Breaking the Disruptor's Dilemma

How to Operationalize a Disaggregated System

Introduction

For the last several years, ONF has been working with network operators to transform their access networks from using closed and proprietary hardware appliances to being able to take advantage of software running on white-box servers, switches, and access devices.

This transformation is partly motivated by the CAPEX savings that come from replacing purpose-built appliances with commodity hardware, but it is mostly driven by the need to increase feature velocity through the softwartization of the access network. The goal is to enable new classes of edge services—e.g., Public Safety, Internet-of-Things (IoT), Immersive User Interfaces, Automated Vehicles—that benefit from low latency connectivity to end users (and their devices). This is all part of the growing trend to move functionality to the network edge.

The first step in this transformation has been to disaggregate and virtualize the legacy hardware appliances that have historically been used to build access networks. This is done by applying SDN principles (decoupling the network control and data planes) and by breaking monolithic applications into a set of microservices. This disaggregation (and associated refactoring) is by its nature an ongoing process, but there has been enough progress to warrant early-stage field trials in major operator networks.

But it is the second step in this transformation—integrating the resulting disaggregated components back into a functioning system ready to be deployed in a production environment—that is proving to be the biggest obstacle to fully realizing the full potential of softwarization. In short, network operators face a *disruptor's dilemma*:

- Disaggregation catalyzes innovation. This is the value proposition of open networking.
- Integration facilitates adoption. This is a key requirement for any operational deployment.

Network operators' instincts in addressing this problem are to turn to large integration teams that are tasked to engineer a solution that can be retrofitted in their operating environment. The problem is the widespread perception that each environment is so unique that any solution deployed in it must be equally unique, which unfortunately (and ironically) results in a point-solution that is difficult to evolve and not particularly open to ongoing innovation. Ideally, you want to keep components free of local modifications so you can consume future releases from upstream sources without having to repeat the integration effort. Augmenting an open source solution with customizations that generate value is one thing, but the brittle nature of

retrofitting puts network operators risk of trading away the value of disaggregation (which enables feature velocity) to regain the operational familiarity and stability of fixed/point solutions.

But is such an outcome preordained? Is there a technical approach to *operationalizing* a collection of disaggregated components in a way that sustains the innovate/deploy/operate cycle? We believe there is a technical solution to this problem, one that automatically assembles a system from a set of disaggregated components and integrates that solution into different deployment environments.

Integration Toolkit

The starting point for operationalizing a collection of disaggregated components is the plethora of CI/CD tools that have proven useful in building cloud services—everything from Kubernetes to Helm to Jenkins—but they are not sufficient because they do not directly address the variability of deployment environments or operator business logic. The goal is an operational system. The CORD project demonstrates how to factor the operator's operational and business requirements into an integrated solution in an automated and repeatable way. The key is the *Network Edge Mediator* (NEM), which plays two important roles.

First, NEM assembles a unified solution. Each operator wants a different subset of the available disaggregated components, which NEM allows them to specify as a configuration-time *Profile*. In the case of SDN-Enabled Broadband Access (SEBA), for example, some operators want BNG to be internal to the solution and some want it to be external, and when internal to the solution, some want the BNG implemented in containers and others want it implemented in the switching fabric.

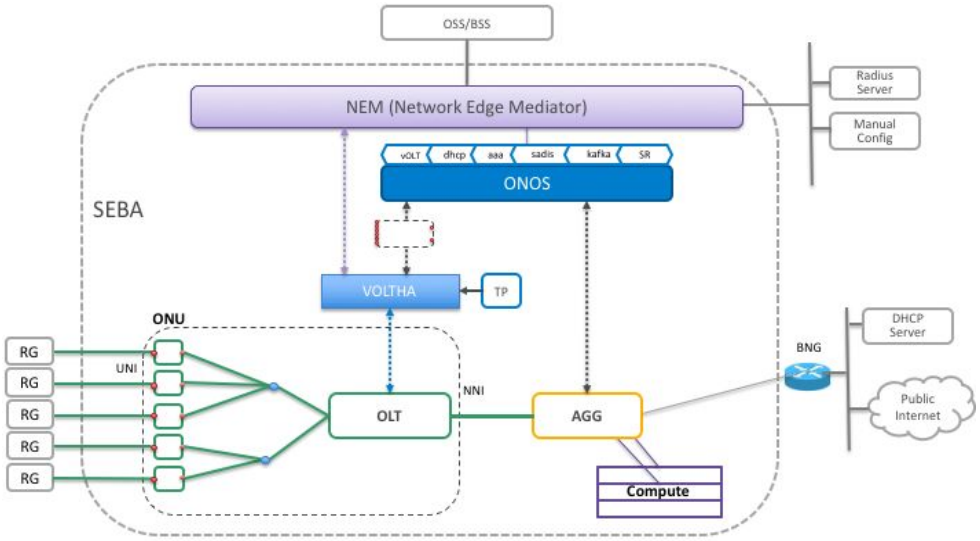

Figure 1. One possible configuration of SEBA (Profile), with NEM mediating access from the containing deployment environment.

NEM assembles the collection of components specified in a declarative profile into a self-contained whole by generating the necessary “glue” code. It presents a coherent Northbound Interface to the operator’s OSS/BSS, and it manages any state that needs to be shared among the components. This avoids hardwiring dependencies into each component. Figure 1 depicts one possible configuration of SEBA, with NEM defining the interface between SEBA and the deployment environment.

Second, NEM explicitly manages environment dependencies by allowing operators to define a runtime-time *Workflow*. Returning to the SEBA example, the workflow specifies how the integrated solution interacts with the surrounding operational environment, as individual subscribers are managed throughout their lifecycle (e.g., coming online/offline). The workflow specifies all aspects of what happens as ONUs are detected and approved, EAPoL packets arrive and subscribers are authenticated, DHCP requests are received and IP addresses assigned, and finally, packets start flowing. Different workflows can be defined for different operators, yet reuse all the same components without modification.

Nothing comes for free. Component developers need to program for reuse, which means not hardwiring dependencies on other components, but instead getting their environment-specific configuration parameters from NEM. Similarly, network operators need to specify the profiles and workflows that satisfy their particular operating and business requirements; they should not modify the individual components that are being integrated. Stated succinctly:

> *Development concerns should be fully decoupled from operational concerns, with the latter specified once (centrally) and in a declarative way.*

By lifting this operational information out of the service implementation and centralizing it in NEM’s declarative data model, it is possible for operators to break the disruptor’s dilemma, and deploy integrated solutions built from disaggregated components, thereby benefiting from a fully agile innovate/deploy/operate cycle.

NEM Internals

At a high level, NEM consists of three subsystems, as depicted in Figure 2:

- Authoritative State Manager → Manages the state needed to configure and control the collection of backend components. Implemented by XOS.
- Monitored State Manager → Collects and manages the logs, metrics, and events produced by the backend components. Implemented by Prometheus, Grafana, Elk Stack, and Kibana.
- Event Bus → Used to share events among all the backend components and NEM subsystems. Implemented by Kafka.

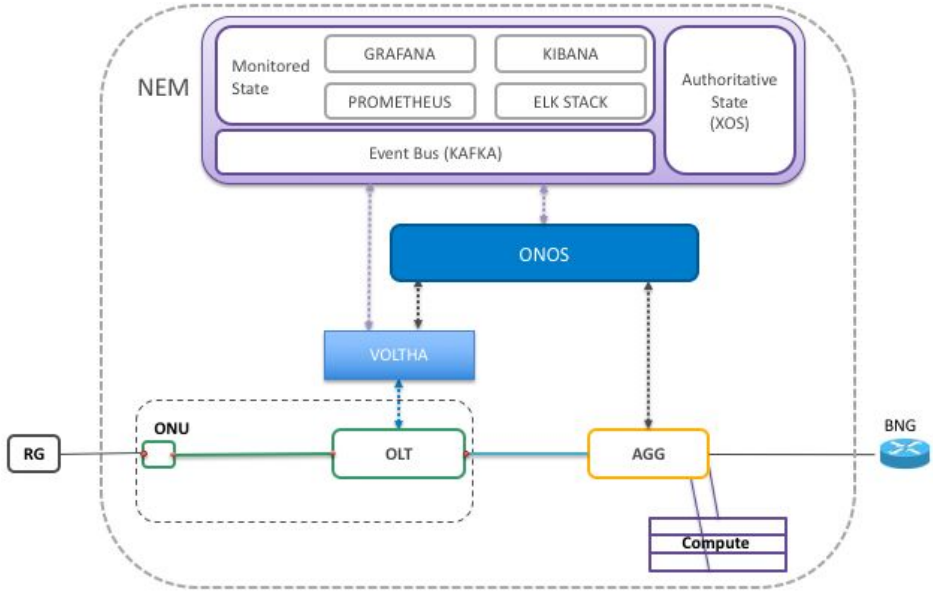

Figure 2. Details of the internal subsystems that comprise NEM, shown in the context of SEBA.

Of these, XOS plays a central role in operationalizing the disaggregated components. It takes a set of model definitions (schema) as input, and auto-generates the glue code needed to integrate those components. The models define both the *Profile* of disaggregated components the operator wishes to deploy, and the *Workflow* required to integrate those components into a surrounding operational environment. The auto-generated code includes the *Northbound Interface* used by the operator to manage the deployment and the *Synchronization Framework* needed to keep the backend components synchronized with the authoritative state (shown in Figure 3), as well as the object-relational mapping needed to make the authoritative state persistent and the security enforcement points used to restrict what principals can read/write which bits of authoritative state (not shown in Figure 3).

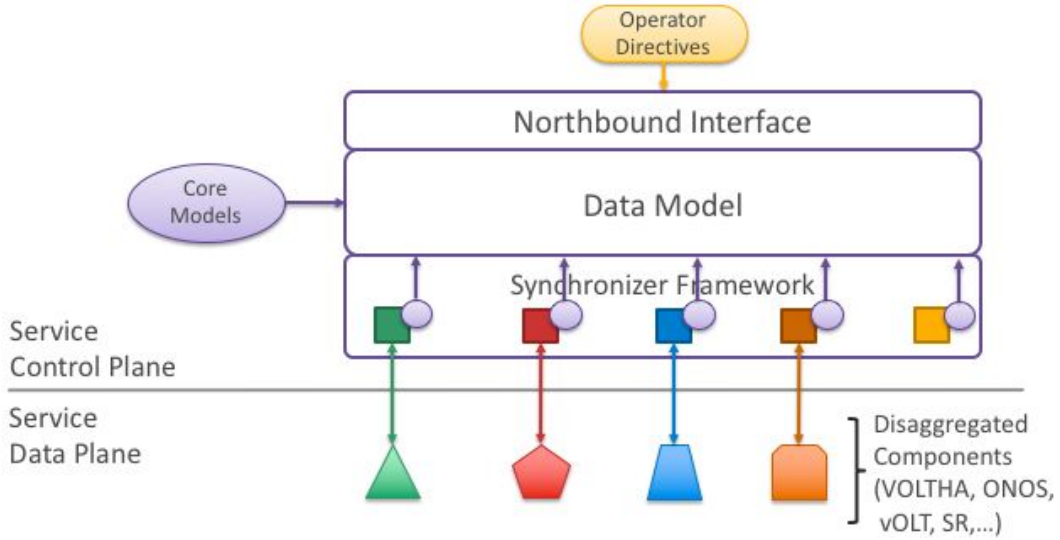

Figure 3. Internal structure of XOS, and its relationship to disaggregated components.

Figure 3 also shows that the XOS Data Model is initialized with a set of *core* models. The core defines common abstractions, such as *Services*, *ServiceDependencies*, *ServiceInstances*, and so on. Each component that is to be integrated into a particular Profile then loads one or more component-specific models into XOS (these typically extend the core models), along with a Synchronizer plugin that keeps the corresponding backend component in sync with the authoritative state. Note that XOS also supports purely logical services that have no corresponding backend component, as exemplified by the rightmost model/plugin in Figure 3. These logical services create new/composite functionality by programming the data model. Deployment-specific workflows are a prime example of such logical services.

Lifecycle Management

Since our goal is continuous integration, it is important to understand the lifecycle management of a NEM-mediated system. Installing and upgrading a deployment is a multi-layer/multi-stage process, as depicted in Figure 4.

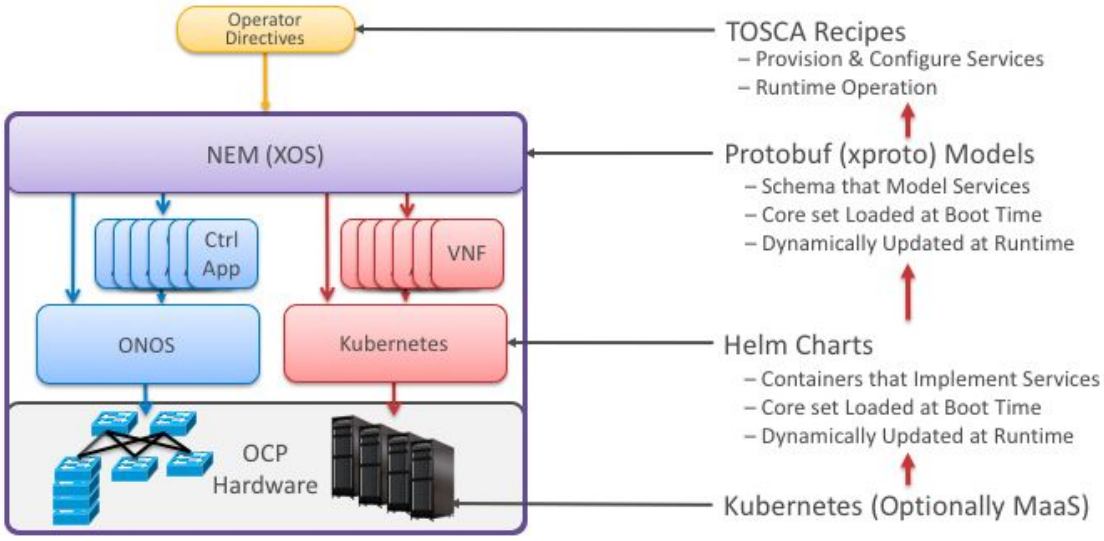

Figure 4. Lifecycle management of a deployed POD.

The figure depicts a generic configuration (called a POD) that includes some combination of SDN Control Apps running on ONOS and VNFs/CNFs running on Kubernetes. Starting from a base platform running Kubernetes, operationalizing these components involves the following three layers:

- Installation (Helm): Installing a POD means installing a collection of Docker containers in a Kubernetes cluster. Helm carries out the installation, with the valid configurations defined by a set of helm-charts. Among the containers deployed by Helm are ones that implement NEM.

- Operations (TOSCA): A running POD supports multiple Northbound Interfaces (e.g., a GUI and REST API), but TOSCA is typically used to specify a recipe for configuring and

provisioning a running system. A freshly installed POD has a set of control plane and platform level containers running (e.g., XOS, ONOS), but until provisioned using TOSCA (or one of the alternative NBIs) there are no active services or service mesh.

- Integration (XOS): XOS implements a middle layer that connects the NBI(s) to the backend components. It takes an *xproto*[1] model definition for all the services that are to be included in the POD as input, and generates the code needed to integrate (on-board) those components. This includes the TOSCA engine used to process the configuration and provisioning workflows.

The resulting system is dynamic. Bringing up new services involves deploying new helm charts and loading new models into XOS, which in turn triggers an upgrade and restart of the TOSCA engine. Upgrading an existing service is similar: Kubernetes incrementally rolls out the containers that implement the service (and rollbacks them back if necessary), and XOS migrates from the old model to the new model (and supports both old and new APIs during the transition period).

For more information about XOS and the role it plays in operationalizing a collection of disaggregated components, see the XOS Guide. For more information about specific configurations like SEBA, see the CORD Guide.

Towards Universality

Integration is difficult, and if not done with an eye towards continuous integration, there is a significant risk of relinquishing the gains of disaggregation by deploying one-off systems. NEM addresses this issue by providing an automated integration toolkit. NEM is a specific implementation, but represents a general architecture that, if widely adopted, has the potential to create a marketplace for a wide range of service vendors and service operators.

Our view is that a schema for services and service-related abstractions is the right place to start. The XOS core models are a candidate set, but they are by no means complete. The definitions are entirely malleable, however, and come with an extensible toolchain that makes it possible to enforce model semantics throughout an operational system.

We recognize that agreeing to an industry-wide definition of these models is a long and arduous process, and that the first step is to establish a common name space for models (and events), with a set of best practices for how the name space is partitioned, and when there is sufficient agreement, reconciled. Our reading of the LeaNFV Initiative is that it recommends a similar starting point and collaborative agenda, although we advocate for a broad scope that encompasses Telco clouds, commodity clouds, edge clouds, and private/enterprise clouds.

[1] xproto is a variant of Protocol Buffers, extended to support inheritance, relations, and predicates. The syntax was selected because of its familiarity, but it is not required by the general architecture.