CNFs and Operators

By Tal Liron, June-August 2021

Acknowledgements: The writing of this document has benefited from generous feedback from engineers at Red Hat as well as upstream communities. Nevertheless, mistakes and opinions are ultimately the author’s.

Introduction

Kubernetes is neither an NFVI (NFV Infrastructure) nor a VIM (Virtualized Infrastructure Manager). It is an orchestrator. Moreover, it’s an extensible orchestrator. It can be extended via plugins, such as CNI, or via custom controllers, which in Kubernetes are called (confusingly, as we’ll see) “operators”.

When developing CNFs (Cloud Native Network Functions) we can deliberately choose *not* to extend Kubernetes and instead use only its built-in controllers and their associated resources: Deployments, DaemonSets, Services, PersistentVolumeClaims, ConfigMaps, Secrets, etc. It is entirely possible to follow cloud native principles without deep integration into the platform itself. However, doing so leaves many of the advantages of Kubernetes on the table and can increase complexity by forcing us to add non-native layers of orchestration.

In this guide we will focus on custom controllers (“operators”) generally as well as how they can and why they would implement the operator design pattern. The operator pattern predates and is not specific to Kubernetes, but when applied it can help us make our Kubernetes extensions more composable and more reusable.

Importantly, too, the operator pattern can be applied in Kubernetes without custom resource definitions (CRDs). We will discuss use cases and alternatives.

The Operator Pattern

An operator is a special kind of controller, so let’s start with what controllers are. And let’s also clarify the context of our definition: we’re in an environment of declarative, intent-oriented, policy-controlled automation. In this context a controller is a service that continuously attempts to realize intent as guided and confined by policy. It must be continuous because the reality might change even if our intent does not, which is a given in clouds. For example, hardware might fail, in which case the controller will attempt to re-realize our intent on different hardware. This process is sometimes called “reconciliation” in reference to the gap between the desired state and the current state.

Controllers in effect encode knowledge and skills. They can replace menial work that would otherwise have to be done by a human, who would need to know how to control the systems and be constantly monitoring them. Another way to think of it is that the system's operations manual here becomes runnable computer code.

An operator is a special kind of controller that, as part of its reconciliation process, declares and manages new intent based on the original intent. It thus has an input and an output, both of which are declarations of intent. We can refer to some operators as "pure", meaning that they do not have side effects other than the output of new intent.

The term "operator" is used here, as it has been used for more than 20 years, in the computational sense. The goal is to decompose complex computational work into smaller, simpler, and possibly reusable components. In this sense an "operator" is somewhat analogous to a "function", just like the arithmetic "a + b" operator can be understood as a "plus(a, b)" function.

Like functions, operators can be chained together into complex graphs in which the output of one operator becomes the input of another. Of course we don't want to generate intent *ad infinitum* and thus the graph must be terminated by controllers that are *not* operators, which manage real-world effects.

This implies a sidedness to the operational graph. One end can be understood as more "high-level" or abstract, with operators along the way generating intent that is increasingly "low-level". However, not all graphs are hierarchical. Indeed the work of the operator can also be understood as a *translation* or a *derivation* of aspects of the intent between domains or systems, rather than a move towards concreteness. This subtle insight allows for additional uses for operators, which we will discuss.

Meanwhile, In Kubernetes

Unfortunately, the above terminology is used in Kubernetes's documentation in confusing and contradictory ways. In Kubernetes the term "controller" refers to built-in controllers while the term "operator" refers to custom controllers.

Built-in controllers are associated with built-in resource types, such as Pod, Deployment, Service, etc. (Note that vanilla Kubernetes does not come with a controller for Ingress resources. Those require an "operator".) The built-in controllers normally run on the host itself. On the other hand, custom controllers are often associated with custom resource definitions (CRDs) and typically run in the cluster as containerized workloads (Pods). Technically they can really run anywhere, such as on the host or even outside the cluster. All they need is access to the custom resource descriptors via the Kubernetes API server.

It is unfortunate that custom controllers are called "operators" because nothing in the definition requires that they implement the operator pattern. Confusingly, most of the built-in controllers *do* implement the operator pattern—but they are not called "operators" in Kubernetes.

There's an inconsistency in the documentation that may be the cause of this misapplication of terminology: it explains the term "operator" in reference to a human operator. However, this metaphor holds for *both* built-in controllers and custom controllers, despite only the latter being called “operators”. Also recall that the term “operator” has a long history as a computational metaphor, not as a metonym for a human role.

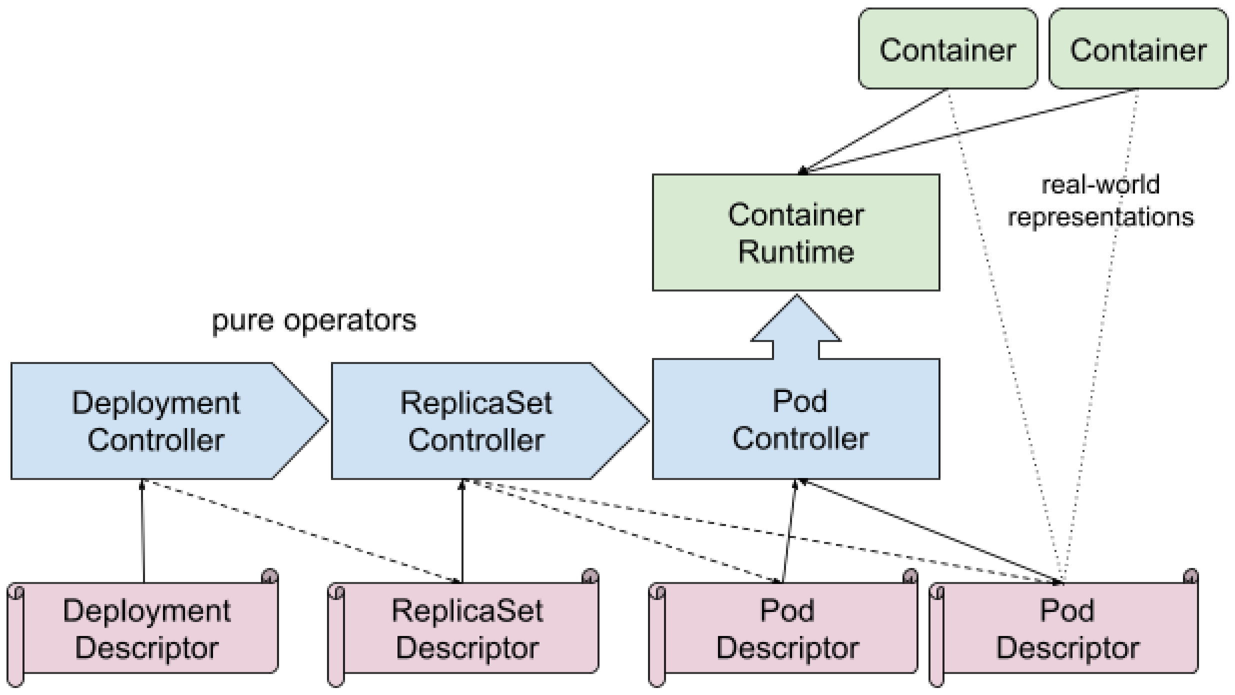
There’s no elegant way out of this mess. When talking about Kubernetes we have to use clumsy phrases such as “a Kubernetes controller that implements the operator pattern” or “a Kubernetes operator that does not implement the operator pattern”.

With this in mind, circling back to the introduction, in this document we address the two distinct sides of the confusion:

- We find value in extending Kubernetes through custom controllers (which Kubernetes insists on calling “operators”).
- We find value in applying the operator pattern in Kubernetes and beyond.

Built-In Examples of the Operator Pattern

Despite the terminological soup, Kubernetes comes with good examples of the operator pattern. Indeed, it has some pure operators:



In this diagram we start with the first implementation of the operator pattern, the Deployment controller, which makes sure to create, delete, and update a ReplicaSet resource to match the Deployment.

The ReplicaSet controller also implements the operator pattern, as it in turn creates, deletes, and updates Pod resources.

The last link in this chain, the Pod controller, does *not* implement the operator pattern. Its role is to use the container runtime (e.g. CRI-O) to create a set of linked containers based on existing images, to destroy them when the Pod resource is destroyed, and to update them, if possible, when the Pod resource is updated. The operational graph is thus terminated here.

Fine-grained Operators

The operator pattern, broadly speaking, does not require that the input and output intent be completely encapsulated. It's possible for *aspects* of intent to be translated into *other aspects* of intent. What this could mean in Kubernetes is that instead of taking a custom resource as an input the metadata of existing resources can be used. Likewise, the output of the operator does not have to be new resources, but rather extra metadata of existing resources. This is still an application of the operator pattern.

One example is Multus CNI, which relies on custom resources (NetworkAttachmentDefinitions) as well as on the "k8s.v1.cni.cncf.io/networks" annotation for Pods. Another example is Knap, which accepts another kind of annotation as an input and outputs Multus's annotation, allowing both of these fine-grained operators to be chained together via metadata.

Off-the-Shelf Operators

The term "custom" does not mean that *you* have to create the operator. It simply means that it's not part of the Kubernetes project. Some Kubernetes distributions, like OpenShift, come with additional operators beyond what the upstream provides.

The open source operator ecosystem is diverse and unfortunately varies widely in terms of quality and reliability. The most common type of operator found in the wild is a database cluster manager. Indeed, as we'll see below, managing state is an attractive use for impure operators. Other popular operators include: cert-manager, KubeVirt, Multus CNI, and Istio.

There are currently few production-ready operators that address telco-specific uses, but hopefully that will change and perhaps this document can inspire more. The CNCF's CNF Working Group could be a good place to conceptualize and kickstart the development of made-for-telco operators, whether they solve general networking needs, such as ENO and Knap, or address specific orchestration features in 5G core, MEC, O-RAN RIC, slicing, etc.

Uses for the Operator Pattern

Stateful Components

Best results are achieved when as many of your components are stateless as is possible. Statelessness can enable trivial redundancy, horizontal scalability, and idempotency, which are all key to service reliability. Where state is necessary a good practice is to disaggregate the stateful components from the stateless ones due to their very different design considerations.

Specifically, it is non-trivial to achieve redundancy, scalability, and idempotency for stateful components, especially in cloud environments. The applied methodology is often product- or domain-specific and makes use of diverse state management paradigms and technologies.

Here's a partial list of stateful functionality and implementation challenges:

- A set of differentiated replicas. Some nodes may be optimized for writing while others are optimized for reading. Some can be leaders while some can be followers.
- Load balancing can happen either on the client, which means the client must know about some or all of the nodes, or on the server via a (reverse) proxy or load balancer, which requires a reliable single endpoint, such as a Kubernetes Service or Ingress.
- Some data must be synchronized while other data can allow for eventual consistency within certain tolerations.
- Cluster-wide persistent sessions may be necessary, which require complex management of disconnects, reconnects, and garbage collection.
- Transactions are special kinds of sessions that add even more constraints, such as cluster-wide synchronicity.
- The lifecycle of the set or individual nodes may be continuous or modal. For example, in some setups new nodes may be able to join and immediately start handling requests, while in others nodes must synchronize data first. Removing a node may require a reconfiguration of the other nodes. Some nodes may occasionally need to move into backup, or cleanup, or resync mode, etc.

To meet these challenges stateful components are often coupled with a manager, which automates the step-by-step operations and continuously monitors the components. It's sensible for such a manager to be a Kubernetes custom controller ("operator") and indeed it's likely that it would implement the operator pattern—impurely, because its main role is the side effects. The operator's input could be a custom resource defining the component while the operator's output would be the built-in or custom resources needed: Deployments, DaemonSets, Services, PersistentVolumeClaims, ConfigMaps, Secrets, cert-manager Certificates, etc.

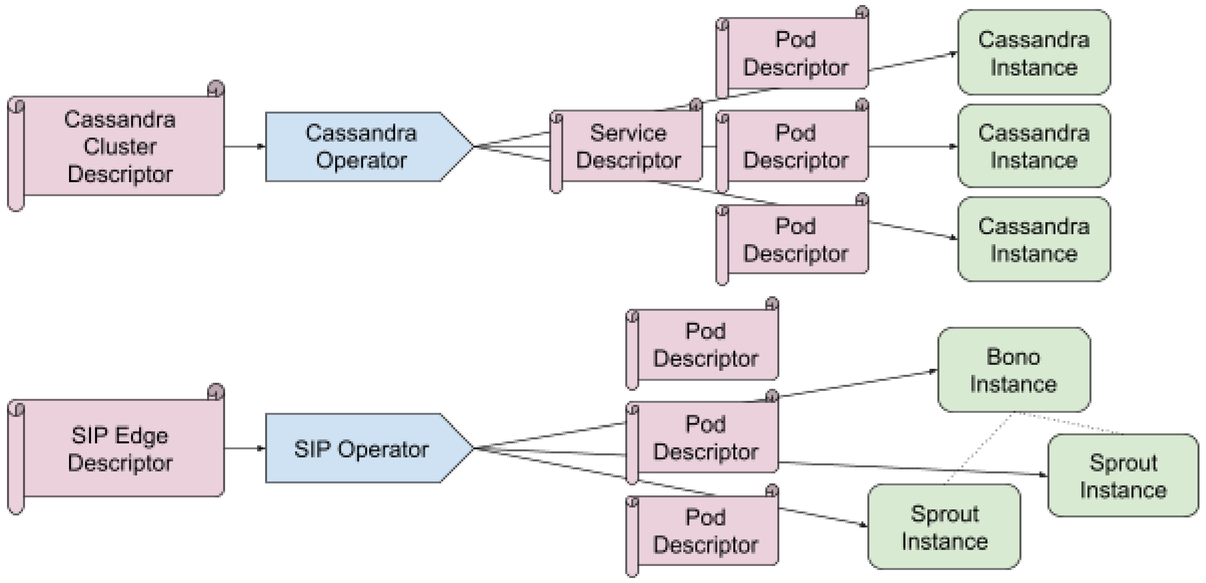
We can imagine a "fully" cloud native stateful component that would not need a manager. Nodes would self-manage and discover each other and coordinate among themselves to create clusters. However, even the most ideal cloud native design could benefit from a lightweight manager that would at least make sure that the minimum number of nodes is up, that a load balancer is configured, that clients have access to a connection URL, etc.

In Practice

An example of this kind of state management in a CNF can be found in the discontinued open-source version of Metaswitch's Clearwater IMS. Clearwater relies on a few third-party databases: Cassandra, memcached, and etcd. Additionally, it has two custom stateful networking components handling SIP connections: Sprout, which is a SIP router, and Bono, which is a SIP edge proxy. Several Sprout instances can run behind a single Bono instance, but they should not be removed or added at will as this would cause existing connections to be terminated.

This version of Clearwater predates Kubernetes and comes with its own cluster lifecycle manager, which implements the operator pattern to manage all the stateful components. It allows actions (scripts) to be queued, generates configuration files (the operator pattern's output), and restarts components as necessary.

We can imagine this CNF redesigned for Kubernetes, in which case we would prefer to have separate operators for Cassandra, memcached, and Sprout/Bono. For the former two we might be able to rely on off-the-shelf support, but Sprout/Bono would have to be deployed with a custom operator designed for its special clustering logic. Diagram:



Configuration Management

Configuration is the meatiest aspect of network function orchestration. More so than most other kinds of software components, network functions tend to expose a staggering array of situational, compositional, interdependent, and dynamic configuration parameters. So much so that standard transactional protocols (NETCONF, RESTCONF, gNMI) and data modeling languages (YANG) have been created just to help tame the depth and scope of network function configuration.

Additionally, network functions may have normal textual configuration files and thus can make use of Kubernetes ConfigMaps and Secrets, which an operator can manage.

Ways in the operator pattern can help:

- **An operator as a translator.** A descriptor can be used to provide a standard, centralized, validated configuration for one or more or all the components. References can be added to relate configuration to dependent components. The operator can resolve the references directly within the cluster or by communicating with other systems, such as an inventory, and finally derive and generate the configurations for each component. Changes to the descriptor can be reconciled and relevant components can be restarted or reloaded as necessary when their configurations are updated. (Note that Kubernetes does not normally restart Pods if their mounted ConfigMaps/Secrets change.)
- **An operator as a protocol client.** Whether it's NETCONF or similar, or a proprietary protocol, the operator can parse a descriptor and then make whatever remote calls are necessary in order to realize the intent. This work can be understood as a kind of translation, but with a more active, programmatic element. For example, the configuration workflow can involve transactions, it can query and update across multiple devices, handle errors via fallbacks and rollbacks, as well as other complex and situational logic. For an example of a programmable, generic approach see the Candice operator.
- **Certificate management.** Configuration protocols invariably involve security, which in turn requires issuing and rotating certificates stored as Kubernetes Secrets and possibly managing a CA. A good generic solution is the cert-manager operator.

A "fully" cloud native component would be able to build its own configuration according to the service in which it is deployed as well as general policies and other system-wide parameters. Unfortunately, reality can introduce modal interdependencies: perhaps one component needs to be fully running before it can generate the necessary parameters that another component needs for its configuration.

A straightforward solution that may work in many situations is simple repetition. Cloud native components would keep trying over and over again to configure themselves until the total system reaches the intended functionality. Perhaps a lightweight manager can be introduced just to make sure there are no endless loops. Alternatively, a custom controller can implement step-by-step lifecycle management. Yet another option is to use a generic workflow engine, such as Argo, which again relies on the operator pattern.

Disaggregation and Modularity

The classical cloud orchestration model is centralized. One big orchestrator sits outside and conceptually "above" all the cloud sites. It holds the complete and total intent for all network functions and services and is in charge of reconciliation (by policy), data collection, inventory management, and more. The bigger the overall network, the bigger the orchestrator has to be. As such there are severe scalability and reliability challenges with this model.

Out of the box Kubernetes already breaks the model. Because it itself is an orchestrator, the big orchestrator is already delegating some work to Kubernetes. A Kubernetes Deployment resource already handles replication, restarts, upgrades, and more. A Kubernetes Service resource already handles IP allocation, DNS registration, load balancing, and more.

The sensible next step is to move more orchestration work to Kubernetes. By introducing operators the big orchestrator would just need to create or update a custom resource instead of directly creating—and thus being responsible for—other Kubernetes resources and the orchestration work that needs to be done. This disaggregation scheme increases scalability and reliability and indeed enables new features:

- **Delegation:** More work is done locally in the cluster, thereby reducing the considerable back-and-forth chatter that would have to happen between a big remote orchestrator and the cluster.
- **Encapsulation and specialization:** Operators can be one-trick ponies that are focused on one orchestration task and that task only ("Unix philosophy"). This allows them to be lightweight, more stable, less dependent on other code and systems, and thus more reliable.
- **Isolation and security:** We can reduce the attack surface by exposing less data and fewer interfaces outside of the cluster. We can completely control and audit what we expose. (We will revisit this topic in "Host Access", below.)
- **Autonomy:** What if the remote orchestrator is not available? What if, even if it were available, it would not be able to respond in time to an urgent event? An operator can be designed to take action autonomously. The key to making this work well is having strong policies in place so that action can be properly confined. AI/ML can be added for optimizing the autonomous decision-making process.

We can refer to such operators as "orchestration modules" with the understanding that they are part of a larger orchestration narrative. Note that they can definitely make use of the operator pattern but can also be terminating controllers. It depends on whether they are somewhere in the middle of an orchestration graph or working exclusively with real-world resources.

Two generic examples of this approach are Red Hat's ACM and Microsoft's Azure Arc. Both provide a classical single-pane-of-glass interface while relying on custom controllers in the clusters to do the heavy lifting.

Note that so far in the discussion we have assumed the classical, layered, hierarchical approach to orchestration, a.k.a. "orchestrators of orchestrators". We have assumed that Kubernetes, plus our operators, would be a layer underneath higher layers of orchestrations. However, the more autonomy we provide to the operators the less authority the big orchestrator has.

Thus, rather than thinking of orchestration as layered, it makes sense to imagine a more horizontally modular architecture. In this scheme all modules have autonomy and the central coordinating task is policy management and reconciliation. Rather than have a big *orchestrator* at the center, perhaps all we need is a small *coordinator* there, while we do all of the real orchestration work locally with operators.

Integration and Control

Some components must work together. From an orchestration standpoint they are a unit. Indeed, the entire CNF can be seen as a single unit—a single custom resource—and its internal composition as an implementation detail that is decoupled from higher-level architectural concerns.

Operators for an entire application are sometimes called “installers”. They indeed allow the entire CNF to be installed from just a single custom resource and can be seen as a way to package a CNF (e.g. as a TOSCA CSAR file). However, the expectation in our context is that they manage the complete lifecycle, from Day 0 modeling through Day 1 installation to Day 2+ maintenance.

“S-VNFMs”

In extreme versions of this scheme the CNF becomes an opaque “black box” that comes with its own specialized manager (an “S-VNFM”), which is the required and only point of entry to deployment, configuration, data collection, and indeed all orchestration.

There are many advantages to managing an entire CNF from one descriptor with its own specialized operator. Complex policies for scaling and healing can be baked into the controller using straightforward programming logic rather than the anemic Day 1 anti-pattern of, say, a Helm chart. Such an operator would own all the resources and have a direct and complete view of the topology. Internally it can be disaggregated, and indeed a top-level operator can own and deploy additional operators to manage specific work. From the user’s perspective the composition is visible (Kubernetes hides nothing) but not modifiable. For example, if a user deletes one of the owned Pods the operator will reconcile the whole CNF and bring it back, doing the work necessary to heal after this unfortunate intervention, e.g. resyncing state, reconfiguring, etc.

The main disadvantage of this approach is that programming logic is hard to change. Adding support for new deployment configurations might require rebuilding the controller and thus very active vendor handholding. On the other hand, this allows the vendor to absolutely disallow unvalidated and unsupported configurations. If running a tight ship is the topmost priority then this is the way to go.

For an example, see the Magma mobile core from Facebook.

“G-VNFMs”

There is room for a middle ground that is not quite as extreme, a “G-VNFM” that integrates the entire CNF in one place but instead of hardcoding or otherwise hiding implementation details it *carefully* exposes the internal architecture for *validated* configurations.

The TOSCA language was created for such a purpose, as it allows the vendor to create custom, strongly-typed models, which the telco can then assemble into valid topologies. An example implementation is [Turandot](), a dynamic Kubernetes operator that allows any custom TOSCA to be translated directly into Kubernetes resources packaged with orchestration artifacts that can run directly in the Pods.

Integration or Disaggregation?

Note the tension between this use, “integration”, and the use we discussed above, “disaggregation”. Though seemingly in contradiction, many practical architectures would likely make use of both. A general rule of thumb is: keep the code physically close to the data. If that requires breaking it up then disaggregate. If that requires moving it into one place then integrate.

We’ll revisit this topic below when discussing “One Operator or Many”.

Management of External Resources

As the saying goes: in for a penny, in for a pound. If you’re already invested in Kubernetes as an orchestrator then it can make sense to use it to orchestrate non-Kubernetes resources. The advantage is that all your orchestration code runs in one place, uses the same paradigms, and can natively interact. Indeed, it would allow you to chain operators together even if some of the resources are external.

Let’s call these “representational operators” because they operate on representations of resources rather than the resources themselves. Of course any Kubernetes resource can be used in tandem: custom resources as descriptors, ConfigMaps and Secrets, and even a Service in order to reserve an IP address and DNS name.

An example of such an operator is the F5 BIG-IP controller.

An obvious use is PNF configuration. For example, if we have a Kubernetes operator handling all our CNF NETCONF work, then why not also let it handle NETCONF calls to our PNFs? Depending on our implementation it might be trivial to add this support. Perhaps all that would be needed is another operator that would automatically populate the cluster with custom resources for each PNF. Or maybe we would want a separate Kind of custom resource specifically for external network functions.

Practical Considerations

Custom Resources and Alternatives

Kubernetes’s custom resources (CRs) are purposely designed for use by operators. They support versioning, schema-based validation, and are fully integrated into the Kubernetes API server. However, they are not the only option available and may not be the best in every situation. They currently have two limitations:

1. Though CRs are namespaced, CRDs are not and thus they require administrative permissions to create or upgrade. In some deployment scenarios it may be impossible to provide these rights. (Upstream recommendation: Kubernetes should get rid of this limitation and allow CRDs to be namespaced.)
2. CRs are, like all Kubernetes resources, limited in size to ~1MB due to relying on etcd for storage.

ConfigMaps Instead of Custom Resources

If administrative permissions cannot be provided, a straightforward alternative to CRDs/CRs is to use ConfigMaps, which are readily available and are similarly general-purpose. A labeling convention can be used to annotate your resource type names and versions, allowing for filtering/selection of just the relevant resources.

Here are side-by-side equivalents of a CR and a ConfigMap:

```
apiVersion: knap.github.com/v1alpha1
kind: Network
metadata:
  name: mynetwork
spec:
  provider: bridge
  hints:
    mtu: 1500
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mynetwork
  labels:
    custom.kubernetes.io/apiVersion: knap.github.com/v1alpha1
    custom.kubernetes.io/kind: Network
data:
  provider: bridge
  hints.mtu: '1500'
```

(Note that the “custom.kubernetes.io” label prefix is not an accepted convention, merely our suggestion here.)

You will lose a few features by using ConfigMaps instead of CRs:

1. The ConfigMap’s “data” field maps strings to strings. One way around this limitation is, as seen above, using “.” paths to traverse nested structures and handle string

conversions in code. Another solution could be to have a single key containing arbitrary data in YAML, JSON, or another notation.
2. You will not have built-in schema-based validation. However, you can provide your own validation in your tooling and indeed it can be more powerful than what Kubernetes provides (OpenAPIv3).
3. ConfigMaps do not have a separate status sub-resource. You can of course decide for yourself that some of your data is “spec” and some is “status”, but you won’t have the ability to update the status separately without marking the resource as changed. Again, if this is required functionality then you can track changes on your own using your own custom ID system, data hashes, etc.

Until Kubernetes allows for CRDs to live in namespaces, there’s room for an open source tool to manage these workarounds for using ConfigMaps instead of CRs: data translation, schema validation, and change tracking. Could someone please create it?

Resources As References

If the ~1MB size limit is at all a concern then obviously data will have to be stored elsewhere. However, this doesn’t mean that Kubernetes can’t help. The resource, whether it’s a CR or a ConfigMap, can be used to hold a reference (pointer) to that data, such as an ID, a repository URL, etc. Note that if authentication credentials are necessary then they should be stored separately in Secrets.

Implementing this approach comes with some challenges. References can be orphaned and may need to be validated before every use and garbage collected. This extra work can be handled by the operator, but it does add complexity. And, of course, that data must be stored somewhere, which requires the deployment and management of a data store.

In some cases that data store might indeed be the starting point, e.g. if we are working with NETCONF against a YANG database, then our Kubernetes references can point to YANG entities.

Non-Kubernetes Resources

It is of course possible for your operator to not use Kubernetes at all for its resource definitions, relying entirely on a separate data store. Indeed, nothing about the operator pattern requires you to use CRs or even ConfigMaps.

Implementing this means that you would have to provide your own APIs and tooling for creating, updating, and querying the resource definitions. Also, you cannot make use of Kubernetes ownership and garbage collection (see below).

GitOps

Note that the data store for your intent could be a git repository. This would allow your operator to fashionably take part in a GitOps workflow. (Turandot indeed supports “git:” URLs.)

Namespaced or Cluster-Wide

An operator can be deployed in the same namespace as the CNF and only have permissions to work on resources in that namespace. This allows for best isolation. However, if multiple CNFs use the same operator then this will come at the cost of duplication. It's important to remember that namespaces provide isolation but *not* true security. However, some security solutions, such as Cilium, do build on top of the namespace feature.

Alternatively, the operator can be deployed in a separate namespace, or even in “kube-system”, and have permissions for some or even all namespaces. For some operators this may be a requirement and thus it would also require administrative permissions to install or upgrade it. If the operator requires a CRD then administrative permissions would be required anyway.

It is generally good practice to allow for both schemes if possible. Many operators do.

Host Access

Depending on the work the operator does it may require privileged access to the hosts. For example, it may need to interact with operating system services, configure NICs, update routing tables, etc.

In these cases a DaemonSet can be used to ensure that an operator Pod runs on all nodes in the cluster. Be aware that this means that you will *always* have multiple copies of your operator running. See “Developing Operators” below for more detail on what this would entail.

Note that privileged host access presents many security challenges that are out of the scope of this document. That said, a common practice is to isolate all privileged access requirements into one component in order to reduce the attack surface. (See “One Operator or Many”, below.)

Workload vs. Platform

When is the operator installed during the lifecycle of the network service? And, relatedly, who installs it?

As discussed above, there's a chance you would need administrative permissions in order to install CRDs and/or enable privileged access to the hosts. This may coincide with procedural organization, for example if only one team within operations has access to administrative permissions then they must be put in charge of installing operators.

And yet there are neither architectural nor conceptual constraints that place operators exclusively in the “workload” or “platform” buckets. An operator can definitely be delivered as part of a CNF, and indeed it may be useless in any other context. Alternatively, a general-purpose operator can be listed as a requirement that should be provided by the “platform”. Similar considerations are often applied to storage components, such as databases, which can be consolidated to serve many workloads. So, then, another way to phrase the question: is the operator providing a “service” to the CNF or is it a component of the CNF?

If it is a “service” then there are practical considerations depending on whether it is exclusive to the CNF or shared with other workloads. If it is shared the door is open to conflicting version requirements. If the operator is fully namespaced then this should not be a problem: different versions can live in different namespaces. However, because CRDs are cluster-wide it is important to also take care not to break the versioning scheme built into the CRD spec. If the API needs to change—even if it’s a semantic change rather than a syntactical one—then the version should be bumped in order to make sure that the right version of the operator will be tracking it.

All things considered, the question of “workload vs. platform” is best answered as part of a general cloud management strategy, which may indeed differ between central and edge sites and even between product categories. For example, some CNFs might be certified as whole stacks of hardware and software, which would include the Kubernetes cluster as well as operators. Others would be more portable and may even be expected to run in public clouds and other something-as-a-service scenarios. Technical as well as procedural constraints must be considered in all cases.

Ownership and Garbage Collection

Kubernetes has various built-in cross-reference types but only one has an orchestration effect: metadata ownerReferences. This feature allows you to tie the lifecycle of resources to each other, such that if you delete the “owner” then the “owned” resources would also soon be deleted.

This effect is obviously useful for the operator pattern: the input intent (the CR) can be made the owner of any generated output intent. Delete the CR and everything that the operator created for it would be deleted.

In some cases, such as in stateful components, this kind of simple garbage collection could be undesirable. There might be a set of operations required before these resources are deleted. In that case you must implement your own lifecycle management and not use this feature.

Be aware that Kubernetes ownership is used only for garbage collection and nothing else. “Owning” a resource does not protect it from being changed by any client that has permissions to do so. A robust operator would thus need to continuously validate that its owned resources exist and are properly configured.

One Operator or Many

There is no technical reason why an operator can’t do many things at once. It can indeed process a few different custom resources. In other words, there’s no rule saying that you need one operator per resource type.

There are certain advantages to centralizing operations. You can deploy your CNF with a single operator that does it all. That’s just one thing to deploy and one thing to upgrade and one thing to publish. This approach can simplify development and operations and increase reliability.

The advantage of dividing the work among many operators is that each would be more focused and potentially more reusable. A certain challenge can be met with a somewhat generic solution that could be used elsewhere, in other CNFs and configurations.

Note, however, that reusability can be achieved at the level of code modularity instead of in packaging and deployment. For example, you can have different code libraries to process different kinds of custom resources. You can then pick and choose, per CNF project, which libraries to compile into its “one big” operator.

Also note that modularity is not only about making individual operators independent and reusable. For example, your solution might involve one “main” operator and several “helper” operators that are designed to work with it and have no use elsewhere. One use case is our “Host Access” scenario discussed above: some work has to be done on every host (a DaemonSet of “helpers”) while we could have a “main” operator coordinating between them.

Developing Operators

It is challenging to develop operators in Kubernetes and even more challenging to develop good operators. A “best practices” guide is out of the scope of this document, but let’s briefly survey the terrain:

- The API server is an asynchronous data environment but in fact is not event-driven. However, our controller design likely *is* event-driven because we trigger reconciliation actions when resources (CRs) are added, deleted, or are changed. One way for a controller to “watch” resources of a certain Kind is to set up a polling mechanism (an “informer” in the Go client) that caches the data, occasionally resyncs it, and can generate events for you. If this cache is per-process you must consider what would happen if you had two or more controller processes running at the same time (e.g. a DaemonSet). If handling the same event twice is undesirable (due to non-idempotency or cost) then some inter-process synchronization is necessary.
- Because these are not true system events they are not persisted anywhere. If your controller was not running when they happened then it’s as if they never happened. Thus you occasionally need to resync and do your own validation, including garbage collection of orphaned resources.
- How will you handle errors? Will you requeue the event to be tried again? Will you implement a backoff algorithm to avoid hammering the API server? Or will you notify the user (in the CR status) and avoid retrying? How do you decide what to do?
- The API server resources are not lockable. Thus, between *any* call to get *or* change data you must plan for other clients to have changed it. This can lead to errors, which may or may not require a retry (see above). It is possible that another controller process (your own code) is already handling the event, which can lead to subtle race conditions. Would the two processes keep interrupting each other and thus reconciliation would never happen? Or, worse, would they keep *correcting* each other and create an endless loop of costly operations?

All of the above challenges can be met. However, success in this kind of development work requires considerable expertise and experience, with race conditions being especially hard to test for. Importantly, developing the operators likely requires a different set of skills than those needed for developing the core CNF components. This implies a diverse skill set in the engineering team.

Go or No-Go

To be clear, Go is absolutely not required for writing Kubernetes operators. Though Kubernetes itself is written almost entirely in Go, its API server is language-agnostic. Here are several open source alternatives:

- 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.

The Kubernetes Go client still has the advantage of being the most native, most up-to-date, and most tested. Are you already doing work in Go? If not, depending on your point of view, introducing a new programming language into your process can represent a hurdle for development or an opportunity for growth. Here are some highlights of what choosing Go entails:

- Strongly-typed compiled language with garbage collection and green threads.
- Fast-growing open source ecosystem.
- The Kubernetes Go client supports code generation for creating typed APIs for custom resources. Alternatively it also supports an “unstructured” (and untyped) code path for working with arbitrary resources, without code generation.

Note, too, that you do not have to use the complete Operator Framework to develop operators in Go. You can instead use Kubebuilder directly, which provides the basic scaffolding for Operator Framework. Or even start with the sample-controller code and build from there. In fact, it’s a good place to learn how to work with the Kubernetes Go client in the API server environment. There are also many blogs that can help you get started, like this one.

Conclusions

TODO