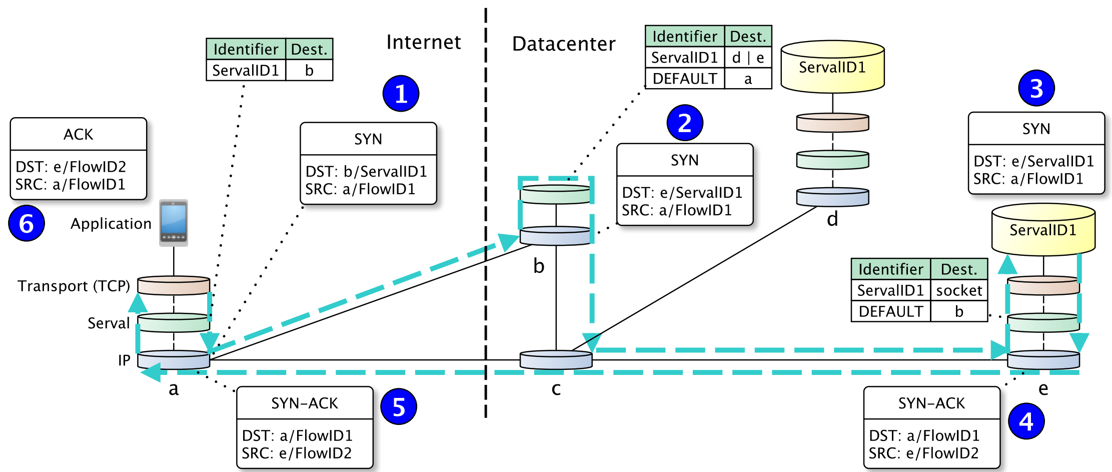


Figure 4·1: Example of Serval's three-way connection handshake between a client on host a and a service instance on e.

number, destination IP address, destination port number) to multiplex connections.

Serval's connection handshake closely resembles TCP's three-way handshake. Figure 4·1 summarizes the process for an application connecting to service $ServalID_1$, which has two instances available in the datacenter at right. Machines a, b, d, e are both Serval and IP enabled, while machine c is a legacy IP router. The handshake starts with a SYN packet destined to $ServalID_1$. This packet also carries a proposed FlowID for the connection from a as the SAL source address. In Step 1, Serval looks up $ServalID_1$ in its service table, resolves $ServalID_1$ to host b, and rewrites the IP and SAL headers accordingly before forwarding. Once the packet arrives at Service Router b (Step 2), b has a forwarding decision to make, and, for example, can route $ServalID_1$ either to host d or e depending on its local load-balancing policy. Here, b rewrites the SAL layer header and the IP header with the chosen destination, e. When the SYN packet reaches host e, Serval resolves $ServalID_1$ to the listening socket (Step 3), which replies with a SYN-ACK packet (Step 4). This SYN-ACK acknowledges a's FlowID, and establishes a FlowID from e. On the return path of

the SYN-ACK (Step 5), the ACK completing the three-way handshake (Step 6), and all subsequent packets for this flow, are forwarded using legacy IP.

We briefly remark on security considerations in Serval. In the original design, ServalIDs lack intrinsic security, that is, connection endpoints cannot verify each other's identity without a third party. To improve security, the Serval design 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.

4.2.1 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 ServalID 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 authors chose to make ServalID 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 ServalIDs, naming services and facilitating routing, makes ServalIDs intrinsically secure in XIA. Thus, in our interpretation of Serval, ServalIDs are the hash of their public key, and scoping is left to other principals to deal with. 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 this field as used in the original Serval and thus, when in use, it is specified as the primary intent in a destination address, but is *always* scoped to a given host. Conceptually, we can think of FlowIDs as a companion principal to

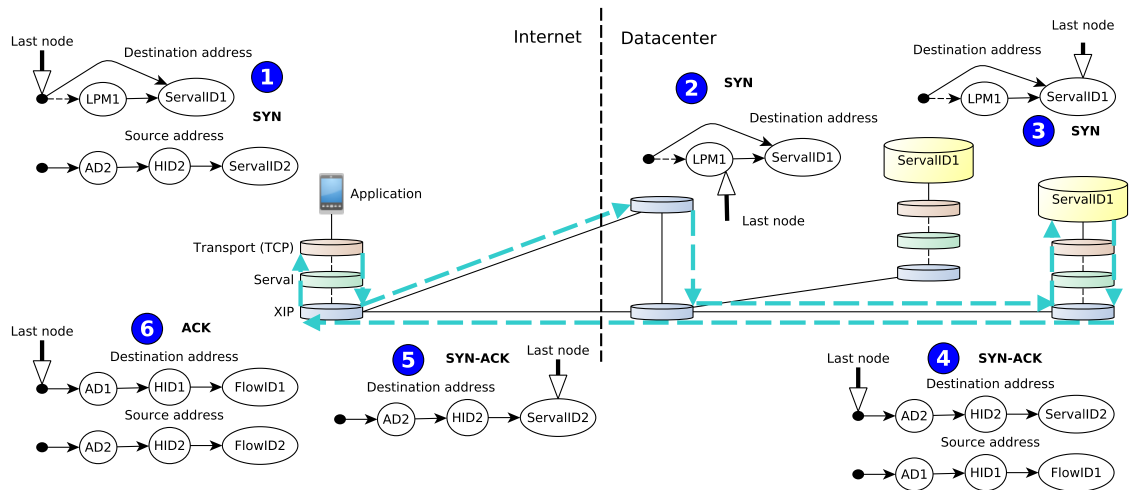
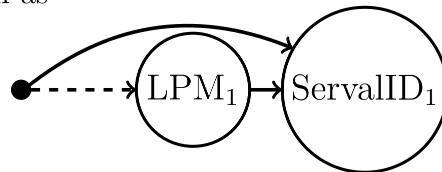


Figure 4·2: Example of XIA Serval's three-way connection handshake between a client and a service instance.

ServalIDs, which together comprise a composite XIA principal which we jointly refer to as the Serval principal.

Finally, note that 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 preserved ServalIDs' hierarchical behavior. But a more modular solution, more in tune with the question how to break Serval into principals, suggests the use of LPM, described in the context of IP, to do the longest prefix matching on its identifiers. Using LPM, Serval's original behavior is fully emulated with addresses such as



We can now quickly review connection establishment in the context of XIA Serval. Figure 4·2 diagrams the packet sequencing, analogous to that depicted in Figure 4·1. The packet sequencing is identical, with the main distinction being provided in the

address formats. In packets where ServalIDs were used as destination addresses in the SAL layer (packets 1-5), those are mapped to XIA destination addresses with XIA ServalIDs as the sink (optionally falling back to LPM XIDs). In packets where FlowIDs were used as the destination (packet 6), those are mapped to XIA destination addresses with XIA FlowIDs as the sink, but always scoped to HIDs.[3] Similarly, when FlowIDs were used as the source address (all packets), those are also mapped to host-scoped FlowIDs. The exception is packet 1. Here, the Serval implementation uses a FlowID as the source, but we require instantiation of a ServalID at both endpoints, and obtain the proposed FlowID from the ACK packet. We find that this small change simplifies connection management, such as migrating a connection endpoint, or adding another path to the connection. More critically, it preserves end-to-end security guarantees, as ServalIDs have a cryptographic meaning, whereas FlowIDs do not. Finally, all subsequent traffic would use $FlowID_1$ and $FlowID_2$ as sink addresses, bypassing the need for today's on-path load balancers to do packet-by-packet preservation of IP-based mappings.

4.2.2 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 layer addresses. Thus, while original Serval can only make forwarding decisions when IP gives it a chance, XIA Serval enables every router to make decisions purely based on network layer information. As a result, placing ServalIDs as the primary intent of connection establishment

[3] In general, the preferred scope of FlowIDs could include both an HID and an AD; this information is available at the host and is either derived from routing daemons, or manually entered by network administrators.

enables routers to seamlessly realize service-layer anycast. This possibility of pushing the instance choice down a path retains Serval's architectural goal of "late binding", in contrast with the "early binding" employed today, e.g., using DNS. Perhaps 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. This is exemplified in the source address used in SYN packets: whereas Serval can take an easy shortcut of inserting a FlowID-based address, the XIA approach enforces the use of a secure ServalID-based address to initiate the communication. Note that this bootstrapping procedure could also be used to harden all subsequent transmissions with cryptographic signing, including the FlowIDs themselves.

As a final remark, we have sucessfully ported Serval onto Linux XIA as described in this section. Serval's codebase is publically available on GitHub (Princeton SNS Group, 2012).

4.3 Case study #3: NDN

NDN (Jacobson et al., 2009), a content-centric architecture, embodies a design that is difficult to instantiate on meta architectures without losing some of its features. Although prior work has sketched how to port generic information-centric networking (ICN) concepts onto translating meta architectures (see, e.g. (Raghavan et al., 2012b)), porting specific features of NDN appears more challenging. These include retaining the anonymity implications associated with the absence of source addresses, deduplicating interest and data packets across regions, and realizing scalable multicast. These features all rely on the Pending Interest Table, a core component of NDN. Leaving out those features implies not supporting protocols that leverage NDN internals, such as ChronoSync (Zhu and Afanasyev, 2013), a distributed pro-

tocol to synchronize dataset state among multiple parties in NDN. We now present two whitepaper designs of how to port NDN to XIA. The first alternative faithfully retains all of NDN's features, whereas the second forgoes support for dynamic content but eliminates the need for public-key management.

NDN abstracts a network as an infrastructure that stores and retrieves content as named data within a hierarchical name system. The name structure is analogous to the one found in a POSIX file system, that is, a sequence of independent names that describe a path from the root of the hierarchical tree toward one of the leaves. Communication proceeds as follows: an application sends an interest packet into the network, each NDN router does a longest-prefix lookup on the name of the content, and performs an action based on the matched rule. These actions take care of forwarding interest packets, replying to interest packets with data packets storing content, and tracking the inbound interface(s) on which each interest packet arrives in order to forward back data packets.

The first challenge in porting NDN to XIA is to find a suitable XID format. The naive approach of embedding NDN names into an XID hits the issue that NDN names can be potentially larger than 20 bytes, the space reserved for an XID identifier. To the best of our knowledge, the NDN project has not committed to a limit for the length of its identifiers, but some limit is necessary; otherwise, a frame with an NDN packet may not have space to accommodate the name of the content that it is interested in (interest packets) or that it carries (data packets). We consider two possibilities for NDN XIDs: (1) the hash of the content name, and (2) the hash of the content.

Having the hash of content names as XIDs closely approximates original NDN. The NDN header, which comes above the XIP header, carries the full name of the content. Thus, XIP caches routing decisions for NDN XIDs, and, whenever a non-cached XID is requested, XIP transfers control to the NDN principal that, then,

looks into the NDN header, runs the NDN routing algorithm *as is*, and returns its routing decision to XIP. This solution reuses virtually every design made for NDN with minimal changes.

Interestingly, the lack of source addresses in NDN interacts with other principals in a non-straightforward way, raising interoperability challenges. An interest packet destined to $\bullet \rightarrow NDN_1$ would work as expected, that is, it queries the network for content NDN_1, and has no source address. The difficulty comes into play with addresses that scope an NDN XID such as $\bullet \rightarrow AD_1 \rightarrow NDN_1$[4]. This address forces an interest packet to be forwarded to a router in the autonomous domain AD_1, and, from there, follows NDN_1. When the data packet flows back on the return path, it will stop at that router in AD_1, not at the anonymous source of the interest packet. One solution is to add a suitable source address to those interest packets, but this spotlights that having the hash of content names as NDN XIDs leads to a different way of thinking about source addresses in XIA. As examples in Section 4.2.1 expose, source addresses in XIA are almost independent of destination addresses. The different NDN source addresses stem from names being used to name content and route interest and data packets.

We now consider the second alternative for porting NDN to XIA, namely, having the hash of a content as its XID. Dynamic content is no longer possible in this alternative because the hash of the content would have to be known prior to sending an interest packet. Nevertheless, this alternative eliminates the need for public-key management to secure content in the original NDN approach (Smetters and Jacobson, 2009), because now identifiers are intrinsically secure. Moreover, since this alternative can rely on other principals to route interest and data packets, its source addresses are not different from other source addresses used in XIA. This second alternative

[4]Such an address has no analogue in NDN, but is admissible in XIA. One relevant use case for scoped NDN XIDs is in the context of network troubleshooting.

for porting NDN to XIA is comparable to the CID principal in XIA (Chapter 2). Considering that a principal such as Serval can take on the responsibility of handling dynamic content, the CID principal offers a pathway to break down NDN into many principals, and exposes the tradeoff between supporting dynamic content and public-key management in NDN.

Porting NDN to XIA enables the community to explore the design space of content-centric architectures, which improves our understanding of the tradeoffs, draws value to each alternative, and may lead to new designs. In addition, this port frees NDN to focus on the features that bring the most value to its design. For example, NDN does not export location abstractions to its applications, which in turn forces NDN to accommodate special-purpose solutions to enable network operators to troubleshoot NDN networks. In XIA, NDN can delegate this chore to appropriate principals.

4.4 Case study #4: ANTS

Given that ANTS is a meta network architecture, one can expect that ANTS and XIA share some common ground, yet, as we show in the following subsection, their relationship goes deeper. This relationship encompasses motivation, some solutions, and a formal, surprising relation between their set of supported principals. Using XIA's lexicon, this section presents ANTS (Wetherall, 1999), shows a whitepaper design of how to port ANTS to XIA, and points out that the reverse is not possible, which, in turn, demonstrates that XIA supports a superset of the principals that ANTS can support.

ANTS and XIA share motivation and solutions, but they have addressed different challenges. The motivation for ANTS, lowering the barriers to evolve the network layer, is just another way to motivate XIA. Solutions such as intrinsic security for

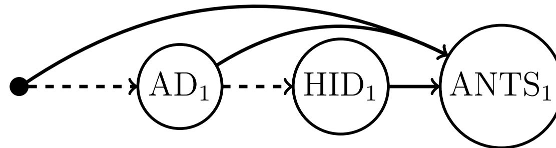
identifiers, a fallback behavior on routers that do not support ANTS, and a principal definition close to ours emphasize the similarity of the problem space of both meta architectures. The addressed challenges have been different due to a key goal of ANTS: supporting mobile code to define new principals, which required a significant amount of effort to design a code distribution protocol, define a suitable running environment for this mobile code, and deal with security issues related to mobile code.

Our port of ANTS sidesteps security issues related to code mobility as well as code distribution to focus on the fundamental porting questions. This is not to say that an XIA version of ANTS can do without these two items, but rather that we do not attempt to improve on the ANTS solution. However, we depart from ANTS' use of Java in favor of Lua (Ierusalimschy et al., 2013) for mobile code. Our rationale is that while ANTS was originally designed to run in user space, the XIA realization of ANTS runs in kernel space, and a lightweight, easily-embedded language simplifies this move.

Different from our previous case studies, the choice of XID format for ANTS is a natural one. ANTS packets, called *capsules*, have a type field that describes how they are forwarded. In ANTS, the type field is a cryptographic hash of the forwarding code that should be applied to packets of this type. Specifically, the hash is a hash of the nodes on the root-to-leaf path of the Merkle tree whose leaves are the blocks of code that define the behavior of an ANTS principal. ANTS uses this Merkle tree to verify that the mobile code that routers exchange among themselves is the one to which the type field maps. Merkle trees are especially convenient here because they enable routers to transfer and verify only the code blocks needed to forward the capsules in hand. In XIA, this type field is ideally suited to be the XID of the ANTS principal since it is already intrinsically secure. In fact, ANTS is, to the best of our knowledge,

the first work to use intrinsic security on network identifiers. Nevertheless, intrinsic security is not pervasive in ANTS; no other class of identifiers in ANTS is defined intrinsically securely.

When a capsule reaches a router that supports ANTS, the router calls the forwarding routine associated with the ANTS XID to make the routing decision. If the routine associated is not available locally, the processing of the capsule is postponed until the necessary mobile code is obtained through the code distribution protocol. If the router does not support ANTS, the router forwards the capsule as a regular IP packet since ANTS stacks its headers above the IP header. The XIA version of ANTS naturally relies on fallback edges to deal with routers that do not support ANTS principals, while giving ANTS highest priority, as in the following XIA address.



Although the address above matches the description of the iterative refinement pattern defined in Section 2.3.2, the semantics here are not necessarily the same; they depend upon the decisions of the corresponding ANTS forwarding code. The following example illustrates the difference: a capsule arrives at a router that supports ANTS and XIP's `LastNode` field points to AD_1. In this case, $ANTS_1$ will be executed, but might choose to inform XIP that its edge cannot be followed, which would force XIP to follow the edge to HID_1. In contrast, the iterative refinement pattern prescribes that XIP would always follow the edge to $ANTS_1$ since $ANTS_1$ is locally available.

ANTS fits well in XIA as a single principal type. Due to its reliance on IP to forward packets as an incremental deployment strategy, ANTS is not a network architecture without IP. The XIA version of ANTS removes this dependency by delegating the scoping of ANTS XIDs to XIA principals such as AD, HID, and 4IDs. At the same time, ANTS both supports a broad spectrum of factors and does not impose

any static dependencies among its supported factors, so ANTS is a meta architecture according to our classification.

We now show that XIA's set of principals is a superset of ANTS. While one can port ANTS to XIA, as shown above, the reverse is not possible due to limitations of the mobile code runtime environment of ANTS. Specifically, ANTS' runtime environment lacks the necessary API calls to implement a reliable transport (Wetherall, 1999, Section 5.4.2); The designers of ANTS deliberately kept the runtime environment to a minimum to cope with security, performance, and technical constraints. In contrast, the port of Serval demonstrates that XIA does not have this limitation.

Among all meta network architectures we have investigated, ANTS is the one most aligned to XIA. We speculate that had ANTS not pursued mobile code as a central goal, it would even more closely resemble XIA. While deploying the XIA version of ANTS in the open Internet is not advisable due to the lack of solutions to curb security issues related to mobile code, we feel that the ANTS principal could well be a useful development tool in controlled environments.

Chapter 5

Evaluation

A burning question for a new Internet architecture, never mind for a meta one, is if it can deliver. This is a complex question that encompasses at least (1) a reason to deploy the new architecture, (2) a deployment plan, (3) enough features to support legacy applications, and (4) good forwarding performance. Failing on any of these items renders a negative answer. This dissertation answers these items for XIA as follows:

1. The reason to deploy XIA is its evolvability. Nobody can predict future network demands, so the Internet should not favor one architecture over another.

2. Our deployment plan is to run natively wherever possible and to interconnect XIA networks through the IP Internet with the help of 4ID principals (Section 4.1) for as long as necessary, or indefinitely.

3. Roughly speaking, all IP applications run over TCP and UDP. These protocols are available in XIA (Figure 3·1), and Chapter 4 shows that there is no reason to doubt that XIA is capable of embracing any other necessary protocol to satisfy legacy applications.

4. Packet-forwarding performance is a key performance limit to any packet-switching network such as IP and XIA. This chapter benchmarks XIA against IP to assess XIA's performance.

Our forwarding performance evaluation consists of simulating an environment close to that seen by a core router, and measuring the impact of 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. Section 5.1 describes our testbed, and Section 5.2 presents the results.

5.1 The testbed

This section covers three aspects of our experiments necessary to the understanding of the results presented in the next section: (1) how we simulated the conditions a core router sees, (2) the software and hardware infrastructure we used, and (3) the conditions under which we took the measurements.

A core router spends most of its time receiving packets from the links connected to it, routing these packets to their output links, and then forwarding the packets. A core router does have other activities such as maintaining its routing table, enforcing network policies, and collecting statistics, but these are supporting activities; they are not meant to take a significant portion of the computational resources. We abstract the other side of the ports of a core router to a packet writer (PW) and a counter of how many routed packets come back. Figure 5·1 depicts this simple topology.

The single field of network headers with highest impact on how packets are routed is the destination address. Some routing features, however, modify how packets are forwarded based on fields other than the destination address, such as reverse-path filtering[1], quality of service, and multipath routing. We focus on the cost of routing itself, therefore our experiments simulate a vanilla router.

All PWs choose the destination addresses of IP packets according to a Zipf distri-

[1] Reverse-path filters check source addresses to enforce that packets come on the same port routers would choose to forward packets to that destination.

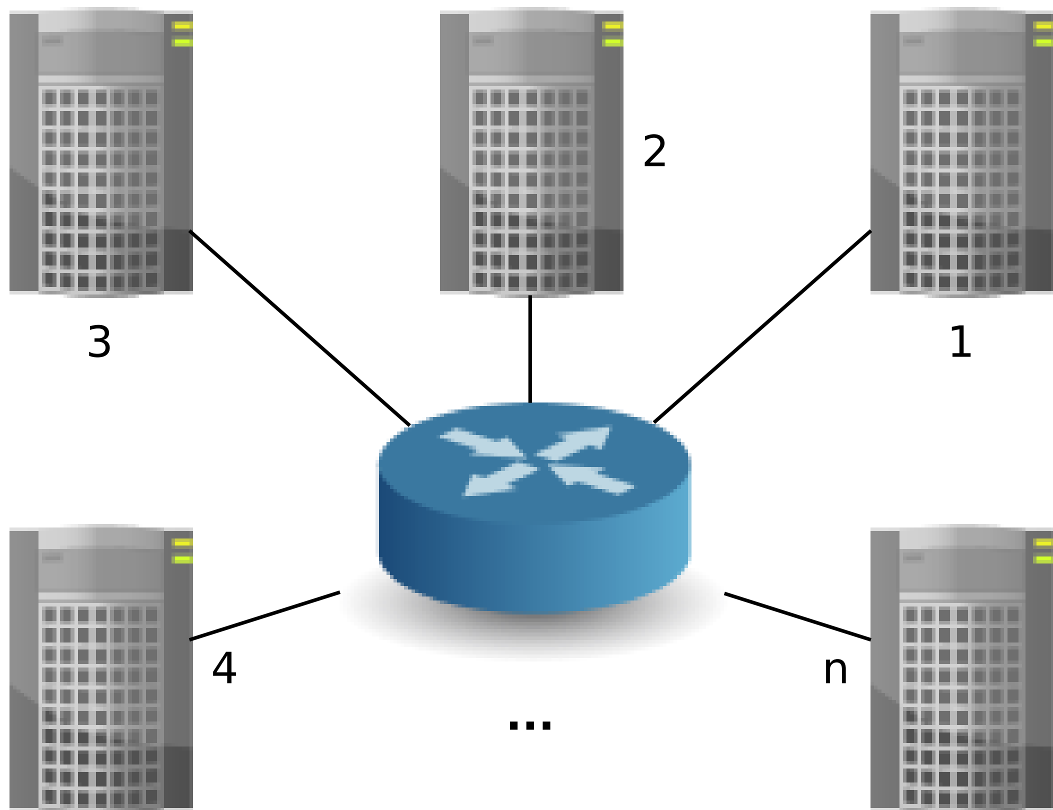


Figure 5·1: Topology used in our forwarding performance evaluation. The environment of our simulated core router is abstracted to packet writers and counters of successfully routed packets.

bution over all possible destinations, to account for the popularity of the destinations. We take as all possible destinations the 462,150 CIDR blocks obtained from Route Views Project (University of Oregon, 2013). At the beginning of each IP experiment, we populate the routing table with these CIDR blocks, choosing the output port uniformly at random among the ports available in the experiment, and randomly choosing the rank of the CIDR blocks, which is shared among all PWs. We leave both the number of ports and the exponent of the Zipf distribution as parameters of the experiments.

All XIA experiments only use addresses with AD XIDs. We could have mixed different principals' XIDs in the addresses, including those of Chapter 4, but core routers are most likely to forward on ADs; at least as long as transit-network operators do not find it valuable to deploy other principals in their networks. Moreover, as the principals described in Chapters 2 and 4 suggest, most principals' XIDs are going to be flat identifiers, like ADs. Therefore, the cost of looking up an AD is representative of the cost of looking up other principals' XIDs. We pessimistically map each CIDR block to a distinct AD XID to produce a worst-case AD routing table and XIP addresses.

Our experiments use different XIP address formats (Figure 5·2) to assess the overhead of processing multiple edges. Address formats FB0, FB1, FB2, and FB3 focus on the cost of having multiple choices, whereas format VIA focuses on the processing-intensive case in which a node of the address is reached. In the specific case of the VIA format, AD node AD_0 is reached when the corresponding packet arrives at a border router of the autonomous domain that AD_0 represents. When a node is reached, XIA routers must first update the `LastNode` field of XIP header to point to the next node in the address before forwarding, and forward accordingly to the new last node.

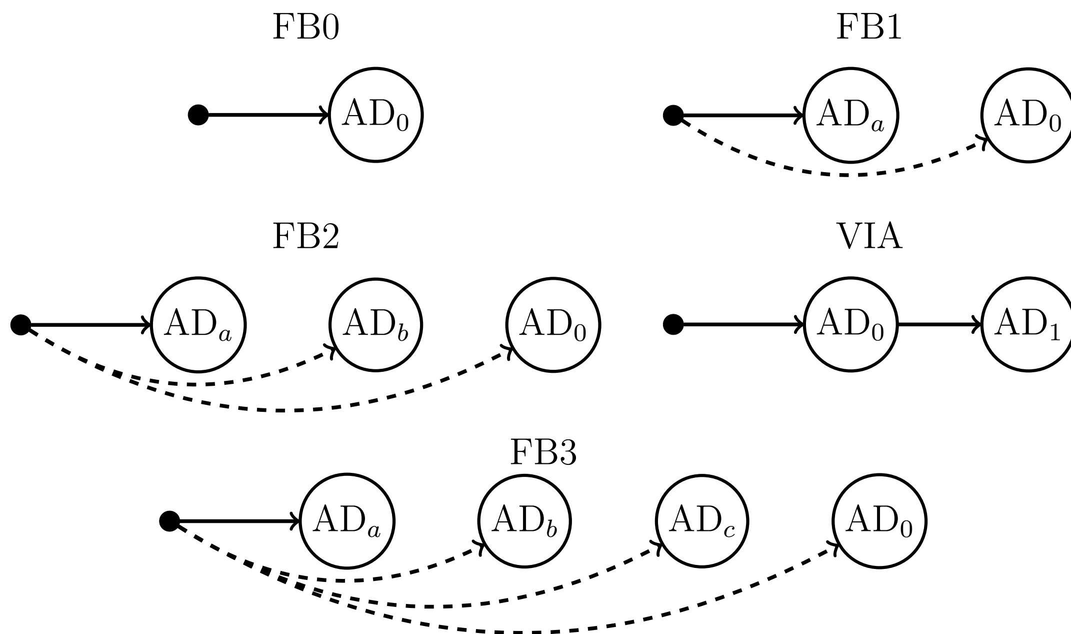


Figure 5·2: Address formats used in the evaluation of forwading performance of Linux XIA. AD_a, AD_b, and AD_c are unknown ADs that force XIP routers to skip their edges, whereas AD_0 and AD_1 are known ADs.

All experiments run in a single machine with the router and PWs isolated by Linux Containers (LXC) (Linux Community, 2013), a lightweight virtualization technology. LXC has been used by others to simplify experiments and make them reproducible (Lantz et al., 2010; Handigol et al., 2012; Cabral et al., 2013), and has helped us to focus on the cost of routing instead of dealing with distracting I/O overload and hardware features that only favor IP. Although LXC has a number of resource constrainers, we have not throttled any resource in our experiments. In special, bandwidth on the virtual links that connect the router and its ports is only bounded by the speed that the hardware can run the kernel code.

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 kernel, which is a fork of Linux 3.11.0-rc7. Our kernel (Machado, 2013a) and our evaluation code (Machado, 2013b) are publicly available on GitHub.

Our evaluation metric is the number of succesfully forwarded packets. All evaluation graphics adopt the unit packets per second instead of throughput or goodput units, such as bytes per second, to abstract the fact that TCP/IP and XIA's headers have different lengths. The forwarded packets are counted with the help of ebtables (Schuymer, 2011) rules that work at the link level and has low measuring overhead.

Our evaluation server can handle IP experiments with up to 10 ports without hardware saturation, and up to 6 ports for XIA experiments (Figure 5·3). Nevertheless, Linux XIA does not scale the number of ports as well as Linux IP does. As a compromise, we chose to run all other experiments with 4 ports to avoid hardware saturation for both stacks, have enough room for the extra load that experiments in Figure 5·5 and 5·6 require, and avoid pushing apart IP and XIA's numbers without

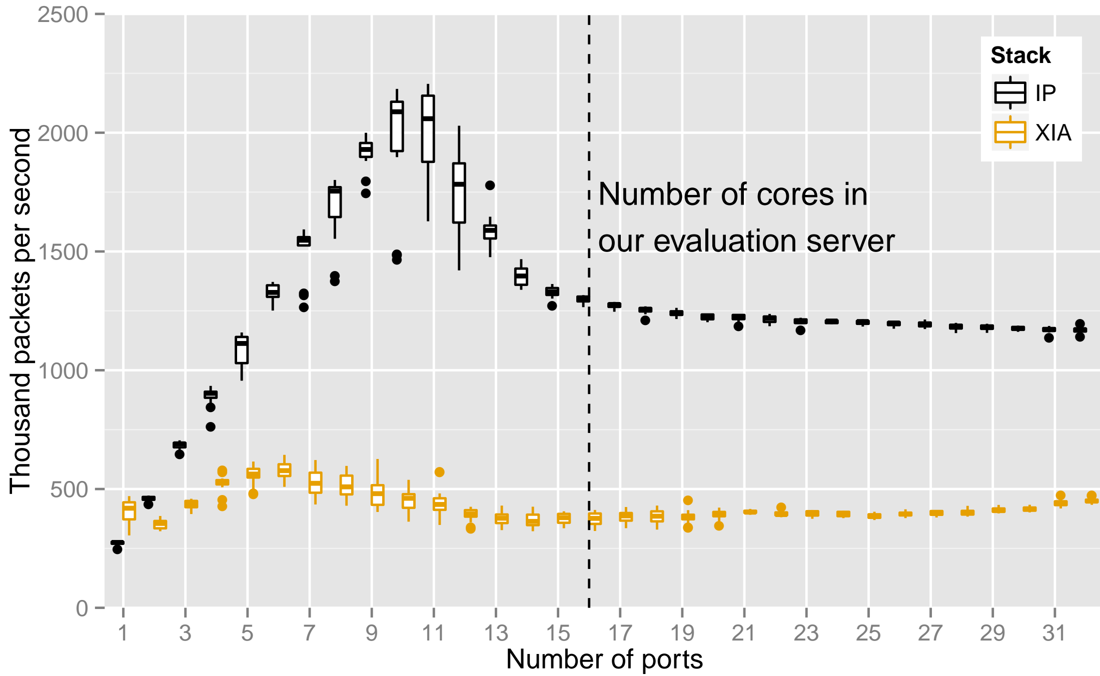


Figure 5·3: Impact of the number of ports on performance
Experiments with 4 ports run without saturating our evaluation server. Fixed parameters: Zipf exponent is 1.0, 256-byte packets, FB0 addresses, and no updates to the routing table.

adding value.

Each experiment run takes roughly 6 minutes. In order to make each experiment run independently from each other, our evaluation scripts reboot the server at the end of each run. The first and last 20 seconds are discarded to account for the transitional phases that takes place before the experiment reaches steady state, and during termination before reboot. The remaining 5 minutes are averaged to obtain the packets per second of that run. Each box in the graphics represents 20 runs of that same experiment. The middle lines of the boxes represent the median of those experiments, whereas the box itself spans the second and third quartile of the runs. The whiskers extend up to 1.5 times the height of the box; any data point beyond

that limit is plotted separately. Thus, Figure 5·3 alone represents approximately 5 days of wall-clock running time; not accounting for the time to reboot between runs. Together, we view this set of experiments as a tough stress test of the code.

5.2 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 complex addresses used in XIA, and high update rates of the routing table. In addition, Linux XIA's solution to routing dependencies, the dependency forest, works so well that it even surpasses IP's saturation update rate by an order of magnitude.

The distribution of packet addresses, which our simulation treats as Zipfian, is the factor that shows the most pronounced effect between the XIA and IP routing algorithms in our experiments (Figure 5·4). Although XIA only approximates IP's forwarding performance in the valuable 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 Zipf exponent is 0.8, in which case XIA's median packets per second is only 42% of that of IP. On the other hand, when the Zipf exponent reaches 1.6, XIA forwards at speeds comparable to IP's performance running with 10 ports (Figure 5·3). Still, IP is faster within a more valuable range of the Zipf exponent. We believe that XIA can do significantly better against IP 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.

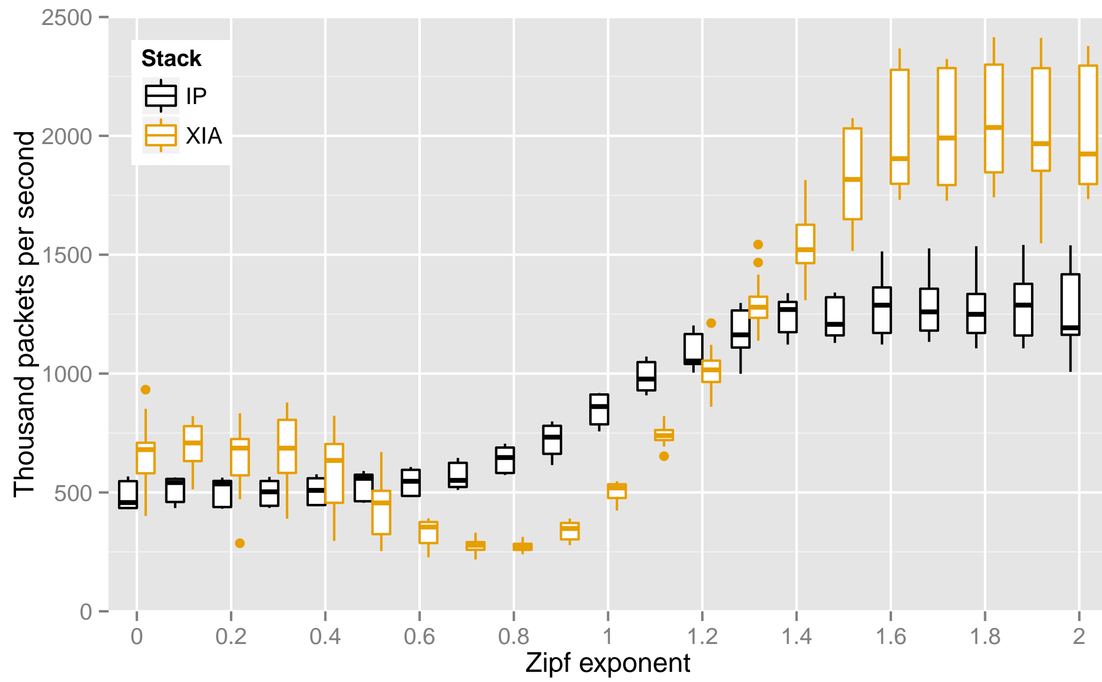


Figure 5·4: Impact of the Zipf exponent on performance. IP has the upper hand on the valuable range $[0.5, 1.2]$ of the Zipf exponent. Zipf exponent equal to zero produces the uniform distribution. Fixed parameters: 4 ports, 256-byte packets, FB0 addresses, and no updates to the routing table.

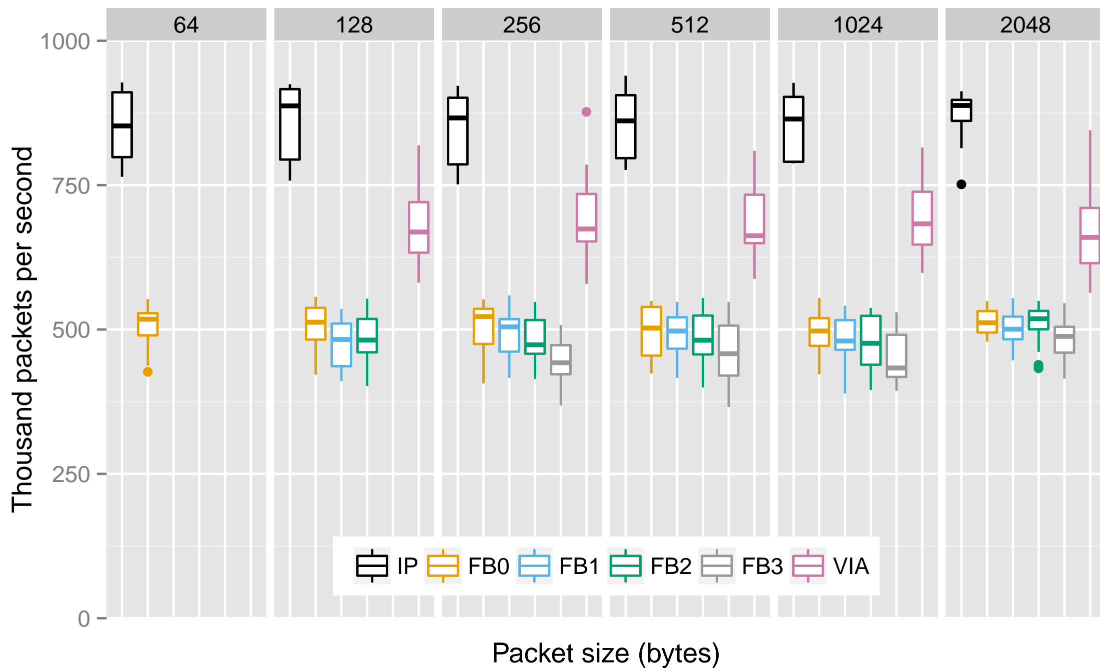


Figure 5·5: Packet size and more complex addresses have a small impact on performance. Addresses FB1, FB2, FB3, and VIA for 64-byte packets, and FB3 for 128-byte packets are not available because they do not fit into these small packets. Fixed parameters: Zipf exponent is 1.0, 4 ports, and no updates to the routing table.

Beyond the distribution of addresses, we conduct sensitivity analyses to quantify the impact of other variables on forwarding performance. Figure 5·5 shows that packet sizes, and more complex addresses in the XIA case, have a small impact on performance for both stacks. The advantage of IP over XIA is in accordance with Figure 5·4 when exponent is 1.0.

We do not yet have adequate justification for the surprising performance of VIA forwarding shown in Figures 5·5 and 5·6, as it seems no easier to forward VIA packets than FB0 packets. We expected that VIA would follow FB1 closely, as happened in the evaluation of XIA prototype (Han et al., 2012, Section 5.1). The most plausible explanation of what is happening is that the underdeveloped code that reclaims entries

of the DST table when the latter is overloaded (see discussion at end of Section 3·2) is trashing valuable DST entries, and the VIA case is somehow avoiding this to happen. In the current version of Linux XIA, the DST table is a hash table with a fixed number of 256 buckets, whereas the previous section points out that there are more than 460,000 possible edge sequences to cache. Although missing a proper justification for the better performance is vexing, this shows that there is room for improving Linux XIA's overall performance, which we will investigate in future work.

The last experiment assesses the effectiveness of our solution, the dependency forest, to keep the DST table synchronized to the routing table while the latter changes. Updating efficiency is relevant due to two factors. First, when disruptive topology changes take place due to hardware failure, traffic engineering, or policy changes, routers may see a large number of route updates, and any delays on reflecting those updates implies more dropped packets. The second factor is XIA-specific: we expect that some principals will show higher update rates than others, for example, large collections of content may require large numbers of CID updates concurrently.

While the update rate of the routing table for both stacks as well as different addresses for XIA show low impact on forwarding performance, IP saturates at an update rate at an order of magnitude lower than XIA (Figure 5·6). Running the router without PWs, we have found that IP saturates around 60K entries per seconds, whereas XIA goes all the way to 900K entries per second.

In our experiments, route updates are sent in batch from userland to the kernel using Route NetLink, Linux's preferred communication channel to manage its routing tables. We assume that each CIDR bock for IP, or each AD for XIA, has the same probability of updating its route, and thus each update is generated uniformly choosing a destination among all destinations. Each update also uniformly chooses a new destination port among all available ports.

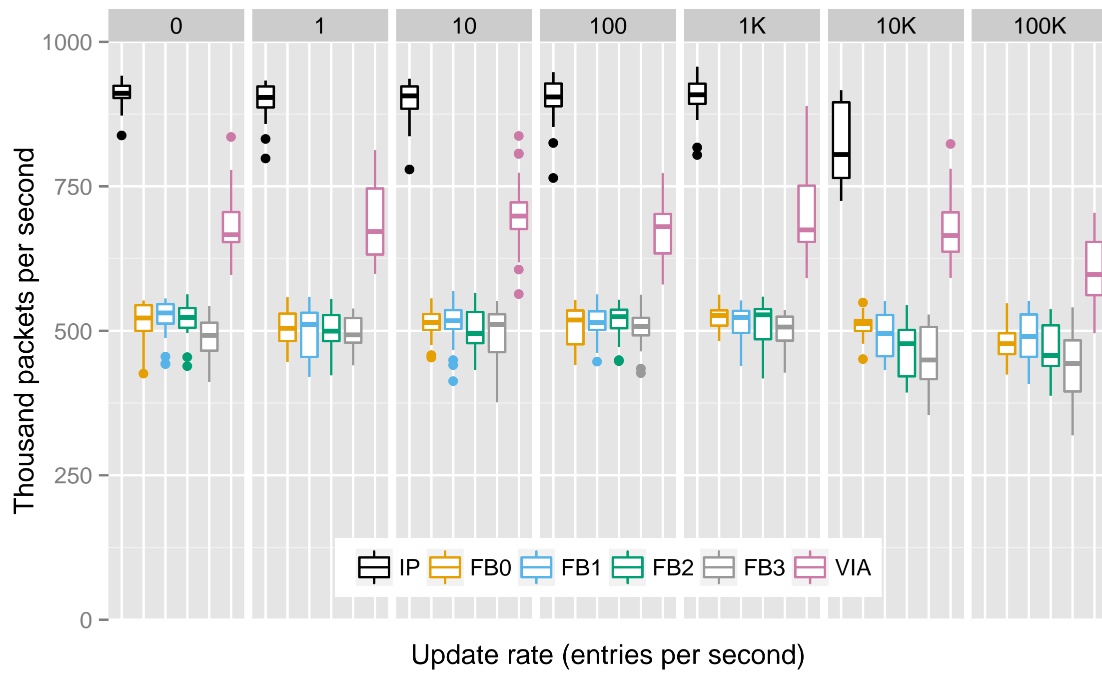


Figure 5·6: Impact of updating the routing table on performance. XIA saturates at a higher update rate of its routing table than IP. IP is not available at 100K entries per second column because it cannot keep up with that update rate. Fixed parameters: Zipf exponent is 1.0, 4 ports, and 256-byte packets.

While our experiments point that Linux XIA may not be ready for prime time, they do make the case that Linux XIA is already a viable platform for exploration of network principals. In addition, we have not found an unsurmountable wall holding us from matching IP's performance. We believe that the performance gap is rooted on the fact that Linux XIA is much younger than Linux IP and, as such, has not been polished as much as Linux IP. Therefore, closing this gap should be a matter of time of implementing solutions available in the literature (Dobrescu et al., 2009; Zhou et al., 2013; Han et al., 2010).

Chapter 6

Conclusions

The previous chapters have laid down a detailed presentation of XIA, an interoperable meta architecture; described Linux XIA, a native implementation of XIA from scratch in the Linux kernel; demonstrated how one can port alien architectures to XIA as well as how to design for XIA; and evaluated XIA benchmarking its forward performance against Linux IP.

XIA is fundamentally different from previous meta architectures. As an interoperable meta architecture, XIA (1) shares the whole network among its supported principals, (2) enables principal composition at every network address, and (3) encourages principal designers to postpone dependencies among principals until runtime through routing redirects. Thanks to these features, XIA principals can delegate functions and responsibilities to other principals, which, in turn, enables XIA principals to specialize and achieve their functionality with minimal form. Of course, the flexibility of XIA comes at a price, which is the noticeable complexity of its concepts: applications are responsible for composing their addresses, principal designers may have to go through a number of design and implementation iterations before they can field the best design form for their principals, and our current forwarding algorithm shows plenty of room for improvements that are not easy to address due to the inherent flexibility that it must support. A key novelty is that in XIA, principals do not have to be self-sufficient, in contrast to previous meta architectures, whose

network factors were forced to be self-sufficient architectures. Notwithstanding the lower principal isolation, XIA is highly expressive, as our demonstration that XIA supports a superset of the principals that ANTS supports substantiates.

Our research has corroborated our view of an interoperable meta architecture being a catalyst to bring future Internet architectures (FIAs) closer to fruition, and to empower the community at large to find *the* future Internet architecture. Finally, this exploration can start today, since XIA is deployable on current hardware and can coexist with TCP/IP.

The remainder of this chapter outlines future work (Section 6.1), presents an interesting alternative view of XIA as a layerless network stack (Section 6.2), and concludes with our final remarks (Section 6.3).

6.1 Future work

As a new network stack, Linux XIA affords plenty of directions to improve upon, and to explore the effects of new features on familiar concepts. In this section, we consider directions that constitute a departure from directions in this dissertation such as intrinisc security, but others that more directly follow from our study of forwarding performance. In addition, we consider porting other architectures to keep pushing the limits of evolution and add value to Linux XIA, evaluating the advantages that a meta architecture brings to applications, and exploring alternative definitions of the factors or how to architect for further specialization.

In spite of intrinsic security being around in the literature for a while now, it is still not part of a typical Internet use case today. Intrinsic security motivates researchers to revisit beaconing and routing protocols like ARP, OSPF, and BGP. For example, leveraging intrinsic security present in HID XIDs to identify poisoning packets, or propagating security properties through peering connections, could lead to

new solutions and protocols. Another venue to explore intrinsic security is to consider alternative definitions for XIDs. For example, the XIA version of Serval opened the door to explore possible alternatives to the intrinsic security that its ServalIDs and FlowIDs could use, and if they can or should depend on each other.

Compared to all the investment that has been poured into optimizing IP's forwarding performance in the past decades, the routing algorithms presented in sections 3.2 and 3.3 represent just a good start. On the software side, the following questions point how to move forward: (1) can Linux XIA forward as fast or faster than Linux IP, (2) is there a locking mechanism that leverages the dependency data structures to reduce contention while updating the data routing table, and (3) how can we achieve graceful degradation of routing performance under extreme stress due to an attack or load above hardware limits? The hardware side offers another set of lines that we have not explored and is just as interesting: (1) how can Linux XIA take advantage of Network Processors, FPGAs, or GPUs, (2) how can one extend OpenFlow to speed Linux XIA up, and (3) what can hardware provide to push Linux XIA to its performance limit?

6.2 Layers vs. factors

After going through the presentation of XIA and the large number of details that comes with it in the previous chapters, one may not see the forest for the trees. This section presents a bird's eye view of TCP/IP and XIA architectures[1] through the key abstractions that modularize these architectures. This angle is motivated by Barbara Liskov's insight on the relationship between modularity and abstraction (Liskov, 2009): "modularity based on abstraction is the way things are done".

The TCP/IP architecture has been modularized into layers[2]. In fact, the layer

[1]We are assuming here any instance of XIA with a set of factors that makes it an architecture.

[2] We borrow the link between Barbara Liskov's insight and layers as the abstraction of the data

abstraction is as old as computer networks, and many network architectures have embraced this abstraction to simplify reasoning and implementation. Layers establish clear functional interfaces that enable evolution within each layer as long as new protocols satisfy the interfaces. The layer abstraction has such deep roots into network stacks that few researchers have questioned its adequacy (Braden et al., 2003). Moreover, according to the surprising claim of the EvoArch model (Akhshabi and Dovrolis, 2011), evolving layered protocol stacks may be the sole cause of hourglass-shaped architectures and network layer stagnation.

It is the factor abstraction that enables meta architectures to evolve. Thanks to its lower isolation of factors (via specialization and interoperability of XIA principals), XIA goes a step further towards dropping the layer abstraction. The fact that XIA does not employ the layer abstraction may not be self-evident due to the impression that factors are above XIP, and XIP is depicted above the link layer in Figure 3·1. While the next paragraphs explain this alternative point of view further, we point out that Figure 3·1 is not in disagreement since it depicts our codebase and its boundaries, not necessarily layers.

Yet XIP is not a layer, but only a protocol that glues XIA principals together through its use of expressive addresses. If XIP were a layer, it would not be clear what abstraction it would export to the above layers. For example, TCP/IP's layers export reliable or unreliable transports, best-effort packet delivery, and physical transfer of frames.

The link layer is a factor whose identifiers are hardware addresses. This view of the link layer as a factor is easy to implement[3]. Consider an Ethernet factor whose XIDs are the numerical identifiers of the network interfaces followed by MAC addresses followed by the appropriate number of zeros. All that the Ethernet factor

plane that modularizes TCP/IP from Scott Shenker (Shenker, 2013)

[3] At the time this dissertation was written, Linux XIA does not implement the link layer as a factor.

on a host would have to do is to encapsulate XIP packets into Ethernet frames and put them on the wire. Ethernet XIDs could either be used directly in XIP addresses, or HID XIDs could redirect to Ethernet XIDs using routing redirects and retaining the same semantics that HID XIDs had in the previous chapters. If Ethernet XID were used directly in XIP addresses, XIA-aware Ethernet cards could forgo the Ethernet header and read the XIP header directly. The processing of those frames would follow regular XIP forwarding: if the `LastNode` field points to an Ethernet XID, deal with the packet accordingly, otherwise attempt a lookup on the following edge. Of course, this hardware modification may not be of interest since Ethernet brings multiple network stacks to share the same medium, but it emphasizes that a link layer realization can be seen as a factor.

Beyond the role of modularity that the layer abstraction takes on in many network architectures, layers establish the order in which packets are passed through protocols. In XIA, this order is pushed from design time to runtime; every XIP address defines the order in which factors are invoked. Ultimately, the layer abstraction reflects the static dependency that exists among the factors in a layered architecture.

The macroscopic view of TCP/IP and XIA from the abstractions that modularize these architectures spotlights the conceptual departure that XIA takes from the common view of computer networks, and further indicates that we are heading onto uncharted waters. While the departure from the common view could make the assimilation of XIA more difficult by the broad community, the fact that we are again entering uncharted waters is an exciting moment for researchers that have watched the incremental evolution of TCP/IP for more than 40 years.

6.3 Conclusions

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 XIA 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 messianic 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*.

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

Bibliography

Agyapong, P. and Sirbu, M. (2011). Economic incentives in content-centric networking: Implications for protocol design and public policy. In *Proceedings of the 39th Research Conference on Communication, Information and Internet Policy (TPRC)*, Arlington, VA, USA.

Akhshabi, S. and Dovrolis, C. (2011). The evolution of layered protocol stacks leads to an hourglass-shaped architecture. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Toronto, Canada.

Anand, A., Dogar, F., Han, D., Li, B., Lim, H., Machado, M., Wu, W., Akella, A., Andersen, D. G., Byers, J. W., Seshan, S., and Steenkiste, P. (2011a). XIA: An architecture for an evolvable and trustworthy internet. Technical Report CMU-CS-11-100, Carnegie Mellon University, Pittsburgh, PA, USA.

Anand, A., Dogar, F., Han, D., Li, B., Lim, H., Machado, M., Wu, W., Akella, A., Andersen, D. G., Byers, J. W., Seshan, S., and Steenkiste, P. (2011b). XIA: An architecture for an evolvable and trustworthy internet. In *Proceedings of the 10th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, Cambridge, MA, USA.

Andersen, D. G., Balakrishnan, H., Feamster, N., Koponen, T., Moon, D., and Shenker, S. (2008). Accountable internet protocol (AIP). *ACM SIGCOMM Computer Communication Review (CCR)*, 38(4):339–350.

Anderson, T., Peterson, L., Shenker, S., and Turner, J. (2005). Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4):34–41.

Arbor Networks, Inc. (2014). DDoS protection, prevention and mitigation. `http://www.arbornetworks.com/`.

Arye, M., Nordström, E., Kiefer, R., Rexford, J., and Freedman, M. J. (2012). A formally-verified migration protocol for mobile, multi-homed hosts. In *Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP)*, Austin, TX, USA.

Balakrishnan, H., Lakshminarayanan, K., Ratnasamy, S., Shenker, S., Stoica, I., and Walfish, M. (2004). A layered naming architecture for the Internet. In *Proceedings*

of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pages 343–352, Portland, OR, USA.

Blanton, E. and Allman, M. (2002). On making TCP more robust to packet reordering. *ACM SIGCOMM Computer Communication Review (CCR)*, 32(1):20–30.

Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. (2013). Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 99–110, Hong Kong, China.

Braden, R., Faber, T., and Handley, M. (2003). From protocol stack to protocol heap – role-based architecture. *ACM SIGCOMM Computer Communication Review (CCR)*, 33(1):17–22.

Cabral, C. M. S., Rothenberg, C. E., and Magalhães, M. F. (2013). Reproducing real NDN experiments using mini-CCNx. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking (ICN)*, pages 45–46, Hong Kong, China.

Challenger, J., Iyengar, A., and Dantzig, P. (1999). A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th IEEE Conference on Computer Communications (INFOCOM)*, volume 1, pages 294–303.

Cheriton, D. R. and Gritter, M. (2000). TRIAD: A new next-generation Internet architecture. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.5878.

Cisco (2013). Cisco visual networking index: Forecast and methodology, 2012-2017. Technical report, Cisco Systems, Inc., San Jose, CA, USA.

Clark, D. D., Wroclawski, J., Sollins, K. R., and Braden, R. (2002). Tussle in cyberspace: Defining tomorrow's Internet. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 347–356, Pittsburgh, PA, USA.

CloudFlare, Inc. (2014). The web performance and security company. https://www.cloudflare.com/.

Crowcroft, J., Hand, S., Mortier, R., Roscoe, T., and Warfield, A. (2003). Plutarch: An argument for network pluralism. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, pages 258–266, Karlsruhe, Germany.

Deering, S. E. (1991). *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University.

Dingledine, R., Mathewson, N., and Syverson, P. (2004). Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, page 303320, San Diego, CA, USA.

Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. (2009). RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 15–28, Big Sky, MT, USA.

Fall, K. (2003). A delay-tolerant network architecture for challenged internets. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 27–34, Karlsruhe, Germany.

Fayazbakhsh, S. K., Lin, Y., Tootoonchian, A., Ghodsi, A., Koponen, T., Maggs, B., Ng, K., Sekar, V., and Shenker, S. (2013). Less pain, most of the gain: Incrementally deployable ICN. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 147–158, Hong Kong, China.

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

Ford, A., Raiciu, C., Handley, M., Barre, S., and Iyengar, J. (2011). Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational).

Ford, B. A. (2008). *UIA: A Global Connectivity Architecture for Mobile Personal Devices*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Freedman, M. J., Arye, M., Gopalan, P., Ko, S. Y., Nordström, E., Rexford, J., and Shue, D. (2010). Service-centric networking with SCAFFOLD. Technical Report TR-885-10, Princeton University, Princeton, NJ, USA.

Ghodsi, A., Koponen, T., Raghavan, B., Shenker, S., Singla, A., and Wilcox, J. (2011). Intelligent design enables architectural evolution. In *Proceedings of the 10th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, pages 1–6, Cambridge, MA, USA.

Govind, S., Govindarajan, R., and Kuri, J. (2007). Packet reordering in network processors. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, Long Beach, CA, USA.

Han, D., Anand, A., Dogar, F., Li, B., Lim, H., Machado, M., Mukundan, A., Wu, W., Akella, A., Andersen, D. G., Byers, J. W., Seshan, S., and Steenkiste, P. (2012). XIA: Efficient support for evolvable internetworking. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA.

Han, S., Jang, K., Park, K., and Moon, S. (2010). PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 40(4), pages 195–206, New Delhi, India.

Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N. (2012). Reproducible network experiments using container-based emulation. In *Proceedings of the 8th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 253–264, Nice, France.

Handley, M. (2006). Why the internet only just works. *BT Technology Journal*, 24(3):119–129.

Hornyak, T. (2014). 1 billion smartphones shipped worldwide in 2013. `http://www.pcworld.com/article/2091940/`.

Ierusalimschy, R., Celes, W., and Figueiredo, L. H. d. (2013). Lua 5.2.3. `http://www.lua.org/`.

Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. (2009). Networking named content. In *Proceedings of the 5th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 1–12, Rome, Italy.

Jokela, P., Zahemszky, A., Rothenberg, C. E., Arianfar, S., and Nikander, P. (2009). LIPSIN: Line speed publish/subscribe inter-networking. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 39, pages 195–206, Barcelona, Spain.

Keromytis, A. D., Misra, V., and Rubenstein, D. (2002). SOS: Secure overlay services. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 61–72, Pittsburgh, PA, USA.

Koetsier, J. (2013). 800 million android smartphones, 300 million iphones in active use by december 2013, study says. `http://venturebeat.com/2013/02/06/800-million-android-smartphones-300-million-iphones-in-active-use/`.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297.

Koponen, T., Chawla, M., Chun, B.-G., Ermolinskiy, A., Kim, K. H., Shenker, S., and Stoica, I. (2007). A data-oriented (and beyond) network architecture. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 37, pages 181–192, Kyoto, Japan.

Koponen, T., Shenker, S., Balakrishnan, H., Feamster, N., Ganichev, I., Ghodsi, A., Godfrey, P. B., McKeown, N., Parulkar, G., Raghavan, B., Rexford, J., Arianfar, S., and Kuptsov, D. (2011). Architecting for innovation. *ACM SIGCOMM Computer Communication Review (CCR)*, 41(3):24–36.

Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, pages 1–6, Monterey, CA, USA.

Lao, L., Cui, J.-H., and Gerla, M. (2005). TOMA: A viable solution for large-scale multicast service support. In Boutaba, R., Almeroth, K., Puigjaner, R., Shen, S., and Black, J. P., editors, *NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems*, volume 3462 of *Lecture Notes in Computer Science*, pages 906–917. Springer.

Lilley, J., Yang, J., Balakrishnan, H., and Seshan, S. (2000). A unified header compression framework for low-bandwidth links. In *Proceedings of the 6th ACM SIGMOBILE Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 131–142, Boston, MA, USA.

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

Liskov, B. (2009). The power of abstraction. Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Triangle Computer Science Distinguished Lecturer Series, `http://www.youtube.com/watch?v=GDVAHA0oyJU`.

Ludwig, R. and Katz, R. H. (2000). The Eifel algorithm: making TCP robust against spurious retransmissions. *ACM SIGCOMM Computer Communication Review (CCR)*, 30(1):30–36.

Machado, M. (2013a). Linux XIA. `https://github.com/AltraMayor/XIA-for-Linux`.

Machado, M. (2013b). Network evaluation environment. `https://github.com/AltraMayor/net-eval`.

Maniatis, P., Roussopoulos, M., Swierk, E., Lai, K., Appenzeller, G., Zhao, X., and Baker, M. (1999). The mobile people architecture. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):36–42.

Mazières, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. (1999). Separating key management from file system security. *17th ACM SIGOPS Operating Systems Review (OSR)*, 33(5):124–139.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(2):69–74.

Moskowitz, R. and Nikander, P. (2006). Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational).

Mukerjee, M. K., Han, D., Seshan, S., and Steenkiste, P. (2013). Understanding tradeoffs in incremental deployment of new network architectures. In *Proceedings of the 9th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 271–282, Santa Barbara, CA, USA.

Naous, J., Walfish, M., Nicolosi, A., Mazières, D., Miller, M., and Seehra, A. (2011). Verifying and enforcing network paths with ICING. In *Proceedings of the 7th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Tokyo, Japan.

Narten, T., Nordmark, E., Simpson, W., and Soliman, H. (2007). Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard). Updated by RFC 5942.

Nguyen, G. T. K., Agarwal, R., Liu, J., Caesar, M., Godfrey, P. B., and Shenker, S. (2011). Slick packets. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 39(1):205–216.

Nordström, E., Shue, D., Gopalan, P., Kiefer, R., Arye, M., Ko, S. Y., Rexford, J., and Freedman, M. J. (2012). Serval: An end-host stack for service-centric networking. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA.

Plummer, D. C. (1982). Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (INTERNET STANDARD). Updated by RFCs 5227, 5494.

Popa, L., Ghodsi, A., and Stoica, I. (2010). HTTP as the narrow waist of the future Internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, Monterey, CA, USA.

Prince, M. (2013). The DDoS that almost broke the internet. `http://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet`.

Princeton SNS Group (2012). Serval. `https://github.com/princeton-sns/serval`.

Raghavan, B., Koponen, T., Ghodsi, A., Brajkovic, V., and Shenker, S. (2012a). Making the internet more evolvable. Technical Report TR-12-011, International Computer Science Institute (ICSI), Berkeley, CA, USA.

Raghavan, B., Koponen, T., Ghodsi, A., Casado, M., Ratnasamy, S., and Shenker, S. (2012b). Software-defined Internet architecture: decoupling architecture from infrastructure. In *Proceedings of the 11th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, pages 43–48, Seattle, WA, USA.

Ratnasamy, S., Shenker, S., and McCanne, S. (2005). Towards an evolvable Internet architecture. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 313–324, Philadelphia, PA, USA.

Raychaudhuri, D., Trappe, W., Gruteser, M., Yates, R., Martin, R., Seskar, I., Zhang, Y., Lindqvist, J., Venkataramani, A., Kurose, J., Towsley, D., Mao, Z. M., Yang, X., Choudhury, R. R., Reiter, M., Lehr, B., Banerjee, S., Chen, G., and Ramamurthy, B. (2010). MobilityFirst Future Internet Architecture Project. `http://mobilityfirst.winlab.rutgers.edu/`.

Rexford, J. and Dovrolis, C. (2010). Future internet architecture: Clean-slate versus evolutionary research. *Communications of the ACM*, 53(9):36–40.

Roscoe, T. (2006). The end of Internet architecture. In *Proceedings of the 5th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, pages 55–60, Irvine, CA, USA.

Saif, U. and Paluska, J. M. (2003). Service-oriented network sockets. In *Proceedings of the 1st ACM SIGMOBILE The International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 159–172, San Francisco, CA, USA.

Sandvine (2013). Global internet phenomena report. Technical Report 2H 2013, Sandvine Incorporated ULC, Waterloo, Canada.

Schuymer, B. D. (2011). ebtables v2.0.10-4. `http://ebtables.sourceforge.net/`.

Shenker, S. (2013). Software-defined networking at the crossroads. Stanford University Colloquium on Computer Systems Seminar Series, `http://www.youtube.com/watch?v=WabdXYzCAOU`.

Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. M. (2010). Can the production network be the testbed? In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 10, pages 1–14, Vancouver, Canada.

Smetters, D. K. and Jacobson, V. (2009). Securing network content. Technical Report TR-2009-01, Xerox PARC, Palo Alto, CA, USA.

Snoeren, A. C. and Balakrishnan, H. (2000). An end-to-end approach to host mobility. In *Proceedings of the 6th ACM SIGMOBILE Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 155–166, Boston, MA, USA.

Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. (2002). Internet indirection infrastructure. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 34, pages 73–86, Pittsburgh, PA, USA.

Subramanian, L., Stoica, I., lakrishnan, H., and Katz, R. H. (2004). OverQoS: An overlay based architecture for hancing internet QoS. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA.

Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., and Minden, G. J. (1997). A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86.

Tennenhouse, D. L. and Wetherall, D. J. (1996). Towards an active network architecture. *ACM SIGCOMM Computer Communication Review (CCR)*, 26(2):5–18.

Trossen, D., Sarela, M., and Sollins, K. (2010). Arguments for an information-centric internetworking architecture. *ACM SIGCOMM Computer Communication Review (CCR)*, 40:26–33.

University of Oregon (2013). Route views project. `http://www.routeviews.org/`.

VeriSign, Inc. (2014). DDoS protection and ddos protection services. `http://www.verisigninc.com/en_US/website-availability/ddos-protection/`.

Wetherall, D. J. (1999). *Service Introduction in an Active Network*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Wheeler, D. A. (2004). SLOCCount 2.26. `http://www.dwheeler.com/sloccount/`.

Wu, W., Demar, P., and Crawford, M. (2009). Sorting reordered packets with interrupt coalescing. *Elsevier Computer Networks*, 53(15):2646–2662.

XIA Team (2013). XIA prototype. https://github.com/XIA-Project/xia-core/.

Yang, X., Wetherall, D., and Anderson, T. (2005). A DoS-limiting network architecture. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 35, pages 241–252, Philadelphia, PA, USA.

Zander, J. and Forchheimer, R. (1983). Softnet – an approach to high level packet communication. In *Proceedings of the 2nd ARRL Computer Networking Conference*, pages 75–76, San Francisco, CA, USA.

Zhang, X., Hsiao, H.-C., Hasker, G., Chan, H., Perrig, A., and Andersen, D. G. (2011). SCION: Scalability, control, and isolation on next-generation networks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 212–227, Oakland, CA, USA.

Zhao, M., Zhou, W., Gurney, A. J. T., Haeberlen, A., Sherr, M., and Loo, B. T. (2012). Private and verifiable interdomain routing decisions. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 383–394, Helsinki, Finland.

Zhou, D., Fan, B., Lim, H., Kaminsky, M., and Andersen, D. G. (2013). Scalable, high performance Ethernet forwarding with CuckooSwitch. In *Proceedings of the 9th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 97–108, Santa Barbara, CA, USA.

Zhu, Z. and Afanasyev, A. (2013). Let's ChronoSync: Decentralized dataset state synchronization in named data networking. In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP)*, Göttingen, Germany.

Zhuang, S. Q., Lai, K., Stoica, I., Katz, R. H., and Shenker, S. (2003). Host mobility using an Internet indirection infrastructure. In *Proceedings of the 1st ACM SIGMOBILE The International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 129–144, San Francisco, CA, USA.

Curriculum Vitae

Michel Silva Machado

Personal

- Born on June 15, 1978.
- Brazilian citizen, and US permanent resident (green card).
- Contact at michel@digirati.com.br

Education

- B.S. Computer Engineering, Pontifícia Universidade Católica do Rio de Janeiro, 2004.
- M.A. Computer Science, Boston University, 2008.
- Ph.D. Candidate Computer Science, Boston University, 2014.

Employment

- Research Assistant, Boston University, 2011–2014.
- Postdoctoral Researcher, Boston University, 2014–present.
- Chief Technology Officer, Digirati Internet, 2001–present.

Publications

- Michel Machado, and John W. Byers. Instantiating the wisdom of the crowd: the case for an interoperable meta network architecture. Submitted to ACM SIGCOMM, Chicago, IL, USA, August 2014.
- Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. XIA: Efficient support for evolvable internetworking. In USENIX NSDI, San Jose, CA, USA, April 2012.

- Ashok Anand, Fahad Dogar, Dongsu Han, Boyan Li, Hyeontaek Lim, Michel Machado, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. An architecture for an evolvable and trustworthy internet. In ACM HotNets, Cambridge, MA, USA, November 2011.