XOS: Modeling-as-a-Service

Abstract

XOS is a logically centralized, model-based service for operating micro-service based cloud applications. XOS complements cloud orchestration tools like Kubernetes, which manage the set of containers that *implement* each micro-service, focusing instead on managing the state necessary to *configure and control* the collection of micro-services as a whole, and doing so at the granularity of individual subscribers. In this sense, XOS can be thought of as augmenting the micro-service based architecture with an *Modeling-as-a-Service* layer.[1]

Value Proposition

XOS defines the single source of truth for all configuration and control information for an application constructed from a set of micro-services. By delegating this responsibility to XOS, individual micro-services benefit from support for saving and recovering configuration state, as well for migrating that state during an in-service-software-upgrade.

But XOS's primary value is to operators that specify and then manage the cloud application as a whole, where a logically centralized data model simplifies the task of reasoning about an extensible graph of micro-services. Importantly, this includes the ability to control the system at the granularity of individual subscribers. The resulting system provides advantages across three dimensions:

- **Uniformity** → XOS generates a uniform, application-wide configuration and control API across an extensible set of micro-services. This makes it easy to:
    - Incorporate the application into an existing operating environment as a single unified entity.
    - Dynamically extend the set of services configured into a deployment, also without introducing OA&M silos.
    - Reason about relationships and invariants across the system as a whole, thereby eliminating the unstated assumptions that often lead to operator errors.
- **Programmability** → XOS provides a programmable framework for modeling and operating on services. This makes it easy to:

[1] A concluding section discusses why we characterize XOS as providing *Modeling-as-a-Service*.

    - Create logical services that do not map onto conventional micro-services, including functionality implemented in a programmable switching fabric.
    - Create composite services by composing multiple micro-services and other disaggregated functionality.
    - Create federated or distributed services that span multiple cloud instantiations.
- **Granularity** → XOS tracks fine-grain context across a sequence of micro-services, forming an end-to-end *service chain* (also called a *network slice*). This makes it easy for:
    - Individual subscribers have visibility into, and control over, their end-to-end service chains.
    - Operators to correlate diagnostic and monitoring information at the granularity of individual subscribers.
    - Operators to allocate resources and isolate performance on a per-subscriber (or subscriber class) basis.
    - Operators to migrate service chains in support of mobile users.

The value XOS provides is fully realized as the cross-product of all three dimensions: Being able to operate a highly configurable system (not just a single/fixed application) built from disaggregated components (including both conventional micro-services running on servers and control applications programmed into the switching fabric), while preserving visibility and controllability at the granularity of individual users.

Background

XOS was originally developed to support service composition in a multi-cloud environment, with some services implemented on-premises, some in commodity cloud datacenters, and some at widely distributed edge sites [1]. In this context, the underlying service implementations could run on different cloud infrastructure (e.g., OpenStack, EC2), as well as be self-contained and independently managed cloud services (e.g., S3, DynamoDB).

Today, the dominant use case for XOS is CORD, an *access-edge cloud* designed for network operators (Telcos and CableCos) [2]. CORD is unique in that in addition to implementing functionality in micro-services, some functionality is also implemented in programmable switches. In this context, XOS implements the service controller [3,4], where the unique requirements of running at the network edge strongly influenced the design [5]. The following summarizes those requirements, and the two design decisions made to support them.

1. CORD was designed to support a wide range of service implementations that are required at the access-edge. This includes legacy VNFs running in VMs, horizontally scalable micro-services running in containers, per-subscriber context isolated in containers or lambdas, and SDN control applications installing flow rules in the switching fabric. To this end, XOS adopted an abstract and implementation-agnostic definition of services.

2. CORD was designed to be a general and extensible platform, able to support a wide range of solutions (e.g., for Residential, Enterprise, or Mobile deployments). To this end, XOS provides a means for operators to configure a given deployment by specifying a service graph of interconnected services, as well as to extend the set of services.

   > ***Design Decision*** → *To cleanly separate the* ***service control plane****, which XOS implements, and the* ***service data plane****, which leverages existing infrastructure services (e.g., OpenStack), container management systems (e.g., Kubernetes), network operating systems (e.g., ONOS), and external cloud resources (e.g., EC2, GCP). The former defines the single source of truth about how the system should behave, and the latter implements the operational system according to these directives.*

3. CORD was designed to allow multiple stakeholders to manage different aspects of an operational system, including support for multi-tenancy. To this end, XOS allows the operator to define the invariants and configuration parameters that govern the behavior of a CORD deployment as a whole, and individual subscribers (end-users) to control the specific “slice” of CORD that provides network access on their behalf.

4. CORD was designed to be a self-contained edge cluster, able to plug into any operator’s global OSS/BSS and orchestration platform. To this end, XOS auto-generates one or more northbound interfaces that mediate all access to the individual components configured into a given deployment.

   > ***Design Decision*** → *To represent the authoritative state of the system and operations on that state with a* ***declarative data model****. All relationships and invariants are explicitly represented in the model schema, with the objects that instantiate these models being the single source of truth for a running system. Both operator-facing and subscriber-facing interfaces are auto-generated from these models. Service developers then write plugins that keep their individual micro-services synchronized with the authoritative data model.*

Design

XOS implements the two key design decisions made for CORD: (1) it decouples how services are implemented (and the infrastructure tools used to implement them) from the configuration and control of those services; and (2) it represents all control and configuration state with a

centralized data model, and allows declarative operations on those models to implement high-level abstractions (e.g., service chains, network slices, logical services, federated services, composite services).

The core of XOS consists of four elements, implemented in ~25k lines of code:

1. ***xproto*** → A declarative language for specifying models. This language is based on ProtoBufs*,* extended to also support inheritance, relationships between models, and first order logic operators on models.
2. ***xosgenx*** → A toolchain for imposing and enforcing the abstract models on an operational system. This toolchain supports an extensible set of code generation targets, including the following defaults:
    - A set of Northbound Interfaces for accessing the system, including a GUI, a REST API, an endpoint for importing TOSCA workflows.
    - The Object Relational Mapping used to map the abstract data model onto a persistent store.
    - A Synchronizer Framework that individual services use to keep their backend services in sync with the XOS-managed data model.
3. ***core.xproto*** → A default set of core models that includes a general definition of *Services*, *ServiceInstances*, and *ServiceDependencies*. These core models codify several years of experience managing systems constructed from a wide-range of service implementations like CORD, but they are malleable by design. This makes it easy for a given system to evolve the definition of what constitutes a service and how services are composed.
4. ***Chart.yaml*** → A set of Helm Charts (plus container images) that deploys XOS into a Kubernetes cluster. These include defaults for three “helper” micro-services: an event bus (redis), a database (Postgres), and a logging framework (ELK Stack).

Given this core, XOS provides service developers with an SDK that they use to write two elements:

1. ***service.xproto*** → A service-specific model, expressed in xproto.
2. ***Dockerfile.synchronizer*** → A *Synchronizer* that keeps the backend micro-service (or other implementation of the service) synchronized with the declarative state defined in the XOS data model. Synchronizers run as a micro-service and consist of two plugins: *sync_step( )* and *model_policy( )*.

More information about these two elements can be found in the *Defining Models* section of the CORD Guide [2].

Roadmap

The XOS core is relatively complete and stable, but how it is packaged, and then applied to different scenarios, is still a work-in-progress. The following implementation tasks and design decisions are currently being addressed.

Packaging

Work is needed to package XOS as a self-contained component:

- Today service developers work with XOS as part of the CORD build system. We need to re-package XOS as a self-contained SDK, breaking the current bundling with CORD's build system.
- Today XOS comes with some baked-in helper services (e.g., redis, Postgres, ELK Stack). Ideally, it should be possible to substitute equivalent counterparts that are already included in the application (e.g., Kafka for redis), rather than making these helper services fixed.
- Today XOS presumes synchronizer plugins are written in Python. Ideally, it should be possible to support other languages, and in general, better isolate and scale synchronizers.

Kubernetes Integration

XOS treats Kubernetes as an exemplar for how to split responsibility for a service between the service data plane (Kubernetes) and the service control plane (XOS). Because XOS is itself a micro-service, it could be used in conjunction with any container management system (e.g., Docker Swarm, Mesos), but we are taking a Kubernetes-first approach.

From a layering perspective, both XOS and Kubernetes expose a Northbound Interface (NBI); that is, neither is layered on top of the other. This means cloud operators can use the Kubernetes NBI to deploy, replicate, scale, and upgrade the set of containers that implement micro-services, and the XOS NBI to control and configure the functionality running in those micro-services. XOS and Kubernetes are interdependent in two ways.

- Kubernetes is used to deploy, replicate, scale, and upgrade XOS. More specifically, we have defined Helm Charts to deploy XOS, along with the other platform services that XOS builds upon.
- XOS models various aspects of the Kubernetes architecture (e.g., deployments, pods, services, containers), which implies the need for a Kubernetes Synchronizer in the same way that XOS requires a Synchronizer for the micro-services it models. This Synchronizer both acts on Kubernetes to reflect the authoritative state specified in the

XOS data model, and acts on the XOS data model to reflect the operational state of the underlying Kubernetes-managed services.

The integration of Kubernetes with XOS is a work-in-progress, so there are open issues that still need to be resolved. For example, there is an issue of how to customize/extend Kubernetes to cover all the use cases XOS expects to support in CORD:

- Horizontally Scalable Services with Built-in Load-Balancing → Containers are spun up/down based on aggregate workload, with responsibility for load balancing requests across containers delegated to Kubernetes. This is the behavior Kubernetes Services support by default.
- Horizontally Scalable Services with External Load-Balancing → Containers are spun up/down based on aggregate workload, with responsibility for load balancing requests across containers delegated to an external load balancer (e.g., RequestRouter in the case of a CDN). This requires each instance be directly addressable. This is known as client-side load balancing, and although not available by default in Kubernetes, it is a common pattern in custom deployments that we will need to reproduce for XOS.
- Per-Subscriber Context → Containers are spun up/down on a per subscriber basis, with each container implementing subscriber-specific context (e.g., vSG in CORD). This includes being able to directly address each subscriber's container. There is no default support for stateful context management in Kubernetes so it will be necessary to develop a custom controller for XOS.

The first two usage patterns involve an operator-initiated action, while the third involves a subscriber-initiated action. Kubernetes assumes operators, and not end-users, create containers, so this is a significant departure. Moreover, it is necessary that XOS respond to subscriber-initiated requests, since it is the source of truth for subscriber-specific state.

Another open issue is how to best accommodate both VM-based and container-based VNFs, a requirement of CORD that we plan to pull forward for XOS. One option is to continue supporting OpenStack. Another is to delegate VM management to Kubernetes, for example, using KubeVirt. In both cases, it is necessary to interconnect VMs and containers, a capability that is currently implemented by the VTN control application running on ONOS. The current plan is to integrate VTN into Kubernetes using the Container Network Interface (CNI).

Istio Integration

Istio and XOS share a high-level objective of making it easier to manage cloud applications built from a set of micro-services, potentially spanning multiple clouds. Both systems decouple the service data and control planes, and both define an application in terms of a service graph (mesh). Beyond that, their respective approaches are largely complementary, with XOS's value-add being to (a) introduce a declarative data model to represent application-wide state,

policy and invariants; (b) support fine-grain service chains on behalf of individual end-users/subscribers; and (c) treat the control applications running in the underlying SDN network as first-class services that can be included in a service graph.

Integrating XOS and Istio will involve reconciling the following touchpoints:

- Istio defines a fine-grain security architecture. XOS complements Istio security by leveraging L2 virtual networks to isolate service interfaces on a per-tenant basis. These two approaches to security will need to be coordinated.
- Istio defines a monitoring architecture. XOS treats monitoring as an orthogonal function provided by one or more other micro-services, with ELK Stack included as a default. XOS will need to be extended to also take advantage of Istio's monitoring capability.
- Istio defines a policy enforcement architecture. XOS complements Isto policy decisions according to the models and invariants it maintains for the composite set of services. XOS will need to be packaged as an *adaptor* that plugs into the Istio policy architecture.

More investigation is needed, but integrating XOS with Istio is likely more complicated than implementing a single *Mixer* adaptor. XOS may also need its own sidecar to augment *Envoy*, an attribute namespace for passing control information among the various components, and a way to ensure that different classes of users can interact directly with the XOS NBI to set policy and control service chains.

Integrating with External Cloud Resources

While our focus is on micro-services, XOS can be applied to other uses cases, including ones that benefit from service chains in a multi-cloud environment. Such service chains might include on-premise and commodity cloud services, virtualized and physical middleboxes, and a combination of private and public network connectivity. Moreover, many of these services are likely to be running in cloud resources that are external to (not orchestrated by) XOS. With this in mind, the goal is to develop and publish a collection of XOS models and synchronizers for a representative set of services hosted in different clouds.

Modeling-as-a-Service

It has been a challenge to succinctly characterize the service XOS provides. This document settles on *Modeling-as-a-Service,* since at its core, XOS provides a mechanism to define a set of models and use them to impose a particular behavior on an underlying set of components.

But as deployed today, the first models loaded into XOS define a core set of service-related abstractions, so it is not unreasonable to conclude that XOS provides *Service-as-a-Service* (or

ServiceChain-as-a-Service). The paper introducing XOS as a multi-tier Cloud OS came to that same conclusion [1].

But once the core models are loaded, the next step is to load models for individual services, representing the components configured into the system. As a result, XOS provides operators with the means to configure and control the system as a whole, meaning XOS effectively supports *Operations-as-a-Service* or *ControlPlane-as-a-Service*.

These alternative descriptions point to two degrees of freedom that XOS provides. First, XOS does not directly execute model-based workflows (specified, for example, in TOSCA), but it instead takes a schema that defines the models (specified in xproto) as input, and generates the interpreter(s) needed to process any model-based workflow. Second, XOS does not have a hardcoded definition of what constitutes a service or service composition, but it instead bootstraps these abstractions from a malleable set of core models (also specified in xproto).

Put differently, XOS is more a *Controller Generator* than it is a specific Controller (or Orchestrator), but one that has proven particularly adept at providing visibility and control at the granularity of individual service chains across a heterogeneous collection of service implementations.

References

[1] XOS: An Extensible Cloud Operating System. *ACM BigSystems 2015*, June 2015.

[2] Central Office Re-architected as a Datacenter. *IEEE Communications, October 2016*.

[3] Defining Models for CORD. CORD Guide.

[4] Beyond Micro-Services: CORD's Model-Driven Design. CORD Design Note.

[5] CORD Architectural Requirements. CORD Wiki.