Making the Internet More Evolvable

Barath Raghavan, ICSI
Teemu Koponen, Nicira
Ali Ghodsi, UC Berkeley
Vjeko Brajkovic, ICSI
Scott Shenker, ICSI/UC Berkeley

Abstract

Why is the Internet so hard to evolve? Some argue that we require a radically different architecture to enable evolution. To the contrary, we contend that a simple re-engineering of the basic Internet interfaces to make them more modular and extensible—as one would in any software system—is sufficient to produce a far more evolvable Internet. We describe our design, called Omega, and report on its implementation and evaluation on PlanetLab.

1 Introduction

The software community has long recognized the value of building large codebases in a way that allows them to evolve over time. To this end, programmers first identify a durable modularity around which to build their system, and then realize this modularity with a set of abstract and extensible interfaces. The evolvability of well-designed software stands in stark contrast to the rigidity of the Internet architecture. While new applications and new networking technologies are easily deployed, the architectural core—by which we primarily mean IP, but also include BGP and DNS—is extremely hard to change. For example, the move from IPv4 to IPv6, which is little more than a change in the header format and addressing scheme, is still ongoing after more than a decade. This bodes ill for the deployability of any architectural proposal emerging from the Internet research community.

As a result, the research community has recently begun focusing on how to foster architectural evolvability. We discuss this literature later in the paper, but here we note that the most prominent proposals [4, 10] require the network to translate between different architectures. In contrast, we propose an approach that is more similar to how software systems evolve: a modular design with extensible interfaces that allows evolution of functionality while maintaining backwards compatibility. As such, our approach involves almost no novelty. However, if the solution is so readily at hand, one might ask why we have had such a hard time evolving the Internet architecture. We believe that this is because of two fundamental "mistakes" made in the early days of the Internet.[1]

First, while the Internet architecture is built around a layered modularity, the actual interfaces between these layers are poorly engineered, undermining that modularity. For instance, TCP uses IP addresses to identify connections, whereas a well-engineered interface would separate the concept of a connection from that of an address. In addition, some interfaces are not extensible, so new functionality precludes backwards compatibility (*e.g.*, `gethostbyname` presumes the name is a DNS name and the response is an IP address). Most of our design involves fixing these interface problems, in ways that would be completely familiar in a software context.

Second, the notion of "architecture" actually involves two separate concepts: (i) the set of components that everyone must agree on, and (ii) the set of components needed to successfully transmit data end-to-end. The Internet community conflated these two, building into IP all the functionality needed to provide end-to-end delivery. In contrast, 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.

We emulate this microkernel approach by not defining a new monolithic Internet "architecture" and instead only specifying a more minimal architectural "framework" (or micro-architecture) called Omega that enables a variety of fully-functional (or comprehensive) architectures to be defined within that framework. Each of these comprehensive architectures must augment the basic framework with various components, and it is these additional components that can evolve over time. As with microkernels, Omega faces dual design challenges: Omega must (i) define as little as possible to allow the maximal degree of

[1] It is easy to label these as mistakes in hindsight, but at the time these decisions were made, merely making the system work was a small miracle.

evolvability, and (ii) define enough so that one can successfully use architectures built within the Omega framework. We will call the first goal *extensibility* and the second goal *sufficiency*. Our current Internet is sufficient but not extensible, while defining nothing would be extensible but not sufficient. Here we seek something that satisfies both goals, and indeed this was the challenge faced in building microkernels.

Meeting these dual challenges requires four key components: (i) an extensible interdomain service model (ISM) interface that allows for parallel deployment of multiple interdomain architectures, (ii) an extensible network API that enables OS support for various parallel network services, (iii) the separation of interdomain and intradomain addressing to preserve modularity, and (iv) a variety of measures to help hosts cope with the resulting architectural heterogeneity.

Evaluating the evolvability that Omega provides is challenging. To this end, we have implemented a prototype of Omega and have implemented several architectures, including BGP, Pathlet routing [9], and DONA [12] on top. We have also implemented several transport protocols, which can be used by native and legacy applications to communicate using different architectures that co-exist on top of Omega via the appropriate APIs.

To further verify the evolvability of Omega we defined a simple language to express architectural evolution and used it to test several scenarios involving architectural changes with our prototype running on PlanetLab. In addition, to demonstrate the feasibility of deploying Omega itself, we built another implementation of Omega's routing components using Open vSwitch and describe how we can leverage SDN and IPv6 to deploy Omega incrementally.

Assumptions and Clarifications. Before describing our design, we must first define our terms and clarify some of our assumptions. When we use the term *architectural evolvability*, we are referring to the ability to make fundamental changes in the entire architecture. Moreover, we want these changes to be relatively easy and frequent rather than extremely difficult and rare. *In short, we want an Internet where architectural innovation can be ongoing and pervasive.* 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.

There are many actors involved in adopting an architectural change, including domain operators (*e.g.*, ISPs), router/switch vendors, OS vendors, application vendors, and application service providers (*e.g.*, CDNs, Google, Facebook). We assume that all of them, with varying degrees of nimbleness, will implement the software changes needed to deploy a new architecture[2], but we also assume that there will always be legacy systems. Hardware changes will be slower in coming, and we will discuss the role of hardware in architectural evolution later. Lastly, domain operators are probably the most conservative actors in this list, as the penalties for malfunctions are higher than the rewards for new functionality, so any change that requires simultaneous adoption by all domains faces a daunting deployment barrier. We also assume that requiring widespread deployment of middleboxes to translate between architectures is a significant barrier to architectural change.

With these preliminaries in hand, we now turn to the Omega design asking first how to make it extensible and then how to make it sufficient.[3]

2 Making Omega Extensible

We address Omega's extensibility by considering various kinds of architectural changes that are hard to effect today and describe how Omega can make such changes easier through relatively minor architectural modifications.

Changing the network API and naming. We start with the network API (hereafter, netAPI) offered by the host OS to applications, which is currently hard to change because it is embedded in applications. This can be avoided by merely providing a layer of indirection—essentially a netAPI identifier—so that all calls first specify which version and flavor of netAPI they want. In fact, some systems already support limited forms of this indirection (*e.g.*, protocol families).[4] It is easy for OS vendors to support multiple netAPIs, and the set of netAPIs on one host need not be identical with those on another, so little

[2]Note that when we say "deploy" we mean that they will *add* support for these newer protocols, but will retain the ability to use the older protocols (which will remain as the default for quite a while).

[3]For convenience, our discussion refers to packet-based designs, but our discussion easily generalizes to non-packet technologies (*e.g.*, optical) by equating the contents of the packet headers with the information conveyed by the signaling mechanism in the non-packet technology.

[4]While it is possible to implement new APIs in user-level libraries, there are two issues with such an approach: fragmentation of functionality and of responsibility. First, building multiple parallel network libraries creates the problem of interoperability; instead, netAPI operates at a different, lower level and provides uniform access to a range of APIs and underlying implementations. Second, introducing new APIs in user-level libraries while protocols and ISMs remain in the kernel divides the responsibility for the codebases between distinct parties; with our approach, it is netAPI developers who are responsible for both interfaces and implementations.

coordination is needed to deploy a new netAPI with this layer of indirection.[5]

Currently DNS names are embedded in applications; this could be avoided by having all applications (as some do now) treat names as semantic-free bits and only handle them via naming operations implemented in the network stack (such as `gethostbyname`). We impose a standard naming syntax where a namespace identifier is followed by the bits representing the name, so the stack can recognize and appropriately handle these names. The stack can support multiple namespaces simultaneously so, as with the netAPI, one can add additional namespaces without revoking old ones.

Changing IP. Our experience with IPv6 suggests that changing IP (*i.e.*, L3) is hard, but this can be overcome with two simple design decisions: (i) we require that all netAPIs pass names, not addresses, so applications are shielded from changes at L3; and (ii) we separate intradomain addressing from interdomain addressing. Thus, a full destination address consists of an interdomain part (a universally-agreed-upon set of domain identifiers) and an intradomain part (which need not be understood by any domain other than the destination domain, see [1]). The interdomain identifiers are the only form of global addressing in Omega, and they are not limited to any particular protocol, so domains can peer with a variety of technologies and layers (such as L3, optical, MPLS, or Ethernet). Separating interdomain from intradomain addressing, by enabling general forms of peering, not only makes changing the L3 protocol easy, it eliminates the concept of a universal internetworking layer altogether.

Changing forwarding. Packet forwarding is typically done by protocol-specific ASICs, so changes to L2/L3 packet headers require an upgrade to a domain's hardware infrastructure. However, we are cautiously optimistic that this deployment barrier will lessen in importance over time, for the following three reasons. First, much of the architectural action is not on the data path itself, but at higher layers or on the control plane, so the need for forwarding changes is less urgent. Second, the OpenFlow forwarding model, which is rapidly gaining traction, provides more flexibility on the datapath, and that flexibility is expected to increase over time. Third, the "fabric" model of forwarding (such as Qfabric from Juniper) puts most of the protocol burden at the edge, where processing can be done with general-purpose CPUs.

Changing interdomain routing and the interdomain service model. Today the Internet's various domains are tied together by BGP, and changing it would require universal adoption by all domains. We overcome this barrier in three ways. First, the use of extensible routing designs lessens the need for broader architectural changes; for instance, the design in [9] supports arbitrary and potentially external route computations, a flexible policy model, innovations in the services offered along paths (such as QoS, middlebox services, monitoring), multipath routing, and flexibility in peering technologies.

Second, we enable partial deployment of new interdomain routing systems, running along side the current one, by having (i) packets carry a label that indicates which routing system computed the route, with the rest of the header understandable only by routers that participate in that particular routing system, and (ii) the routing systems expose which domains are reachable via that routing system (as BGP does today). In this way, whomever is choosing the route (end host, domain, external route computation agent, etc.) can inspect the various routing systems to find one that provides connectivity.

Third, we note that once we insert this level of indirection (through the label in the packet), the service offered by a new "routing" system need not resemble traditional routing; for instance, such a mechanism might offer a pub/sub interface. Thus, Omega's general approach enables the incremental introduction of new *interdomain service models* (ISMs); new ISMs would be invoked by upgraded applications via new netAPIs, allowing legacy applications to use their previous ISMs through their previous netAPI. Of course, the functionality of an ISM is only available to those domains supporting that ISM.

So far we have only specified an extensible syntax for names and the netAPI, inserted an ISM identifier in packets, and required that the netAPI pass names, not addresses. This leaves the rest of the architecture free to employ arbitrary naming systems, netAPIs, and ISMs. While this certainly provides a high degree of extensibility, we must now ask how to make this minimal framework sufficient to support comprehensive architectures.

3 Making Omega Sufficient

When considering Omega's sufficiency, there are three main areas of concern. Given all the freedom architectures have within the Omega framework, Omega must enable applications and hosts to discover what their options are. As this freedom allows domains to adopt different designs, Omega must help hosts cope with this heterogeneity. And, of course, we must consider what aspects of security must be built into Omega.

Discovery. In order to take advantage of the variety of

[5]Note that two communicating hosts can use different netAPIs, as long as the implementing protocols they use are compatible.

Init	**Bootstrap**(DomainTech #, Address, ISM # $^{+}$) $\rightarrow$ Domain #, (DomainTech #, Address, Directory, Aux), ISM Num^{+}

Figure 1: Bootstrap API. DomainTech # refers to the domain technology, Address refers to an intradomain address, ISM # $^{+}$ refers to a list of possible ISMs, Domain # refers to the domain's identifier, Directory refers to the location of the resource directory, and Aux refers to any auxiliary information.

Init	**Metanegotiate**(Negotiation # $^{+}$) $\rightarrow$ Negotiation #

Figure 2: Metanegotiation API. Negotiation # $^{+}$ refers to a list of possible negotiation protocols.

Component	Description
netAPI	API identifier
	Pass names not addresses
	Bootstrap API
Addressing	Separation of intra-/inter-domain identifiers
ISMs	ISM identifier in packet header
	DoS interface
	Path visibility (not a strict requirement)
Domains	Resource directory
	Support for bootstrap interface
Hosts	Metanegotiation protocol

Table 1: Summary of the Omega design.

netAPI semantics available, applications should be able to query the netAPI to determine which netAPI identifiers are supported by that OS. Similarly, the domain must provide a *resource directory* so that the host can determine the location of critical services (such as name resolvers).

In addition, when a host boots up in a new domain, it must discover the properties of its environment. To this end, the netAPI must include a Bootstrap API (see Figure 1) that enables a bootstrap daemon (or the equivalent) to collect this information. The host stack must interact with the communication technology in the local domain via compatible software in the stack. The domain technology (and the corresponding stack software) must implement the bootstrap interface (but the means of implementation can be technology-specific).

More specifically, we define the bootstrap interface as follows: when initiating the bootstrap, the host optionally specifies which intradomain technology it would prefer, an address it would like to be assigned within the domain, and a list of ISMs the host would like to use. The domain returns its domain identifier, a tuple containing essential intradomain information—including the intradomain technology (either the one requested, or another), the host address (for that technology), the resource directory address, and any auxiliary information—and a list of supported ISMs (a subset of those requested by the initiating host). The resource directory allows the host to find local facilities such as name resolution servers, first hop routers for the various ISMs, and other resources needed to function. The bootstrap interface could also return auxiliary information about the domain in an extensible manner, but the content above is the minimal necessary for the host to function within the domain.

Heterogeneity. Two communicating hosts must use compatible protocols, and they should be able to negotiate this set of protocols (as in [6]). Omega does not specify the negotiation protocol, only a metanegotiation protocol (see Figure 2) that allows two hosts to decide on a negotiation protocol to use, allowing the set of negotiation protocols to evolve over time. During metanegotiation, the initiating host (if it has no prior information about the protocols supported by the other host) sends a list of supported negotiation protocols, and the receiving host returns a single one (which the initiating host then uses to begin the negotiation process). Metanegotiation is a last resort: if two hosts have recently communicated, or if the name resolution process provides the supported protocols (or negotiation protocols) in metadata, then it is unnecessary.

Another area where heterogeneity becomes an issue is in path properties; different paths will support different QoS features or router-assisted congestion control mechanisms. While Omega does not mandate a specific interface for this, it is recommended that any ISM provide a reasonable degree of visibility and choice, so that hosts can pick paths that meet their needs. If the ISM allows the equivalent of source-based policy-compliant route computation, then a new capability can be used if *any* policy-compliant path supports it (as opposed to today where one can use the capability only if the default path supports it).

Security. Many have argued (see [1, 13, 15]) that, except for availability, *all* aspects of network security (narrowly construed) can be implemented at the end hosts. Preserving availability falls into two categories: dealing with attacks on the infrastructural components (such as hacking a router or injecting false routing advertisements), and dealing with denial of service attacks. Each architecture is responsible for protecting itself against infrastructural attacks, but since denial-of-service attacks can be launched across architectures we require that every new ISM (whether routing for traditional destination-based packet delivery, or new service models like optical) provide an interface that ensures protection against denial of service. For traditional packet delivery, this interface could *enable delivery* (for capability-like approaches to DoS, such as in [16]) or *prevent delivery* (for filter-based approaches to DoS, such as in [3, 13]). Thus, for a domain to support a new ISM, it must support both the delivery *and* DoS interfaces.[6] While Omega does require that ISMs support a DoS interface, it does not specify the nature of that interface.

Comments. Note that here we are not asking what additional components are needed to build a comprehensive architecture, but what features must be built into Omega itself, and not left to individual architectures; Table 1 provides a summary of Omega's key components. The ability to discover what architectures are available locally, and to tolerate the inevitable diversity in architectures, is clearly needed within Omega itself, as is a mandate to provide some form of DoS interface in every ISM. What is not needed, however, is some explicit agreement about how architectures share resources (though this is an objection we frequently hear); each domain that is deploying multiple designs in parallel (such as supporting multiple congestion control schemes in their routers) can independently determine how to allocate their resources among them (*e.g.*, by assigning relative bandwidth shares to traffic classes using different congestion control schemes); there is no need for a global agreement on this.

Init	**Open**(Schema, Resource, Options) → Handle **Listen**(Schema, Resource, Options) → Handle
Access	**Put**(Handle, Data, Options, Aux. Data) → Result **Get**(Handle, Options, Aux. Data) → Data **Accept**(Listen Handle, Options) → Handle
Meta	**Control**(Handle, Mode, Option) → Result
Cleanup	**Close**(Handle) → Result

Figure 3: Sockets API.

Init	**Subscribe**(Schema, Resource, Options) → Handle **Publish**(Schema, Resource, Data, Opt.) → Handle
Access	**Get**(Handle, Options, Aux. Data) → Data
Meta	**Control**(Handle, Mode, Option) → Result
Cleanup	**Cancel**(Handle) → Result

Figure 4: Pub/Sub API.

Init	**Client**(Resource, Options) → Handle **Server**(Resource, Options) → Handle **Register**(Server Handle, Function) → Result
Access	**Invoke**(Handle, Function, Options) → Result, Token **GetReply**(Handle, Token, Options) → Result, (Data) **Process**(Server Handle, Options) → Result
Meta	**Control**(Handle, Mode, Option) → Result
Cleanup	**Close**(Handle) → Result

Figure 5: RPC API.

4 Using Omega

To make our discussion more concrete, we next look at a few example interfaces, and then trace the life of a packet.

4.1 Example Interfaces

In Figure 3 we give our version of the Sockets API, which is quite similar to BSD sockets. Our implementation of netAPI interfaces such as the Sockets API rely upon *schemas*, which are classes of protocols that provide the same functionality. For example, in our C++ implementation of Omega, an application can use the Sockets API to open an unreliable socket by calling `Socket::Open(SCHEMA_DATAGRAM_UNRELIABLE, name, options)` where `name` is the name of the remote host and `options` is used by the schema to select and configure the underlying protocol.

An application can also publish content using the Pub/Sub API we implemented (Figure 4), which interfaces with our DONA ISM. We also built a simple RPC interface (Figure 5) that lets applications ignore the underlying protocols in use. The interface provides high-level semantics—that of function calls and arguments—and hides details of ISMs and schemas within the stack. Such high-level constructs are commonplace; we merely wanted to demonstrate that Omega can accommodate them in its netAPI model.

In Figure 6 we give a definition of an ISM providing best-effort packet delivery, as might be provided through traditional IP interdomain routing using BGP or through a scheme such as pathlet routing [9], both of which we implement as backends for this ISM interface. This interface includes capability-like security primitives RequestToSend and AllowToSend. Figure 7 describes the DONA interface, which is similar to many other ICN designs (each of which uses its own terminology for the same set of calls). Note that there is no security portion of the interface, as such publish/subscribe models do not suffer from standard denial-of-service flooding attacks. Figure 8 describes the interface to an ISM that provides end-to-end optical connections (without any packetization); the signaling mechanism supports the RequestToSend and AllowToSend calls, which must precede any transmission and provide inherent support for selectively allowing or denying traffic.

[6]Of course, there are many other aspects of security that do not fit within the narrow definition we adopt here (see [13] in particular for an explicit discussion of this), but Omega's role in security is not to define particular security mechanisms but to allow the Internet to adopt the necessary mechanisms through architectural evolution. The one place where we place a security requirement on Omega directly is to require all ISMs to incorporate their own DoS interfaces.

Init	**SelectRouting**(Name, Filter) → Result
Access	**IsReachable**(Name) → Result **SendPayload**(Name, Payload) → Result
Security	**RequestToSend**(Name) → Result, (Capability) **AllowToSend**(Sender Spec) → Result

Figure 6: Packet Delivery ISM.

Init	**SelectRouting**(Name, Options) → Result
Access	**Find**(Name, Options) → Result, Data **Register**(Name, Data, Options) → Result

Figure 7: DONA ISM.

Init	**RequestToSend**(Name, Filter) → Result **AllowToSend**(Sender Spec) → Result
Access	**IsReachable**(Name) → Result **SendPayload**(Name, Payload) → Result

Figure 8: Optical ISM.

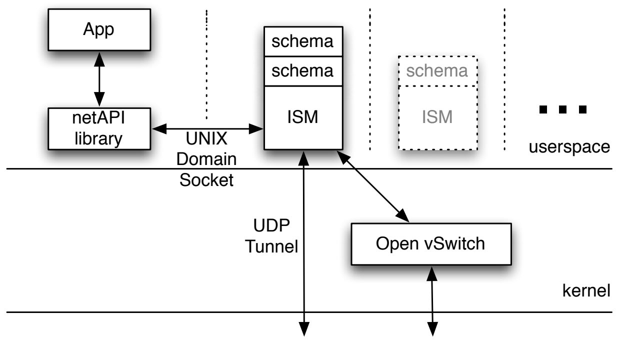

Figure 9: Omega prototype implementation, end host view with an application and ISM in use. UDP encapsulation of traffic is used in PlanetLab tests, whereas Open vSwitch is used with our IPv6 variant. The grayed ISM highlights the fact that multiple ISM daemons co-exist and have their own appropriate schemas.

4.2 Life of a Packet

All of our previous discussion focused on individual aspects of Omega's design. Here we try to put them together by describing the life of a packet. Consider host X in domain A communicating with host Y in domain B using a packet-based ISM Q. When the packet leaves X, its outer headers reflect A's internal technologies (say IP over Ethernet), followed by the Q-specific header; when the packet arrives at the first-hop-router for ISM Q, the outer headers headers are removed (leaving the Q-specific headers) and replaced, if needed, by headers the ISM Q routers use to communicate hop-by-hop; when the packet arrives at the last-hop router for ISM Q, these outer headers are replaced by headers appropriate for B's internal technologies.

By keeping intradomain and interdomain addressing separate, Omega allows domains to change their internal technologies without coordination. While today's Internet got some interfaces wrong, the L2 interface did indeed have the right modularity, being shielded from all applications and other domains. Thus, Omega does little more than to turn domain-internal technologies into what we now think of as L2. In Omega, the ISMs play the role of L3, but by carefully ensuring extensibility of the various interfaces, Omega allows many of these ISMs to coexist side-by-side.

5 Omega in Action

We have argued that Omega enables evolution but, given the amorphous nature of Internet architecture, we have no way of making this argument rigorous. To provide some concrete evidence, we implemented Omega and then used it on PlanetLab to verify its support for architectural evolution. In the following, we first describe our implementation and then several of our test scenarios. We return to the deployment of Omega itself in Section 6.

5.1 Prototype Implementation

Our objective was to implement Omega's interfaces and sufficient back-end mechanisms to test the framework in the context of varied architectures. Since our aim is not to faithfully reproduce the details of various mechanisms of the architectures used in these scenarios (all of which have been implemented in other contexts), but rather to evaluate Omega's support for evolution, the mechanisms we built are minimal.[7] That is, we built the full-fledged Omega framework and only sufficient but minimal architectures and architectural components to live within it.

Our implementation (which we will make available for others to use) executes at user level on Linux and Mac OS X and includes components to model the application-side APIs which link to applications, daemons to model ISMs, and daemons for routers and gateways. The end host view is depicted in Figure 9; routers and gateways are simply standalone daemons. We also implemented a library interposition-based wrapper to enable simple unmodified BSD Sockets-based applications to run over Omega. Since our aim was to test our implementation on today's networks and on PlanetLab, for the scenarios we describe in this

[7]Specifically, we have avoided implementing complex control planes for protocols that would use them in the wild — *e.g.*, gossip-based route dissemination for pathlet routing —and instead have encoded such control information in the scenarios themselves.

Keyword	Arguments	Description
routervendor	*name feature list*	Specifies a common set of router features.
hostvendor	*name feature list*	Specifies a common set of host features.
domain	*name feature/setting list*	An abstract entity containing hosts, gateways, and routers.
router	*name vendor domain setting list*	An interdomain router entity.
gateway	*name domain*	An intradomain gateway entity.
host	*name namespace vendor setting list*	A host entity.
instance	*name host specification list*	An application instance running on a host; specifies scenario.
connect	*entityname entityname*	Designates the logical or physical connection of two entities.
start	*instancename*	Starts execution of the scenario.
wait	*seconds*	Pauses execution of the script.
wait	*instancename list*	Creates an execution barrier until instance(s) complete(s).
ism	*ismname setting list*	Specifies settings or policies for a given ISM.

Table 2: A concise specification of Evol as used in our architectural evolution scenarios.

section, Omega packets are encapsulated within UDP packets. For controlling packet forwarding of Omega built on IPv6, we implemented a simple OpenFlow controller for Open vSwitch, which we return to in Section 6. Our Omega implementation consists of over 10k lines of C++ and 1200 lines of Python, not including protocol definitions or third party libraries.

At a high-level, the prototype has two key components: (i) an application-facing library that provides the interfaces to (ii) schemas and ISM daemons that can be considered as part of the network stack. The two sides interface via IPC calls, modeling system calls.[8]

Each ISM exposes its functionality as appropriate to its design; for example, a packet-oriented ISM such as BGP exposes functionality to send packets to a given destination and the ability to determine whether a destination is reachable. The content-oriented ISM exposes functionality only to publish or subscribe to content by name, with no packet sending primitives. Coupled with each ISM are schema implementations appropriate to that ISM; they provide the translation between the API primitives and the ISM primitives. For packet-oriented ISMs like pathlet routing and BGP, we implemented schemas that provided basic but useful functionality—an unreliable datagram schema similar to UDP and a reliable, congestion-controlled byte stream similar to TCP; for our DONA ISM we implemented a schema to publish and subscribe to content. We also implemented a basic meta-negotiation protocol as described earlier.

We implemented the routers and gateways as standalone daemons that communicate via UDP tunnels with other nodes (both with end host nodes and other router / gateway nodes) but operate largely in the same manner as the end host ISM daemons. The primary distinction between an Omega router daemon and an end-host ISM daemon is that Omega router daemons (i) first multiplex between ISM types before passing a packet to the relevant ISM module, (ii) do not implement schemas (since these are handled at the end host), and (iii) have support for on-path operations that are required (*e.g.*, for the DONA ISM, routers track content requests that have been made and cache content along the path). The gateway daemons are somewhat simpler; their primary function is to engage in a bootstrap protocol with end hosts, conveying supported ISMs, default routing information, and name resolution configuration for end hosts.

5.2 Evolution Scenarios

With this implementation in hand, we then sought to explore different scenarios that test the evolvability of Omega. To run these tests, we defined a simple architectural evolution language, *Evol*, and an interpreter for it, implemented in Python.[9] Each scenario controls the actions of numerous instances of our implementation that perform the actual communication as well as the architectural evolution that may occur as a scenario proceeds. The language adheres to several constraints, including: hosts must connect to gateways; gateways must connect to routers or hosts; routers must connect to routers or gateways; if an entity name is given again with a new feature list, that constitutes an architectural upgrade of that entity; and "wait" pauses execution of the scenario until the given instance terminates to allow for an ordered sequence of architectural transitions to test evolution. Vendor feature lists denote support for the given technologies while for

[8]In a full-fledged implementation, we would expect that the application-side library would reside in a standard shared library (and its associated header files), the daemon would be replaced by modules within the OS kernel, and IPC would be replaced by system calls.

[9]We don't claim *Evol* as a contribution; we simply needed a language to assist the verification and defined one that met our needs.

	ISMs				
Scenario	Pathlets	BGP	DONA	Optical	**Notes**
Basics	●				Communication between two compatible pathlets-based hosts in two different domains.
Address Format	●	●	●		Communication between domains with different address formats (IPv4 and flat addresses).
Path Selection	●	○			BGP is stuck with the default path which violates the given path requirement (medium topology).
Pub/Sub API			●		Content published and subscribed to/from network.
No DONA ISM			○		Applications attempt to use Pub/Sub API without domain support for DONA.
Optical				●	Communication between domains after path establishment.
Security	●				Destination host allows only from sender's domain.
Several Changes	●	●	●	●	Transition from BGP-only adding an ISM per step (medium topology).

Table 3: A subset of Omega evolution verification tests run on PlanetLab; for each scenario we denote which ISMs were in use and indicate whether the scenario succeeded (●) or failed (○).

entities denote the enabling of those features. We describe the main keywords of Evol in Table 2 to give a flavor of the types of evolutionary changes we can test.

We now discuss several scenarios we ran using our Omega implementation on PlanetLab to test various aspects of evolution. Table 3 summarizes the results.

Basics. In this scenario we defined an end-to-end communication test between two hosts within different domains, each of which uses IPv4 as its intradomain architecture, pathlets [9] as its ISM, and Omega Sockets as its API. There is no evolution involved and the scenario merely verifies the hosts' ability to exchange packets. This two host, two domain topology acts as our base topology. However, for several scenarios, we defined a topology of eight domains (six stubs and two transit) and their constituent hosts. We label it as the *medium topology* and indicate in Table 3 when the scenario used it. Because these scenario definitions are verbose (the medium topology scenarios being over 70 lines), we do not show the definitions we used, but their structure and content should be clear from our descriptions.

Address Format. Here we defined a scenario in which one domain evolved its internal addressing scheme from IPv4 to semantic-free flat addresses while no other domains were aware of this change. A host outside this changing domain was able to communicate with a host inside this domain before and after the transition.

Path Selection. Here we ran a series of communication tests where hosts required a specific property from their end-to-end path (*e.g.*, in-network congestion control, geographic region, and QoS properties). In one version of the checks we performed this request using the BGP ISM, which does not support flexible path selection. When the default path does not have the appropriate property, the hosts are not able to communicate. We then considered a case in which the domains along the path evolved their ISM support to include pathlets; here the hosts were able to find a path with the appropriate properties. The success of this is clearly due to the choice of routing system—that pathlets enables intelligent endpoint path selection—and not due to Omega, but it is Omega here that enables pathlets and other new ISMs to be incrementally deployed.

Publish/Subscribe API. We ran a scenario where hosts, routers, and the relevant domains evolved to support DONA [12]. With these changes, hosts were able to publish and subscribe to content in a new DONA-content namespace using the publish/subscribe API and the DONA ISM. The main takeaway is that the deployment of DONA required no modification of non-DONA code since Omega enables easy multiplexing on the ISM, API, and protocol. To confirm that our implementation and scenarios are indeed capable of failing when deploying a new ISM, we performed a version of this experiment in which a host attempts to use the publish/subscribe API and DONA ISM before its domain's routers support it.

Optical. We similarly tested a scenario that deployed an optical-like ISM (*i.e.*, an ISM that requires signaling for in-network path setup before end-to-end communication can begin) which was then used by hosts to communicate.

No changes to applications, or to nonparticipating domains, were necessary for this to work as only domains along the path were required to establish state and end-system stacks (not applications) were required to initiate path setup.

Security. Our pathlets implementation integrates a capability-based mechanism to allow hosts to issue capabilities for inbound packets. If the capabilities are invalid, the associated packets are dropped at the border of the domain by the ingress pathlets router. In this scenario we used pathlets and verified hosts were able to block access to themselves using capabilities. While this is not novel functionality, Omega enabled its deployment without any changes in nonparticipating hosts or domains (or applications).

Sequence of Changes. The above scenarios each focused on a particular architectural transition, but our hope is that Omega enables ongoing and pervasive changes. To verify this, we devised a few scenarios where multiple evolutionary changes occurred in parallel. For instance, in one we started with BGP as the only ISM. Then a few domains evolved to support a pathlet-based ISM in addition to the BGP-based one. Then another set of domains evolved to support DONA as an ISM. At the end of the scenario, we had pairs of hosts using BGP and/or pathlets depending on what ISM the domain provided and what properties BGP's default path supported. Similarly, some hosts used the publish/subscribe functionality of DONA to access content, while other hosts could only communicate end to end. Co-existing with these were hosts using the end-to-end optical-like ISM (which requires an initial path setup before actual communication). In total here we used eight domains and four ISMs.

Larger Networks. Finally, we also ran scenarios for larger networks (of 144 PlanetLab nodes, representing entities in 48 domains), with a sequence of architectural changes. The scenarios began with all domains using BGP, and hosts communicating via BGP. Subsequently, some of the domains evolved to enable pathlets and their hosts switched to communicate via pathlets (while unchanged domains / hosts continued to use BGP). Finally, several domains evolved to enable DONA and their hosts began using DONA. In the scenario's final state, we had 16 BGP-only domains, 16 domains with BGP and pathlets, and 16 domains with BGP, pathlets, and DONA support, with their hosts communicating using those three different ISMs in parallel. All of these evolutionary steps behaved as expected.

Component	**Router**	**OS**	**App**	**Domain**
Domain tech.	$\exists$ D	$\exists$ H		$\exists$ D
ISM	$\star$ D	H-H		$\star$ D
API		$\exists$ H	$\exists$ A	
End-to-end		H-H		
Path property	$\star$ D	H-H		$\star$ D

Table 4: Evolutionary constraints for the deployment of new network functionality in Omega. In the table, $\exists$ indicates that the affected / relevant local party—a domain (D), host (H), or application (A)—must be changed, H-H indicates that communicating endhosts must both be changed, and $\star$ indicates that all the participating parties must be changed.

5.3 Evolutionary Constraints

We now step back and summarize what we have learned about which evolutionary transitions are allowed. Table 4 identifies the inherent constraints on changes, which we think are not specific to Omega but rather apply more generally.[10] For instance, when a domain wants to use a new domain technology (*e.g.*, a new version of IP or Ethernet), the router vendor used by that domain must support the technology, and the hosts within that domain who want to use that technology must enable support supplied by their OS vendor. To deploy a new ISM, participating domains must support it, and the hosts involved must support it (for simplicity the table assumes these are pairwise interactions, but that need not be the case). Deploying a new API requires support by the OS vendor (on hosts that want to provide that API) and App vendor (for applications that seek to use the new API). The table also lists the constraints that apply to architectural changes in end-to-end protocols and to properties or protocols that cover the entire path. We omit for space changes to namespaces or name resolution infrastructures.

The bottom line is that these are *all* the constraints that evolutionary transitions face. Omega has removed artificial constraints (such as tying applications to a particular addressing or naming scheme).

6 Transitioning to Omega

The previous section provided evidence that Omega does indeed make architectural evolution easier, but these

[10]However, it is the case that in other architectures and frameworks, including today's Internet architecture, there are likely additional constraints beyond those given here for Omega. More generally, it is an interesting research topic for how to rigorously catalog these constraints. We produced this table by inspecting various dependencies between architectural modules, but this begs for a more formal treatment.

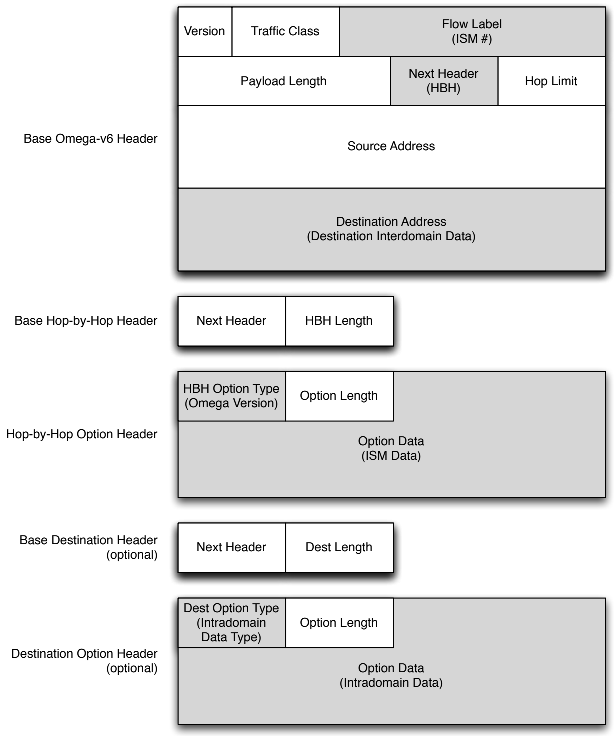

Figure 10: Structure of the Omega-v6 header. Shaded fields represent re-use of IPv6 fields for encoding of Omega values.

benefits will remain theoretical if Omega is never deployed. Thus, we now turn to the daunting question of how Omega might be deployed. The most promising way forward is to leverage the ongoing efforts to deploy IPv6 and SDN. Thus, in this section we demonstrate how to use IPv6 and SDN to deploy Omega, an approach we call Omega-v6.

6.1 Building on IPv6 Extensibility

We rely upon IPv6 extension headers to carry necessary Omega information in packets as follows:

Flow label. To identify the packet's ISM, we use IPv6's Flow Label field. In doing so, we allow use of existing hardware mechanisms designed for IPv6 to match against the ISM type—we will return to the motivation for this shortly.

Hop-by-Hop. We use the IPv6 Hop-by-Hop options header to encode ISM-specific information. As per the IPv6 specification, any devices along a path can ignore the header if so specified by the sender.[11]

Destination options. Any header information that is local to the receiving domain (but not specific to the end host) and that is not already encoded in the destination address or HBH header is encoded within the IPv6 destination options header. End-to-end headers are contained within the packet's payload as usual so they do not require special consideration, and are decoupled from the details of the ISM in use.[12]

The syntax of the headers we use for Omega-v6 is already predefined by IPv6, which meets our needs. Any unmet future requirements can be accommodated by a new extension type, also within IPv6's header structure. We show the overall layout of an Omega-v6 header in Figure 10.

6.2 Forwarding Omega-v6 Packets

To enable forwarding of Omega-v6 packets, we built our implementation around an unmodified OpenFlow data-plane, Open vSwitch (OVS), accompanied with a custom, centralized control plane. On the end-host side, we introduced a minimal set of changes to an existing Linux IPv6 networking stack that enable applications to choose a specific ISM and instrument its operational parameters. To validate our Omega-v6 implementation, we set up a testbed composed of Linux Containers which represent the end hosts and a series of inter-connected Open vSwitch instances that act as routers within domains that are tasked with forwarding Omega-v6 packets.

Our implementation for forwarding Omega-v6 packets differs little from the implementation we described in Section 5. Specifically, in the end host portion of the implementation, only packet emission code within ISMs needed alteration, since the remainder of the code concerns only end-to-end behavior (which includes schemas, bootstrapping, and meta-negotiation). The key change in the ISM code is to encode ISM information within an IPv6 header as shown in the figure.

The Omega router from our user-level implementation is the most significantly changed component in Omega-v6. In Figure 11 we show the packet processing done by an OVS node to appropriately handle an Omega-v6 packet. The changes required for an IPv6-supporting router to enable support for Omega-v6 and then incrementally add support for a new ISM—matching on the flow label and passing

[11] While today's router hardware tends to rate limit the processing of packets with Hop-by-Hop headers (and does so on the slow path), these headers would not be exposed to unmodified routers due to the tunneling discussed in next section.

[12] For example, since our TCP and UDP-like schemas are not TCP and UDP themselves, they do not have an inherent coupling to a specific address scheme or L3 (*i.e.*, the schemas treat the source connection identifier, however the ISM returns it to us, as an opaque bag of bits for the purpose of flow labeling).

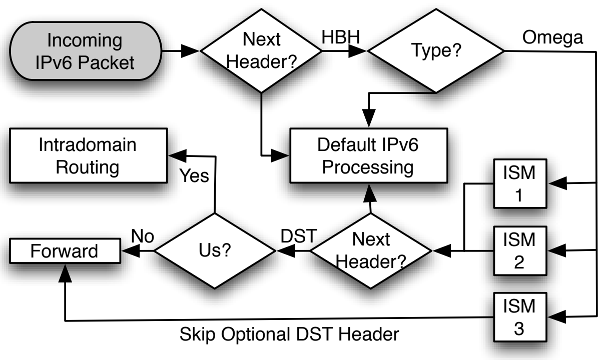

Figure 11: Packet processing by our Omega-v6 Open vSwitch router.

the matching packets to an ISM module—are modest. In the following, we'll consider the overall transition process from IPv6 to Omega-v6.

6.3 Deploying Omega-v6

Our proposed deployment path for Omega-v6 involves nothing new—just parallel protocol stacks and tunneling techniques—but they are useful to review to make a simple point: the difficulty of deploying Omega on top of IPv6 is no greater than deploying IPv6, and yet the benefits would outweigh that of IPv6 (or other single-shot clean-slate designs) because of the evolution it would enable.

Networks. Initially, we envision Omega-aware devices deployed only at the network edges allowing for parallel architectures to co-exist. As core routers start reaching their end-of-life, they will be slowly phased out of production and replaced with more flexible forwarding devices. Once critical mass is achieved, parallel architectural forwarding planes will merge, allowing a single router to understand multiple ISMs.

This leaves us with two requirements: (i) to make the traffic pass through legacy routers but still (ii) to instruct them to route any Omega traffic towards Omega-aware routers. For passing traffic through, we turn to tunneling. This corresponds to tunneling approaches developed for carrying IPv6 over IPv4—mechanisms that range from header extensions to UDP tunnels to IP-in-IP encapsulation—and any of them would apply here as well.

For steering Omega traffic towards Omega-aware routers, we assume that Omega routers know their next Omega hops, and thus, can send the tunneled Omega-v6 packets towards the proper IPv6 destination address. Omega traffic could be routed by software routers [5], and only when the traffic volumes increase, the implementation would have to move to hardware. Alternatively, due to our IPv6 based approach, hardware router platforms could also be extended with software forwarding; hardware platforms would match the Omega-v6 flow label in their hardware forwarding path to identify the packets requiring software forwarding.

In addition, a network with SDN-enabled routers could take advantage of hardware forwarding support for some ISMs without requiring upgrades of routers. To do so, the controller would set up rules for forwarding packets for these new ISMs in hardware; for example, pathlet routing can be implemented given an ASIC that can be directed by the controller to do masking and bit matching, since pathlets only contain short, opaque labels.[13]

End hosts. To enable applications to use Omega, we need to introduce a new API in OS protocol stacks and standard libraries. OS stack support consists of three pieces: support for ISMs, schemas, and new APIs, though they can be deployed independently. The former two are network-facing and the latter is application-facing, yet we contend that since both are introduced with regularity by OS vendors, introducing support for them will not be difficult. In addition, Omega support will not come at the expense of legacy application support, as OSes can continue to support BSD sockets-based applications and corresponding protocols in the stack.

To verify the feasibility of adding such end-host functionality, we implemented a proof of concept approach in Linux for introducing new APIs, schemas, and ISMs in the kernel. Our approach is extensible, uses kernel modules, and does not introduce any new system calls or modify existing networking code, though we only envision its use during the transition to Omega. (We did not implement the full suite of schemas and ISMs here, just enough to verify the feasibility of the approach.)

Suppose a vendor develops a publish/subscribe API with corresponding schema(s) and ISM(s). An application that wants to use this API links to a new library, `libpubsub`, which we built to provide `publish()` and `subscribe()` interfaces. Besides the library itself, the vendor then provides a kernel module; in this we implemented the corresponding (skeleton) schema and ISM. We added a new schema by adding a new protocol family via Linux's `proto_register()` and the associated handler via `sock_register()`; to handle Omega-v6 packets we used `nf_register_hook()`. To use the new functionality, `libpubsub` creates an underlying socket with the desired

[13]Today's forwarding ASICs often have significant constraints in their abilities, enabling hardware forwarding for only the simplest ISMs. While we expect their flexibility to increase, of course more complex ISMs are unlikely to ever be implementable without custom hardware.

schema (family) and uses `sendmsg()` and `recvmsg()`, which pass very general structs, for all communication with the kernel module (in a manner akin to IPC). While this approach does not lead to an elegant implementation, vendors can use it to create new modules for APIs, schemas, and ISMs without having to add new functionality to a legacy kernel's network stack, modify libc, introduce new system calls, or other actions that require coordination and agreement.

7 Related Work and Discussion

There have been many approaches to architectural innovation, the earliest being overlay networks. However, while overlays make it easy to deploy a new architecture on top of an old one, they do *nothing* to make it easier for two architectures to interact. That is, one can deploy new architectures—say, AIP [1]—using an overlay, but this only allows AIP hosts to talk to other AIP hosts; the overlay does not enable an AIP host to exchange packets with an IPv4 host. Thus, while overlays are useful for experimental deployments (particularly virtualized overlays like GENI [7]) and to achieve a wholesale replacement of one architecture by another (a grindingly slow process), they do not enable pervasive and ongoing change.

Active networks [14] enables certain forms of architectural evolution, particularly on the datapath, but this approach does not address architectural aspects such as naming and interdomain routing. Nebula [2] offers a great deal of extensibility in network paths and services, which is an important dimension of evolvability. However, the core of the architecture (*i.e.*, the datapath) is universal within and across domains; it is not clear how independently domains can evolve internally.

Plutarch [4] represents an entirely different approach to evolvability, 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 between the different architectures. The more recent XIA proposal [10] enables the introduction of new services through the definition of *principals*. To cope with partially deployed services, 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 could have paths for CCN [11] and a source route over IP addresses, with edges permitting intermediate routers to pick either means of reaching the data.

Thus, XIA's main 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.

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 XIA and Plutarch are unfortunate repudiations of these two principles, requiring architectural translations within the network to achieve evolvability. Moreover, we feel that architectural evolution can easily be accomplished while leaving these principles intact.

The design of Omega is similar to the FII proposal outlined in a recent CCR editorial [13]. While there are important differences (particularly in the realms of interdomain routing and DoS, where the approaches are quite different), the more fundamental distinction is that the previous discussion was motivated by high-level architectural arguments; here we arrive at these design decisions by looking for simple modifications to the Internet's current interfaces that would enhance modularity and extensibility. A similar approach was taken in [8], but with no implementation or evaluation.

In Omega, we take a more software-like approach to architectural evolution. Domains will likely support many ISMs with some of these being widely supported, and others in the early stage of adoption. The decision about which ISM to use is made at the edge, not internal to the network, so there is no need for ISMs to be translated into each other. This is fully consistent with the Internet's founding principles, and more in tune with how we design software systems.

We arrived at our design not by inventing new mechanisms, but through a basic code review of the current architecture, pointing out where modularity and extensibility were being ignored in the interface designs. All modern software systems are built with modularity and extensibility in mind, but these concepts have gotten lost in discussions of Internet architecture because the networking community tends to focus on the details of Internet protocols rather than view the various headers and mechanisms as abstract interfaces subject to the same scrutiny that we apply to software. Whether one agrees or not with our claims about evolvability, we hope the

Internet community begins to take a more systems-oriented approach to Internet architecture, so that we can avail ourselves of the decades of experience building evolvable software systems.

References

[1] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. of SIGCOMM*, 2008.

[2] T. Anderson et al. NEBULA - A Future Internet That Supports Trustworthy Cloud Computing. `http://nebula.cis.upenn.edu/NEBULA-WP.pdf`.

[3] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and Delay Accountability for the Internet. In *Proc. IEEE ICNP*, 2007.

[4] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. In *Proc. of SIGCOMM FDNA*, 2003.

[5] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, 2009.

[6] B. Ford and J. Iyengar. Efficient Cross-Layer Negotiation. In *Proc. of HotNets*, 2009.

[7] GENI: Global Environment for Network Innovation. `http://www.geni.net/`.

[8] A. Ghodsi, T. Koponen, B. Raghavan, S. Shenker, A. Singla, and J. Wilcox. Intelligent Design Enables Architectural Evolution. In *Proc. 10th ACM Workshop on Hot Topics in Networks (Hotnets-X)*, November 2011.

[9] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet Routing. In *Proc. of SIGCOMM*, 2009.

[10] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient Support for Evolvable Internetworking. In *Proc. of NSDI*, 2012.

[11] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proc. of CoNEXT*, 2009.

[12] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and Beyond) Network Architecture. In *Proc. of SIGCOMM*, 2007.

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

[14] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1), 1997.

[15] D. Wendlandt, I. Avramopoulos, D. Andersen, and J. Rexford. Don't Secure Routing Protocols, Secure Data Delivery. In *Proc. of HotNets*, 2006.

[16] X. Yang, D. Wetherall, and T. Anderson. A DoS-Limiting Network Architecture. In *Proc. SIGCOMM*, 2005.