Service Assembly & Composition in CORD

Scott Baker, Andy Bavier, Sapan Bhatia, and Larry Peterson
Open Networking Lab

February 29, 2015

Introduction

The business case for CORD is rooted in service providers wanting to benefit from both the economies of scale (infrastructure constructed from a few commodity building blocks) and the agility (the ability to rapidly deploy and elastically scale services) that cloud providers like Amazon and Google enjoy today [1].

The first challenge in realizing CORD is to re-architect the legacy hardware to take advantage of commodity servers, white-box switches, and open source software. The second challenge is to forge the resulting collection of hardware and software elements into a coherent system that is economical, scalable, and agile. CORD defines a framework that addresses this second challenge, but it is more accurate to think of these challenges are two sides of the same coin: the abstractions that define the framework also inform how the system is factored into building block elements.

Our approach is inspired by successful cloud providers like Amazon and Google, who offer a large and varied collection of services (e.g., the AWS console lists 46 services), many of which are built by combining and leveraging other services in their portfolio. For example, Google App Engine is effectively a front-end to a collection of Google services that originally included BigTable, a NoSQL database built on the Google File System (GFS) and the Chubby Locking Service, but has recently been enhanced with a fault-tolerant datastore called Spanner and a new file service called Colossus.

To this end, CORD adopts a general organizing principle—*Everything-as-a-Service (XaaS)*—that models all functionality as scalable, multi-tenant services. XaaS is another way of saying “micro-services architecture” so we use the two interchangeably, although

we usually drop the “micro” and just say “services.”[1] Some services are purely building blocks used by other services, some are both building blocks and directly accessed by users, and some are purely user-facing.

CORD then provides the means to organize all of these services into a coherent whole, making it possible to create, name, operationalize, manage and compose services as first-class operations. It is through service composition that CORD lowers the barrier to creating new functionality from existing components.

Essentially, services are the software modules of the cloud, with a clean separation between (composable) interfaces and (hidden) implementations. The overarching goal of CORD is to provide a framework that makes it easy to build and manage scalable services based on the principle of XaaS. In pursuing this goal, four insights have influenced our design:

- **Unify SDN, NFV, and the Cloud:** Although initially conceived and developed as largely independent technologies, full agility is realized only when SDN (making the network control plane programmable), NFV (making the network data plane programmable), and the Cloud (elastically scaling programs based on demand) are seamlessly incorporated into a single coherent framework. Without a common framework, there is a risk that SDN-based, NFV-based, and Cloud-based functionality remain siloed and unable to seamlessly build on each other. The key insight is that services offer the most general blueprint for building scalable functionality, and so CORD models SDN, NFV and cloud applications in exactly the same way, including support for elasticity and multi-tenancy.

- **Support Multiple Actors:** To fully partition responsibility, the solution must accommodate multiple actors, including service developers (third-party service vendors) that implement functionality; service providers (cloud operators) that decide what services to deploy and control their operational parameters; and end-users (subscribers) that get value from those services. The key insight is that each actor requires a different perspective on the system, and so CORD defines a rich set of roles (i..e., access privileges on the underlying objects) and interfaces (i.e., control knobs) that allow each actor to control those aspects of the system that they are responsible for.

- **Factor State Management:** Operationalizing a portfolio of services on top of a set of cloud resources is a complex task. Much of this complexity boils down to distributed state management. Instead of viewing all state as being equal, the

[1] While there is no precise definition of micro-services architecture, the specific requirements for CORD are spelled out in a companion paper [2].

key insight is to distinguish between *authoritative state* that defines the desired behavior of the system and the *operational state* that defines the ongoing, fluctuating, and sometimes erroneous state of the actual system. CORD represents the authoritative state with a declarative data model that is easy for humans to reason about and automated tools to analyze, and provides a collection of imperative runtime mechanisms to continually "drive" the system's operational state towards this authoritative definition.

- **Leverage Virtual Networks:** End-users typically access commodity cloud services through the public Internet, but not all communication is between end-users and the services they access. There is also private intra-service communication among software components that implement the service, privileged management communication used to configure and control services, and service-to-service communication in support of composition. Moreover, it is sometimes the case that the network actually provides the service (i.e., the software components that implement the service *control* the network rather than *use* the network). The key insight is that virtual networks provide a powerful mechanism for building scalable services, and so CORD makes liberal use of virtual networks to both enforce the principle of least privilege (restricting each virtual network to just those entities that need to communicate) and implement functionality (software both controls and uses each virtual network).

These four insights follow from the requirements for CORD [2], and in particular, support CORD's security goals [3].

Layers of Abstractions

CORD defines a model—a layered collection of abstractions—that represents the services configured into a given Central Office. These abstractions impose a structure on the set of services in a way that allows operators to express and enforce policies on them. We sometimes say these abstractions form a *service control plane*.

XOS is the software component in CORD that defines a concrete representation for this structure. It includes a language for writing service control programs (policy statements) that one can reason about, and a runtime system that applies these policies to the operational system. In CORD's reference implementation, this includes an authoritative *Data Model* implemented in Django, both RESTful and TOSCA-based programmatic interfaces for specifying policy, and a *Synchronizer* that uses Ansible to keep the operational state of the underlying system in sync with the CORD's authoritative state.

The rest of this section introduces CORD's abstractions, with the following sections discussing the role they play in more detail and sketching the XOS implementation.

At the base, ONOS defines a network graph abstraction on top of a set of white-box switches and OpenStack defines a set of primitive cloud resources (e.g., instances, images, flavors, networks, ports) on top of a set of commodity servers. On top of this foundation, CORD defines three layers of abstraction as summarized in Figure 1:

- **Slice:** Represents a system-wide resource container, including the means to specify how those resources are embedded in the underlying infrastructure. CORD models a slice as a set of *Virtual Machines (VMs)* and a set of *Virtual Networks (VNs)*. Slices are similar to OpenStack tenants or projects, except they permit more operational control, especially with respect to VNs.[2]

- **Service:** Represents an elastically scalable, multi-tenant program, including the means to instantiate, control, and scale functionality. CORD models a service as a *Service Controller* that exports a multi-tenant interface and an elastically scalable set of *VMs* interconnected by one or more *VNs* (the set of VMs and VNs are collectively represented as a Slice). XOS includes mechanisms to assemble a CORD-ready service from both greenfield and legacy components.

- **Service Graph:** Represents a dependency relationship among a set of services, including the means to create new functionality by composing building block services. CORD models service composition as a *Tenancy* relationship between a *provider* service and a *tenant* (i.e., client) service. Service tenancy is anchored in a *Tenant Principle* that may be bound to one or more authenticated *Users*.

CORD also defines a programming environment—including **User**, **Tenant**, and **Role** abstractions—that makes it possible to customize access for different actors, such as service developers, network operators and end-users. Support for role-based access control makes it possible to associate fine-grained privileges with an extensible range of roles, including global operators, site-specific operators, service operators, service developers, and service tenants (other services) and service subscribers (end-users).

[2] This note describes the "virtualized compute instance" as a Virtual Machine (VM) and assumes OpenStack as the underlying mechanism for creating and managing VMs, but CORD is not limited to OpenStack or VMs. For example, the reference implementation also supports Docker containers, running both inside VMs and on bare metal. To simplify the exposition, we sometimes say "VM" in lieu of technology-agnostic alternatives like "compute instance."

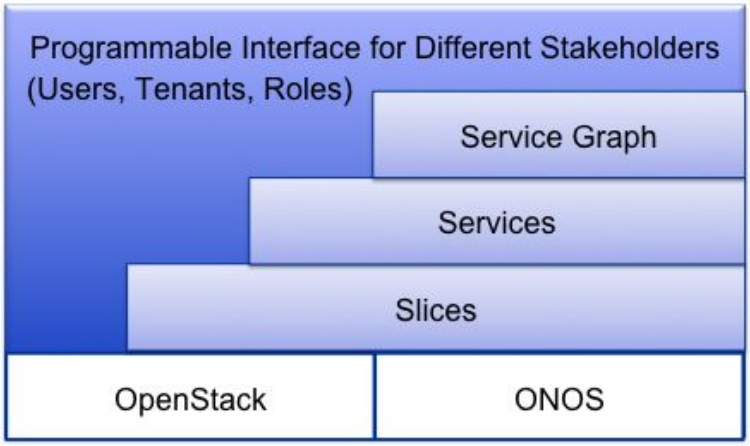

Figure 1. Abstractions layered on top of OpenStack and ONOS.

Sometimes it is helpful to visualize these abstraction as compound (nested) objects built out of primitive objects. As shown in Figure 2, for example, a Slice is an aggregate object created from a set of VM instances, a Service is a combination of a Slice and a Controller that manages the instances in the Slice, and a Service Graph is defined by a dependency relationship among a set of Services, rooted by some class of principal. (Figure 2 uses CORD services, in particular vSG, as an illustrative example.) Although not shown in this particular depiction, each Slice also contains a set of VNs and the dependency between services is parameterized by the VN that interconnects them. In other words, if we view XOS as providing mechanisms that can be used to assemble a CORD service, then the sequence shown in Figure 2 trace that assembly process.

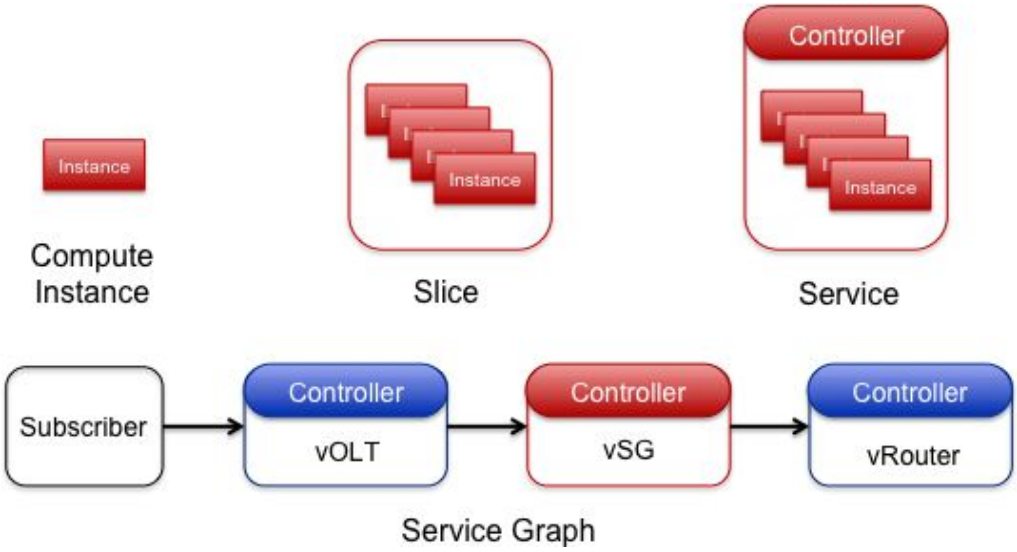

Figure 2. Schematic depiction of a Slice, Service, and Service Graph. The containing box denoting a Slice implies a VN used by tenants to access the slice's VMs, and the box labelled "Subscriber" is an example of a Tenant Principal.

Multi-Tenancy

Multi-tenancy is a central aspect of CORD Service, by which we mean authenticated principals can acquire tenancy in a service, much like they do with commercial cloud services. But the CORD data model goes beyond commodity clouds to also provide support for services being tenants of other services, which is the cornerstone of service composition.

Tenant Abstraction

The overarching design of CORD assumes each service defines some notion of a *service instance*—a virtualized "copy" of itself returned to an individual tenant (e.g., S3's service instance is called a Bucket)—where the service is responsible for isolating tenants (service instances) from each other. Minimally, this implies namespace isolation (one tenant cannot name/access another tenant's resources) and failure isolation (a failure by one tenant does not impact another tenant). It also implies some level of performance isolation, although specifications vary from best-effort to guaranteed.

In defining a service, the developer naturally faces an issue of service granularity—how narrow or broad to make its service instances. A "volume" or "bucket" is an obvious abstraction for a storage service (as opposed, to say, a single file or directory), but the same is not necessarily true for traditional Telco services. CORD permits a wide range of service designs, but there are two specific cases worth mentioning because they correspond to different interpretations of "service" and "tenant" in the cloud provider and service provider domains (and so are a common source of confusion).

In the first case, the vSG service in CORD defines a "subscriber bundle" as its service instance, where each bundle consists of one or more "features" (e.g., basic connectivity, parental control, firewall, and so on). Telcos commonly refer to these customer-facing features as "consumer services." While it is certainly possible to package each such feature as an independently scalable multi-tenant service—with the bundle as a whole implemented as the composition of these narrow services—we decided to package vSG as a coarse-grain bundle with each feature (customer-facing service) implemented through the proper configuration of subsystems within the Linux container bound to each subscriber. For example, firewall rules are implemented by configuring `iptables` and parental controls are implemented by configuring `dnsmasq`. But this is just one possible implementation strategy for vSG. An alternative implementation that still preserves the subscriber bundle abstraction would have been to replace containers with a "plugin" framework like UC Berkeley's E2.

In the second case, each CORD service is multi-tenant, where tenants can be as fine-grain as individual subscribers. In contrast, Telcos often use the term “tenant” in a coarse-grain way, corresponding to major business units—e.g., mobile, residential, enterprise—that share a common underlying physical infrastructure. CORD also applies tenancy to infrastructure-based services (i.e., all services are tenants of OpenStack’s IaaS), but adopts a more general interpretation of tenancy that provides a means to acquire both physical and logical resources. This means fine-grain service instances at one layer might be nested in coarse-grain service instances at another layer.

Note that business rules can enforce resource allocation policy among business units, but such partitioning is not explicit in the Slice and Service abstractions. This makes it is possible to share services across business units rather than isolate units all the way down to the hardware.

Tenants and Instances

It is important to decouple how *service instances* (which are one-to-one with some tenant) and *compute instances* (which are instantiated as a VM or container) support isolation: each service instance isolates tenant state, and each compute instance isolates execution state, but these are not necessarily the same thing. There is an implementation choice as to how service instances map onto compute instances, with two common approaches illustrated in Figure 3.

In the first implementation—shown in on the left in Figure 3, which we sometimes call *explicit isolation*—the service implements each service instance in its own compute instance. (From an architectural perspective, container-based instances and VM-based instances are equivalent.) The vSG service in CORD is an example of this approach, where each subscriber bundle (vSG’s service instance) executes in its own Docker container. In this approach, scaling the number of tenants and scaling the number of compute instances happen in lock-step.

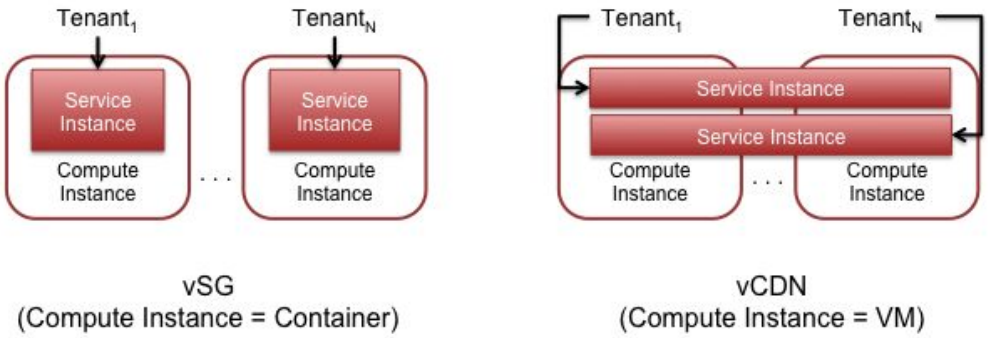

Figure 3. Two different Tenant-to-Instance Mappings.

In the second implementation—shown in on the right in Figure 3, which we sometimes call *implicit isolation*—the service multiplexes multiple tenants (and their corresponding service instances) across one or more compute instances. The CDN service in CORD is an example of this approach, where the service uses a logical partitioning of the URL namespace (called a CDN-Domain) to isolate one tenant's content from another's, but the number of compute instances that the CDN scales across is determined entirely by the workload; it is independent of the number of tenants (and in all likelihood, a given tenant's content is spread across multiple compute instances to avoid hot-spots).

Note that these two strategies strongly influence how packets flow through a sequence of services on behalf of a given user. On the one hand, explicit isolation means that having a handle (reference) for a service instance can be equivalent to a handle for the corresponding compute instance, such that the chain of VMs/containers is known statically for a given tenant. On the other hand, implicit isolation means the a handle for a service instance tells you nothing about the corresponding compute instance(s), such that the chain of VMs cannot be determined until runtime (e.g., a RequestRouter or load balancer selects the next compute instance at each step).

Note that both of these examples assume tenants are implemented by a scalable number of compute instances, but another possibility is that a given service implements service instances "in the data plane," as a collection of flow rules installed in the underlying white-box switches. In this case, there are still compute instances involved, but they control the network rather than use the network to communicate. We discuss this possibility in more detail in the Virtual Network section.

Tenancy and Composition

How services implement tenancy also plays a role in service composition, which again varies from service to service. Two common patterns are depicted in Figure 4. In the first case, tenancy is one-to-one between the consumer and provider services, such that each tenant (service instance) in the former corresponds to exactly one service instance in the latter. This happens, with the vOLT, vSG, and vRouter services in CORD, for example (topmost chain in Figure 4): A Subscriber holds a reference to a service instance in vOLT, which holds a reference to a service instance in vSG, which holds a reference to a service instance in vRouter.

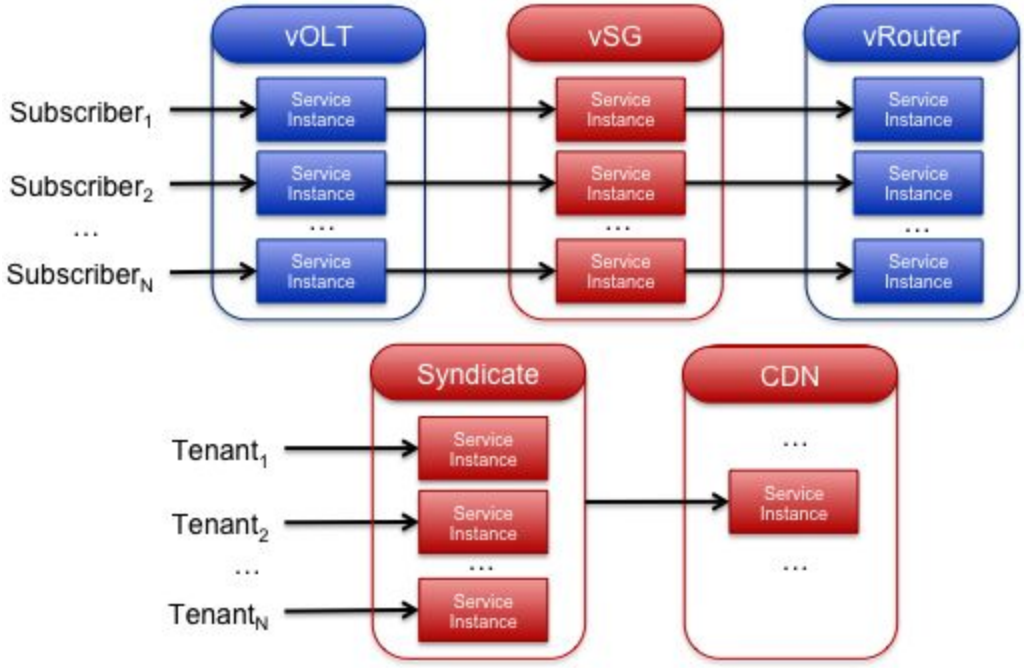

Figure 4. Two common inter-service dependency mappings.

In the second case, tenancy is many-to-one between the consumer and provider services, such that all the tenants (service instances) in the former correspond to just one service instance in the latter. This happens with the Syndicate Storage Service and CDN services, for example (bottom-most in Figure 4): Each tenant holds a reference to its corresponding service instance in Syndicate (a Volume), and Syndicate multiplexes all of its Volumes onto a single service instance in the CDN (a CDN-Domain) [4].

Note that Figure 4 shows references to service instances, not compute instances (VMs or containers). The exact form of each of these references depends on how service instances are mapped onto compute instances, as outlined in the previous subsection (and illustrated in Figure 3). If the service supports explicit isolation, then the reference is direct (e.g., it corresponds to the address of a container or VM). If the service supports implicit isolation, then the reference is indirect (i.e., additional mechanisms are needed to map the reference into the address of an appropriate container or VM). How such references are interpreted is a function of the Virtual Networks that interconnect services, as discussed in the next section.

Figure 4 can also be viewed through the lens of Network Functions Virtualization (NFV). Roughly speaking, each service instance in CORD corresponds to a Virtualized Network Function (VNF) in the NFV architecture. How a sequence of such VNFs—sometimes called a *service chain*—map onto a sequence of VMs depends on four things: (1) whether the service instance is on the network control plane or data

plane, (2) how each service instance maps onto one or more compute instances, (3) the relationship between services instances in composed services, and (4) how virtual networks interconnect compute instances.

Said another way, a linear chain of compute instances corresponding to single subscriber—an implement of service chaining assumed by many service orchestrators—is just one of many possible outcomes of service composition in CORD. Our experience with a wide collection of services that span the full NFV, SDN, and Cloud space is that a more general model of service composition is required, and this experience informs CORD's design.

Virtual Networks

CORD supports two different VM/VN relationships. In the first, a VN interconnects a set of VMs; OpenStack *ports* define the VM/VN interface (i.e., the interface through which the VMs send and receive packets over the VN). In the second, a set of VMs control a VN; the ONOS API defines the VM/VN interface (i.e., the interface through with the VMs apply flow rules that manage the VN).

In both cases, ONOS—a multi-tenant CORD service—implements VNs as its service instance, returns a VN in response to a request to create one, and applies subsequent control directives to a specific VN. The difference is that in the first case, Neutron requests the VN on behalf of a service that wants to use the it to communicate (i.e., a Neutron plugin effectively implements the control program that defines the VN's semantics), and in the second case, the service directly controls the VN by running some control program (hosted in one or more VMs) and invoking ONOS directives on that VN.

Standard Virtual Networks

Focusing on the VNs created on behalf of services that want to use them to communicate (the first case above), CORD currently supports the following three network types:

- **Management:** All Slice instances are connected to a management network, which is used to create, delete, and control instances. The topology of this management network is deployment specific, as discussed briefly below.
- **Private:** Supports communication among the set of instances within a Slice. This VN also plays a role in service composition, where each service specifies how the VN delivers requests to the service. There are two options, both implemented by the VTN control application running on ONOS [5]:

    - **Direct:** Local (CORD-resident) clients request service by directly addressing each instance in the Slice. Typically, we would expect the service to use some external load balancing mechanism (e.g., Request Router) to evenly distribute requests across its instances.
    - **Indirect:** Local (CORD-resident) clients request service by addressing the service as a whole. In this case, the VN implements load balancing among instances.
- **Public:** Behaves the same as a Private VN, but does so in a way that allows Internet clients to request service from (address packets to) the service using a publically routable IP address.

The structure of the management network is deployment specific. The CORD POD, for example, includes a global management network that connects all the servers and fabric switches to the control processes running on the CORD head nodes (it is implemented by a physical switch that is distinct from the white-box switches that implement CORD's data plane), and a local (per-server) management network that interconnects the instances running on that server. But these management networks are not interconnected. Instead, certain management functions are implemented by proxies or application bridges that are connected to the global management network to the set of local (per-server) management networks. For example, an ssh proxy is used to log into an instance for management purposes.

Shared Responsibility

An explicit goal of CORD is to define abstractions and interfaces that separate concerns, with different actors responsible for different aspects of the system. There are two actors that are particularly important to call out: Cloud Operators and Service Developers.

In addition to defining the semantics of their service—including what constitutes a meaningful abstraction for a service instance—service developers are responsible for all aspects of how their service achieves scalable performance and high availability. They determine how to best balance load among the compute instances that implement the service (e.g., DHT, layer-N load balancer, Request Router); how to best distribute state over the compute instances (e.g., sharding, replication); how to best embed the service in the underlying infrastructure (e.g., VM placement and affinity, VN topology and control); and how to isolate tenants and map service instances onto compute instances.

On the other hand, the cloud operator is responsible for determining what services are included in the service portfolio and the dependency among those services, as well as

for setting all policies that constrain service operation. This includes defining the service-level dependency graph, setting limits on how many resources each service may consume, arbitrating resource allocation among services, and telling each service when it is appropriate to scale up-or-down based on observed or anticipated workloads. To the extent the global cloud spans multiple deployments (clusters), the operator may also be responsible for constructing end-to-end-solutions from localized components.

With respect to the CORD data model and programmatic interface, this means services export operations that allow operators to (1) set resource quotas and (2) scale the service up/down, while the service is responsible for (1) responding to the scale-up/down request and (2) notifying the operator when there are insufficient resources to scale up.

Service Assembly

CORD is built from a set of services. XOS provides a collection of mechanisms to help service developers assemble services and incorporate them into CORD. Broadly, these mechanisms include Slices (resource containers) that host service instances, and building block services (e.g., RequestRouter-as-a-Service, Instrumentation-as-a-Service) that perform common tasks.

But more importantly, the collection of CORD abstractions provides the means to construct a Service Controller. This includes support for a tenant-facing service interface, a data model that represents the state of the service, and mechanisms by which the controller configures and manages the individual compute instances. Collectively, these mechanisms make it possible to express and enforce policies on a collection of services: It provides a programmatic interface for expressing policy, and state management mechanisms for ensuring that the underlying operational system adhere to those policies.

Service Control Plane

XOS represents each service (and collectively, a graph of services) with a two-part implementation. The first part is a declarative *data model* that defines the *authoritative state* associated with the service, along with an interface for creating and operating on the service's tenant abstraction. The second part is a *synchronization framework* that directs how this declarative state is translated into primitive operations on the underlying OpenStack (and other) resources.

As illustrated in Figure 5, the implementation leverages a two existing open source components (Django and Ansible), but also includes a new mechanism built specifically for XOS (Synchronizer). Taken together, the framework works as follows.

The data model is implemented in Django, which includes mechanisms to define the state associated with each object and the relationships among objects, along with tools for generating both programmatic and graphical interfaces. Django is a powerful modeling platform; XOS defines a more restrictive set of conventions that guide how it is used to model its core abstractions. The value of Django is that it provides a declarative means to define the authoritative state of the system.

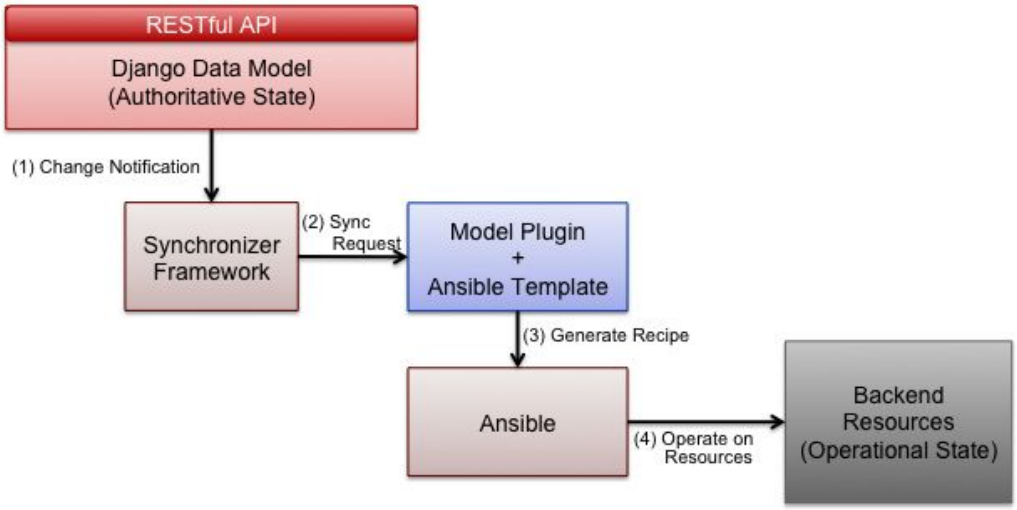

Figure 5. Component subsystems of the XOS Service Control Plane.

The imperative half of the framework is a three-part mechanism for keeping the operational state of the underlying cloud resources in sync with the authoritative state: (1) an XOS-provided Synchronizer, (2) a set of developer-provided model policy plugins, and (3) Ansible. Briefly, Ansible is the target language for expressing the actions that need to be taken on the underlying resources to bring their operational state in line with the corresponding authoritative state. It is by executing these Ansible playbooks that XOS directly operates on the underlying resources.

The Synchronizer is responsible for generating these recipes. It does this by first analyzing the data model to determine what objects require attention and to compute the dependencies among those objects. It then executes the appropriate model policy plugins to generate the required Ansible playbooks. The plugins are written according to a stylized Synchronizer template that requires the model developer to write four methods: *sync_record*, *delete_record*, *check_sync* and *check_delete*. The first two methods synchronize records and delete objects respectively, and the last two check

the results of the synchronization and deletion, signaling the Synchronizer to mark the objects as having been processed, or to invoke error handling mechanisms.

The value of this particular state management framework is that it provides a pragmatic solution to the challenge of keeping the system's operational state in sync with its authoritative state, but it does so in a way that frees from service developer from having to address several cross-cutting issues: dependency management, error handling and propagation, tolerating partial failures, and authentication and access control.

It is also the case that by constructing data models and policy plugins that reflect the control aspects of the individual services it is possible to reason about both the individual services and the composition of services defined by a service graph. This plays a particularly important role in defining and enforcing a security policy for the system as a whole, as discussed in more detail in a companion paper [3].

Examples

XOS provides a toolkit that can be used to assemble a service, but how much or how little of this toolkit a given developer exploits depends on how complete the existing service implementation is. Figure 6 shows a range of examples, where blue denotes elements defined or managed by XOS and red denotes elements external to CORD.

In the left-most example, the developer starts with a VM image that implements a single instance of the service (perhaps extracted from legacy hardware appliance), and the XOS mechanisms are used to construct a service—the controller is implemented from scratch and the instances run in XOS-managed slices.

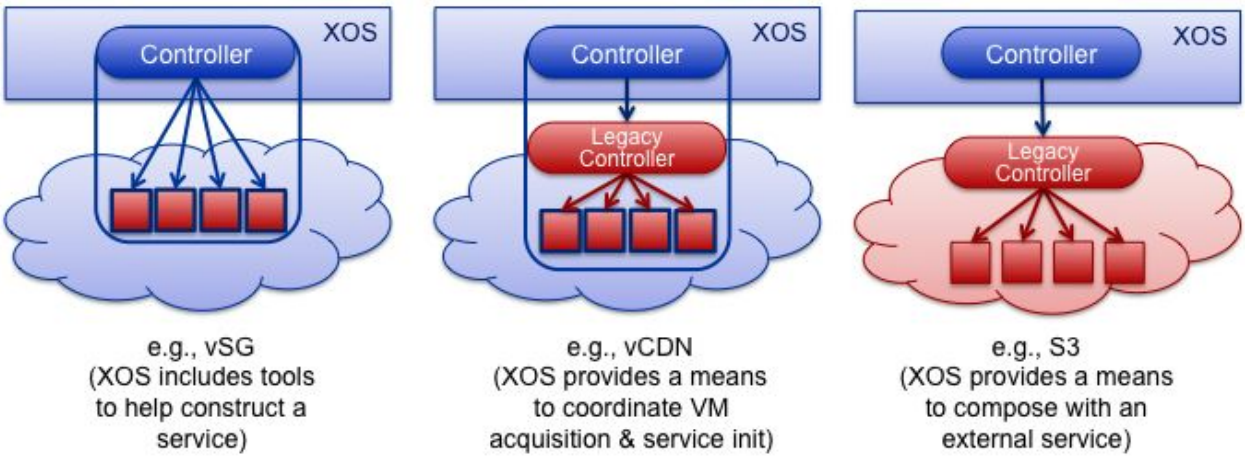

Figure 6. Examples of how to use XOS to import a service into CORD.

In the middle example, the developer starts with a scalable service—including a legacy controller—and XOS provides a slice to host a scalable number of instances. (The

legacy controller may be co-located with XOS, run in an XOS-provided VM, or be external to the deployment.) There is also a Service data model that represents (is a proxy for) the legacy controller, which provides a means to import the service interface into the XOS service portfolio, which in turn facilitates securely composing this service with other services.

In the right-most example, the full service already runs in a cloud external to XOS, in which case the XOS data model provides a means to represent the service in the CORD service portfolio (i.e., there is a service object in the XOS data model that represents the service). Note that in this and the previous (middle) cases, the XOS state management framework still plays a role in keeping the XOS-managed state synchronized with the state managed by the legacy controller.

End-to-End Example

CORD manages complexity through abstraction, but all the actors still have work to do. The service developer must program the service abstraction (including the model plugin shown in Figure 5), and define how the service is to scale. The operator must specify the desired service dependencies, define quotas on the number of resources each service is allowed to consume, and instruct services to scale up or down. Having done that, however, CORD provisions, configures, and interconnects the underlying OpenStack and ONOS resources according to a declarative specification.

CORD supports a RESTful API that is auto-generated from the data model defined in Django, but CORD also accepts operator-defined specifications in TOSCA, an industry standard for cloud lifecycle management. When viewed from an end-to-end perspective (see Figure 7), XOS effectively defines a directly executable representation of the operator's policy comprised of service-provided elements organized according to the Slice, Service, and Composition abstractions.

The figure highlights the importance of Virtual Networks (VNs) in CORD. The configuration includes two VNs: one named *LAN_side* connects the vSG service to subscribers (it is controlled by the vOLT service) and one named *WAN_side* connects vSG to the Internet (it is controlled by vRouter). The vCDN slice is also connected to WAN_side so the caches can access the Internet to fetch content when there is a cache miss. These VNs are specified as part of Slice definition, and their role interconnecting slices is specified as part of the Service dependency definition.

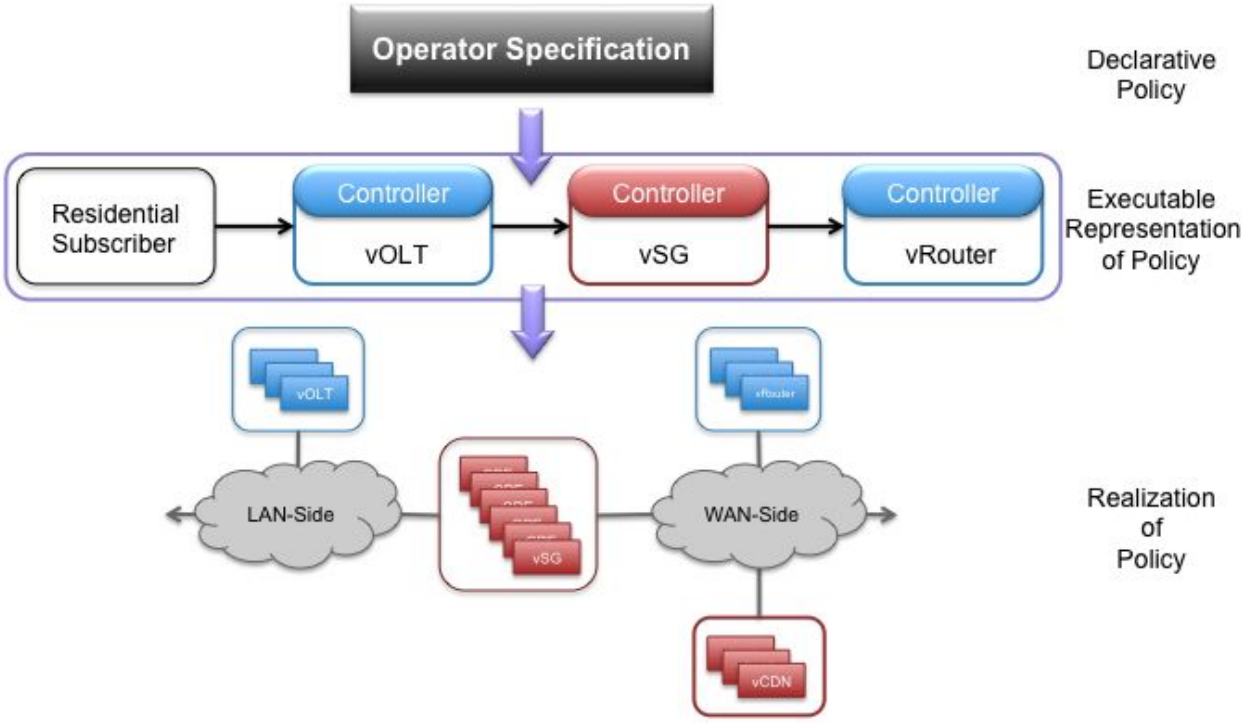

Figure 7. Configuration of the underlying cloud resources (VMs and VNs), as instantiated by CORD under the control of operator-specified policy.

Also, although not explicitly shown in Figure 7, the TOSCA specification is written to call out narrow “subscriber-visible” services—corresponding to vSG features—rather than limit itself to only those broader services instantiated in CORD. Such user-level service specifications are translated into the corresponding operations on the vSG service to enable and configure the features it supports.

Summary

Translating a high-level specification of a set of services into an operational deployment of virtual machines and switch flow rules is a challenging task. CORD addresses the challenge by breaking it down into a collection of interrelated interfaces and mechanisms:

- Operators define inter-service dependencies, and the virtual networks that will allow tenants to request service from providers.
- Operators define limits on how many resources each service is allowed to consume and specifies the workload each service needs to support.
- Services define abstract service instances and how those abstractions map onto a set of compute instances.

- Services define how the service scales across a set of compute instances to meet a particular workload, including instance placement and virtual network behavior.
- Virtual networks, and the SDN applications that control them, define how packets are forwarded to a particular compute instances.

In short, CORD provides a unifying set of abstractions and associated development tools that allow these independent elements to work in concert to meet the target specification.

References

[1] Central Office Re-architected as a Datacenter (CORD). *CORD Design Notes* (March 2016).

[2] CORD Reference Implementation. *CORD Design Notes* (March, 2016).

[3] Security in CORD. *CORD Design Notes* (March 2016).

[4] J. Nelson and L. Peterson. Syndicate: Virtual Cloud Storage Through Provider Composition. *ACM BigSystems 2014* (June 2014).

[5] CORD Fabric, Overlay Virtualization, and Service Composition. *CORD Design Notes* (March 2016).