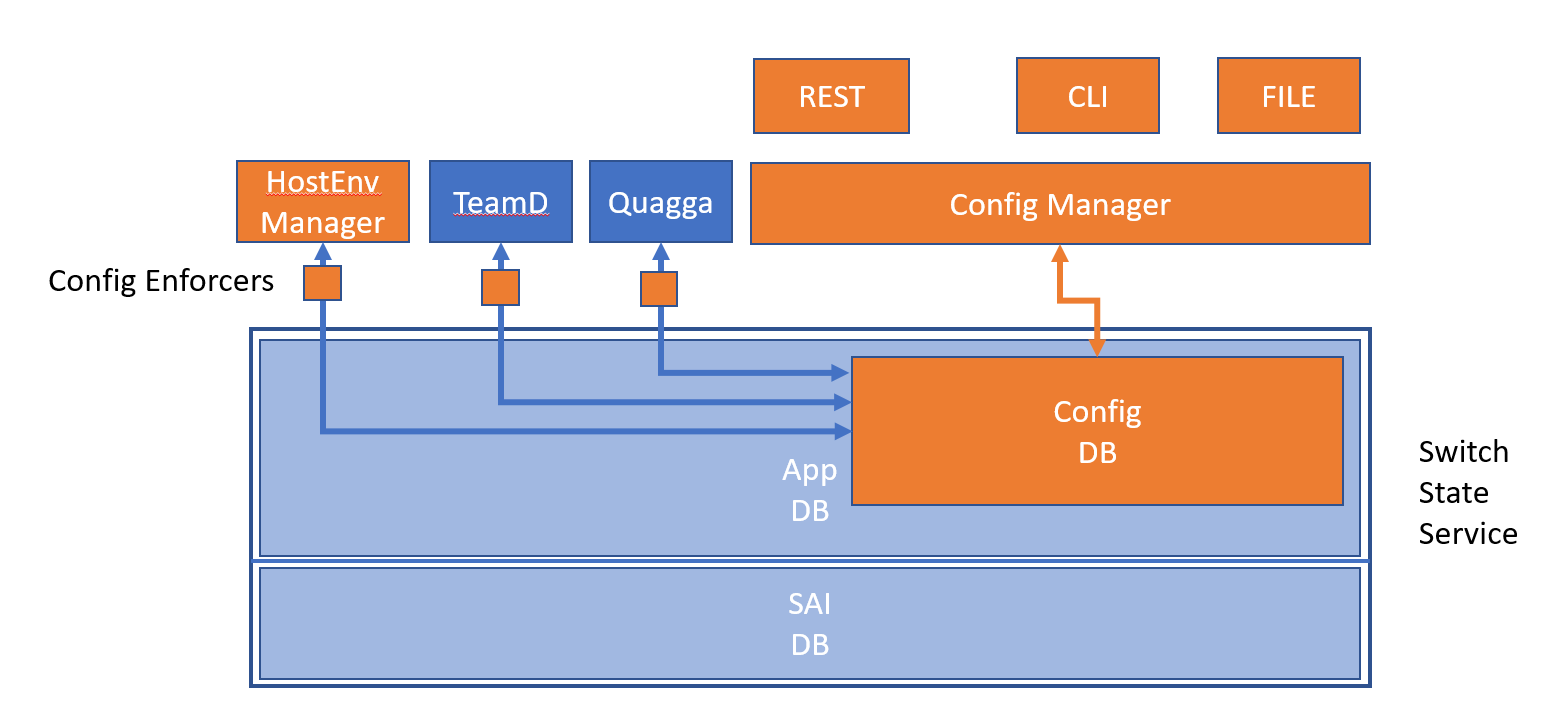
SONiC Configuration Management

# Overview

This document outlines how SONiC support configuration management. Configuration management in SONiC is highly driven by the concept of “desired state”, aka, “intention” or “gold config”.



# System Components



Broadly, configuration management in SONiC consists of three system components.

1. ConfigDB: where the authoritative configuration is stored on the box.
2. ConfigManager: which provides the north-bound interface to ConfigDB.
3. ConfigEnforcer: which enforces the desired configuration in ConfigDB to the whole system.

ConfigDB**:** ConfigDB is basically table in the Redis DB for storing the configuration. With this DB, one can save the current configuration to disk by dumping the table, and restore to a set of config by loading the DB. The internal schema of ConfigDB follows the philosophy of Switch State Service, exposing essentially a set of table that stores a set of Entity/Attribute/Values, which maps to the DB-centric CRUD semantics naturally.

We intend to make the data model used in the ConfigDB close to the OpenConfig/Yang model – as defined in <https://github.com/openconfig/public/tree/master/release/models> . This will enable the REST layer to be easily adapted to the OpenConfig model and the ecosystem there.

However, we intend to

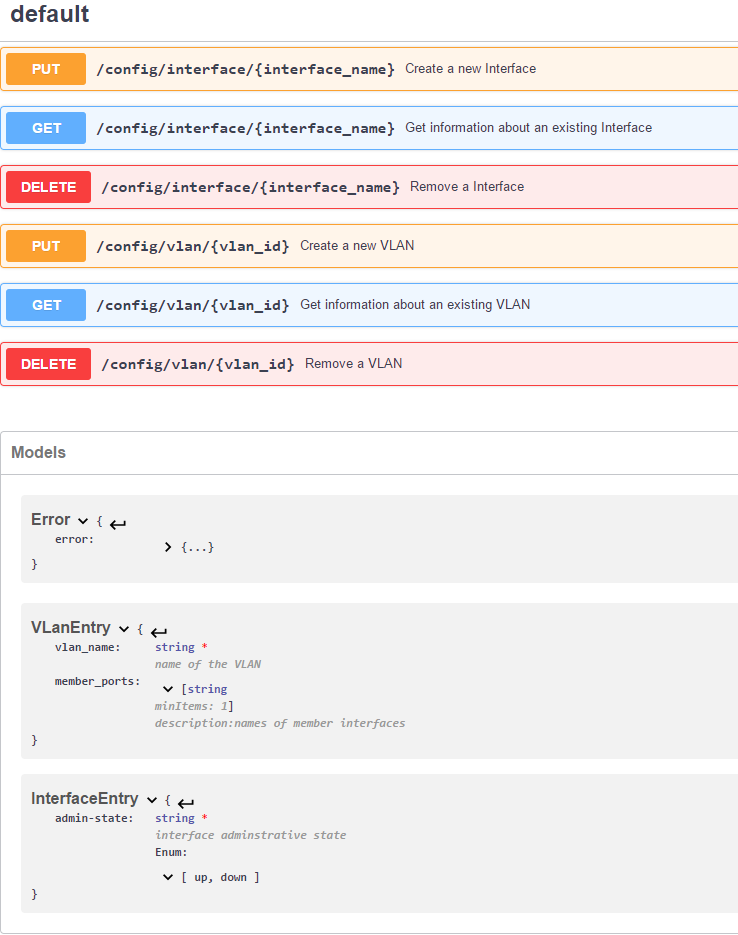
(1) only adopt the minimum portion of the Yang model that is needed instead of trying to be feature complete on all the models specified on day one.

(2) keep a clean separation of desired state (“config”) and observed state (“state”). The ConfigDB will only have the configuration related models. This is also the general principle used in the OpenConfig.

Config Manger**:** ConfigManager provides the interface to manage the desired state of SONiC. If a configuration is accepted by ConfigManager, then it becomes the desired state of the SONiC box, and the software stack will drive SONiC to behave according the desired state. Therefore, ConfigManager assumes that all configuration change will go through itself, and takes the responsibility of validating every configuration change before it is accepted. For example. ConfigManager needs to explicit fail the creation of VLAN with bad vlan id and prevent the duplication of IP addresses on two different interfaces.

At the beginning stage, SONiC Config Manager will support three kinds on interfaces, RESTFul API, CLI over SSH, and file-based interface.

RESTful API:SONiC intend to use swagger/OpenAPI 2.0 as the API specification language.



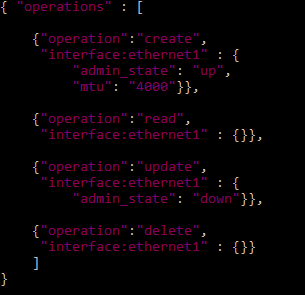
Above is an example of REST API specified in OpenAPI/Swagger 2.0, with only VLan and Interface as examples. As mentioned earlier, we will adopt the Yang model defined in OpenConfig if possible.

### CLI:

We propose to implement the CLI interface as a wrapper of the REST interface with a lot of syntactic sugar. Following is a mocked up example.

|  |
| --- |
|  |

The above command line operation is essentially driven by CRUD operations defined in the following json document.



## File-Based Config:

ConfigManager assumes every SONiC box will have a file (/etc/sonic.startup.config) that defines the options available to load the startup configuration.

|  |
| --- |
| {"configpath": [  {"priority": "1", "model": "RedisDB", "path": "/etc/sonic.configDB.redis.dump"}  {"priority": "2", "model": "MiniGraph", "path": "/etc/sonic.minigraph.config"},  {"priority": "3", "model": "MiniGraph", "path": "http://configserver/hostname.minigraph.config"},  ]  } |

We intend to support both loading from the static file in the JSON format that is largely defined by the MiniGraph + OpenConfig/Yang, and RedisDB dump. We also intend to support retrieving config file from a http server.

The RedisDB can be dumped from SwSS at any time and stored. This would enable config to be dumped from one box and restored on another box easily.

We believe it is feasible to build a RedisDB/MiniGraph translator.

At this early phase, we plan to load start up config from a single source, instead of merge between multiple sources. This is mostly due to the complexity of building validation capabilities and conflict resolution. Building the validation capability for a single source is already challenge enough, so until that is done, we don’t intend to support startup from multiple sources.

## Configuration Application Modules:

Once a set of configuration is accepted by ConfigManager and written to ConfigDB, they will need to be applied to each application containers as well as the host environments.

**Distributed App Config Modules**

This is a set of per-container module that is responsible to reading the relevant portion of data in ConfigDB and translates to the proper in-container app configuration. Depending on the application’s requirement, they can either choose to scan the relevant fields in ConfigDB periodically or use the pub/sub model provided by SwSS.

For example, Quagga will need to understand the configurations related to BGP (peers, keepalive timers etc.), and then generates its own internal configurations and behave accordingly. This can be easily achieved by inserting a configuration module inside the Quagga container. This module needs to know how to read and understand the ConfigDB, and serves as a translator between the common SONiC configuration semantics and Quagga’s internal semantics. It might be implemented a pub/sub model for peer up/down state changes, but maybe only scan the local keepalive timer change periodically.

Instead of having the Configuration Manager centralize the functionality off pushing the configuration to all the containers, we elect to have the configuration module to be distributed into individual containers. The motivation here is to avoid bloating ConfigManager with the liability to understand all the possible containers. If a container is not needed in a particular SONiC environment, the Config Module will not be present as well. In the scenario of SONiC swapping Quagga with GoBGP, the Quagga container and the GoBGP container will each have its own configuration module for this translation purpose.

There is concerns on how can we keep the application container and the configuration modules relatively isolated from each other. E.g. assuming SONiC gets BGP container using FRR, but the general purpose FRR is not for SONiC only, so the default FRR image may not contain the SONiC-specific configuration module. This issue is addressed by the “OverlayFS” used in SONiC.

Quote: <https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/#image-layering-and-sharing-with-overlayfs-overlay>

======================= BEGIN QUOTE=======================

OverlayFS takes two directories on a single Linux host, layers one on top of the other, and provides a single unified view. These directories are often referred to as layers and the technology used to layer them is known as a union mount. The OverlayFS terminology is “lowerdir” for the bottom layer and “upperdir” for the top layer. The unified view is exposed through its own directory called “merged”.

The diagram below shows how a Docker image and a Docker container are layered. The image layer is the “lowerdir” and the container layer is the “upperdir”. The unified view is exposed through a directory called “merged” which is effectively the containers mount point. The diagram shows how Docker constructs map to OverlayFS constructs.



===================END QUOTE===========================

Incremental Config Update:

Each application container has the responsibility to monitor the ConfigDB, and react to changes in the DB that are relevant to itself and in the most appropriate way according to its own capabilities. For example, the SNMP container, knowing restarting itself will not cause a data plane failure, may choose to simply reload itself to pick up the new configurations. On the contrary, the TeamD container might choose to incrementally add a link to the LAG group instead of reloading itself completely. This enables the SONiC community to enable incremental config update on the modules that matters first. From a user scenario perspective, we see the need of incremental update for ACL, Credentials, Link bundles, but less so for interface break out or interface IP change etc.

With the distributed configuration module design, it certainly brings the challenge that the order of change propagation may not follow certain expected order, and therefore, may cause some failures. For example, if a BGP peering config change happened before the HostEnv changes the interface IP configuration, the peering will certainly not come up. The counter arguments here are (1) one app container should not make assumption about other app containers and needs to handle errors gracefully, and (2) the standard “eventual consistency” model, that the changes to the ConfigDB may or may not be picked up by individual containers in the original order. However, as long as they eventually converged to the final state, the system should end up in the right state.

**HostEnv Manager:** SONiC leverage many utilities provided by the hosting Linux environment and the kernel, e.g. the physical interfaces. HostEnv Manager is then responsible for configuring the Linux host environment.

# Clarification Questions:

### OpenConfig:

SONiC currently uses the MiniGraph model, with the exception for the case of ACL, which SONiC uses the Yang-model defined in OpenConfig. We’ll adopt the OpenConfig/Yang model for the components we need.

### Source of Truth:

ConfigDB is the only source of truth in SONiC. We believe this is necessary to keep the simplicity and predictability of operations. We recognize it is quite possible for people to go to the linux host environment and modify the host environment configuration directly. Similarly, it is possible to go into individual containers and modify the configs there. But if it is not reflected in the ConfigDB accordingly, they could be overwritten by the ConfigEnforcer next time it sync with the ConfigDB.

We may make future improvements to automatically sync kernel changes to ConfigDB, but that is not the priority for now.

### BootUp:

SONiC will load the config based on the order defined in /etc/sonic-startup.config. It will only load one, instead of trying to merge multiple data source at start up phase.

### Config Validation:

ConfigManager will have the overall system level config validation before any config is accepted into ConfigDB.

### MiniGraph and ConfigDB:

MiniGraph is JSON/XML format, which is easier for human to interpret, ConfigDB is Redis format, which is more geared for storage and query interface. Their schema can be very different. We don’t recommend applications to use the RedisDB directly.

### Central Controller:

They can choose to (1) upload the MiniGraph file and then reload the box, or (2) they can use the REST API to configure the box.

SONiC’s CLI interface is PURELY for human administrator, and will be difficult to automate around.

### ConfigDB and AppDB

Yes, ConfigDB and AppDB are separate. There is no dedicated StateDB – App states are in AppDB.

### CLI vs. REST:

CLI implementation is wrapped around the REST interface. REST can be served via HTTPS, so can use cert-based auth.