# Enabling a Permanent Revolution in Internet Architecture

James McCauley
UC Berkeley & ICSI
jmccauley@cs.berkeley.edu

Yotam Harchol
UC Berkeley
yotamhc@berkeley.edu

Aurojit Panda
New York University
apanda@cs.nyu.edu

Barath Raghavan
University of Southern California
barathra@usc.edu

Scott Shenker
UC Berkeley & ICSI
shenker@icsi.berkeley.edu

## ABSTRACT

*Recent Internet research has been driven by two facts and their contradictory implications: the current Internet architecture is both* inherently flawed *(so we should explore radically different alternative designs) and* deeply entrenched *(so we should restrict ourselves to backwards-compatible and therefore incrementally deployable improvements). In this paper, we try to reconcile these two perspectives by proposing a backwards-compatible architectural framework called Trotsky in which one can incrementally deploy radically new designs. We show how this can lead to a permanent revolution in Internet architecture by (i) easing the deployment of new architectures and (ii) allowing multiple coexisting architectures to be used simultaneously by applications. By enabling both architectural* evolution *and architectural* diversity*, Trotsky would create a far more* extensible *Internet whose functionality is not defined by a single narrow waist, but by the union of many coexisting architectures. By being incrementally deployable, Trotsky is not just an interesting but unrealistic clean-slate design, but a step forward that is clearly within our reach.*

## CCS CONCEPTS

• **Networks** → **Network architectures**; **Public Internet**;

## KEYWORDS

Internet architecture, Internet evolution

**ACM Reference Format:**
James McCauley, Yotam Harchol, Aurojit Panda, Barath Raghavan, and Scott Shenker. 2019. Enabling a Permanent Revolution in Internet Architecture. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3341302.3342075

## 1 INTRODUCTION

### 1.1 Motivation

In discussions about changes to the Internet architecture, there is widespread agreement on two points. The first is that the Internet architecture is seriously deficient along one or more dimensions. The most frequently cited architectural flaw is the lack of a coherent security design that has left the Internet vulnerable to various attacks such as DDoS and route spoofing. In addition, many question whether the basic service model of the Internet (point-to-point packet delivery) is appropriate now that the current usage model is so heavily dominated by content-oriented activities (where the content, not the location, is what matters). These and many other critiques reflect a broad agreement that despite the Internet's great success, it has significant architectural problems.

However, the second widely-accepted observation is that the current Internet architecture is firmly entrenched, so attempts to significantly change it are unlikely to succeed, no matter how well motivated these changes are by the aforementioned architectural flaws. More specifically, the IP protocol is deeply embedded in host networking and application software, as well as in router hardware and software. As a result, architectural changes face extremely high deployment barriers (as evinced by the decades-long effort to move from IPv4 to IPv6).[1]

These two beliefs have moved the research community in contradictory directions. Spurred by the first – that the Internet architecture is deeply flawed – projects like NewArch [8] and others began (in the late '90s) to look at what later became known as *clean slate* redesigns of the Internet.[2] In subsequent years, there were large NSF research (FIND, FIA) and infrastructure (GENI) programs (along with their numerous EU counterparts) devoted to this clean-slate quest. Many good ideas emerged from these efforts, ranging from new security approaches [2, 41] to mobility-oriented architectures [33] to service-oriented architectures [26] to new interdomain designs [13, 38] to information-centric networking (ICN) [18, 19] to entirely new conceptions of the Internet [3, 9, 15, 24, 31, 37].

Despite their motivating needs and useful insights, and the long history of sizable research funding, none of these clean-slate designs had significant impact on the commercial Internet. This eventually led to a backlash against clean-slate designs, with many in the field viewing them as useless exercises that are hopelessly detached from reality. As a result of this backlash, the second observation – that the Internet architecture is, and will remain, difficult to change – then became the dominant attitude, and there has been a marked decrease in clean-slate research. The community's emphasis is now


*SIGCOMM '19, August 19–23, 2019, Beijing, China*

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342075

[1] Of course, there has been great progress in various network technologies, such as novel transport protocols (*e.g.,* QUIC), datacenter congestion control, reconfigurable datacenter topologies, and host networking stacks (*e.g.,* DPDK and RDMA); however, as we will see later when we define the term "architecture", these changes are not architectural in nature.

[2] This term was first popularized by Stanford's Clean Slate program.

on backwards-compatible designs that have been tested in large-scale operational networks, or at least evaluated with data drawn from such networks. In this paper, we try to reconcile these two perspectives by proposing an *architectural framework* called Trotsky that provides a *backwards-compatible* path to an *extensible* Internet. There are five key words in our mission statement that we now clarify.

We delve deeper into this definition in the next section, but in brief we define an Internet *architecture* to be the specification of the hop-by-hop handling of packets as they travel between sender and receiver(s). By architectural *framework* we mean a minimal set of design components that enable the deployment and coexistence of new architectures. Saying a new design is *backwards-compatible* or *incrementally deployable* means that one can deploy it in a way that does not prevent legacy applications or hosts from functioning, and that one can reap some benefits while only partially deployed. One may require that systems be upgraded to take advantage of the new capabilities of this design, but legacy systems can continue to function unchanged.

By *extensible* Internet we mean an Internet where new architectures can be deployed in a backwards-compatible manner, and multiple of these architectures can exist side-by-side. This latter property directly contradicts our current notion of IP as the Internet's narrow waist. We chose architectural extensibility as our central goal because it: (i) allows us to easily deploy radical new architectures without breaking legacy hosts or domains, (ii) allows these new architectures to "evolve" over time (new versions are just another architecture, and hosts/domains can smoothly transition from one to another), and (iii) allows applications to avail themselves of multiple architectures, so they can leverage the diversity of functionality to achieve their communication goals.

This last point is crucial. The ability for an application to simultaneously use multiple architectures changes the nature of an architecture from something that must try to meet all the needs of all applications, to something that more fully meets some of the needs of some applications. Thus, architectural extensibility would hopefully lead to a broader and ever-evolving ecosystem of architectures which, *in their union*, can better meet all the needs of all applications. Our hope is that by creating an extensible Internet, the Trotsky architectural framework will be true to its name by enabling a *permanent revolution* in architectural design.

### 1.2 Contribution

Trotsky has two key properties. The first is that the framework itself is backwards-compatible; nothing breaks while it is being deployed, and those adopting it accrue benefits even while it is only partially deployed. Second, once deployed, Trotsky enables radical new architectures to be deployed in a backwards-compatible fashion and to coexist with other architectures. To our knowledge, Trotsky is the first proposal that can simultaneously achieve both of these goals. *As such, it is the first proposal that provides a backwards-compatible path to an extensible Internet.*

This paper is unabashedly architectural, and thus our contribution is conceptual rather than mechanistic. When designing systems to be extensible, the key questions are typically about the modularity of the design (*i.e., are there aspects that break legacy systems, or limit the scope of new extensions?*), not about the novelty of algorithms or protocols. The same is true here. The basic mechanisms we use within Trotsky are extraordinarily simple; what is novel about Trotsky is the way in which these mechanisms are applied. Trotsky is based on a better understanding of the modularity needed to achieve architectural extensibility, not on a better design of the necessary mechanisms. In fact, to the extent possible, we try to reuse current mechanisms, and our resulting design strongly resembles the current Internet in all but one important aspect: how the Internet handles interdomain traffic.

After providing some background in the next section, we make our conceptual case for Trotsky in three steps. First, we analyze why architectural evolution is currently so hard. We find (in Section 3) that rather than this difficulty being inherent, it is due to a missed opportunity when ASes became a part of the Internet's structure, and argue that this mistake can be corrected by treating interdomain delivery as an overlay on intradomain delivery. *Trotsky (whose design is presented in Section 4) is merely the set of infrastructural pieces needed to support this overlay.*

Second, we have implemented Trotsky (see Section 5), producing a prototype of a Trotsky-Processor (Trotsky's version of a router) that supports Trotsky and some selected architectures using a mixture of hardware and software forwarding.

Third, in Section 6 we use this prototype to demonstrate that with Trotsky: (i) one can incrementally deploy a variety of new architectures, and (ii) applications can use multiple of these architectures simultaneously.

Note that our focus here is not on what future Internet architectures should be, or what problems they should solve, but rather on how we can change the Internet to the point where we can more easily deploy such architectures. Of course, our paper rests on the premise that architectural extensibility is an important goal. In claiming this, we are not saying that there is a specific new architecture that needs to be deployed immediately. Rather, our contention is merely that the Internet would be far better if it could easily adopt new architectures, and if applications could make use of multiple architectures. Such a development would turn the Internet into a more dynamic and diverse infrastructure.

The alternatives to architectural extensibility are (i) forever using *ad hoc* architectural workarounds for security, resilience, and other shortcomings of the current Internet, or (ii) eventually replacing the Internet entirely. Given that architectural extensibility is reachable in a backwards compatible manner, it seems preferable than either of these two alternatives. And if we think we will need to eventually evolve the Internet's architecture, then we should start laying the intellectual foundation now. This foundation, rather than immediate adoption and deployment, is the goal of our paper.

### 1.3 Ethical Considerations

This work does not raise any ethical issues.

## 2 CLARIFICATIONS, ASSUMPTIONS, AND A QUESTION

### 2.1 Two Clarifications

First, in what follows we typically refer to packets as the means of communication, with the packet header containing crucial signalling information (source, destination, QoS, etc.). Everything we describe

can also be applied to connection-oriented designs where the signalling information is carried out-of-band. While we briefly explain how this can be done within Trotsky in Section 6, for the sake of clarity our basic presentation focuses only on packet-based designs.

Second, there are several other works on architectural evolution. We postpone a detailed comparison of our approach with these until after we have more fully explained our design (see Section 7), but here we merely note that we have borrowed heavily from the insights in [12, 20, 28] but extended them to provide an incrementally deployable evolutionary process, which is the central contribution of this work.

### 2.2 Three Key Assumptions

First, while we wish to enable architectural innovation, one aspect of the Internet we do not expect to change any time soon is its domain structure (*i.e.,* being organized around ASes). This structure is not a technical design decision but rather a manifestation of infrastructure ownership and the need for autonomous administrative control – factors that we do not see changing in the foreseeable future. Thus, we assume that future architectures involve the cooperative behavior of multiple autonomous domains.

Second, given the momentum behind Network Function Virtualization (NFV) and Edge Computing (as in [6, 17]), our designs assume that domains have software packet processing capabilities at their edges (typically in the form of racks of commodity servers). This assumption has not yet been fully realized, but it is where NFV and Edge Computing are rapidly moving, so it seems a reasonable assumption when considering a future Internet.

Third, we assume that support for IPv4 (or IPv6) remains in place indefinitely (perhaps eventually replaced by a similar successor) so that hosts and applications that are Trotsky-unaware can continue to function without change. We will refer to this as the *default* architecture.

### 2.3 What Is An Architecture?

In a paper about architectural extensibility, we should clearly define the term "architecture". Informally, people often say an Internet architecture is what entities must agree on in order to interoperate, but what does this really mean? While there are multiple L1, L2, and L4 designs, the crucial aspect of today's Internet is that it has a single design for L3. This "narrow waist" is the component that different networks must agree on in order to interoperate, and the necessity of such a narrow waist is one of the central dogmas of the Internet. Thus, when we talk about the current Internet architecture we usually mean its L3 design, IP, which dictates the dataplane behavior at both hosts and routers. In addition, in order for different domains to interoperate, they must also agree on the interdomain routing protocol (currently BGP).

The literature about new architectures is then mostly about alternative designs for L3 (along with interdomain routing). When looking only at L3 in the current architecture, a packet's trajectory starts at the sending host, then traverses some number of routers (perhaps zero), and then arrives at the destination host (or hosts).

Generalizing this to not focus solely on L3 (for reasons that will soon be clear), we will define an Internet *architecture* to be a design for handling packets that travel from an originating host, through a series of intermediate network nodes that process them according to the architecture's dataplane (which, in turn, is guided by its control plane), to eventually arrive at one or more receiving hosts anywhere else in the Internet (*i.e.,* to be an architecture, these designs cannot be limited to local networks or particular domains, but must be capable of spanning the entire Internet).

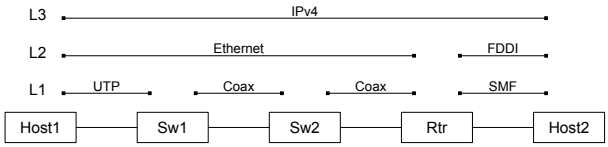


**Figure 1: Simplified hybrid layer diagram of a modern network with two L2 switches and one L3 router. L1 depicts the physical media (the lowest sublayer of L1 in the OSI model). L2 depicts the media access layer (the lowest sublayer of L2). L3 is the Internet layer (described in RFC 1122).**

As noted earlier, Trotsky is not an architecture but an architectural *framework*. This is because Trotsky does not encompass all the features needed to handle packets end-to-end; instead, it defines only those features necessary to create an extensible Internet capable of deploying new architectures in a backwards-compatible manner. While this distinction between architecture and framework is new to the networking community, the operating systems community has long understood that these concepts can be quite different; microkernels taught us that, rather than having a rigid monolithic OS, one can enable innovation by standardizing only a small subset of OS functionality and allowing the rest of the functionality to be supplied at user-level. While the microkernel is not enough, by itself, to support applications, it represents the key portion that enables extensibility. Similarly, Trotsky is not enough, by itself, to handle packets end-to-end, but it represents the key components that enable architectural extensibility.

## 3 MOTIVATING TROTSKY'S DESIGN

We noted earlier, without explanation, that the key to achieving extensibility is to treat interdomain delivery as an overlay on intradomain delivery. Here we provide the motivation for this statement.

### 3.1 The Internet Is A Series of Intrinsic Overlays

Today when we hear the term "overlay" we typically think of an *ad hoc* approach for deploying one protocol on top of another (as in IPv6 over IPv4), or a virtual infrastructure over a physical one (as in RON [1]). However, the Internet's layering is essentially the inherent and recursive use of overlays [36]. In these overlays, which are built into the architecture, nodes in each layer *logically* connect directly with their peers, but *physically* are communicating using technologies from the layer below (which are, recursively, built out of technologies below them). For instance, as shown in Figure 1: neighboring L2 endpoints directly communicate with each other at a logical level, but are connected via physical L1 links; neighboring L3 endpoints directly communicate with each other at a logical level, but are physically connected via "logical links" provided by L2 networks. This extends to L4, where two L4 endpoints directly communicate at the logical level with L3 providing "logical pipes"

connecting the two endpoints. Thus, L2 is an inherent overlay on L1, L3 is an inherent overlay on L2, and L4 is an inherent overlay on L3.[3]

The intrinsic use of overlays allowed the Internet to accommodate heterogeneity – and thereby spurred innovation at L1, L2 and L4 – through the presence of three factors. First, these layers provided relatively clean interfaces so, for instance, an L2 network could use multiple L1 technologies, as long as each supported the required L2 interface. Second, there was a unifying point of control to manage the heterogeneity. Each L2 network provided the unifying point for multiple L1 technologies, and L3 provided the unifying point for multiple L2 technologies. Applications were the unifying point for L4, as each application typically uses a default L4 protocol (so the two ends of an application would automatically use the same transport protocol). Third, the underlying layers L2 and L3 provide the equivalent of a next-header field, so that the packet could be handed to the appropriate next-layer protocol during processing.

If we consider the Internet before the advent of Autonomous Systems (or domains), then the one layer where there was no way of managing heterogeneity was L3. If there were multiple L3 protocols then (i) two L2 networks that only supported disjoint subsets of these L3 protocols could not interconnect and (ii) even if all adjacent L2 networks shared a common L3 design, there would be no way to guarantee end-to-end connectivity for any particular L3 design. Thus, because there was no way to manage heterogeneity at L3 before domains, we assumed that there could only be one L3 protocol (the "narrow waist" of the architecture). In short, in the absence of domains, a narrow waist at L3 was needed to enable arbitrary sets of networks to easily connect and provide end-to-end connectivity, while at the other layers the overlay approach could support diversity.

This made architectural evolution extremely difficult because, when there is a single L3 protocol that must be deployed everywhere, converting to a new L3 design requires (eventually) changing the L3 implementation in all routers. Moreover, if the new L3 design is not compatible with current router hardware, then this is not just a configuration or firmware change in routers, but a hardware one as well. In the case of the IPv4 to IPv6 transition, we have had decades to prepare all routers to support IPv6, and yet the transition still requires many *ad hoc* measures such as building temporary overlays (so that IPv6 traffic could be carried over IPv4 networks) and deploying special-purpose translators (to translate between IPv4 and IPv6 packets). These techniques have been effective, but it required decades of preparatory work; this is not an example we can follow for more general architectural change.

When domains arose, they presented a unique opportunity to manage the heterogeneity at L3. A domain could ensure *internal* connectivity (by carefully managing which L3 designs their internal networks supported) and also manage *external* connections to ensure global end-to-end delivery (using remote peering, which we describe later in this section). Managing internal and external heterogeneity independently – *i.e.,* decoupling the internal choices from the external ones – requires making a clean architectural distinction between interdomain and intradomain dataplanes (just as we make a distinction between L1, L2, and L3 dataplanes). The natural way to do this, following what had been successfully done at each of the other layers, would have been to make interdomain connectivity an intrinsic overlay on intradomain connectivity. But this path was not chosen, and the opportunity to support heterogeneity at L3 was squandered.

Instead, it was decided to connect domains by using the same dataplane L3 protocol as used within domains, and to devise a new control plane protocol to handle interdomain routing. This allowed domains to choose their own intradomain routing protocols, but there was still a single universal L3 dataplane. It is this L3 universality in today's architecture – an architecture which in so many other ways is designed to support heterogeneity – that makes the current architecture so hard to change.

### 3.2 Two Design Decisions

Our goal for Trotsky is to make the mechanisms needed to incrementally deploy new architectures a fundamental part of the infrastructure, rather than something deployed for a specific architectural transition (such as IPv4 to IPv6). This involves two major decisions.

First, to use domains as a way of managing heterogeneity at the L3 layer, we reverse the decision to use the same dataplane for both intradomain and interdomain delivery. Instead, we decouple the distinct tasks of interconnecting networks within a domain (which we leave to L3) and interconnecting different domains, for which we introduce a new layer (which we will call L3.5) which is an intrinsic overlay on L3. This overlay approach – which allows the Internet to (as we argue later) support multiple L3.5 designs and rely on domains to manage the resulting heterogeneity – is the key to creating an architecturally extensible Internet. *The role of Trotsky is little more than providing the intrinsic support for this overlay.*

Second, we initially deploy any new L3.5 design only at domain edges[4] (in entities we will call "Trotsky-Processors" or TPs) and at hosts. This decision has two implications: (i) it greatly limits the required scope of deployment for any new L3.5 design, and (ii) the aforementioned presence of software packet processing at the edge (due to NFV and Edge Computing) means that the required deployments can be in software rather than hardware (and we later show in Section 6 that this is feasible). Thus, this second decision means that supporting a new L3.5 protocol requires only a software modification at domain edges, rather than (as today) hardware modifications in all routers.

How can we limit these changes only to the edge? Just as L3 abstracts L2 connectivity using logical links, L3.5 abstracts L3 connectivity as "logical pipes". That is, when looking only at L3.5, TPs are connected (to each other and to hosts) by logical pipes which use technologies from the layers below to deliver packets from one end of the logical pipe to the other (see Figure 2). Thus, even when only deployed at the edge, packets still travel from a host through a series of L3.5-supporting boxes to the destination host(s), all connected by logical pipes.[5]

[3] The discussion here and what follows is not a summary of what the original Internet designers thought at the time, but is a description of how we might think about the Internet's design in hindsight.

[4] Edge here means all ingress/egress points in a domain, which includes peering points (bilateral and IXPs) and customer attachment points.

[5] One might ask whether putting functionality only at the edge gives adequate performance for content-oriented architectures (which rely on caching). This has been studied in [11] where it was shown that supporting ICN designs only at the edge provided a very high fraction of the performance benefit.

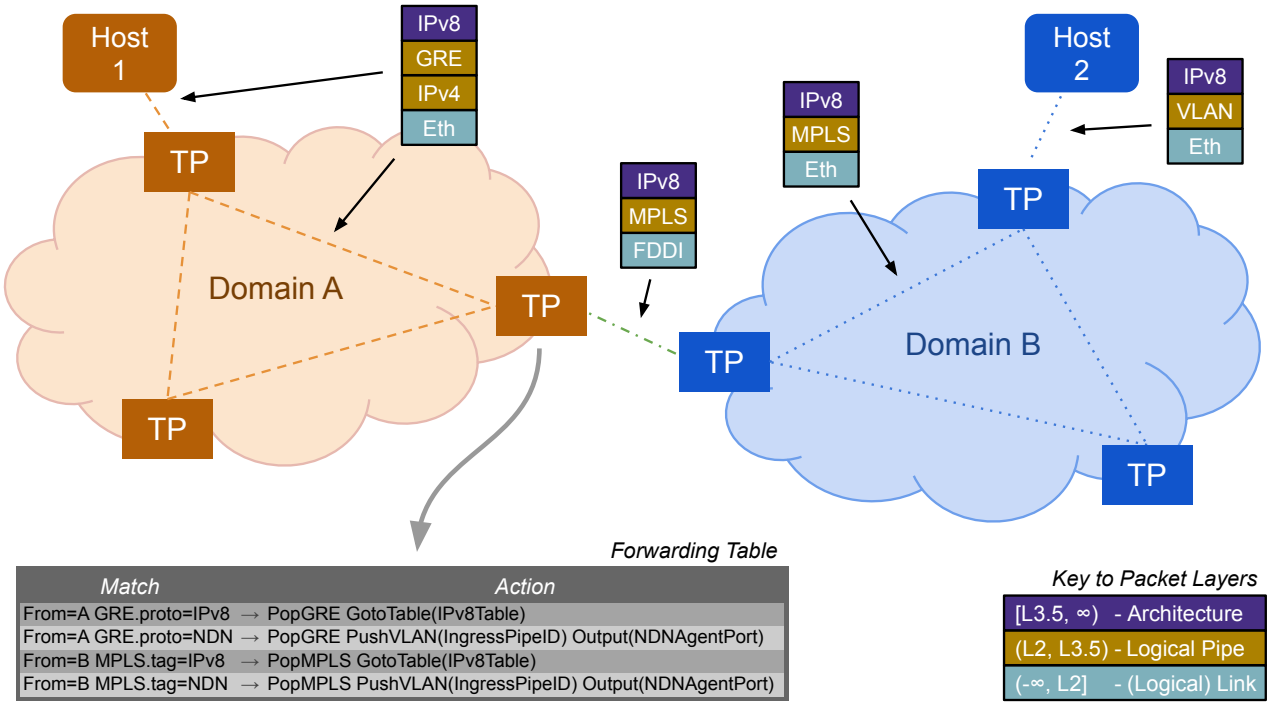


**Figure 2: Illustration of a possible deployment, with headers for a hypothetical IPv8 L3.5 packet traveling between two hosts. Different logical pipes are implemented differently (using GRE+IPv4, MPLS, and VLANs). A partial and abstract forwarding table for one TP's Pipe Terminus (see Section 5.1) describes how different packets are handled: IPv8 packets are sent to a hardware IPv8 forwarding table, and NDN packets (another L3.5 supported by the domain) are handled by a software NDN agent attached to a physical port (the ingress pipe ID being encoded in a VLAN tag so the agent is aware of it).**

### 3.3 Making Interdomain An Inherent Overlay

For L3.5 designs to be overlaid on L3 (and layers below), L3 must provide logical pipes to connect pairs of L3.5-aware nodes (*i.e.,* TPs and hosts). These pipes must support, at the very least, best-effort packet delivery (below we describe optional enhancements to the pipe model) and also provide the equivalent of a *next-header* field (as Ethernet does at L2, and IP does at L3), so that a receiving TP knows what L3.5 protocol should be applied to an arriving packet. Note that a logical pipe is nothing more than an intradomain packet delivery service (like IP) that delivers packets from one location to another, and need not entail establishing $n^2$ tunnels. As shown in Figure 2, logical pipes can be constructed in various ways, and encapsulate packets as they travel between TPs.

An L3.5 design defines the dataplane behavior (*e.g.,* packet format, forwarding rules, etc.) executed in the TPs, along with whatever interdomain control plane protocol is needed to support the L3.5 design.[6] This is no different than what a new L3 design would define in the current Internet; thus, the Trotsky approach imposes no additional architectural restrictions.

An L3.5 design's data and control plane determines an end-to-end service model. This service model applies even when the communicating nodes are in the same domain: just as IP determines (via host L3 code) the service model when two hosts in the same subnet (L2 network) communicate, the host L3.5 implementation determines the service model even when two hosts in the same domain connect. Thus, one should not think of an L3.5 design as *only* an interdomain dataplane, but as the only dataplane that can span domains and which therefore determines the nature of the end-to-end service. In contrast, in our approach L3 protocols only provide logical pipes within domains, and do not determine the Internet's service model.

The service model defined by an L3.5 design is not restricted to basic best-effort packet delivery; instead, it could be service-oriented, or information-centric, or mobility-oriented, or something entirely different. In addition, these L3.5 designs need not support all applications, but instead could focus on more specific use cases (such as supporting in-network aggregation for IoT, or providing a particular security model). Because of such restricted use cases, L3.5 designs can impose restrictions on the underlying logical pipes. For instance, an L3.5 design could require the underlying technologies to be all-optical, or provide end-to-end encryption, or various multipoint delivery functions (such as broadcast or multicast). While these restrictions would limit the applicability of such designs, this is appropriate for L3.5 designs that are intended for specific use cases.

[6] For convenience, we will assume the familiar model in which this interdomain control plane protocol is executed by one or more TPs within each domain, but it could be implemented anywhere.

Thus, we expect that L3.5 designs might support a wide variety of service models, and therefore hosts must have a network programming API (or *NetAPI*) that can support a range of communication semantics. Fortunately, this is already provided by existing mechanisms for adding support for new protocols to the socket API (*e.g.,* protocol families and the flexible *sendmsg()* function), which is supported by virtually all current operating systems. Applications need to be aware of the new NetAPI semantics in order to use the associated L3.5 designs (*e.g.,* the service model of ICN designs is different from that of IP, and this must be reflected in the NetAPI; an application must be aware of these different semantics in order to invoke them).

### 3.4 Implications for Extensibility

Before delving into the design of Trotsky in the next section, we should clarify why inserting a new interdomain layer (L3.5) addresses the issues of extensibility. There are three factors here: independence, deployment, and coexistence. First, by decoupling interdomain (L3.5) from intradomain (L3) designs via a clean "logical pipes" interface, one can evolve one without changing the other. Second, the deployment of new L3.5 designs is made easier by our only requiring that they be deployed at domain edges, and initially in software. Third, multiple L3.5 designs can coexist, and this is made possible by the existence of an extensible NetAPI (which is already in use) and the fact that the L3 logical pipes must support a next-header field that identifies which L3.5 design should handle the packet. Because of these three factors, one can deploy a new L3.5 design without changing any L3 designs, this deployment can initially be limited to only software at domain edges, and these L3.5 designs can operate in parallel. But three questions remain:

*How general is the Trotsky approach?* The definition of L3.5 involves the forwarding and receiving behavior of hosts and TPs (all connected by logical pipes), along with an interdomain control plane. Note that this picture is directly analogous to the current picture of L3 except with TPs replacing routers, and with logical pipes replacing logical links (we use different terms – pipes vs. links – because the pipe abstraction is global as in L3, whereas the link abstraction is local as in L2). Thus, any design that could be seen as a candidate for an L3 architecture in today's Internet could also be seen as a candidate for L3.5 in our approach. The reverse is not true, because (as mentioned above) we allow L3.5 designs to put restrictions on the nature of the logical pipes. However, one could imagine designs that required specifying more than a per-hop dataplane and a control plane (*e.g.,* designs requiring some large-scale physical-layer coordination) that would not be implementable within Trotsky.

*How does Trotsky handle partial deployment of an L3.5 design?* When a new L3.5 design arises, it obviously won't be deployed by all domains simultaneously. Trotsky deals with partial deployment by the well-known technique of *remote peering*. Two domains that are not directly physically connected but wish to use the new L3.5 can peer by (i) building a tunnel between them (using, for example, the default L3.5 design) and then (ii) establishing a logical pipe over this tunnel (which treats the tunnel like a logical link). In this way, L3.5 designs can be initially deployed by a subset of domains which connect via remote peering.

*How does Trotsky handle partial deployment of Trotsky itself?* In terms of functionality, domains that are not Trotsky-aware are roughly equivalent to Trotsky-aware domains that support only IP at L3.5.[7] To peer with a domain that has not adopted Trotsky, Trotsky-aware domains use special logical pipes that strip off all but the IP headers.

## 4 TROTSKY DESIGN

We now turn to the design of Trotsky. This is anticlimactic, as the design is minimal and straightforward. However, since Trotsky is the framework that remains constant, while various L3.5 architectures come and go, it is important that we properly identify the crucial functionality and leave everything else to other (more dynamic) components in the ecosystem. Thus, in this section, we start by describing the necessary pieces of Trotsky (whose implementation we describe in the next section), and then discuss issues that are left to either L3.5 designs or to the domains. We then end this section with a walk-through of what happens when a laptop arrives in a domain to illustrate how the various pieces fit together.

### 4.1 Design Overview

Trotsky must provide intrinsic support for overlaying L3.5 designs on L3, which involves two key components.

**Logical Pipes.** The lower layers must provide logical pipes (*i.e.,* connectivity) between every two L3.5 nodes. Constructing these pipes requires datapath mechanisms (to support basic delivery and a next-header field) and control path mechanisms (to support functions such as MTU negotiation).

**Host Bootstrapping.** We need a mechanism – a service managed by a domain and accessed by a host, akin to today's DHCP – for providing a host with basic bootstrapping information. This includes the set of L3.5 protocols supported by the domain and the specific protocols it uses for logical pipes.[8]

Trotsky is nothing more than the union of these two functions, with everything else left to L3.5 designs and the domains. We now list some of those other functions.

### 4.2 Functions Left to L3.5 Designs

**Naming and Addressing.** In the simplest case, all L3.5 designs would have their own set of names, name resolution services, and addresses. However, we expect that names and name resolution systems might be shared between L3.5 designs; similarly, addressing schemes might be used by more than one L3.5 design (*e.g.,* IPv4 addresses might be used for an end-to-end optical design). To accommodate this, we assume that each L3.5 design identifies the naming and addressing schemes it is compatible with, and the host OS and domain bootstrapping mechanisms can direct resolution requests to the appropriate infrastructure. We assume that the responsibility for

[7] Note that in Trotsky, IP can be used both at the L3 layer (to support logical pipes) and at the L3.5 layer (to provide end-to-end connectivity); the address spaces for these two uses are separate.

[8] The question of what functions were built into Trotsky and which were left to L3.5 designs is a subtle one, and we could imagine alternative designs that arrived at slightly different conclusions. For instance, some bootstrapping information could be left for hosts to discover on their own, rather than be supplied via a single query, but we prefer the cleaner approach where the information is more explicit.

establishing a new name resolution infrastructure does not lie with individual domains, but with third-parties (such as ICAAN).

**Interlayer Mapping.** In the current Internet, ARP is commonly used to map between L3 addresses and L2 addresses. Similarly, in Trotsky, L3.5 designs must provide a mechanism to map between L3.5 addresses (which determine end-to-end-delivery, but are not necessarily traditional host locators and could instead refer to content or a service) and the pipe addresses (addresses used by logical pipes to deliver packets within a domain). This can be done by the L3.5 packet itself (by having a field in the packet to carry the pipe address) or with some ARP-like mechanism defined by the L3.5 design. One might question why this is being left to L3.5 designs rather than via a generic Trotsky functionality. In fact, early versions of Trotsky included a generic ARP-like mechanism that could be used by all L3.5 protocols. This approach shared a weakness with ARP: the required state scales with the number of endpoints (e.g., host addresses). Unfortunately, this is potentially more problematic in Trotsky than with ARP because the required state for a domain is much larger than a single L2 network. While we believe the state requirements are currently feasible, we think it unwise to assume this remains true in a design that should underlie the Internet for the foreseeable future. Accordingly, we moved the responsibility for this task from the Trotsky framework to the individual L3.5 protocols. This allows for L3.5s to perform optimizations that are specific to their designs and even allows building solutions to the mapping problem into their design from the outset (for example, designing addresses to allow for the mappings to operate on aggregations, assigning addresses to facilitate a programmatic rather than state-based mapping, or embedding L3 addresses within their own L3.5 protocol).

**Reachability.** With multiple L3.5 designs coexisting, and with not all domains supporting all L3.5 designs, how does a host decide which to use to reach a particular destination or content or service? There are several answers to this. First, if the name of the destination/object/service is tied to a specific L3.5, then the sending/requesting host knows which L3.5 to use (assuming that L3.5 is supported by its own domain). Second, L3.5 designs could (but need not) support an interface that allows hosts to query reachability (a negative result returning something like the ICMP Host Unreachable message). Third, names and/or the name resolution service might contain hints about which L3.5 designs are supported. Ultimately, however, applications can simply attempt using the name with multiple L3.5 designs.

**Congestion Control Support.** Congestion control will still be implemented in L4, but there are various proposals (*e.g.,* RCP [7] and FQ [10]) where routers provide support for congestion control. L3.5 designs can specify what congestion control mechanisms TPs and logical pipes must provide.

**Security.** In discussing security, it is important to make two distinctions. First, we must distinguish between the broad concerns of operating systems and applications, and the more narrow responsibilities of the network itself. Second, for those concerns that are the responsibility of the network, we must distinguish between those which must be addressed within the Trotsky framework itself, and those which can be left to L3.5 designs.

Turning to the first question, as has been observed by many (see [5, 20]), the goal of network security, very narrowly construed, is to provide communicating parties with availability (ensuring that they can reach each other), identity (knowing whom they are communicating with), provenance (knowing the source of data they have received), authenticity (knowing that this data has not been tampered with), and privacy (ensuring that data is accessible only to the desired parties). All but availability can be handled by cryptographic means at the endpoints (which may be made easier by the choice of naming system and other aspects of an L3.5 design). But availability – in particular the ability to withstand DDoS attacks – requires a network-level solution. In addition, there are a broader set of network security concerns such as anonymity and accountability, as discussed in [2, 20, 22, 25] among other places, and these too require network-level solutions.

The question, then, is whether issues not handled solely by endpoints should be left to individual L3.5 designs, or incorporated into Trotsky itself. Here we firmly believe that these security concerns should be addressed within individual L3.5 designs for the simple reason that, as security threats change and security approaches improve, we should allow these security solutions to evolve over time. For instance, for DDoS, there are a range of approaches, ranging from capabilities [39] to filters [4] to shut-up-messages [2, 20], and we think it presumptuous to bake a single approach into Trotsky itself. Moreover, it is not even clear how effectively Trotsky *could* lay out a security framework sufficient to address the needs of all potential L3.5 designs (with their own potentially idiosyncratic principals, communication patterns, and so on).

All the above said, the protocols of Trotsky itself (*e.g.,* the bootstrapping protocol) must be designed to ensure that they do not create additional security vulnerabilities. To that end, we have designed Trotsky so that it can be implemented in a secure fashion.

### 4.3 Functions Left to Domains

**Resource Sharing.** When multiple L3.5 designs are supported by a domain, the domain can control how its internal bandwidth is shared between these designs using standard methods available in current routers.

**Internal L3.5 Deployment.** While we envision that initially L3.5 designs are deployed only at the edge, domains can decide to provide additional internal support later if desired (*e.g.,* to provide closer-by caches if the L3.5 design involves content caching). This can be done by adding additional TPs, and whatever internal control plane mechanisms are needed. This decision can be taken on a domain-by-domain basis.

### 4.4 How Does This All Fit Together?

Consider a Trotsky-aware laptop that arrives in a domain, and proceeds to start a game application. The laptop connects, as it does today, to an L2 technology (*e.g.,* plugging into an Ethernet port, or attaching wirelessly) and uses the Trotsky bootstrap mechanism to determine which L3.5 designs are available, as well as what protocols are used for logical pipes. It then uses the bootstrapping mechanisms relevant to the internal domain protocols (*e.g.,* DHCP if IP is being used for logical pipes), and whatever L3.5-specific bootstrapping mechanisms are available for the L3.5 designs it wants to

leverage. The game invokes the NetAPI for content-oriented retrieval (to download graphical and other game assets) and also invokes the NetAPI for a special-purpose multiagent-consistency L3.5 design ideal for multiplayer gaming.

Each packet leaves the host in a logical pipe that uses its *next-header* field to identify which L3.5 the packet is associated with. If the destination for the packet is within the origin domain, then the L3.5 design describes how to reach it using an internal logical pipe without having to send packets to an intermediate TP. If the destination for the packet is in a different domain, the initial logical pipe takes the packet to a TP on the edge of the origin domain, where it is processed by the appropriate L3.5 design and then sent over a logical pipe to a TP running in a peer domain. The peer domain sends the packet through an appropriate logical pipe either to the destination (if the destination is in that domain), or to another peer domain (if the destination is elsewhere).

Note that if the user starts a legacy application (*e.g.,* an application that is not Trotsky-aware), the OS automatically invokes the default L3.5 design (*e.g.,* IPv4) and the application proceeds as today. Similarly, when a host arrives in a domain and does not use the Trotsky bootstrap mechanisms, the domain automatically assumes all of its traffic is using the default L3.5 design. Thus, Trotsky is incrementally deployable since it does not break hosts or applications that are not Trotsky-aware.

## 5 IMPLEMENTATION

In the previous section we identified Trotsky's two key technical tasks: providing logical pipes and host bootstrapping. Here, we describe how these tasks are accomplished in the two entities where Trotsky is implemented: Trotsky-Processors (TPs) and Trotsky-capable hosts. We do so by first describing the implementation challenges in general, and then how they are met in our prototype implementation. Before starting, however, recall that logical pipes (just like L2's logical links do for L3) essentially encapsulate L3.5 packets as they traverse between two L3.5-aware nodes, with each endpoint serving as a terminus for these logical pipes. These logical pipes, which must support best-effort packet delivery and the equivalent of a next-header field, can be constructed in many ways using protocols at or below L3 (such as using IP at the L3 level for intradomain connectivity, and VLAN IDs as the next-header field). In addition, different domains can use different pipe constructions, and a single domain can use different pipe constructions for different contexts (*i.e.,* logical pipes connecting two TPs in neighboring domains might be constructed differently than those connecting internal hosts to edge TPs).

### 5.1 Trotsky-Processors

The essential functional units of a Trotsky-Processor are illustrated in Figure 3. We now discuss them in turn.

**Pipe Terminus.** Packets carried by logical pipes arrive at physical interfaces of a TP, enter the pipe terminus (which removes the logical pipe encapsulation), and are then handed to the appropriate L3.5 implementation based on the next-header field. Essentially the opposite operations are performed for packets exiting the TP. In addition, the TP has a unique ID for each logical pipe it terminates, and the pipe terminus provides a mechanism to map each packet

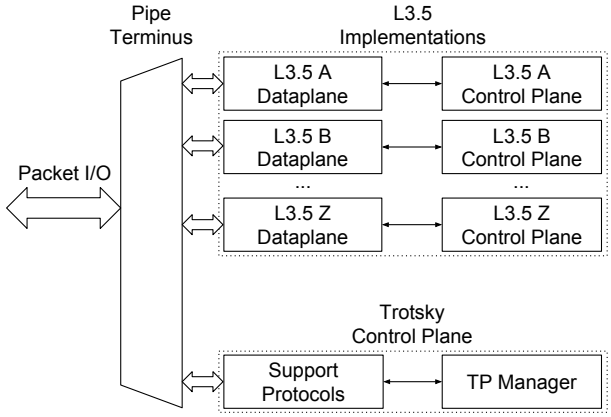


**Figure 3: Trotsky Processor System Diagram, supporting multiple L3.5 designs.**

with the logical pipe ID, which can be implemented by either storing this ID in the packet header or through other means.

**L3.5 Implementations.** Every TP supports the default L3.5 design along with possibly several other L3.5 designs. Such support includes an L3.5 dataplane and any associated control plane protocols. While control protocols are typically implemented in software, dataplanes for new protocols are likely to begin in software but can eventually transition to hardware for better performance and power usage. As we discuss further in Section 6.1, in addition to a default IPv4 L3.5 implementation, we integrated existing software codebases for two different L3.5 designs into our prototype TP, wrote code for a third, and investigated how one might support several other L3.5 designs. Should hardware support for any of these designs eventually arise, it would be simple to migrate to it. Moreover, existing hardware for the efficient handling of IP already exists, and – as we describe later in this section – it can be directly used in support of interdomain (L3.5) IP. Thus, Trotsky can accommodate both hardware and software dataplane implementations.

**The Trotsky Control Plane.** TPs also contain control/management infrastructure that primarily consists of an extensible store of configuration information which is used for internal purposes (*e.g.,* information about the logical pipes – how many there are, what protocols they use, whether they are intradomain or peering, etc.), and can also be used by L3.5 implementations via an API. Configuration data can be marked for sharing with other TPs via the Trotsky Neighbor Protocol. This protocol also allows for sanity-checking (*e.g.,* that logical pipes are connected to their intended peers), and can probe logical pipes to maintain an accurate view of the pipe bottleneck MTU. This last aspect becomes necessary when logical pipes span multiple logical links with different MTUs, so the bottleneck may not be directly observable from the pipe endpoints.

**Our Implementation.** We have implemented the pipe terminus in hardware by using an OpenFlow switch. Arriving packets are de-encapsulated (using any encapsulation formats supported by the switch), and a next-header field extracted to determine the appropriate L3.5 protocol. In the case of the default L3.5 (IPv4), OpenFlow also performs IP prefix matching. For other L3.5s, the packet is

forwarded to an attached server where the software implementation of the L3.5 protocol is running, but first, an OpenFlow action encodes the input pipe ID into the packet as a VLAN tag so that the L3.5 implementation knows which pipe the packet arrived on. Packets arriving *from* an attached server have the VLAN tag set (by the server) to the desired *output* pipe ID; OpenFlow rules match on the pipe ID, perform the appropriate logical pipe encapsulation, and forward the packet out the appropriate physical port. All of our control code – which controls the OpenFlow switch, implements the Trotsky Neighbor Protocol, and automates configuration of the integrated L3.5 protocols – is written in Python and runs in user mode on a server attached to the switch.

### 5.2 Trotsky-Capable Hosts

Our prototype host code converts a standard Linux distribution into a Trotsky-capable one. This is done almost entirely via user mode code, some of which (re)configures standard Linux networking features (such as tap interfaces). Host support for Trotsky includes a pipe terminus and implementations of L3.5 protocols, which we do not describe because they are so conceptually similar to the case for TPs. Unlike TPs, hosts incorporate bootstrapping and a NetAPI, which we now discuss.

**Bootstrapping.** When a new host arrives on a domain, it needs answers to the following three questions: (i) Is this domain Trotsky-capable? (ii) What type of logical pipes does this domain support and what configuration parameters (if any) are required to use them? and (iii) What L3.5 protocols does this domain support? The Trotsky Bootstrap Protocol provides answers to these questions. The server side is implemented with a small, stateless server that must run on each L2 segment and responds to requests sent by hosts to a multicast Ethernet address. If the server answers with empty responses (*i.e.,* no pipe types and no L3.5 protocols supported), or if repeated requests produce no response, the host can conclude that the domain does not support Trotsky and can fall back on legacy IP bootstrapping (*e.g.,* DHCP). Otherwise, two things occur. First, the local pipe terminus is configured. In our prototype, the terminus is a userspace process that implements two types of logical pipe based on IP encapsulation as well as a VLAN-based one, and pipes show up as tap interfaces. Secondly, the supported L3.5s are configured, *i.e.,* DHCP is run over the logical pipe to acquire an L3.5 IP address, and the userspace L3.5 daemons are (re)started attached to the logical pipes.

As running a new bootstrap server on every L2 network may be impractical in the short term, a simplified version of the bootstrap protocol can be encapsulated in a DHCP option. On some DHCP servers, this can be implemented with only a configuration change; if the server code must be changed, the change is minimal. A modified DHCP client on the host can then send a DHCP request and use legacy IP or Trotsky configuration depending on whether the response contains the Trotsky bootstrap option.

**The NetAPI.** Within hosts, an API is needed to allow applications to utilize L3.5 protocols. Rather than design a new API, our prototype uses the standard Berkeley sockets interface for this. This choice is based on two factors.

First, several existing prototypes for new architectures (*i.e.,* L3.5 protocols, from a Trotsky perspective) already use APIs closely modeled on the sockets API. Second, as we imagine an ecosystem of L3.5 protocols, it makes sense to – as much as practical – share an API between them, such that software can be written that is agnostic to which L3.5 it is run on as long as it shares common semantics. This is, in fact, already an idea embraced by the sockets API; for example, much code can be written to be agnostic to whether it is run on a unix socket or an IP socket as long as it properly implements the semantics appropriate to its type (e.g., SOCK_STREAM). Moreover, architectures with different primitives can often be usefully mapped to common semantics (in Section 6.1, we briefly discuss how we mapped a common content-centric networking pattern – where a sequence of named chunks constitute a larger unit of data – as a streaming socket).[9]

To achieve this in our prototype, we undertook a FUSE-like [30] approach, by developing a kernel module which can redirect API calls on sockets of specified protocol families to a userspace backend. The backend then performs the actual operation. As mentioned above, it is typical for existing interdomain protocol prototypes to already offer a sockets-like API (*e.g.,* XIA's Xsocket interface), so the backend in such cases is straightforward glue code. This approach has a performance hit relative to kernel-based networking, but we use it only for protocols that do not have a kernel implementation (*e.g.,* there is no performance hit for IP).

## 6 EVALUATION

In this section we substantiate our architectural claims by: (i) describing how we were able to deploy a variety of architectures within Trotsky, (ii) how within Trotsky, applications can seamlessly use more than one architecture, and (iii) illustrating how a domain can deploy a new L3.5 design. We then provide performance numbers that show that (iv) Trotsky itself does not introduce significant performance overheads and (v) software forwarding is sufficient for simple L3.5s.

### 6.1 Implementing L3.5 Designs within Trotsky

In addition to the IPv4 L3.5 implemented using hardware via OpenFlow as described in Section 5.1, we also integrated two existing software implementations of new architectures, prototyped a third L3.5 from scratch, and investigated the implementability of several others. We discuss these here.

**NDN.** As discussed in Section 5.1, integrating an existing codebase as an L3.5 protocol is largely a matter of installing the existing NDN codebase adjacent to the pipe terminus, and directing the pipe terminus to forward packets to the NDN forwarding daemon when the next-header field corresponds to NDN. Additional "glue code" translates from the TP configuration to an NDN configuration file (crucially, this configures an NDN "face" for each logical pipe).

NDN usually provides its own unique API to applications. As discussed in Section 5.2, we favor the reuse of the sockets API when possible. To demonstrate this, we used our FUSE-like mechanism to map requests for an NDN *stream* socket to the NDN *chunks* protocol as implemented by the *ndncatchunks* and *ndnputchunks*

[9] While we focus on the sockets API in this paper, we admit that some L3.5s may well benefit from more than trivial additions to it. We have no objection to this if done with a focus on general semantics rather than low-level details, and we note that the IETF TAPS Working Group is already working along these lines (*e.g.,* in [35]).

tools maintained by the NDN project. The *chunks* protocol already functions much like a stream transport, providing a mechanism to download more data than fits in a single named packet while adding reliability and congestion control. This allows applications to simply open an NDN streaming socket, connect it to an address which is just the root name of some NDN chunks, and read data from the socket.

**XIA.** XIA is a flexible architecture that we discuss in Section 7. Much like the NDN use case, we incorporated XIA as an L3.5 design in Trotsky with minimal modification to the public XIA source code. XIA packets arriving at the TP pipe terminus are forwarded to the XIA daemons and from the XIA daemons back through the terminus to pipes. We made minor changes to the XIA code to account for running on logical pipes rather than directly on Ethernet (essentially disabling the XARP protocol). The "glue code" for our XIA L3.5 implementation leverages the Trotsky neighbor protocol to propagate configuration information between TPs.

**Trotsky Optical Protocol.** To illustrate how Trotsky can support non-packet L3.5 designs, we devised the Trotsky Optical Protocol (TOP) which supports the establishment and management of end-to-end (interdomain) optical paths. We assume that each domain participating in TOP has at least one *optical controller* (OC) that manages the optical switches of the domain. In addition, we assume there is a global *path computation engine* (PCE) that keeps track of all optical links, their wavelength (lambda) usage, etc., and computes paths in response to requests.

The TOP L3.5 implementation in TPs is simple. When a host requests the establishment of an optical path to a host in another domain, the host sends a TOP request, which reaches the TOP L3.5 implemention in a TP of the originating domain. The TOP L3.5 code contacts the domain's OC, which requests a path from the PCE. When the OC receives the path, it communicates the relevant portions to the domain's optical network, and also sends the path information to the TP that handles the peering relationship with the next hop domain. The TOP L3.5 implementation in that TP hands the path to the peer's TP, which requests a path from the OC in its domain. The process recurses until the entire path is established.

**Other potential L3.5 Designs.** For the aforementioned L3.5 designs, we either used existing codebases or (for TOP) wrote our own L3.5 design. To broaden our perspective, we also looked carefully at several other potential L3.5 designs, including IPv6, AIP [2], SCION [41], and Pathlets [13]. In each case, the porting of the design to Trotsky appeared to be straightforward. We did not actually do the porting because in some cases little was to be learned (IPv6), or codebases were not available (Pathlets and AIP), or lower layers in the current codebase were tightly coupled such that porting the code to use pipes would have been overly time-consuming (SCION).

### 6.2 Diversity

We modified the GNOME Web (aka Epiphany) browser to allow it to use both IP and NDN L3.5 designs simultaneously. This required adding an `ndnchunks:` URL scheme, which involved 156 lines of code, and which allows HTML documents to contain URLs of the form `ndnchunks:<ndn name>` (in addition to the usual `http://` and `https://` URLs). Such URLs can read NDN *chunks* published via the *ndnputchunks* tool (see Section 6.1). One immediate benefit of this is the ability to leverage NDN's caching benefits when retrieving static objects that today are often served via CDNs. Thus, a single application can utilize multiple L3.5 designs, leveraging the entire suite of available L3.5 functionality rather than being restricted to a single design.

### 6.3 Deploying a New L3.5

We performed functional testing using a software networking tool roughly similar to the well-known Mininet [21]. With this tool, we could create virtual networking environments composed of Linux network namespaces and Docker/LXC containers ([14, 23]) connected in arbitrary topologies by virtual network links. Within the nodes (*i.e.,* namespaces/containers), we can run software TPs, routers, the XIA and NDN protocol stacks, and so on, or simply emulate hosts. While the tool has low performance fidelity, it provided a flexible way to experiment with our prototype and its administration.

For example, by treating a group of nodes as a domain and configuring things accordingly (*e.g.,* choosing a common L3 protocol, configuring the TP nodes within the domain appropriately), we could role-play as the administrators of two domains deploying a new L3.5 design. Consider two peering domains *AS*1 and *AS*2 that both support the IPv4 L3.5 design, but *AS*2 is also running the NDN L3.5 design. *AS*1 wants to add support for NDN and add this to its peering with *AS*2. Here are the steps we take in our test environment to instantiate this support, which mirrors reality.

First, acting as the administrators of *AS*2, we alter the configuration of the *AS*2 TP adjacent to the peering logical pipe to allow NDN on it; this is the only action that must be taken for *AS*2. Acting as administrators for *AS*1, we then enable NDN for its end of the peering pipe as well as for other logical pipes internal to *AS*1. Following this, we install the NDN software on the *AS*1 TPs and reconfigure the pipe terminus to send NDN packets to it (in our prototype, software L3.5 code runs in containers, and we envision a standard container-based "package format" for L3.5 processing code in the future).

The final step is to update *AS*1's bootstrap daemons to advertise NDN as an available L3.5. After this, Trotsky-capable hosts attaching to *AS*1 see NDN as available and are able to make use of it (provided they also have NDN installed).

This is all it takes to deploy new L3.5s. In a large domain, changing configurations obviously poses additional challenges (*e.g.,* testing, training, etc.), but our point is that Trotsky removes all the *architectural* barriers to extensibility.

### 6.4 Overhead Microbenchmarks

We evaluated the performance overhead of the TP design presented in Section 5 for two different L3.5 designs: IPv4 (in hardware), and NDN (in software). We used another server as a traffic generator and compared two setups for each L3.5 design: with and without Trotsky. For Trotsky, we encoded the logical pipe next-header field using a VLAN tag.

Our experiments show no measurable difference in the throughput and latency with and without Trotsky for both L3.5 designs. For IPv4, in hardware, the roundtrip latency is $2\mu s$ in both cases and the link gets saturated ($10Gbps$ throughput). Using the NDN software, roundtrip latency is around $1.2 - 1.8ms$ and the median goodput is

$49Mbps$ in both cases. Thus, the presence of Trotsky does not impose significant performance overheads.

### 6.5 Viability of Software L3.5 Implementations

One might worry that software L3.5 implementations will always be too slow. To provide some context for this issue, we ran some software packet forwarding experiments using BESS [16] on an Intel® Xeon® E5-2690 v4 (Broadwell) CPU running at $2.60GHz$. We achieved roughly $35Mpps$ for IP forwarding on a single core; roughly $18Gbps$ for minimal-sized packets.[10] The server, which costs roughly $10,000, has 28 cores, so this is roughly $50Gbps$ per $1,000. While this is clearly less than one can achieve with a networking ASIC, the question is whether it is fast enough to be viable.

In terms of handling the load at a single interdomain connection point, the measurements above show that it would only take six cores to provide $100Gbps$ IP connectivity for min-sized packets. This is clearly feasible, but is the overall cost of provisioning cores at the edge too much? While there is little public data on the load entering a domain, the best information we can collect results in an estimate on the order of $10Tbps$. To handle this load with minimum-sized packets, we would need to spend roughly $200,000 on servers to handle the *entire* load entering the domain. This number contains many approximations (moving from min-sized packets to average sized packets would reduce the cost by an order of magnitude, considering more complicated L3.5 designs could add two or three orders of magnitude, etc.). But with the capital budgets of large carriers running into the billions of dollars (up to tens of billions), this number will likely be a tiny fraction of their overall costs. Thus, it is clearly viable for a domain to use software packet processing to handle their incoming load. Moreover, we envision that it is only the *initial* deployment of an architecture that is in software; once it becomes widely and intensely used, we assume hardware support will become available if needed.

## 7 RELATED WORK

While our work refers to the Internet architecture as having cleanly defined layers, the actual usage patterns of the Internet are far more complicated. A recent paper [40] notes that domains can use multiple levels of tunneling (*e.g.,* one can have multiple IP headers and multiple MPLS headers in a packet), and observes that it might be best to think of the Internet as an exercise in composing networks rather than being a simple set of four layers. We agree with this sentiment, but think that making the distinction between L3 and L3.5 remains crucially important in enabling an extensible Internet. Domains will presumably continue to use complicated sets of overlays to, for instance, carry wireless traffic over the public Internet, but this does not change our basic observation that separating L3 from L3.5 will allow new L3.5 designs to be incrementally deployed and for multiple of them to coexist.

There have been several other approaches to architectural innovation. The earliest, and most incrementally deployable, is that of overlay networks. However, in most overlay approaches the process of constructing the overlay network (and eventually transitioning from overlay to "native") is typically ad hoc. By contrast, in Trotsky the overlay is fully intrinsic. More precisely, of course one can use a purpose-built overlay to deploy a new clean-slate architecture. Trotsky is a framework in which the interdomain delivery is an intrinsic overlay that allows all such clean-slate designs to be deployed straightforwardly and simultaneously.

MPLS can be considered an *underlay*, which for some domains handles most intradomain delivery but leaves interdomain delivery to IP. However, in contrast to Trotsky, MPLS is strictly a mechanistic separation, with IP remaining as the universal L3 design.

Beyond overlays and underlays there are several interesting clean-slate proposals for architectural change. Active networking [34] innovated on the datapath, but did not address the issue of the NetAPI or interdomain interactions. Nebula [3] offers a great deal of extensibility in network paths and services, which is an important dimension. However, the core of the architecture (*i.e.,* the datapath) is universal within and across domains and therefore hard to change.

Plutarch [9] represents an entirely different approach to evolution, stitching together architectural contexts, which are sets of network elements that share the same architecture in terms of naming, addressing, packet formats and transport protocols. These contexts communicate through interstitial functions that translate different architectures. It is interesting to contrast this with the current Internet, which was founded on two principles. The first is that there should be a universal connectivity layer, so that in order to support $n$ different internal network designs (what we call L2 today) we would not need $n^2$ translators. The second is the end-to-end principle, which pushes (to the extent possible) intelligence to the edge. We feel that Plutarch runs counter to these two principles, requiring architectural translations within the network to achieve evolvability.

XIA [15, 24] enables the introduction of new service models by defining *principals*, and XIA itself defines a number of broadly-useful classes of identifiers. To cope with partial deployment, XIA relies on a directed acyclic graph in the packet header that allows the packet to "fall back" to other services that will (when composed) provide the same service. For instance, a DAG can have paths for CCN [18] and a source route via IP, with edges permitting intermediate routers to select either. Thus, XIA's approach to partial deployment (which is a key step in enabling evolution), much like Plutarch before it, is to require translations between architectures at network elements that understand both. In this respect, both Plutarch and XIA deploy new architectures "in series", and any heterogeneity along the path is dealt with by having the network explicitly translate between architectures. In contrast, in Trotsky, one simply uses any of the L3.5 designs mutually supported by both endhosts and their domains, which is an end-to-end approach. Naming systems can provide hints about how hosts and objects can be reached (*e.g.,* a name might be tied to one or more L3.5 designs), but the network does not try to translate between designs.

In terms of the many proposed architectures arising from Clean Slate research (such as NDN or SCION), none are incrementally-deployable general frameworks for deploying arbitrary architectures, so our goals are different from theirs. However, Trotsky could serve as a general deployment vehicle for all of them, so they need not explore deployment mechanisms on a design-by-design basis.

Sambasivan *et al.* observe that were it possible to make two clean-slate changes to BGP, it would then be possible to evolve routing

[10] We used IP forwarding because it is well-known and fairly simple. Obviously the performance numbers would be significantly worse for designs that required extensive cryptography or other computation. However, even for IPsec, one can achieve roughly $20Gbps$ per core with MTU-sized packets.

thereafter [32]. Their work highlights the value of enabling architectural evolution, and presents an alternative path to achieving it in the context of interdomain routing. ChoiceNet [31, 37] aims to improve the Internet's architecture by fostering a "network services economy", focusing on service offerings in a competitive marketplace; this could be complementary to Trotsky. The FII [20] proposal is also concerned with architectural evolution, though through a non-backwards-compatible clean-slate approach. In addition, it focuses largely on specific solutions to routing (specifically Pathlet routing [13]) and security, rather than general interdomain services. A similar but more general approach was taken in [12, 28].

Thus, to our knowledge, Trotsky is the first attempt to make a backwards-compatible change in the current architecture which then enables incremental deployment of radically different architectures (which need not be backwards compatible with any current architecture). Whatever one thinks about clean-slate design, the ability to make a single backwards-compatible change in the infrastructure (migrating to Trotsky) – one that is conceptually simple and mechanistically mundane – that then unleashes our ability to incrementally deploy new architectures is a significant step forward.

## 8 CONCLUSION

Our research was guided by six axioms, which we now review.

First: Trotsky's basic goal of architectural extensibility is *desirable*. This derives from (i) our belief that despite rapid innovation at higher layers (such as QUIC, SPDY, and the like) there are more fundamental architectural improvements that we would like to deploy – ranging from security mechanisms [2, 41] to ICN designs [18, 19] to mobility enhancements [33] to service-centric architectures [26] to more readily available Edge Computing [27] – and (ii) that there are currently high barriers to the deployment of these new architectures.

Second: Trotsky's approach is *novel*. In particular, we are the first to observe that making the interdomain datapath an inherent overlay over the intradomain datapath L3 would both (i) be incrementally deployable and (ii) enable the incremental deployment of radical new architectures. As such, Trotsky is not just a typical *ad hoc* overlay, but a structural rearranging of the Internet architecture to create an intrinsic overlay.

Third: Trotsky is *effective*. We have shown via implementation how Trotsky can seamlessly deploy a wide range of architectures and demonstrated that applications can leverage the resulting architectural diversity. Moreover, Trotsky can support any L3.5 design that could have been deployed as a clean-slate L3 design on routers, so there is no significant limit to its generality. While we cannot eliminate all operational barriers, Trotsky does eliminate the architectural barriers to extensibility.

Fourth: software packet processing, as currently being deployed in NFV and Edge Computing, is a key enabler for easy initial deployment. There is no question that software processing is far slower than hardware forwarding, but there is also no question that software packet processing is sufficient to handle the load at the edge for simple L3.5 designs, particularly when they are first deployed.

Fifth: *enabling architectural extensibility could greatly change the nature of the Internet*. Trotsky can transform the Internet not just because it can deploy multiple coexisting architectures, but also because it allows applications to use a diversity of architectures. This means that individual L3.5 designs need not meet the needs of all applications, only that the union of L3.5 designs can meet those needs in a way that is superior to today's Internet. This greatly lowers the bar on what new architectures must do in order to be a valuable part of the overall ecosystem.

Sixth: Trotsky may change the *incentives* surrounding architectural change. Our motivation in designing Trotsky is to show that some long-held beliefs about Internet architecture, such as the necessity of a narrow waist and the inherent difficulty of architectural evolution, are wrong. But Trotsky will remain merely an academic exercise if there are no incentives to deploy it, or to adopt new L3.5 designs once Trotsky itself is deployed. While the long history of architectural stagnation and lack of carrier innovation dampens our optimism, we do believe that now the incentives may be more favorable than in the past. The ISP business is rapidly becoming commodity, and carriers are desperate to offer new revenue-generating services. However, such services are of limited value if they can only be offered by a single carrier. Because L3.5 designs can initially be deployed in software running in the TPs, Trotsky provides a way for carriers to offer new services – in the form of new L3.5 designs – without endless standards battles and long hardware replacement cycles, and to easily peer with other carriers supporting those services. Only if such a design attracts significant traffic does the carrier need to contemplate more extensive support (in the form of hardware or internal deployment). Thus, Trotsky may play a crucial role creating the incentives for deploying new cross-domain services (in the form of L3.5 designs). In so doing, Trotsky will have created a permanent revolution in Internet architecture.

## ACKNOWLEDGEMENTS

This work owes a debt to a great many people. First off, we wish to thank our shepherd Marco Canini, our anonymous referees, and three undergraduates at UC Berkeley who helped bring this paper to fruition: Ian Rodney, Brian Kim, and Michael Dong. More fundamentally, this paper is a culmination of almost a decade's worth of research and reflection. Many other people have shaped the ideas presented here, through either their participation in related joint papers [12, 20, 28, 29] or in-depth discussions. While we take sole credit for any of the bad ideas in this paper, we gratefully acknowledge the following for contributing to whatever good ideas it contains: Hari Balakrishnan, Vjekoslav Brajkovic, Martin Casado, Nick Feamster, Igor Ganichev, Ali Ghodsi, P. Brighten Godfrey, Dirk Hasselbalch, Teemu Koponen, Nick McKeown, Sylvia Ratnasamy, Jennifer Rexford, Ankit Singla, Amin Tootoonchian, James Wilcox, and others. We also wish to acknowledge financial support from Intel, VMware, Ericsson, and Huawei, as well as NSF grant 1817115.

## REFERENCES

[1] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. 2001. Resilient Overlay Networks. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 131–145. https://doi.org/10.1145/502034.502048

[2] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. 2008. Accountable Internet Protocol (AIP). In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/1402958.1402997

[3] Tom Anderson, Ken Birman, Robert M. Broberg, Matthew Caesar, Douglas Comer, Chase Cotton, Michael J. Freedman, Andreas Haeberlen, Zachary G. Ives, Arvind

Krishnamurthy, William Lehr, Boon Thau Loo, David Mazières, Antonio Nicolosi, Jonathan M. Smith, Ion Stoica, Robbert van Renesse, Michael Walfish, Hakim Weatherspoon, and Christopher S. Yoo. 2013. The NEBULA Future Internet Architecture. In *The Future Internet - Future Internet Assembly 2013: Validated Results and New Horizons*. Springer, Berlin, Heidelberg, 16–26. https://doi.org/10.1007/978-3-642-38082-2_2

[4] Katerina Argyraki and David R. Cheriton. 2005. Active Internet Traffic Filtering: Real-time Response to Denial-of-service Attacks. In *Proceedings of the USENIX Annual Technical Conference (ATC '05)*. USENIX Association, Berkeley, CA, USA, 135–148.

[5] Steven M. Bellovin, David D. Clark, Adrian Perrig, and Dawn Song. 2005. A Clean-Slate Design for the Next-Generation Secure Internet. GENI Design Document 05-05. (July 2005). Report on NSF workshop.

[6] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC '12)*. ACM, New York, NY, USA, 13–16. https://doi.org/10.1145/2342509.2342513

[7] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. 2005. Design and Implementation of a Routing Control Platform. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 15–28.

[8] David Clark, Karen R. Sollins, John Wroclawski, Dina Katabi, Joanna Kulik, Xiaowei Yang, Robert Braden, Ted Faber, Aaron Falk, Venkata K. Pingali, Mark Handley, and Noel Chiappa. 2003. *New Arch: Future Generation Internet Architecture*. Technical Report. ISI. https://www.isi.edu/newarch/iDOCS/final.finalreport.pdf

[9] Jon Crowcroft, Steven Hand, Richard Mortier, Timothy Roscoe, and Andrew Warfield. 2003. Plutarch: An Argument for Network Pluralism. *SIGCOMM Comput. Commun. Rev.* 33, 4 (Aug. 2003), 258–266. https://doi.org/10.1145/972426.944763

[10] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures & Protocols (SIGCOMM '89)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/75246.75248

[11] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, K.C. Ng, Vyas Sekar, and Scott Shenker. 2013. Less Pain, Most of the Gain: Incrementally Deployable ICN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/2486001.2486023

[12] Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and James Wilcox. 2011. Intelligent Design Enables Architectural Evolution. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets-X)*. ACM, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/2070562.2070565

[13] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. 2009. Pathlet Routing. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/1592568.1592583

[14] Serge Hallyn, Stéphane Graber, Dwight Engen, Christian Brauner, and Wolfgang Bumiller. 2019. Linux Containers. https://linuxcontainers.org/. (2019).

[15] 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. 2012. XIA: Efficient Support for Evolvable Internetworking. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 309–322.

[16] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report UCB/EECS-2015-155. University of California at Berkeley.

[17] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. 2015. *Mobile edge computing – A key technology towards 5G*. White paper 11. ETSI.

[18] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. 2009. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1658939.1658941

[19] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. 2007. A Data-oriented (and Beyond) Network Architecture. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/1282380.1282402

[20] Teemu Koponen, Scott Shenker, Hari Balakrishnan, Nick Feamster, Igor Ganichev, Ali Ghodsi, P. Brighten Godfrey, Nick McKeown, Guru Parulkar, Barath Raghavan, Jennifer Rexford, Somaya Arianfar, and Dmitriy Kuptsov. 2011. Architecting for Innovation. *SIGCOMM Comput. Commun. Rev.* 41, 3 (July 2011), 24–36. https://doi.org/10.1145/2002250.2002256

[21] Bob Lantz, Brandon Heller, and Nick McKeown. 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-IX)*. ACM, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/1868447.1868466

[22] Taeho Lee, Christos Pappas, David Barrera, Pawel Szalachowski, and Adrian Perrig. 2016. Source Accountability with Domain-brokered Privacy. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 345–358. https://doi.org/10.1145/2999572.2999581

[23] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239 (March 2014), 2.

[24] David Naylor, Matthew K. Mukerjee, Patrick Agyapong, Robert Grandl, Ruogu Kang, Michel Machado, Stephanie Brown, Cody Doucette, Hsu-Chun Hsiao, Dongsu Han, Tiffany Hyun-Jin Kim, Hyeontaek Lim, Carol Ovon, Dong Zhou, Soo Bum Lee, Yue-Hsun Lin, Colleen Stuart, Daniel Barrett, Aditya Akella, David Andersen, John Byers, Laura Dabbish, Michael Kaminsky, Sara Kiesler, Jon Peha, Adrian Perrig, Srinivasan Seshan, Marvin Sirbu, and Peter Steenkiste. 2014. XIA: Architecting a More Trustworthy and Evolvable Internet. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 50–57. https://doi.org/10.1145/2656877.2656885

[25] David Naylor, Matthew K. Mukerjee, and Peter Steenkiste. 2014. Balancing Accountability and Privacy in the Network. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/2619239.2626306

[26] Erik Nordström, David Shue, Prem Gopalan, Robert Kiefer, Matvey Arye, Steven Y. Ko, Jennifer Rexford, and Michael J. Freedman. 2012. Serval: An End-host Stack for Service-centric Networking. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 85–98.

[27] Aurojit Panda, James Murphy McCauley, Amin Tootoonchian, Justine Sherry, Teemu Koponen, Syliva Ratnasamy, and Scott Shenker. 2016. Open Network Interfaces for Carrier Networks. *SIGCOMM Comput. Commun. Rev.* 46, 1 (Jan. 2016), 5–11. https://doi.org/10.1145/2875951.2875953

[28] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. 2012. Software-defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets-XI)*. ACM, New York, NY, USA, 43–48. https://doi.org/10.1145/2390231.2390239

[29] Barath Raghavan, Teemu Koponen, Ali Ghodsi, Vjeko Brajkovic, and Scott Shenker. 2012. *Making the Internet More Evolvable*. Technical Report. International Computer Science Institute. http://www.icsi.berkeley.edu/pubs/techreports/ICSI_TR-12-011.pdf

[30] Nikolaus Rath. 2019. libfuse: Filesystem in UserSpace. https://github.com/libfuse/libfuse. (2019).

[31] George N Rouskas, Ilia Baldine, Ken Calvert, Rudra Dutta, Jim Griffioen, Anna Nagurney, and Tilman Wolf. 2013. ChoiceNet: Network Innovation through Choice. In *2013 17th International Conference on Optical Networking Design and Modeling (ONDM)*. IEEE, Piscataway, NJ, USA, 1–6.

[32] Raja R. Sambasivan, David Tran-Lam, Aditya Akella, and Peter Steenkiste. 2017. Bootstrapping Evolvability for Inter-domain Routing with D-BGP. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 474–487. https://doi.org/10.1145/3098822.3098857

[33] Ivan Seskar, Kiran Nagaraja, Sam Nelson, and Dipankar Raychaudhuri. 2011. MobilityFirst Future Internet Architecture Project. In *Proceedings of the 7th Asian Internet Engineering Conference (AINTEC '11)*. ACM, New York, NY, USA, 1–3. https://doi.org/10.1145/2089016.2089017

[34] David L. Tennenhouse and David J. Wetherall. 1996. Towards an Active Network Architecture. *SIGCOMM Comput. Commun. Rev.* 26, 2 (April 1996), 5–17. https://doi.org/10.1145/231699.231701

[35] Brian Trammell, Michael Welzl, Theresa Enghardt, Gorry Fairhurst, Mirja Kühlewind, Colin Perkins, Philipp S. Tiesel, and Christopher A. Wood. 2019. *An Abstract Application Layer Interface to Transport Services*. Internet-Draft draft-ietf-taps-interface-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-03 Work in Progress.

[36] Yuefeng Wang, Ibrahim Matta, Flavio Esposito, and John Day. 2014. Introducing ProtoRINA: A Prototype for Programming Recursive-networking Policies. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 129–131. https://doi.org/10.1145/2656877.2656897

[37] Tilman Wolf, James Griffioen, Kenneth L. Calvert, Rudra Dutta, George N. Rouskas, Ilya Baldin, and Anna Nagurney. 2014. ChoiceNet: Toward an Economy Plane for the Internet. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 58–65. https://doi.org/10.1145/2656877.2656886

[38] Xiaowei Yang, David Clark, and Arthur W. Berger. 2007. NIRA: A New Inter-domain Routing Architecture. *IEEE/ACM Trans. Netw.* 15, 4 (Aug. 2007), 775–788. https://doi.org/10.1109/TNET.2007.893888

[39] Xiaowei Yang, David Wetherall, and Thomas Anderson. 2008. TVA: A DoS-limiting Network Architecture. *IEEE/ACM Trans. Netw.* 16, 6 (Dec. 2008), 1267–1280. https://doi.org/10.1109/TNET.2007.914506

[40] Pamela Zave and Jennifer Rexford. 2019. The Compositional Architecture of the Internet. *Commun. ACM* 62, 3 (Feb. 2019), 78–87. https://doi.org/10.1145/3226588

[41] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. 2011. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. IEEE Computer Society, Washington, DC, USA, 212–227. https://doi.org/10.1109/SP.2011.45