Dependency graph for configuration items

Motivation

Zedrouter and NIM are already one of the most complicated microservices within EVE. This is mostly due to the fact that they have to deal with many configuration items (routes, bridges, interfaces, etc.). A single high-level configuration object defined by the controller, such as a device port or a network instance, is actually built using multiple configuration primitives inside the (Linux) network stack. EVE may even start additional processes, like dnsmasq or radvd, configured accordingly to serve requested network services.

EVE has to deal with the *intended* (also called *desired*) state on one side, which in this case is `EdgeDevConfig` received from the controller, and with the *actual* (also known as *the current*) state on the other side - the configuration currently applied and running on the device. The intended state is defined (using protobuf) to be more high-level, describing the intent but not the implementation. EVE first has to map it to low-level configuration primitives that implement the desired functionality. Next, it has to determine the difference between the currently running configuration and the new intended state. Finally, it performs all changes necessary in the form of Create/Modify/Delete operations to transition from the current state to the new intended state. Ordering of these operations matters and it has to respect any dependencies that exist between configuration items. For example, a virtual interface (VIF) of an application cannot be inserted into a bridge if that bridge has not yet been created. This is further complicated by the fact that the configuration space is split between microservices, sending updates between each other using pubsub.

To summarize, the problem described above can be split into the following tasks (for an EVE microservice):

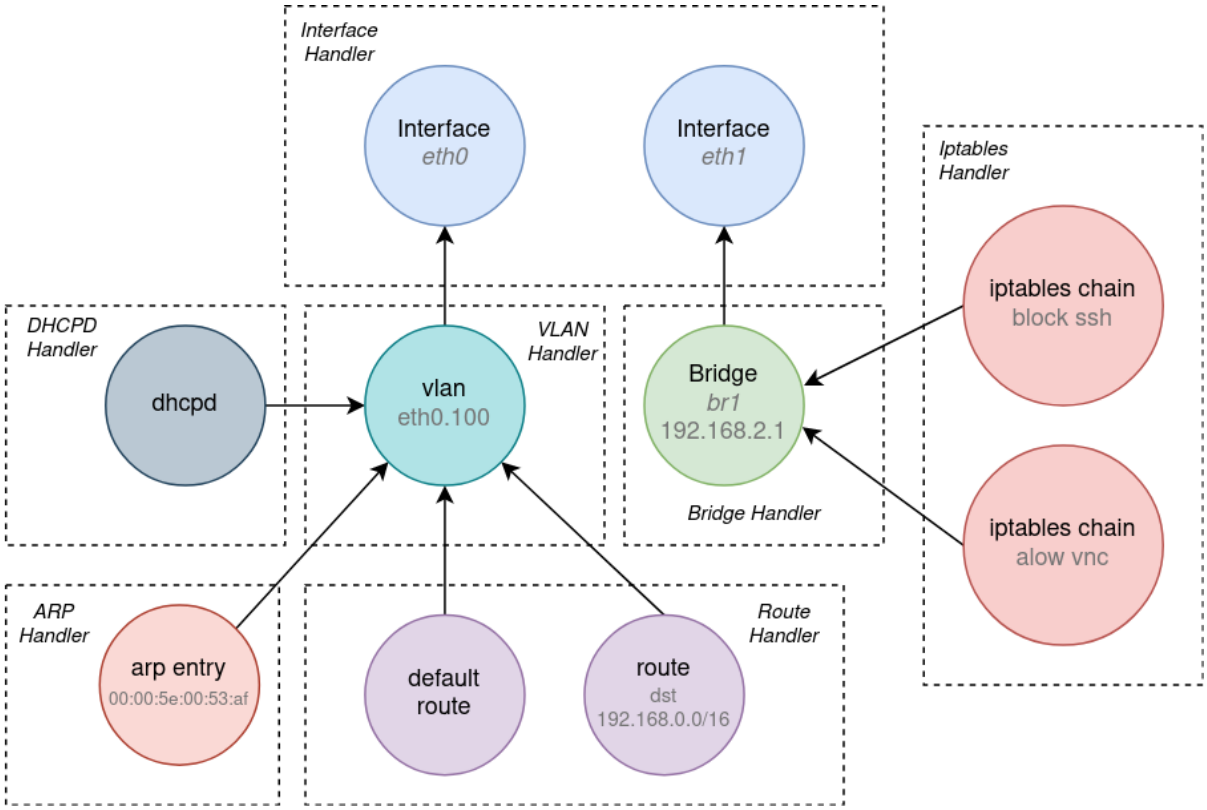
1. From (high-level) `EdgeDevConfig` (or a portion of it that the microservice receives) *render* the corresponding set of configuration primitives, processes or files to create/run/write in the running system
2. Determine the difference between the current system state and the new intended state
3. Determine the set of Create/Modify/Delete operations to execute and find a valid ordering (might depend on objects configured by other microservices)
4. Execute operations, collect any errors and report back to the controller (or to other microservices)

The task no. 1 is specific to each microservice. It may also depend on the hardware (e.g. which virtualization technology to configure). Note that making this part of the code separated using interfaces/structs, would allow us to have the configuration implementation *replaceable.* For example, it might be possible to easily switch from Linux network stack to some vswitch, like OVS.

However, the topic of this proposal are the tasks 2. and 3., and as shown below, they can be tackled in a generic way and in one place.

Proposal

We propose to solve the problem of the currentintended state reconciliation generically using a dependency graph. Every configuration primitive (rendered from the task 1.) can be represented by a single graph node. Dependencies between items are modeled using directed edges. For the start, we will need to support dependency with the semantics "*must exist*". For example, if a route depends on a particular network interface to be configured first, there would be an edge originating at the route's graph node and pointing to the node of the network interface. A visual example with a dependency graph representing network configuration is shown below:



The graph should be able to:

1. Store the currently running state
2. Allow to prepare the new intended state
3. Move the current state towards the new intended state - this means to:
    a. Determine "diff" between the current and the new state
    b. Find ordering of Create/Modify/Delete operations that respects the dependencies. Erik: is there some assumption that a modification to a parent (e.g., vlan above) also means there is a need to run a modify, or a delete+create, of a child? There might be some cases where the parent modify results in a delete and recreate which would implicitly delete e.g., routes using that interface. Milan: Yes, this may happen. If modification is done through re-create, items that depend on it are first deleted and then created after the modify. Also, even if modification is done in-place (not by recreate), it is possible to explicitly request re-create of items that depend on it. This may be needed in some cases.
    c. Run operations. Erik: Can it handle Run that needs to be asynchronous e.g., the nim case of asking domainmgr to get back eth1 from pciback? Would it make sense for the reconciler to return a list of functions to call and have the caller execute those functions? Then the functions can return done, error, waitforX. Milan: Currently not supported, but this is planned (scroll down to "limitations and Future plans"). First version of the graph I wanted to have as simple as possible, but the next improvement will be to add support for asynchronous operations.
    d. Store and expose any errors returned by Create/Modify/Delete operations

For the step 1., the graph must also allow to represent configuration items managed by microservices other that the one that owns the graph (let's label them as "*external*"). For those, the graph will never trigger Create/Modify/Delete operations and will use them only for the purposes of dependency management (e.g. create A only after another microservice has already created B).

For the step 3.c, the graph needs to have an access to *handlers* of configuration items (those which are not *external*). For the graph this can be a structure that implements an interface with Create/Modify/Delete methods. For every distinct configuration item type (like "Linux route", "container", "Linux bridge", "dnsmasq"), there will be a separate handler registered with the graph. For the graph, these handlers (in the preliminary implementation called "Configurators") are essentially backends or drivers, that the graph calls as needed to synchronize the current state with the latest desired state.

Additional Benefits

Dependency graph will not only allow to solve a common problem in one place, therefore shortening the code size and the complexity of microservices that will use it, but it will also enforce a much more readable and sustainable programming style. Consider the following comparison between the current and the new programming style (note that the code here is only symbolic, not actually taken from EVE):

Current programming style	**New programming style**

```
whenNetworkConfigChanges() {
        determineObsoleteVlans()
        removeObsoleteVlans()
        determineObsoleteBonds()
        removeObsoleteBonds()
        changeInterfaceIPsIfNeeded()

ifSomethingChangedRestartDnsmasq()
        addNewBonds()
        addNewVlans()
        ...
}
```

```
whenNetworkConfigChanges() {
        newConfig := []ConfigItem{
                interface(params),
                arpEntry(params),
                arpEntry(params),
                route(params),
                route(params),
                bridge(params),
                dnsmasq(params),
                // A comment explaining why this config item is
here…
                iptablesChain(params),
                iptablesChain(params),
            ...
        }
        graph.Cluster(<network-name>).Put(newConfig)
        err := graph.Sync()
    ...
}
```

Note that the example also presents the concept of clustering (subgraphs), that the depency graph will support and which was borrowed from graphviz. Having support for graph clustering will allow us to group items which are in some way related to each other. For example, all components of the same application (domain, volume, VIFs) could be grouped under one cluster. This will be mostly done to simplify modifications to the intended state. As demonstrated in the example, the intended state of a single specific network can be replaced with just one function call: `graph.Cluster(<network-name>).Put(newConfig)`

Note that the new approach is not only easier for the developer and therefore less bug-prone, but also allows to *explicitly* express the *intent (*= newConfig*),* while the steps (the sequence of configuration changes) needed to take to get there are implicit. Compare that with the current approach, where the steps are explicit, but the programmer's intent is implicit. To determine what the program is trying to configure, one must study the code thoroughly and build a mental image of the intended state. If the programmer made a mistake in that complex code, one might get a wrong idea of what the intended state is.

Lastly, with the dependency graph, it will be much easier to add new features. A programmer will only need to implement handlers for new configuration items and describe their dependencies. The rest is being taken care of by the graph.