Kubernetes Operators for Telco Workloads

OpenShift Commons Briefing, September 8 2021

Speaker: Tal Liron
Red Hat, Telco Solutions and Enablement

Part 1
How Did We Get Here?
A History of Decoupling

Prehistory: Network Equipment

("Network" in the telecommunications sense)

Since 19th century telegraph

Signal processing, switching, multiplexing, etc.

Decoupling the network operator from the network equipment (via standards)

("Operator" in the telecommunications sense)

Equipment sold and serviced by Network Equipment Providers (NEPs)

Step 1: Computerization

Some “network equipment” contains or indeed is specialized computers

Evolution from analog to digital

Evolution from telephony to Internet

Step 2: Virtualization

Decoupling network equipment hardware and software

Pushed by operators to reduce costs and avoid lock-in

Three players: operators, hardware vendors, software vendors

Instead of “equipment” we call them “network functions”

Physical Network Functions (PNFs) vs. Virtual Network Functions (VNFs)

Evolution to off-the-shelf hardware

Evolution to off-the-shelf software (more in next slide)

Step 3: Cloudification

Cloud Native Network Functions (CNFs)

Four players: operators, hardware vendors, platform vendors, CNF vendors

Evolution from virtual machines to containers

Evolution from management (e.g. OpenStack) to orchestration (e.g. Kubernetes)

(You heard that right! Kubernetes is an *extensible* orchestrator)

There's even some use of public clouds

Part 2
Functions and Operators

The Operator Pattern

Functions and operators as computational building blocks

It's not arithmetic! Our functions and operators do **continuous work**

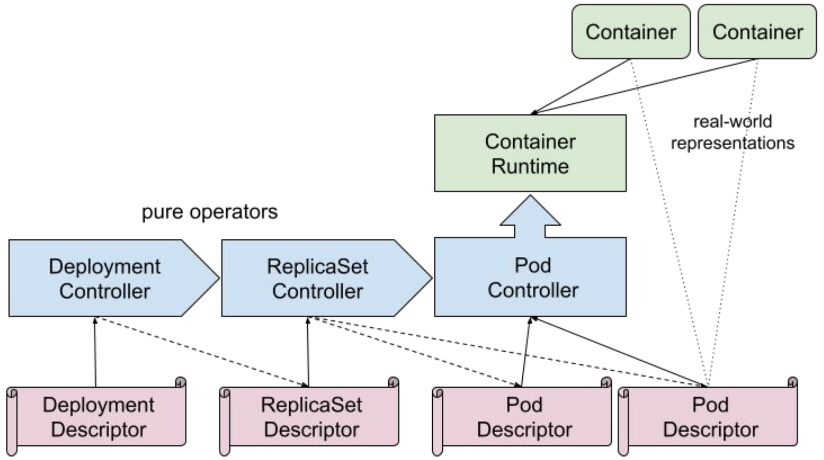
When applied to a declarative intent-oriented orchestration context (like K8s):

- the operator consumes intent as its input
- and emits intent as its output
- continuously monitors input and manages output

A "pure operator" has no side effects

An "impure operator" can do orchestration work as well as emitting intent

Pure Operators in Kubernetes



Unfortunately… *[insert sad trombone]*

Kubernetes official terminology is confusing and misleading:

All built-in operators are called “controllers”, but *not* operators, even when they implement the operator pattern

And *any* custom controller is called an “operator”, even if it does not implement the operator pattern

No elegant way out of this mess, we’re stuck with phrases like this:

- “a Kubernetes controller that implements the operator pattern”
- “a Kubernetes operator that does not implement the operator pattern”

“Controller” vs. “Operator”

Kubernetes operators are useful even without the operator pattern

Ask yourself:

Should you emit intent or terminate intent and handle all the work yourself?

Consider your overall architecture:

- Is your solution is meant to work with other solutions?
- Is your solution specific to one project or is it meant to be reused?
- “Unix philosophy”: a focused tool that does just one thing and does it well

Try Not to Reinvent Wheels

A growing ecosystem of off-the-shelf generic operators:

- cert-manager
- KubeVirt
- Multus CNI
- Istio
- ...

And of course there's the Operator Hub

Part 3
Telco Use Cases

Use #1: Stateful Components

Statelessness provides trivial redundancy, scalability, and idempotency

Non-trivial to achieve these with stateful components (e.g. database clusters)

Impure operators can handle the specialized lifecycle management

State in network functions:

- Connections and links must be maintained
- Persistent sessions
- Cross-component transactions
- Live scalability and healing with no packet loss

Use #2: Configuration Management

The meatiest aspect of network function orchestration

Whole suites of software and standards (NETCONF, YANG, gNMI)

Involves configuration workflows

An operator as a local “configurator” (instead of a far away orchestrator)

Check out Candice, which enables configuration “micro-workflows” in Python

Use #3: Disaggregation and Modularity

The “big orchestrator” can delegate work to operators

Doing the work locally means:

- faster and more autonomous decisions
- specialization: the operator does one thing well (“Unix philosophy”)

Such operators are then “orchestration modules” within our larger orchestration narrative

(which has always been modular: a long history of disaggregation)

Use #4: Integration and Control

Operators as “installers” (including Day 2 changes)

Historically:

- “S-VNFM”: special operator for one CNF
- “G-VNFM”: generic operator that can be configured to handle various CNFs

Turandot is an example of a generic operator based on TOSCA

When to modularize and when to integrate?

Keep the code close to the data!

Use #4: Management of External Resources

Why not do all our orchestration work in Kubernetes?

CNFs and PNFs!

A lot of the work is the same, especially for configuration (e.g. NETCONF)

Example: F5 BIG-IP controller

Part 4
A Few Practical Considerations

Limitations of Custom Resource Definitions

Unfortunately *[sad trombone]* though CRs are namespaced, CRDs are cluster-wide

Meaning you need cluster privileges to install them (Kubernetes should fix this!)

Another problem: etcd limits CR size to ~1MB

Alternatives:

- Use ConfigMaps instead of CRs
- Use annotations (“fine-grained operators”)
- Use your CR (or ConfigMap or annotation) to point to somewhere else
- Store your intent elsewhere (e.g. GitOps)

Namespaced or Cluster-Wide?

Should your operator be installed in its own namespace?

Or in all the namespaces in which it is used?

Why not support both options?

(Many operators do)

Ownership and Garbage Collection

Kubernetes can automatically delete any resource when its owning resource is deleted

Perfect for the operator pattern!

But not good for stateful components with specialized lifecycle management

Developing Operators

Complex and daunting: Kubernetes is a highly asynchronous environment, and then there's Go

But there are many options:

- Kopf is a comprehensive operator SDK in Python
- Ansible Operators (Red Hat) allow you to use Ansible playbooks as event handlers, which of course can be extended via Python with custom modules, e.g. the Kubernetes collection and the Galaxy ecosystem
- Charmed Operators (Canonical) is written in Go but allows you to write Python code within the Charm ecosystem
- KUDO aims to do all the heavy lifting for you by implementing best practices and exposing a declarative interface
- CRAFT (SalesForce) enables straightforward CRUD operations using any language.
- Shell-operator allows you to use shell scripts

Thank You!

Link to full document

Find me: tliron@redhat.com