Open Network Interfaces for Carrier Networks

Aurojit Panda[‡], James McCauley[‡], Amin Tootoonchian[♮†], Justine Sherry[‡]
Teemu Koponen[⋄], Sylvia Ratnasamy[‡], Scott Shenker[‡♮]
[‡] UC Berkeley, [♮] ICSI, [†] UToronto, [⋄] Styra
{apanda, jmccauley, justine, sylvia}@cs.berkeley.edu, amin@cs.toronto.edu
koponen@styra.com, shenker@icsi.berkeley.edu

ABSTRACT

With the increasing prevalence of middleboxes, networks today are capable of doing far more than merely delivering packets. In fact, to realize their full potential for both supporting innovation and generating revenue, we should think of carrier networks as service-delivery platforms. This requires providing open interfaces that allow third-parties to leverage carrier-network infrastructures in building global-scale services. In this position paper, we take the first steps towards making this vision concrete by identifying a few such interfaces that are both simple-to-support and safe-to-deploy (for the carrier) while being flexibly useful (for third-parties).

1. INTRODUCTION

In the early days of the Internet, there was a very clear division of labor between the network infrastructure and application developers. It was the network's job to deliver packets quickly and reliably, while it was the application developer's job to build applications using the end-to-end model (*i.e.,* expecting no direct network support other than best-effort packet delivery). These early efforts resulted in the seminal Internet applications on which we now rely (email, the web, etc.). Most of these applications were built around a client-server model, so that as the age of datacenter-scale services arose the end-to-end protocols needed little change (though significant innovation was needed to scale the internals of the datacenter, including the networking infrastructure).

However, the world has changed greatly since those early days. Two developments in particular have blurred this previously clear division of labor between application developers and the network: the rise of middleboxes (or network appliances) and the increasing importance of the network edge.

Middleboxes have become the most common way of deploying new in-network functionality, and according to a recent study [7] most networks have roughly an equal number of routers, switches, and middleboxes. Among the many middleboxes found in these networks are HTTP proxies, SIP proxies, WAN optimization, deep-packet-inspection, content caching, and content transcoding.

In the Internet age, Ben Franklin's adage "time is money" has become an empirically verified fact. Lowering response latency keeps viewers longer, which translates into increased advertising revenue. Placing services at the network edge helps reduce both latency and the load on the network backbone. As a result, many companies (*e.g.,* Netflix, Google, and Akamai) have placed some of their services at the edge. And this trend may accelerate as the Internet-of-Things, with its potential for generating large volumes of streaming data, will require even more edge data processing.

As a result of these trends, network involvement in Internet applications is both *possible* (due to the presence of middleboxes) and *desirable* (given the importance of placing functionality at the network edge, to reduce latency and backbone bandwidth). While we continue to pay lip-service to a clean division between applications and network infrastructure, and that model still suffices for some applications where latency and bandwidth are not major concerns (such as online banking), this is no longer the dominant reality. Rather than resisting this trend, we should embrace it by viewing network infrastructure not merely as a packet delivery mechanism, but as a more general *platform for supporting services*.

Carriers (*e.g.,* NTT and Verizon) are perfectly positioned to take advantage of this trend. They have ubiquitous presence at the edge of their own network and, due to the rise of SDN and NFV, can flexibly insert middleboxes there. In addition, carriers have extensive experience with 24/7 operations on their infrastructure. However, carriers have not effectively responded to this opportunity. They have attempted to build services themselves, such as CDNs, but their efforts are widely seen as too-little-too-late and have had little impact on the Internet ecosystem. While carriers can provide premium connectivity to those providing popular services (*e.g.,* the recent Netflix-Comcast deal), and allow third-parties (such as Akamai and Netflix) to place equipment at the carrier edge, their infrastructures are still designed primarily around packet delivery.

We advocate a much more active role for carrier networks, one that is best motivated by considering the history of Amazon's EC2 service. To support their own business, Amazon built a set of large datacenters. They then recognized that these datacenters could be useful to others, and that offering up this computational infrastructure as a service could be profitable. To take advantage of this opportunity, they developed a tenant-facing service interface (the EC2 VM interface) that had six important properties (where we use the term tenant to refer to EC2 customers):

- **Simple to support**: The VM interface is well-established, and required little technical innovation to support.
- **Safe to deploy**: VMs provide isolation (protecting both other tenants and Amazon itself) and Amazon carefully manages resource allocations; the combination ensures that a tenant's use of EC2 poses no threat to Amazon, or other tenants (in terms of security and resource usage).
- **Self-service**: No manual intervention by Amazon is needed for a tenant to use the EC2 service.
- **Usage-based**: Tenants are charged based on usage, and (for small resource requirements) do not have to reserve resources in advance. Moreover, tenants that are more forgiving of failures can use cheaper options, such as spot-pricing, that are less reliable.
- **Flexible**: One can develop useful services using this interface, and the space of services enabled by the infrastructure far exceeds what anyone could have envisioned. In fact, the Amazon business model had the flavor of "*we don't know*

the future, but we sure hope someone builds it on our infrastructure", which is a way of profiting from grass-roots innovation.

- **Narrow**: The service interface is narrow enough that Amazon had the freedom to innovate in how it was implemented, allowing it to improve the efficiency of its infrastructure without changing the interface.

The fact that EC2 was self-service and usage-based lowered the barrier-to-entry for these datacenter services, opening the world of large-scale computing to everyone (in fact, in 2011 one could rent the 30th fastest supercomputer for a little over $1000/hour!). This enabled everyone to scale services without running their own infrastructure. Amazon's ability to support scalable services, while protecting their own infrastructure, has spurred innovation in Internet services and made Amazon a tidy profit in the process.

In this paper, we propose an initial step carriers could take to emulate the EC2 example. We call our system NSS, for *Network Service Support*, and put it forward not for the sake of the carriers, whose business prospects are not our concern, but to hopefully increase the rate of innovation in network services and broaden how we think of the network service model. Our focus is not on the design details or implementation issues, but instead on the basic interfaces operators could expose and how they might be used.

NSS is exceedingly simple for carriers to build, and exposes an interface with three main components:

- **Tenant-facing invocation interface:** This is how the tenant invokes NSS, and describes the desired high-level application interaction pattern. This interface abstracts away low-level details, enabling seamless changes (by both tenants and carriers) in the low-level infrastructure.
- **Client-facing primitives:** These include registration and name resolution.
- **Edge services:** These are services, such as caches or firewalls, that the tenant can instantiate at the edge (through the invocation interface).

NSS has the same six properties as listed above for EC2, and enables third-parties (tenants) to build services, offered to clients (customers of the tenant), that leverage the network infrastructure. Thus, NSS allows application designers to focus on what they do best – searching for unmet needs and figuring out the right way to meet them – and lets the network infrastructure make deployment simpler (because the application designer can use a set of basic primitives, and need not worry about how to scale the network-based portion of the service) and more effective (given the proximity to the edge).

This obviously resembles current systems like Akamai, which uses edge caches and name resolution to speed content delivery. In fact, *that's our point!* Systems like Akamai are extremely valuable and our goal is to make them easier to deploy. We discuss later how NSS could trivially support an Akamai-like service with very little tenant-supplied infrastructure.

Our approach imposes no significant changes on applications, in terms of the interactions between clients and servers. This is because we are not exploring new application architectures, only new deployment and configuration models for network-based applications. Our approach lets functionality be placed at the edge where needed (*i.e.,* where the clients are), without the tenant needing to know beforehand where these clients might arise nor having to deploy the edge functionality themselves. Moreover, the configuration (in terms of who the client contacts to access a particular service) is handled by the service primitives we have designed. In this sense, our approach is similar to SDN, which did not change how packets are forwarded, but did change how that forwarding behavior is computed and configured.

We are certainly not the first to write about the role of networks in supporting services. There has been recent research on enabling service *invocation* [8], which exposes various network services to end users. This rightly moves service invocation from implicit dataplane actions to explicit control plane interfaces. But our focus is on service *construction*, using the existing network infrastructure, which is quite different (and complementary). There has also been a long history of work on service *composition* [5, 6], and more recent developments with an emphasis on the cloud [4], but this typically involves chaining together several high-level services, such as file servers, databases, and the like, and requires interfaces suitable for general distributed programs. Our focus is on utilizing a narrow set of low-level network services, so our interfaces need not be so general or work in such a broad set of contexts. Thus, our problem is far easier, and we do not have to confront the deeper problems explored in the general service composition literature.

Of course, the technical community has been actively developing the SDN and NFV paradigms (with a burgeoning literature), which are useful mechanisms for implementing what we describe here, but neither directly address the question of how one uses the network to seamlessly support third-party applications.

2. IDEALIZED SCENARIO

In this section we move beyond motivation to a discussion of how this NSS might be used in a highly-idealized scenario (where all complicating details are omitted). Consider for instance an application designer (henceforth referred to as the tenant) who would like to deploy a new CDN specifically tuned for video content that would benefit from network support. She begins by dividing functionality between the client (which runs on one or more user devices), a cache service (running at the network edge) and the back-end servers (running in a cloud) where the content is hosted.

She then picks cache and server implementations that meet her application requirements, with the only NSS-specific requirement being that the client code uses NSS-supplied primitives for bootstrapping. To deploy this service she now contacts the carrier to find the name of a *Coordinator* (a carrier portal) and provides the Coordinator with an *instantiation* consisting of:(i) a *template* that describes and names the components of an application (*i.e.,* client, cache and origin server in this case) and indicates how they interface; and (ii) *metadata* specifying the location of back-end servers, application code that needs to be executed at the network edge, and other configuration parameters.

The carrier then activates instances of this service at various network edges in response to clients connecting at those edges. When clients first connect to the network, a bootstrap mechanism (DHCP or others) provides the client with the address for the Coordinator which the client then registers with. On registration the Coordinator provides the client with a handle for a *Discovery* service. The application then uses the Discovery service to find the cache.

While the template – which dictates the overall structure of the application – remains fixed, a tenant can evolve an application by changing the metadata, which can: change the set of services invoked at the network edge (*e.g.,* to introduce transcoding), change the software deployed at the network edge (*e.g.,* upgrade the software or update its configuration), or change the location of back-end servers. None of this requires manual intervention by the carrier; these updates are sent by the tenant to the Coordinator (using either a programmatic interface or manually through a portal), which then implements these changes where appropriate (at the edges, or in the Discovery service).

We now provide more technical details to flesh out how this works in practice.

3. ENTITIES AND INTERFACES

We now provide more details on the entities involved and the basic tenant and client interfaces. We hope it is self-evident that these interfaces are simple, safe, self-service, flexible, and narrow, and could easily support usage-based charging (which are the properties we cited in the introduction as having been crucial for EC2).

3.1 Entities

In addition to allowing applications to invoke an extensible set of edge services (*i.e.,* VMs running tenant-supplied or operator-supplied code), NSS provides each application with two simple entities: a Coordinator and a Discovery service.

- **Coordinator:** The Coordinator is responsible for bootstrapping applications. The Coordinator is provided by the carrier and is shared across applications.[1] The coordinator also provides the tenant interface; tenants first invoke NSS by contacting the Coordinator and handing it an instantiation consisting of a template and metadata.

- **Discovery service:** The Discovery service is responsible for binding application-specific names to addresses, and can require authentication before using. The Discovery services uses tenant-specified metadata for these resolutions.

These two entities are responsible for supporting the tenant and client interfaces.

3.2 Tenant Interface

Tenants interact with the carrier through the carrier's Coordinator. The Coordinator provides an interface for instantiating applications, and the input to this interface is comprised of two parts:

- **Template:** The template describes the basic structure of the application, by naming the components involved and specifying the dataflow between them. The application components include tenant-managed servers and carrier-managed edge services. Tenants can in turn delegate the management of their servers to others: the term tenant-managed only implies that the tenant is responsible for providing them, as opposed to the carrier-managed edge services which are operated by the carrier.

  As an example, for the CDN service (Section 2) the invoking template would be:

  $$\text{Client} \rightarrow \text{Cache Service} \rightarrow \text{Origin Server}$$

  Upon receiving such a template, the Coordinator returns (to the tenant) a handle for each of the application components, so the tenant can change the metadata associated with each of these components.

- **Metadata:** The tenant supplies the Coordinator with metadata for each application component. This metadata includes:

  - Names for all application components, and whether they are carrier-managed or tenant-managed
  - For tenant-managed servers, the metadata includes IP addresses for tenant-managed servers and other relevant information (*e.g.,* certificates to ensure the validity of the server).
  - For carrier-managed edge services, the metadata specifies what services to provide at the edge. If the edge service is carrier-supplied, the metadata contains enough information to specify and configure that code. If the edge services is tenant-supplied, the metadata includes a pointer to the executable and related information (resource requirements, version number, etc.). This metadata can be updated as needed.

3.3 Client Interface

Client code can have an arbitrary application-specific interface (*e.g.,* in our CDN example, NSS does not constrain the interface between the client and the cache), but the client interfaces that concern us here involve how the client interacts with NSS. Clients interact with NSS in two specific ways:

- **Coordinator:** Each client, when they connect to a new network, register with the Coordinator. The Coordinator might redirect this registration so that the client first authenticates (as specified in the template). Once a client is registered, it is given the location of the tenant-specific Discovery service.

- **Discovery service:** Clients use the Discovery service to resolve application-specific names (the Discovery service is not just a local copy of DNS, but rather can implement its own name resolution mechanisms). The resolution results can be local in nature, in that the tenant-specified metadata can contain results tailored to each edge, or they can be global (*e.g.,* application back-end servers). Client can include authentication token with request, as some bindings might be available only to appropriately authenticated clients.

For certain classes of applications (*e.g.,* mobile applications), it might be beneficial to avoid the additional RTT involved with sending requests to the discovery service. In this case NSS allows the use of a more proactive version where the Coordinator returns name-address bindings for the components of a tenant edge application rather than the location of a Discovery service. While this avoids the round-trip to get name bindings it reduces the amount of flexibility available: servers and instances of network components must preserve the same address over the course of the application's lifetime.

3.4 Carrier Implementation

While the purpose of NSS is to make life easier for application developers (to invoke network support), does this make life hard for the carrier? As the preceding description hopefully makes clear, the answer is clearly no. To build and deploy NSS, a carrier needs only: (a) build a simple portal to serve as Coordinator, (b) build a simple name resolver to serve as the Directory service, and (c) deploy racks at their edge that can spin up VMs on demand, thereby instantiating the required edge services (most of which are easily available, like firewalls, or are supplied by the tenants themselves). To validate our idea we built a prototype of the primitives and several applications which made use of them. In our implementation the primitives themselves took only 2,600 lines of C++ code[2] and took less than a week of programmer time. Thus, we believe building NSS involves only standard software engineering best practices.

[1] While the Coordinator is shared across applications, its state is partitioned by application and hence applications are isolated from each other.

[2] As measured by David A. Wheeler's SLOCCount tool.

4. EDGE SERVICES

Allowing applications to utilize services running at the network edge are the key feature enabled by NSS. We envision that these will be provided by both tenants (for services tuned to a specific applications) and carriers (for commonly used services). We assume most edge services will run as VMs (as in NFV) on traditional software servers deployed at the edge of the network. Many existing service implementations can be deployed at the network edge almost unmodified. However, since carriers will deploy these edge services on-demand, at multiple network edges, these services must meet a few requirements that we highlight below.

4.1 Edge Service Requirements

To allow carrier to deploy services at multiple network edges and so these services can be started and stopped on-demand, we require that:

- Edge services rely on configuration that is location and scale agnostic. In particular, the correct and efficient functioning of edge services should not depend on the number of copies running on the same network, and the edge service should make no assumption about the geographic location or the IP address for the VM in which it is executed.
- Edge services send requests to other services by name, *i.e.,* edge services must themselves use the *Discovery* mechanism provided by NSS rather than using hardcoded addresses for services. The use of names instead of addresses allows carriers to launch and teardown individual edge service instances without affecting application functionality.
- Edge services store state in a manner than enables elastic scaling; *i.e.,* state that needs to persist across multiple sessions from the same user is persisted elsewhere (not on the edge services).
- Consistency between edges must be handled explicitly, for instance with the use of a central server to which all edge services talk. This limitation means that applications supporting transparent mobility must provide mechanisms to synchronize user sessions across edges.

While these requirements seem stringent, many existing services (caches, SIP proxies, etc.) already meet these requirements. Further, services meeting these requirements can be trivially scaled (even on the same edge) by launching additional copies.

We expect that these requirements themselves would evolve over time, in particular many of these requirements can be eliminated by the addition of other features. For instance, carriers can divide the network edge into coarse grained availability zones and allow tenants to restrict their instances to particular availability zones. Similarly, carriers can provide state management services and thus enable edge services that require more permanent state. In addition, in the future one might want to add monitoring hooks as a requirement, so that tenants could more easily monitor (and debug) the global operation of their applications. Thus, the requirements here are merely a first step and are designed to simplify the development and deployment of NSS.

4.2 Example Services

Next we present a few example edge services. We mostly list services that are general and commonly used, and hence might be provided by carriers. Some applications would undoubtedly provide their own edge services (*e.g.,* for application-specific transcoding or data processing).

Caches: The ability to deploy caches at the network edge can greatly reduce traffic through the network core and perceived client latency. Additionally, many carriers already provide caches as a part of CDNs.

Client Registration: A client registration service can add name bindings (accessible through the Discovery mechanism) for clients at the network edge. This can be used for a variety of purposes including finding all devices belonging to a user connected to a particular edge and finding all devices that have specific content. The registration service can accept an authentication token that must be provided to discover certain bindings.

Authentication: A carrier (or other provider, for instance Google) might choose to offer an authentication mechanism that can be optionally used by tenants. Such an authentication service would be token based (*e.g.,* based on Kerberos), and requires clients to authenticate using credentials provided by the provider.

5. EXAMPLE USAGE

Moving beyond the CDN example of Section 2, and obvious extensions such as SIP proxies, we now look at a few examples of more complicated applications that could be supported by NSS.

5.1 Edge Based Multicast

Video content providers often need to transfer the same content to multiple devices. This might be either for live streaming (for sports events, lectures, etc.) or for other cases where we want users to be able to watch the same video stream on multiple devices seamlessly. While one can solve this problem by running several unicast streams, doing so increases utilization on the transit links. In a network where services can be placed at the edge of a network one can instead send a single stream to an edge service, which can replicate and send it to all local clients. A multicast overlay like this requires no internal network support from the carriers, and works across carriers.

Instantiation and Usage

Template:

$$\text{Origin Server} \rightarrow \text{Edge Proxy} \rightarrow \text{Clients}$$

Metadata: The Edge Proxy is the only carrier-instantiated application component. Metadata in this case provides an address for the origin server. The tenant can require that clients authenticate before getting access.

Usage: Once this template has been instantiated by the carrier, clients initiate sessions (for particular streams) with the edge proxy. The edge proxy is responsible for receiving the stream from the origin server and multicasting it to all local clients. Edge proxies can subscribe to streams from the origin server either on demand (*i.e.,* when they have active sessions for a stream) or proactively.

As an alternative, the edge proxy can proactively initiate sessions for popular streams (for instance World Cup games) with the origin server ahead of time thus reducing initial latency for all clients.

5.2 ICN

Previous work has argued that Information Centric Networking [2,3], where content can be accessed using names instead of requiring clients to provide IP addresses, can lower response latency, provide additional security and better mobility and reduce bandwidth consumption. Recent work [1] has shown that one can achieve most of these benefits using name-based caches at the edges. Using our mechanisms a tenant can deploy a system similar to what was envisioned by [1] without requiring any additional carrier support.

Instantiation and Usage

Template:

Client → Proxy → Name Resolution → Origin Server

Metadata: The proxy and possibly the name resolution service are carrier instantiated. The tenant can require clients to authenticate before obtaining access.

Usage: Clients can use this service exactly as described in [1]: requests for named data are sent to the proxy which either responds to these from its local cache or uses the *name resolution* service to locate the data in the origin server. This name resolution service can be tuned to support flat names, thereby supporting arbitrary naming systems (and, in particular, content-centric names).

5.3 Storage Synchronization

Storage services such as Dropbox, Box and AeroFS synchronize content across user devices. To lower latency and reduce data sent through the network core these services include support for synchronizing local clients (*i.e.,* devices connected to the same local network) without involving a remote storage server. In this case, the Discovery service can allow clients to discover all other local clients, greatly simplifying the development of such applications.

Instantiation and Usage

Template:

Client → Authentication → Client Registration

Metadata: Application registration is carrier instantiated. Depending on the service an application writer can choose to use a carrier-provided authentication service or one provided by third-parties.

Usage: Once authenticated, clients can contact the *application registration* service to be added to the list of clients belonging to a user currently connected to a network edge. This mapping is maintained (and queried) using the Discovery service at the local edge.

5.4 Edge Processing for Sensors

Recently, there is growing interest in collecting and aggregating data from sensors embedded in "smart" objects, which form the *Internet of Things* (IoT). As a whole these sensors can generate huge amounts of data; for instance it is reported that the engines on a Boeing 777 generate over 4 terabytes of data during a trans-Atlantic flight. Transferring this raw data to data centers for processing can be costly and contribute to congestion at the core. Often, the data can be processed, either to limit its size while retaining information required for processing or to speed up initial analysis [9]. Tenants can use our mechanism to perform this processing, on-demand, at the edge.

Instantiation and Usage

Template:

Sensor → Processing → Data Center

Metadata: Tenants supply code for the processing service: since the functionality is dependent on both the sensor and the query being executed this might be highly-specialized for each client.

Usage: Sensors lookup and send traffic to edge processing units which then forward the processing results to the data center. Clients can issue queries and look at results by connecting to the data center.

5.5 Middlebox Outsourcing

Recent work [7] has also proposed outsourcing middlebox functionality to data centers, both to simplify administration and take advantage of consolidating this functionality for several enterprises. This work proposed outsourcing this functionality to a data center, where these middleboxes are run on virtual machines. Using NSS one can instead outsource this functionality to the network edge, rather than to the middleboxes and thus potentially reduce latency. This is also an example where traffic is sent through the service transparently.

Instantiation and Usage

Template:

Traffic Class → Middlebox Pipeline

Metadata: The metadata in this case specifies the middlebox pipeline (a sequence of middleboxes, or more generally a DAG of middleboxes), and the configuration of each middlebox.

Usage: All packets belonging to the specified traffic class are forwarded to the middlebox, which then processes this traffic and forwards it as appropriate.

6. DISCUSSION

Before EC2, a company could only offer a global service once it had learned how to scale its infrastructure (*e.g.,* run a datacenter, etc.) and sufficiently invested in the infrastructure. After EC2, companies could focus on meeting customer needs and let Amazon worry about scaling the infrastructure. This has had made it far easier to bring new service ideas to large markets. Network operators are in a similar position to where Amazon was when first building EC2: the demands of NFV have meant that they are deploying general purpose compute servers at the network edge. These servers are meant to help ISPs rapidly deploy new features, but are necessarily underutilized at most times. NSS, similar to many other recent proposals, is therefore inspired by the presence of this spare capacity, but aims to allow application developers, not just ISPs make use of this capacity.

We hope that, similar to EC2, NSS will enable application developers to build and deploy network supported services, without worrying about scaling or building out network infrastructure. Right now there are several companies that have significant deployments at the edge (*e.g.,* Google, Akamai, Netflix); competing with them would require a new entrant to make a similar infrastructure investment, precluding all but extremely well-funded and targeted efforts from mounting a challenge. With NSS, one could deploy a wide variety of edge-based services with very low barriers-to-entry, and we think this might facilitate more rapid innovation in this space.

While one can debate whether NSS will have impact, it is clearly feasible for carriers to build and deploy. The Coordinator and Discovery components are straightforward, and carriers can easily deploy racks at the edge to support the required edge services. In this sense, NSS's lack of technical depth is a feature, not a bug. The point of our position paper is defining and supporting open network interfaces for application support is trivially within our reach.

The most challenging open question that remains is this: do carriers compete or collaborate in offering NSS? If they compete, then each carrier offers a NSS-like interface, and individual third-parties can decide how many they need to sign up with to provide adequate edge deployment. Competition might lead to faster adoption of NSS-like interfaces, as carriers seek to beat their competitors to market.

If they collaborate, then NSS's interface becomes a standard and network interface generalizes from simple packet delivery to a more service platform. This would represent the next step in the evolution of the Internet, in which edge services become a fully integrated aspect of carrier networks. This would require solving the question of how carriers peer at the NSS level, so that deployment happens at all edges, regardless of the carriers (*i.e.,*, if a tenant has signed up for service with one carrier, how does that carrier arrange for other carriers to support that tenant). This is less a technical question than an economic one; technically, it is trivial to disseminate the instantiation information across carriers, but economically it may be hard to agree on the compensation for such peering.

Regardless of how the competition/collaboration is resolved, we believe that incorporating service support is an opportunity whose time has come. The relevant technology (particularly given the advances in SDN and NFV) is readily available, and the overall architecture is conceptually simple and straightforward to build.

7. ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments and suggestions which have helped improve this paper. This research was supported in part by NSF grants 1343947 and 1420064, and funding provided by Intel Corporation.

8. REFERENCES

[1] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker. Less pain, most of the gain: incrementally deployable icn. In *Proc. of SIGCOMM*, 2013.

[2] V. Jacobson et al. Networking Named Content. In *Proc. of CoNEXT*, 2009.

[3] 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*, August 2007.

[4] OpenCloud. Retrieved 07/16/2014: `http://opencloud.us/`.

[5] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. Service placement in a shared wide-area platform. USENIX ATC, 2006.

[6] F. A. Samimi and P. K. McKinley. Dynamis: Dynamic overlay service composition for distributed stream processing. SEKE, 2008.

[7] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. of SIGCOMM*, 2012.

[8] J. Sherry, D. C. Kim, S. S. Mahalingam, A. Tang, S. Wang, and S. Ratnasamy. Netcalls: End Host Function Calls to Network Traffic Processing Services. Technical Report EECS-2012-175, UCB, 2012.

[9] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. CIDR, 2011.