XOS: An Extensible Cloud Operating System

Larry Peterson
Scott Baker
Marc De Leenheer
Open Networking Lab

Andy Bavier
Sapan Bhatia
Jude Nelson
Mike Wawrzoniak
Princeton University

John Hartman
University of Arizona

Abstract

This paper describes XOS, a cloud operating system designed to manage hardware and software resources across a multi-tiered cloud. XOS raises the level of abstraction in an IaaS cloud architecture by elevating scalable software *services* to first-class objects. This involves adopting three design principles: (1) Everything-as-a Service (XaaS) (services are building blocks, and combinations of those building blocks are also services); (2) Multi-tenancy (a tenant relationship links one service to another, and facilitates reasoning about safety, privacy and efficiency); and (3) Control/Data-plane separation (services are configured through a logically centralized service controller interface, but the controller is not on the data path between services). XOS applies these principles through the lens of an operating system—it defines a set of abstractions that support constructing multi-tenant services that can be folded back into XOS as available building blocks, while also extending the capabilities of conventional IaaS. The paper shows how these abstractions can be used to build a functional, evolvable, service-oriented cloud.

1. INTRODUCTION

The Cloud is evolving into a multi-dimensional space that includes virtualized clusters and software-defined networks; large data centers and smaller clusters deployed deeply in the network; specialized private clusters and commodity public facilities; building block services and new applications. A key challenge for architects and designers of future clouds is to provide the right set of abstractions so that application builders can fully leverage these new opportunities, while avoiding the ossification that often arises when such a large diversity of factors is involved.

This paper explores the value of adopting an operating system perspective on the problem of programming the future cloud. We describe XOS, an extensible cloud operating system that defines unifying abstractions on top of a collection of cloud services. We call XOS an operating system because it plays much the same role in a geo-distributed cloud as a traditional operating system does on a conventional computer: it provides general programming abstractions to support a wide range of applications, while at the same time safely multiplexing the underlying hardware resources and software services between them. By regarding XOS as an operating system, we bring 50 years of proven concepts and best practices to the problem of tranforming the cloud into a general-purpose computing environment.

XOS is designed for the multi-tier, network-wide cloud shown in Figure 1. Its abstractions unify access to cloud infrastructure across multiple sites—from commodity clouds to private datacenters to wide-area network routing centers to edge access sites. XOS leverages existing datacenter cloud management systems to implement low-level scheduling and resource allocation mechanisms for each autonomous site, and SDN-based network customization to connect the sites.

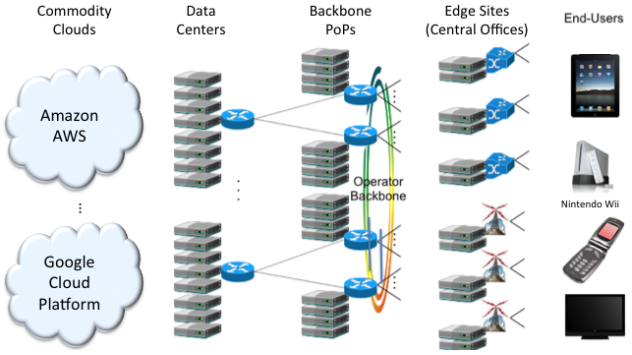

Figure 1: *Multi-tiered cloud that XOS is designed to run on, spanning commodity clouds, private datacenters, and modest clusters embedded deep in the network.*

XOS provides explicit support for multi-tenant services, making it possible to create, name, operationalize, manage and compose services as first-class operations. Well-known multi-tenant clouds are designed to host applications, but they usually treat these as single-tenant services that run on top of the cloud, for the benefit of the user. In contrast, XOS provides a framework for implementing multi-tenant services that become part of the cloud, thereby lowering the barrier for services to build on each other.

Focus on support for multi-tenant services is inspired by service ecosystems like Amazon Web Services (AWS) [3] and the Google Cloud Platform [9], which consist of a rich set of mutually supportive services. For example, Google App Engine [11] is effectively a front-end to a collection of Google services, which over time has included different combinations of Spanner [6] (a fault-tolerant database), BigTable [5] (a NoSQL database), Colossus and its predecessor the Google File System [10], and the Chubby Locking Service [4]. Unlike AWS and Google Cloud Platform, however, XOS is an open system—both in terms of source code (giv-

BigSystem 2015, June 23, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-3568-3/15/06 ...$15.00.
http://dx.doi.org/10.1145/2756594.2756598.

ing others the opportunity to build their own service ecosystems) and in terms of serving as an artifact to study (giving the community the opportunity to identify common patterns and building blocks). We return to this point, and discuss related open systems like Docker [7] and OpenStack [16] in Section 5.

2. APPROACH

This section outlines our approach, which considers the problem of architecting a cloud from the perspective of designing an operating system. It also sketches the software structure we arrived at in prototyping XOS—bootstrapping XOS with existing public services and open source software—which itself is an important element of our approach.

2.1 OS Perspective

An OS provides many inter-related mechanisms to empower users. If we define an OS by a successful example, Unix [18], then an OS provides isolated resource containers in which programs run (e.g., processes); mechanisms for programs to communicate with each other (e.g., pipes); conventions about how programs are named (e.g., /usr/bin), configured (e.g., /etc), and started (e.g., init); a mechanism to program new functionality through the composition of existing programs (e.g., shell); a means to identify and authorize principles (e.g., users); and a means to incorporate new hardware into the system (e.g., device drivers).

XOS provides counterparts to all these mechanism—as described in the next section—but in general terms, XOS adopts much the same design philosophy as Unix. Both are organized around a single cohesive idea—everything is a file in Unix and Everything-as-a-Service (XaaS) in XOS. Both also aim to have a minimal core (kernel) and are easily extended to include new functionality. In Unix, the set of extensions correspond to the applications running top of the Unix kernel. OS experts might debate whether /bin/bash is a fundamental part of Unix, a feature of a particular Unix distribution, or an application running on top of Unix, but from the user's perspective, the distinction between the kernel and commands they can invoke is an implementation detail. This is also the case with XOS, where as depicted in Figure 2, the core is minimal and the interesting functionality is provided by a collection of services. Moreover, XOS supports a shell-like mechanism that makes it possible to program new functionality from a combination of existing services.

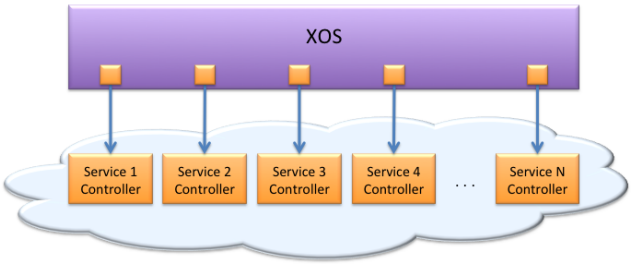

Figure 2: *XOS layers an OS on top of a set of cloud services (service controllers).*

Implicit in Figure 2 is an underlying model of exactly what constitutes a service. Our model is that every service incorporated into XOS provides a *service controller* that exports a programmable interface to network-wide functionality, with the service implemented by an elastically scalable number of *service instances*. Figure 3 depicts the anatomy of a service in this way, where the instances (for example, VMs) are potentially distributed widely over a geo-distributed set of clusters. For example, some VMs might be concentrated in one or more datacenters, while others are distributed across many edge sites.

The separation of service controller from service instances is central to our design. The controller maintains all authoritative state for the service, and is responsible for configuring the underlying instances. Service users (and service operators) interact with the controller, which exposes a global interface; any per-instance or per-device interface is an implementation detail that is hidden behind the controller.

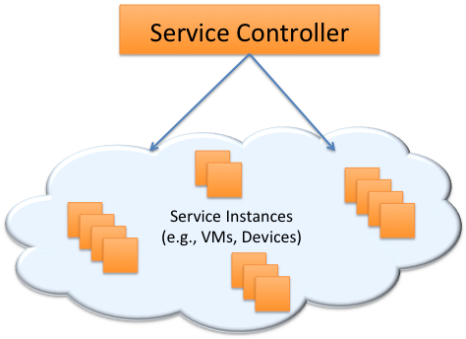

Figure 3: *Anatomy of a Service: A single service controller provides a logically centralized interface to network-wide functionality, with service instances providing many points of implementation distributed across the network.*

To continue the Unix analogy, there is a spectrum of implementation choices for providing the capabilities outlined above, including monolithic kernels (e.g., Linux) and microkernels (e.g., Mach [1] and L4 [14]). Because XOS unifies a collection of services, many of which exist independent from XOS, it most naturally maps onto a microkernel structure. Figure 4 shows a canonical microkernel side-by-side with XOS, with the latter depicted as a simple reorientation of Figure 2.

Figure 4: *XOS can be viewed as adopting a microkernel structure.*

There are, of course, important differences between the two systems in Figure 4. The most notable is the extent to which each decouples the control and data planes. In the case of XOS, controllers represent services—each controls a scalable set of service instances distributed throughout the network—but these instances interact with each other without the controller being directly on the data path. In contrast, a conventional microkernel-based system bundles a server's control and data plane in a single component. Sometimes operations flow from a client to a remote server via RPC, but both control operations and data operations traverse the same sequence of local and remote modules. This explicit separation of a service's control and data plane is an important aspect of XOS's design.

2.2 Software Structure

We have built a prototype of XOS and deployed it on hardware similar to that shown in Figure 1. The following highlights a few key attributes of the implementation.

XOS is organized around three layers, as illustrated in Figure 5. At the core is a *Data Model*, which records the logically centralized state of the system. It is the Data Model that ties all of the services together, and enables them to interoperate reliably and efficiently. The logical centralization of this state is achieved through a clearcut separation between this *authoritative* state and the ongoing, fluctuating, and sometimes erroneous state of the remainder of the system—the so-called *operational* state. The ability to distinguish between the overall state of the system at these two levels—authoritative Data Model and operational backend—is a distinguishing property of XOS.

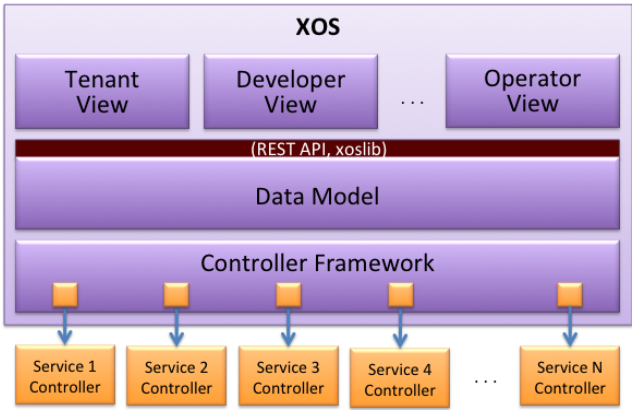

Figure 5: *Block diagram of the XOS software structure.*

The Data Model encapsulates the abstract objects, relationships among those objects, and operations on those objects. The operations are exported as a RESTful HTTP interface, as well as via a library (*xoslib*) that provides a higher-level programming interface. On top of this Data Model, a set of *Views* defines the lens through which different users interact with XOS. For example, Figure 5 shows a view tailored for tenants, one tailored for service developers, and one customized for service operators. Finally, a *Controller Framework* is responsible for distributed state management; that is, keeping the state represented by a distributed set of service controllers in sync with the authoritative state maintained by the Data Model.

The Controller Framework is a critical component of XOS—it binds the logically centralized authoritative state to the rest of the system, synchronizes the policies specified in its upper levels to it lower-level mechanisms, and keeps the policies themselves consistent. Instead of transmitting changes in the authoritative state in the form of deltas sent out to various service controllers, it computes these deltas from the service controller's vantage point. With this strategy, the authoritative state of the system can be determined unambiguously at any given time, even if the operational state of the rest of the system is lagging behind, or is even erroneous.

Furthermore, any consistency properties encoded as policies in the form as functions of the state can be applied to the central state independent of the service controllers. This decoupling of managing the central state and orchestrating backend mechanisms is instrumental in enabling the logical centralization of distributed resources. And the resulting parallel construct is not an accident. Just as individual services have a logically centralized controller and a scalable set of instances, XOS is structured to provide a logically centralized interface on top of a cloud-wide set of services.

The current implementation of XOS uses a combination of open source software and commodity services. The Data Model is implemented in Django, and leverages a substantial Django ecosystem (e.g., Django Suit is used to automatically generate an admin GUI). Views are Javascript programs running on the user's browser, where xoslib is a client/server library that uses Backbone.js and Marionette.js over the HTTP-based REST API. The Controller Framework is a from-scratch program (called the Observer) for executing service controller plug-ins. It leverages Ansible to handle low-level configuration with the back-end controllers.

XOS runs on top of a mix of service controllers, some instantiated as part of an XOS deployment, and some made available by commodity providers. Today, these include OpenStack (Nova, Neutron, Keystone, Ceilometer, and Glance), EC2, HyperCache [12] and RequestRouter [17] (proprietary CDN services from Akamai), ONOS [13] and OpenVirtex [2] (a network operating system and network hypervisor, respectively), and Syndicate [15] (a research storage service built on top of S3, Dropbox, HyperCache, and Google App Engine). We have also prototyped multi-tenant services using several open source projects, including Cassandra, Kairos, Swift, and Nagios.

3. ABSTRACTIONS

Returning to the claim that XOS provides a useful collection of OS abstractions, this section describes the support XOS provides to first implement services, and to then fold those services back into XOS for other applications to build upon. Because these abstractions are represented in the XOS Data Model, this section is organized around the key objects in the model, where for each, we give a high-level pseudo-specification. (A detailed definition of the resulting REST API is available online at http://portal.opencloud.us/docs/.)

3.1 Resource Containers

XOS provides resource containers (*Slices*) in which programs (*Services*) run. Services are a first-class abstraction (object) in XOS, meaning they are explicitly created, named, and managed. This makes it possible to define relationships among a collection of services, for example, to say that Service A "composes with" Service B. We summarize the key objects that comprise a service as follows:

```
Service = ({Slice,...}, Controller)
Slice = ({VM,...}, {VN,...})
VM = (Placement, Image, Resources)
VN = (Topology, NetworkOS, Resources)
Controller = (URL, Credentials, Plugin)
```

A Service is defined in terms of two other objects: (1) a set of one or more Slices, each of which is a resource container in which the instances that implements the service runs, and (2) a Controller that represents the XOS end-point for the service's controller. Most services run in a single slice, but XOS allows for the possibility that a given service has been factored into a set of slices—because each sub-program benefits from being isolated in its own resource container—akin to a Unix application running across multiple processes.

A Slice is defined in terms of two additional objects: (1) a set of Virtual Machines (VM), each of which is instantiated on some physical server, and (2) a set of Virtual Networks (VN), each of which is embedded in the underlying physical network and interconnects the VMs in the slice. Both VMs and VNs isolate services from each other, where XOS is (indirectly) responsible for allocating resources to slices. We say "indirectly" because still other services imported into XOS are directly responsible for resource allocation, but these services operate under the control of XOS. For example, a common configuration uses OpenStack's Nova service

to directly implement XOS VMs and OpenVirteX (a network hypervisor) to directly implement XOS VNs.

Service developers are given considerable operational control over their slices. For example, they can control VM placement, what image is booted in each VM, and how many resources (e.g., cores) are to be allocated to each VM. Similarly, they control virtual network topology, what NetworkOS (e.g., ONOS [13]) controls each virtual network, and how many resources (e.g., link bandwidth) are to be allocated to each VN. Note that the VM Image and VN NetworkOS are parallel constructs: Both a VM and a VN are empty containers unless "booted" with the appropriate Image and NetworkOS, respectively.

The Controller bound to each Service records configuration state needed to invoke operations on the back-end service controller. There is also a Python plug-in that executes in the XOS controller framework, invoking operations on the back-end controller according to state it sees in the XOS data model. Note that it is possible to import an existing/external service into XOS by creating a Controller object for it, but without creating a slice in which the service runs. This is how we incorporate a service like S3 or EC2 into a given deployment of XOS. (See Section 3.4 for elaboration.)

3.2 Service Composition

The power of building a system from modular components is realized when the system provides a means (framework) for composing those components. XOS provides explicit support for composing services, or to be more specific, for service providers to declare that "Service A is a tenant of Service B." We represent this relationship in the XOS Data Model as a *Tenant* object:

$$\text{Tenant} = (\text{Service}_T, \text{Service}_P, \text{Connect}, \text{Attributes})$$

The first two fields identify the tenant and provider services, respectively. The third argument indicates how the two services are to be connected in the underlying network and the last argument records state necessary to implement the tenancy. These last two fields are explained below.

While multi-tenancy is a staple of cloud services, there are typically two assumptions that do not apply in the general case that XOS addresses. The first assumption is that the tenant is a user (e.g., John Smith is a tenant of Amazon's EC2 service). Tenancy raises additional challenges when the tenant is itself an elastically scalable service; all the service instances must collectively be able to access the provider service. The second assumption is that all services are autonomous—that is, each is an independently operated service that runs *on* some cloud (e.g., EC2). But XOS is also designed to support services that are *part of* the cloud, which more closely corresponds to the ecosystem of services offered by AWS (as opposed to runs on AWS). These services build upon each other and are all operated by a single cloud provider (Amazon).

While we could leave these challenges to the individual services to address, XOS provides mechanisms that lower the operational costs of Service A being a tenant of Service B under these more general circumstances. We broadly characterize them as enabling composition in the data plane and enabling composition in the control plane.

The first aspect of service composition is data plane connectivity: the ability of one service to connect to (and exchange packets with) another service. All services in XOS are connected to one or more virtual networks. These virtual networks are designed to provide isolation. This is the primary role they play in a multi-tenant cloud, but for two services to compose means that the constituent service instances (i.e., VMs) must be able to communicate with each other. This could happen over the public Internet, as happens on public clouds, but doing so violates the principle of least privilege. There are deployment scenarios in which internal services are composed with each other, but without offering a publicly reachable interface.

XOS provides three ways to interconnect a pair of services (as indicated by the *Connect* field of the Tenant object). The first is the default on commodity clouds: the services communicate over the public Internet. The second is to create a single VN that is shared by two or more slices. Such a shared VN interconnects the union of all VMs belonging to the participating slices. The third leverages OpenVirteX to install the appropriate flow rules in the underlying switches so as to pass packets from one VN to another. The "gateway" between the two VNs is logical—packets do not traverse a "router process" as they cross from one VN to another.

The second aspect of service composition is control plane tenancy: managing the tenancy of one service relative to another. XOS provides mechanisms that address two challenges. First, each instance of service A (i.e., each VM that implements A) must have the requisite tenancy credentials to access B. For example, if Service A is a tenant of a scalable storage service (B), then each VM in A needs the credentials that allow it to read and write data stored in the VMs of B. Mechanistically, XOS records the all tenancy state corresponding to A being a tenant of B in its Data Model (the *Attributes* field of the Tenant object) and has a means to distribute this state to all the service instances.

Second, one service might need to take some action when one of its tenants changes its instances. For example, if Service A takes advantage of a scalable storage service (B) that mounts volumes in each of the service instances (VMs) that implement A, then Service B needs to be alerted when Service A adds a new VM to one of its slices. Mechanistically, because this dependency is explicitly recorded in the Data Model, when there is a change in the state maintained for Service A, XOS notifies the Controller plug-in for Service B, thereby giving it an opportunity to take service-specific actions (e.g., mount a volume in the newly created VM belonging to Service A).

Finally, by focusing on Services as tenants we do not mean to imply that users cannot also be tenants. Although not described in this paper, XOS includes a *User* object, where Users can be assigned a set of *Roles*. This includes Roles with sufficient privilege to operate a service. The key is that XOS decouples tenants and users, with former representing a virtual instance of a service and the latter being one example of a principal that can request tenancy in a service.

3.3 Programming Environment

Analogous to a Unix shell, XOS provides a programmable environment for creating alternative views of the set of services represented in the XOS data model. Views are typically tailored for a particular user community or workflow. For example, we have implemented a traditional cloud tenant view (it mimics the EC2 interface and is tailored for someone that only wants to acquire a set of VMs to run a scalable application); a service developer view (it is designed for someone that requires low-level control over VM placement, VN topology, and service composition); a service operator view (it is designed for someone that is managing an already developed and deployed service); and a cloud operator view (it lets operators define global policies and configuration parameters for a suite of services).

Views are themselves represented in the XOS data model. They defined by a URL that renders the view, and a type that indicates whether the view is implemented as Javascript or an iframe imported from some other web page.

$$\text{View} = (\text{Type}, \text{URL})$$

Unix	XOS	Comments
Processes	Slices (VMs + VNs)	Resource container that provides isolation
User, setuid	Tenancy	Access control in the Control Plane
Shell	Views	Programming environment for composition
Devices	Controllers	Means to connect new resources
Applications	Services	Run on top of OS / Fold back in OS
Syscall interface	REST API	Interface to kernel / Data Model
libc	xoslib	Common building block library

Table 1: Unix-to-XOS comparison.

Views access and control one or more scalable services, but a view is not itself a scalable service. It is an interface (portal) that runs in the end-user's browser. For any function that requires scalable computation or state, the XOS design philosophy is to (1) encapsulate the scalable aspects of the function in a service (with a corresponding service controller and scalable set of service instances), (2) import the service into XOS via a Controller object, and (3) run only the interface to that service in a view, accessing the service controller indirectly through xoslib and the underlying XOS data model.

3.4 Importing Resources

The abstractions described to this point are about defining an environment for programmers building services on top of XOS. Like any operating system, XOS must also manage a set of underlying hardware resources. In our case, this means importing Infrastructure-as-a-Service (IaaS) into XOS.

Fortunately, IaaS is no different than any other service in XOS. A given IaaS-based cloud exports a controller that is, in turn, imported into the XOS data model, with a corresponding Controller plug-in running in the XOS Observer. For example, we have imported resources from both EC2 and multiple OpenStack clusters into a global deployment of XOS called OpenCloud. Each of these resources sets is modeled in XOS by an abstract object, called a *Deployment*, but in terms of underlying XOS mechanisms, a Deployment is no different than any other service, it just happens to make infrastructure resources available across some set of sites.

Deployment = (Controller, Sites, Policies)

OpenCloud is an operational system that spans the four tiers depicted in Figure 1. The first tier (right-most in Figure 1) is a widely distributed set of edge sites. These sites are currently located at Universities and research labs, and each runs as an independent OpenStack cluster. The second tier is a set of eleven small clusters co-located in routing centers of Internet2. All eleven sites run as a single distributed OpenStack cluster. The third tier is a set of five modest data centers—two on the east coast of the US, two on the west coast of the US, and one in Europe—each of which runs as an independent OpenStack cluster. The fourth tier (left-most in Figure 1) is a set of available commodity clouds. (Figure 1 shows two for generality, but currently only EC2 has been imported into OpenCloud.)

3.5 Summary

We conclude with a brief summary of mechanisms that XOS provides, and how they map onto their Unix counterparts (Table1). Using Unix as a model for XOS is not perfect, but the mapping is helpful in understanding the roles played by the various XOS components. Interestingly, while we consciously adopted the Unix design philosophy as we defined XOS—particularly with respect to extensibility—we were not trying to match Unix feature-for-feature. Recognizing the per-component mapping happened after we had completed XOS and started using it to build and compose services. That this correspondence between Unix and XOS holds at both the design level and the mechanism level is reassuring.

4. EXAMPLES

This section walks through three examples of how we configure and use XOS, which sets the table for discussing how services benefit from XOS (and vice versa).

4.1 End-to-End Path

The first example takes a narrow view (Figure 6), showing the path through XOS corresponding to a single service. At the top, a View might define a GUI through which users access the service. This view is implemented in Javascript and makes reference to (reads and writes fields from) a service-specific object in the XOS data model. As the state of this object changes, a service-specific plugin running in the controller framework observes the changes and makes the necessary calls on the back-end service controller. The controller plugin has access to related state in the XOS data model, including the Controller object bound to the Service object (giving it the credentials it needs to talk to the back-end controller) and the Slice object bound to the Service objects (giving it the information it needs to identify the VMs and VNs that correspond to service instances). The service controller then manages the set of service instances running in a set of VMs.

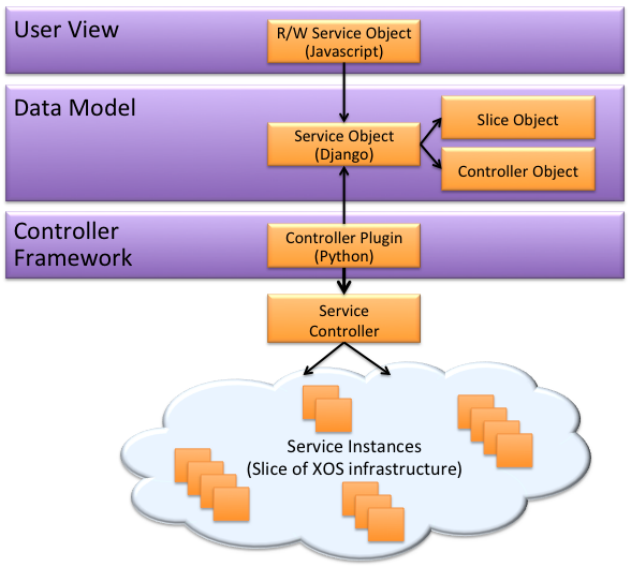

Figure 6: *End-to-End path through XOS for a single service.*

Figure 6 is a simple example, but it provides a frame of reference for discussing several variations. First, XOS can include external

services like S3 and EC2, as long as they support a well-defined controller interface. Doing so makes such services available to Views through *xoslib*, thereby making it easier for others to build on them. In this case, the service would not run on the XOS-managed infrastructure (i.e., the slice in Figure 6 would be null), with the implementation hidden behind the external service controller.

Second, XOS can be configured to provide a service controller for services that do not already have one. This would be the case if the service instances are managed by directly installing a configuration file on each instance, as is the case with single-tenant open source services like Cassandra and Kairos. In such cases, XOS implements the service controller as a combination of the corresponding Service object (which defines the service's interface) and the controller plugin (which pushes the necessary configuration information to the individual instances). Note that supporting a logically centralized controller interface is not sufficient for creating a multi-tenant service. The underlying implementation must also support some form of tenant abstraction. For example, our prototype added a "KeyStore" tenant abstraction to Cassandra.

Third, while we have been describing views as graphical interfaces through which human users interact with XOS, it is also possible to build views that interface with other programs, for example, high-level decision-making applications (commonly called OSS/BSS in telecommunication environments) that monitor the load on a service and decide to scale the service up or down. To this end, we anticipate implementing views that support existing service modeling languages (e.g., yang[19]).

Fourth, XOS provides a collection of mechanisms to support tenancy, but any given service is free to use only the subset that it needs. If we imagine a parallel tenant service stack composed with the provider service stack in Figure 6, then these mechanisms include: (1) interconnected virtual networks in the infrastructure (enabling communication in the data plane); (2) the controller plugin for the provider service would have a reference to the Service object for the tenant service (enabling the provider to take any necessary action should the tenant Service object change); and (3) XOS pushes credentials for provider service to the instances for the tenant service (enabling the tenant to request service from the provider).

4.2 Service Ecosystem

The second example takes a broad view (Figure 7), across the set of services configured into the OpenCloud deployment of XOS. Each node in Figure 7 represents a service and the edges represent the "tenant of" relationship. For example, HyperCache (HPC) is a tenant of the RequestRouter (RR) service.

This particular configuration leverages three external services (shown in red): S3, GAE (Google App Engine), and EC2. They are external in the sense that their instances do not run on XOS-managed infrastructure. Also, we denote OpenStack in brackets to indicate that there are a set of OpenStack clusters. NewService (in green) denotes a new service someone might create. In this case, it is a tenant of both XOS (which provides multi-site IaaS) and Syndicate (which provides a secure bootstrapping service).

We make four high-level observations. First, it is accurate to view XOS, itself, as a multi-tenant service. Its primary tenant abstraction is called a Service, analogous to Volumes being the tenant abstraction of a scalable storage service and KeyStores being the tenant abstraction of a NoSQL database. Moreover, XOS is itself a tenant of multiple IaaS platforms. This gives us a convenient way to think about XOS: it runs on top of a collection IaaS-based services and offers tenancy to higher level services that want to be deployed across the breadth of underlying clouds.

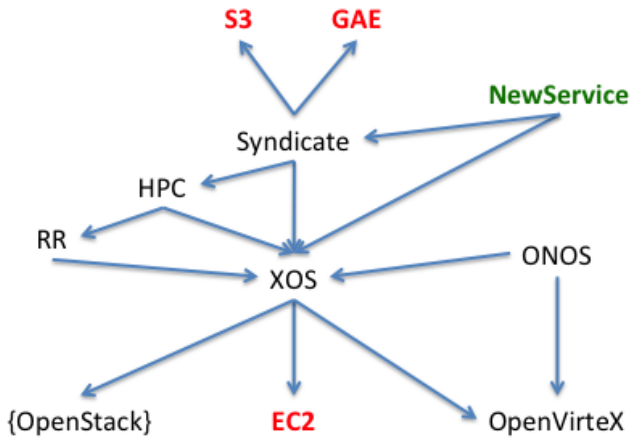

Figure 7: *Tenant relationship among a set of services.*

Second, the service composition graph shown in Figure 7 represents an important security policy statement made by the cloud operator. Specifically, the tenancy objects that the graph model include directives about how services are connected in the data plane—that is, how they are connected by isolated virtual networks. The interconnection of VMs and VNs (not shown in the Figure) are derived from this specification.

Third, the tenancy graph is a static, operator-defined specification of the available services and the dependencies among those services, but it neither specifies nor restricts how combinations of services can be leveraged by a View on behalf some user community. For example, one might create an "Analytics View" that leverages Syndicate as a data store and EC2 for its compute resources, loading Hadoop into each EC2 VM. Used in this way, XOS Views are effectively a programming environment for creating customized Platforms-as-a-Service.

Fourth, making the tenancy graph explicit does not ensure that service APIs and semantics remain in sync as they evolve over time. An operator can run multiple versions of a given service and incrementally migrating tenants from one version to another, but the evolution of a given service remains outside the purview of XOS. Understanding the service life cycle and identifying opportunities for XOS to support in-service-software-upgrades is a promising direction for additional research.

4.3 Central Office as a Datacenter

The services in the previous subsection informed XOS's design. This section presents another example that shows how we have recently started using XOS to address a much different usage scenario: re-architecting the Telco central office as a datacenter.

The central office is the facility at the edge an Telco network. For example, AT&T operates over four thousand central offices around the US, each of which serves 10–100k residential customers. With a diverse collection of network devices and appliances that have accumulated over the last 50 years, the central office is a enormous source of capital and operational costs for Telcos, which would like to take advantage of the same economies of commodity hardware and agility as cloud providers [8].

To this end, we are prototyping the service tenancy graph depicted in Figure 8. It includes three new services (vOLT, vCPE, and vBNG), which correspond virtualized incarnations of three physical devices currently deployed in Central Offices: Optical Line Termination (OLT), Customer Premises Equipment (CPE), and Broadband Network Gateway (BNG), respectively. Although an in-depth

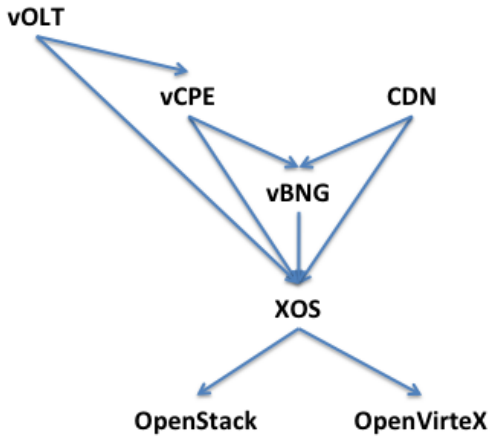

Figure 8: *Tenant relationship among a set of services running in a Telco Central Office.*

description is beyond the scope of this paper, the resulting three multi-tenant services are implemented on a leaf-spine fabric of white-box switches and scalable software running in VMs on commodity servers.

vOLT – Terminates the optical link in the Central Office and authenticates customer access. Each tenant corresponds to a *Subscriber VLAN*.

vCPE – Runs a bundle of functions (e.g., Firewall, DHCP, NAT, VoIP, Parental Control) on behalf of a given subscriber. Each tenant corresponds to a *Subscriber Bundle*.

vBNG – Connects subscribers (and other cloud services running in the central office) to the public Internet. Each tenant corresponds to a *Routable Subnet*.

That is, a subscriber becomes a tenant of vOLT (acquiring a VLAN), which becomes a tenant of vCPE (acquiring a service bundle that is attached to that VLAN), which becomes a tenant of vBNG (acquiring a routable subnet). Figure 8 also includes a CDN service, which corresponds to a combined HPC+RR service from earlier. The CDN service also uses vBNG to acquire routable addresses, thereby allowing it to acquire content from origin servers when there is a cache miss.

There are two important takeaways from this example. First, XOS is general enough to be applied to a significantly different use case than it was originally designed to support. Most notably, this involves applying the Everything-as-a-Service organizing principle (with multi-tenancy) to what has historically been organized and operated at the device level.

Second, being able to isolate services on private virtual networks is an important part of the central office's security architecture. In this case, the VMs that implement the vOLT and vCPE services are connected by one VN that is not publicly routable, while the vCPE and CDN services are connected by a second VN that is publicly routable. vBNG is the scalable service that implements routing for this second VN. It is essentially a logical router—implemented as a control application running on ONOS, that in turn installs flow rules in the underlying switching fabric—connecting the other services running inside the datacenter to the rest of the Interent. Although the initial prototype is minimal, vBNG generalizes to also support QoS, VPNs, and various forms of tunneling.

5. DISCUSSION

This section discusses the value the XOS architecture provides for the cloud, and in doing so, draws additional parallels between XOS and well-understood operating system design alternatives.

5.1 Everything-as-a-Service

Nearly all of the capabilities attributed to XOS in the proceeding sections are provided by services; the XOS core merely provides a framework for accessing them. The key value of this approach is its ability to deal with a space of new facilities—namely, the cloud—whose capabilities is still being defined. Rather than claiming that we can predict these capabilities in advance, we propose that such capabilities evolve over time, with services as the building blocks.

This is the case for acquiring resources (which leverages infrastructure-as-a-service like OpenStack and EC2), as well as for creating SDN-based virtual networks (which leverages the OpenVirteX and ONOS as examples of network-as-a-service). It is also the case for securely bootstrapping and configuring services (which leverages the Syndicate storage service, which in turn builds on the XOS-hosted RequestRouter and HyperCache services, along with external services like S3 and Google App Engine); providing a load balancer to help services scale (which leverages RequestRouter, but could use a directory service like ZooKeeper or load balancing application running on top of ONOS); and a monitoring service (which leverages a combination of Nagios, Ceilometer, and the Google BigQuery service).

Again, XOS merely provides the framework for integrating these services into a cloud, but any function that needs to scale is implemented as a service. This, as well as all of the other conveniences mentioned above, are obtained automatically when a developer uses the service abstraction to build his service.

5.2 Multi-Tenant Services

In order for capabilities to evolve with services as building blocks, services have to link to one another. In XOS, the link between services is captured by tenancy. A service that leverages another service becomes a tenant of that second service.

The focus on multi-tenant services is a key characteristic of XOS, and differentiates it from service deployment tools like Docker, which provide mechanisms for composing single-tenant services on behalf of a single user/application. The best way to compare the two approaches is that they represent well-understood OS structures: XOS most closely matches a microkernel-based OS (single instantiation of each multi-tenant service), whereas Docker most closely matches a library-OS (multiple instantiations of single-tenant services).

While microkernels fell out of favor in the 1990s, largely due to performance penalties on single-core/single-machine deployments serving single-user workloads, the multi-core/multi-machine/multi-tenant environment of the cloud makes the microkernel-inspired structure competitive today. Moreover, isolating functionality in distinct multi-tenant services is more consistent with the principle of separating concerns—as a tenant, I am not responsible for maintaining or scaling the services upon which I depend. In the end, we will likely see a combination of both approaches, but we do observe that the same forces driving the industry from software-as-a-library to software-as-a- service argues in favor of treating everything-as-a-service.

5.3 Bootstrap and Evolve

The evolution of the Cloud must begin with a set of base capabilities as a starting point. To this end, XOS heavily leverages OpenStack, which raises the question of what additional value it pro-

vides. One perspective is that XOS is a distribution of OpenStack, bundled with a particular set of configurations, building block services, and policies. Another perspective is that XOS defines a meta-cloud that includes OpenStack as just one of possibly many base clouds—some offering infrastructure (EC2) and others offering higher level services (S3). And while it is certainly possible that OpenStack over time will also gain such support, XOS is purposely designed to combine these base clouds, along with other building block services built on top of this base, in arbitrary ways.

Both of these perspectives are accurate, but they don't capture the full value of XOS. One analogy that sharpens the differentiation is that if XOS is the microkernel of the cloud and Docker is the library-OS of the cloud, then OpenStack is its monolithic kernel. It is easy to see examples of this OS structure today—for example, the tight integrated of Nova, Neutron, and Keystone—but because the system continues to evolve, we argue that the open source community would be best served by evolving OpenStack towards a more principled design in which multi-tenant services are treated as first-class objects. XOS defines a starting collection of core abstractions and mechanisms in support of such a design, thereby enabling the widest combination of third-party services and new capabilities created through service composition.

6. CONCLUSIONS

XOS is a general-purpose operating system for a multi-tier cloud. Our goal is to enable an evolving service ecosystem. The central characteristic that makes this possible is a well-defined model for extensibility. That XOS resembles Unix in this respect demonstrates the applicability of the saying: *Those who ignore Unix are destined to reinvent it.*

We believe a combination of abstractions constitutes an OS when they converge to a steady state. In Unix, the combination of files, pipes and processes supports both isolation and composition. In XOS, isolation and composition are the result of services, virtual networks, and slices. Unix provides shells and XOS provides a javascript abstraction layer built on top of the REST API. The parallels go on.

In this light, XOS can itself be seen as a multi-tenant service, where its tenants are services. You could say that while XaaS was the goal of the design, its outcome is a related but more powerful paradigm: *Service-as-a-Service*.

7. REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX 1986 Summer Conference*. USENIX, 1986.

[2] A. Al-Shabibi, M. DeLeenheer, A. Koshibe, G. Parulkar, W. Snow, M. Gerola, and E. Salvadori. OpenVirteX: Make Your Virtual SDNs Programmable. In *ACM SIGCOMM Workshop on Hot Topics in Software-Defined Networking (HotSDN)*. ACM, 2014.

[3] Amazon AWS. `http://aws.amazon.com`.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 205–218, 2006.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX, 2012.

[7] Docker. `https://www.docker.com/whatisdocker/`.

[8] AT&T Vision Alignment Challenge Technology Survey (AT&T Domain 2.0 Vision White Paper). `http://http://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf`.

[9] Google Cloud Platform. `https://cloud.google.com`.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[11] Google App Engine. `https://developers.google.com/appengine/`.

[12] Akamai HyperCache. `http://www.akamai.com/html/solutions/hypercache.html`.

[13] B. L. Lantz, B. O'Connor, J. Hart, P. Berde, P. Radoslavov, M. Kobayashi, T. Koide, Y. Higuchi, M. Gerola, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *ACM SIGCOMM Workshop on Hot Topics in Software-Defined Networking (HotSDN)*. ACM, 2014.

[14] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[15] J. Nelson and L. Peterson. Syndicate: Virtual cloud storage through provider composition. In *Proceedings of BigSystem 2014*. ACM, 2014.

[16] OpenStack. `http://www.openstack.org/software/`.

[17] Akamai Request Router. `http://www.akamai.com/html/solutions/request-router.html`.

[18] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7), July 1974.

[19] YANG – A Data Modeling Language for the Network Configuration Protocol (NETCONF). `http://tools.ietf.org/html/rfc6020`.