Architecture Yanzhao Zhang edited this page Jul 21, 2022 · 13 revisions Referenced Documents

Document Title Document Identifier & Link

1 SONiC official wiki https://github.com/sonic-net/SONiC/wiki 2 SONiC architecture https://github.com/sonic-net/SONiC/wiki/Architecture 3 SAI API https://github.com/opencomputeproject/SAI 4 Redis documentation https://redis.io/documentation 5 Click module http://click.pocoo.org/5/ 6 JSON introduction https://www.json.org/ 7 SONiC supported platforms https://github.com/sonic-net/SONiC/wiki/Supported-Devices-and-Platforms 8 SONiC Software Architecture 2018 Workshop Videos Slides SONiC System Architecture

SONiC system's architecture comprises of various modules that interact among each other through a centralized and scalable infrastructure. This infrastructure relies on the use of a redis-database engine: a key-value database to provide a language independent interface, a method for data persistence, replication and multi-process communication among all SONiC subsystems.

By relying on the publisher/subscriber messaging paradigm offered by the redis-engine infrastructure, applications can subscribe only to the data-views that they require, and avoid implementation details that are irrelevant to their functionality.

SONiC places each module in independent docker containers to keep high cohesion among semantically-affine components, while reducing coupling between disjointed ones. Each of these components are written to be entirely independent of the platform-specific details required to interact with lower-layer abstractions.

As of today, SONiC breaks its main functional components into the following docker containers:

`Dhcp-relay`

`Pmon`

`Snmp`

`Lldp`

`Bgp`

`Teamd`

`Database`

```
Swss
```

```
Syncd
```

The following diagram displays a high-level view of the functionality enclosed within each docker-container, and how these containers interact among themselves. Notice that not all SONiC applications interact with other SONiC components, as some of these collect their state from external entities. We are making use of blue-arrows to represent the interactions with the centralized redis-engine, and black-arrows for all the others (netlink, /sys file-system, etc).

Even though most of SONiC's main components are held within docker containers, there are some key modules seating within the linux-host system itself. That is the case of SONiC's configuration module (sonic-cfggen) and SONiC's CLI.

A more complete picture of all the possible component interactions and the associated state being transferred, will be covered in subsequent sections of this document.

Sec4Img1 SONiC Subsystems Description

This section aims to provide a description of the functionality enclosed within each docker container, as well as key SONiC components that operate from the linux-host system. The goal here is to provide the reader with a high-level introduction; a more graphical and (hopefully) intuitive approach will be followed in subsequent sections.

Teamd container: Runs Link Aggregation functionality (LAG) in SONiC devices. "teamd" is a linux-based open-source implementation of LAG protocol. "teamsyncd" process allows the interaction between "teamd" and south-bound subsystems.

Pmon container: In charge of running "sensord", a daemon used to periodically log sensor readings from hardware components and to alert when an alarm is signaled. Pmon container also hosts "fancontrol" process to collect fan-related state from the corresponding platform drivers.

Snmp container: Hosts snmp features. There are two relevant processes within this container:

```
Snmpd: Actual snmp server in charge of handling incoming snmp polls from e
```

```
Snmp-agent (sonic_ax_impl): This is SONiC's implementation of an AgentX sn
```

Dhcp-relay container: The dhcp-relay agent enables the relay of DHCP requests from a subnet with no DHCP server, to one or more DHCP servers on other subnets.

Lldp container: As its name implies, this container hosts lldp functionality. These are the relevant processes running in this container:

```
Lldp: Actual lldp daemon featuring lldp functionality. This is the process

Lldp_syncd: Process in charge of uploading lldp's discovered state to the

Lldpmgr: Process provides incremental-configuration capabilities to lldp d
```

Bgp container: Runs one of the supported routing-stacks: Quagga or FRR. Even though the container is named after the routing-protocol being used (bgp), in reality, these routing-stacks can run various other protocols (such as ospf, isis, ldp, etc).

BGP container functionalities are broken down as follows:

```
bgpd: regular bgp implementation. Routing state from external parties is r

zebra: acts as a traditional IP routing-manager; that is, it provides kern

fpmsyncd: small daemon in charge of collecting the FIB state generated by
```

Database container: Hosts the redis-database engine. Databases held within this engine are accessible to SONiC applications through a UNIX socket exposed for this purpose by the redis-daemon. These are the main databases hosted by the redis engine:

```
APPL_DB: Stores the state generated by all application containers -- route

CONFIG_DB: Stores the configuration state created by SONiC applications --

STATE_DB: Stores "key" operational state for entities configured in the sy

ASIC_DB: Stores the necessary state to drive asic's configuration and oper

COUNTERS_DB: Stores counters/statistics associated to each port in the sys
```

Swss container: The Switch State Service (SwSS) container comprises of a collection of tools to allow an effective communication among all SONiC modules. If the database container excel at providing storage capabilities, Swss mainly focuses on offering mechanisms to foster communication and arbitration between all the different parties.

Swss also hosts the processes in charge of the north-bound interaction with the SONiC application layer. The exception to this, as previously seen, is fpmsyncd, teamsyncd and lldp_syncd processes which run within the context of the bgp, teamd and lldp containers respectively. Regardless of the context under which these processes operate (inside or outside the swss container), they all have the same goals: provide the means to allow connectivity between SONiC applications and SONiC's centralized message infrastructure (redis-engine). These daemons are typically identified by the naming convention being utilized: *syncd.

```
Portsyncd: Listens to port-related netlink events. During boot-up, portsyn

Intfsyncd: Listens to interface-related netlink events and push collected
```

```
Neighsyncd: Listens to neighbor-related netlink events triggered by newly
```

```
Teamsyncd: Previously discussed -- running within teamd docker container.
```

```
Fpmsyncd: Previously discussed -- running within bgp docker container. Aga
```

```
Lldp_syncd: Also previously discussed -- running within lldp docker contai
```

The above processes clearly act as state producers as they inject information into the publisher-subscriber pipeline represented by the redis-engine. But obviously, there must be another set of processes acting as subscribers willing to consume and redistribute all this incoming state. This is precisely the case of the following daemons:

```
Orchagent: The most critical component in the Swss subsystem. Orchagent co
```

```
IntfMgrd: Reacts to state arriving from APPL_DB, CONFIG_DB and STATE_DB to
```

```
VlanMgrd: Reacts to state arriving from APPL_DB, CONFIG_DB and STATE_DB to
```

Syncd container: In a nutshell, syncd's container goal is to provide a mechanism to allow the synchronization of the switch's network state with the switch's actual hardware/ASIC. This includes the initialization, the configuration and the collection of the switch's ASIC current status.

These are the main logical components present in syncd container:

```
Syncd: Process in charge of executing the synchronization logic mentioned
```

```
SAI API: The Switch Abstraction Interface (SAI) defines the API to provide
```

```
ASIC SDK: Hardware vendors are expected to provide a SAI-friendly implemen
```

CLI / sonic-cfggen: SONiC modules in charge of providing CLI functionality and system configuration capabilities.

```
CLI component heavily relies on Python's Click library [5] to provide user
```

```
Sonic-cfggen component is invoked by SONiC's CLI to perform configuration
```

SONiC Subsystems Interactions

This section aims to provide reader with a detailed understanding of the set of interactions that take place among the various SONiC components. To make information more digestible, we have bundled all the system interactions we can envision, attending to the particular state being exchanged by each major functionality. LLDP-state interactions

The following diagram depicts the set of interactions observed during LLDP-state transfer episodes. In this particular example we are iterating through the sequence of steps that take place upon the arrival of an LLDP message carrying state changes.

Sec4Img2

(1) During LLDP container initialization, lldpmgrd subscribes to STATE_DB to get a live-feed of the state of the physical ports in the system -- lldpmgrd's polling cycle runs every 5 seconds. Based on this information, Lldpd (and its network peers), will be kept aware of changes in the system's port-state and any configuration change affecting its operation.

(2) At certain point a new LLDP packet arrives at lldp's socket in kernel space. Kernel's network-stack eventually delivers the associated payload to lldp process.

(3) Lldp parses and digests this new state, which is eventually picked up by lldp_syncd during its execution of lldpctl cli command -- which typically runs every 10 seconds.

(4) Lldp_syncd pushes this new state into APPL_DB, concretely to table LLDP_ENTRY_TABLE.

(5) From this moment on, all entities subscribed to this table should receive a copy of the new state (currently, snmp is the only interested listener). SNMP-state interactions.

As previously mentioned, snmp container hosts both a snmp master-agent (snmpd) as well as a SONiC-specific agentX process (snmp_subagent). This subagent interacts with all those redis databases/tables that provide information from which MIB state can be derived. Concretely, snmp-agent subscribes to the following databases/tables:

```
APPL_DB: PORT_TABLE, LAG_TABLE, LAG_MEMBER_TABLE, LLDP_ENTRY_TABLE
```

```
STATE_DB: *
```

```
COUNTERS_DB: *
```

```
ASIC_DB: ASIC_STATE:SAI_OBJECT_TYPE_FDB*
```

The following diagram depicts a typical interaction among various SONiC components during the time an incoming snmp query is processed by the system.

Sec4Img3

(0) During the initialization of the different MIB subcomponents supported in snmp-subagent process, this one establishes connectivity with the various DBs mentioned above. From this moment on, the state obtained from all these DBs is cached locally within snmp-subagent. This information is refreshed every few seconds (< 60) to ensure that DBs and snmp-subagent are fully in-sync.

(1) A snmp query arrives at snmp's socket in kernel space. Kernel's network-stack delivers the packet to snmpd process.

(2) The snmp message is parsed and an associated request is sent towards SONiC's agentX subagent (i.e. sonic_ax_impl).

(3) Snmp-subagent serves the query out of the state cached in its local data-structures, and sends the information back to snmpd process.

(4) Snmpd eventually sends a reply back to the originator through the usual socket interface. Routing-state interactions.

In this section we will iterate through the sequence of steps that take place in SONiC to process a new route received from an eBGP peer. We will assume that this session is already established and that we are learning a new route that makes use of a directly connected peer as its next-hop.

The following figure displays the elements involved in this process. Notice that I'm deliberately obviating details that are not relevant to this SONiC's architectural description.

Sec4Img4

(0) During BGP container initialization, zebra connects to fpmsyncd through a regular TCP socket. In stable/non-transient conditions, the routing tables held within zebra, the linux kernel, APPL_DB and ASIC_DB are expected to be fully consistent/equivalent.

(1) A new TCP packet arrives at bgp's socket in kernel space. Kernel's network-stack eventually delivers the associated payload to bgpd process.

(2) Bgpd parses the new packet, process the bgp-update and notifies zebra of the existence of this new prefix and its associated protocol next-hop.

(3) Upon determination by zebra of the feasibility/reachability of this prefix (e.g. existing forwarding nh), zebra generates a route-netlink message to inject this new state in kernel.

(4) Zebra makes use of the FPM interface to deliver this netlink-route message to fpmsyncd.

(5) Fpmsyncd processes the netlink message and pushes this state into APPL_DB.

(6) Being orchagentd an APPL_DB subscriber, it will receive the content of the information previously pushed to APPL_DB.

(7) After processing the received information, orchagentd will invoke sairedis APIs to inject the route information into ASIC_DB.

(8) Being syncd an ASIC_DB subscriber, it will receive the new state generated by orchagentd.

(9) Syncd will process the information and invoke SAI APIs to inject this state into the corresponding asic-driver.

(10) New route is finally pushed to hardware. Port-state interactions.

This section describes the system interactions that take place during the transference of port-related information. Taking into account the key-role that portsyncd plays, as well as the dependencies that it imposes in other SONiC subsystems, we start this section by covering its initialization process.

The goal of this exercise is twofold. Firstly, we are exposing the multiple components in the system that are interested in either producing or consuming port-related information. Secondly, we are taking the reader through a graphical example of how the STATE_DB is used in the system, and how different applications rely on its information for their internal operations.

Sec4Img5

(0) During initialization, portsyncd establishes communication channels with the main databases in the redis-engine. Portsyncd declares its intention to act as a publisher towards APPL_DB and STATE_DB, and as a subscriber for CONFIG_DB. Likewise, portsyncd also subscribes to the system's netlink channel responsible for carrying port/link-state information.

(1) Portsyncd commences by parsing the port-configuration file (port_config.ini) associated to the hardware-profile/sku being utilized in the system (refer to configuration section for more details). Port-related information such as lanes, interface name, interface alias, speed, etc., is transmitted through this channel on its way to APPL_DB.

(2) Orchagent hears about all this new state but will defer acting on it till portsyncd notifies that it is fully done parsing port_config.ini information. Once this happens, orchagent will proceed with the initialization of the corresponding port interfaces in hardware/kernel. Orchagent invokes sairedis APIs to deliver this request to syncd through the usual ASIC_DB interface.

(3) Syncd receives this new request through ASIC_DB and prepares to invoke the SAI APIs required to satisfy Orchagent's request.

(4) Syncd makes use of SAI APIs + ASIC SDK to create kernel host-interfaces associated to the physical ports being initialized.

(5) Previous step will generate a netlink message that will be received by portsyncd. Upon arrival to portsyncd of the messages associated to all the ports previously parsed from port_config.ini (in step 1), portsyncd will proceed to declare the 'initialization' process completed.

(6) As part of the previous step, portsyncd writes a record-entry into STATE_DB corresponding to each of the ports that were successfully initialized.

(7) From this moment on, applications previously subscribed to STATE_DB content, will receive a notification to allow these ones to start making use of the ports they are relying on. In other words, if no valid entry is found in STATE_DB for a particular port, no application will be able to make use of it.

NOTE : As of today, these are the applications actively listening to the changes in STATE_DB: teamsyncd, intfmgrd, vlanmgrd and lldpmgr. We will cover all these components in subsequent sections -- lldpmgr has been already tackled above.

Now, let's iterate through the sequence of steps that take place when a physical port goes down:

Sec4Img6

(0) As previously mentioned in the overview section, syncd performs both as a publisher and as a subscriber within the context of ASIC_DB. The 'subscriber' mode is clearly justified by the need for syncd to receive state from the north-bound applications, as has been the case for all the module interactions seen so far. The 'publisher' mode is required to allow syncd to notify higher-level components of the arrival of hardware-spawned events.

(1) Upon detection of the loss-of-carrier by the corresponding ASIC's optical module, a notification is sent towards the associated driver, which in turn delivers this information to syncd.

(2) Syncd invokes the proper notification-handler and sends the port-down event towards ASIC_DB.

(3) Orchagent makes use of its notification-thread (exclusively dedicated to this task) to collect the new state from ASIC_DB, and executes the 'port-state-change' handler to:

a. Generate an update to APPL_DB to alert applications relying on this state for their operation (e.g. CLI -- "show interface status").

b. Invoke sairedis APIs to alert syncd of the need to update the kernel state associated to the host-interface of the port being brought down. Again, orchagent delivers this request to syncd through the usual ASIC_DB interface.

(4) Syncd receives this new request through ASIC_DB and prepares to invoke the SAI APIs required to satisfy orchagent's request.

(5) Syncd makes use of SAI APIs + ASIC SDK to update the kernel with the latest operational state (DOWN) of the affected host-interface.

(6) A netlink message associated with the previous step is received at portsyncd, which is silently discarded as all SONiC components are by now fully aware of the port-down event. Interface-state interactions.

TBD Neighbor-state interactions.

TBD LAG-Interface-state interactions.

TBD Configuration-state interactions.

TBD