XFLOW RESEARCH

SONiC- SWSS

April 2022

Revision History

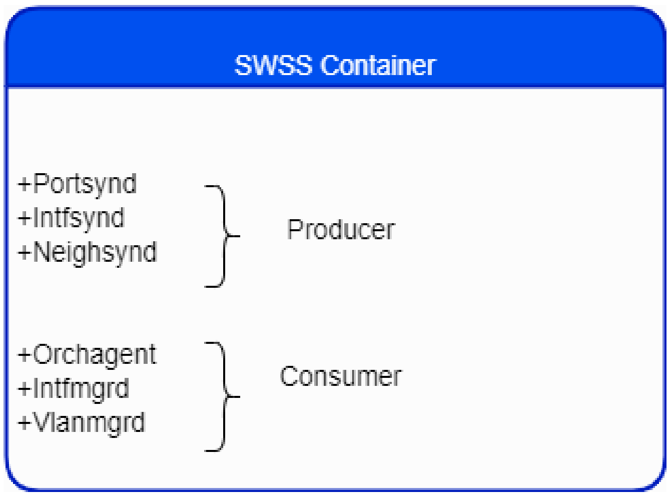
Revision No.	Description	Editor	Date
1.0	SONIC-SWSS	Rida Hanif	Apr 21, 2022

Table of Contents

SONIC SWSS Container

SWSS Container: The Switch State Service (SwSS) container comprises a collection of tools to allow an effective communication among all SONiC modules. If the database container excels 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.

SWSS Goals: The goal is to provide 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.



SWSS Container Daemons:

SWSS Container has following daemons:

- **Portsyncd:** Listens to port-related netlink events. During boot-up, portsyncd obtains physical-port information by parsing system's hardware-profile config files. In all these cases, portsyncd ends up pushing all the collected state into APPL_DB. Attributes such as port-speeds, lanes and mtu are transferred through this channel. Portsyncd also injects state into STATE_DB.

- **Intfsyncd:** Listens to interface-related netlink events and pushes collected state into APPL_DB. Attributes such as new/changed ip-addresses associated with an interface are handled by this process.

- **Neighsyncd:** Listens to neighbor-related netlink events triggered by newly discovered neighbors as a result of ARP processing. Attributes such as the mac-address and neighbor's address-family are handled by this daemon. This state will be eventually used to build the adjacency-table required in the data-plane for L2-rewrite purposes. Once again, all collected state ends up being transferred to APPL_DB.

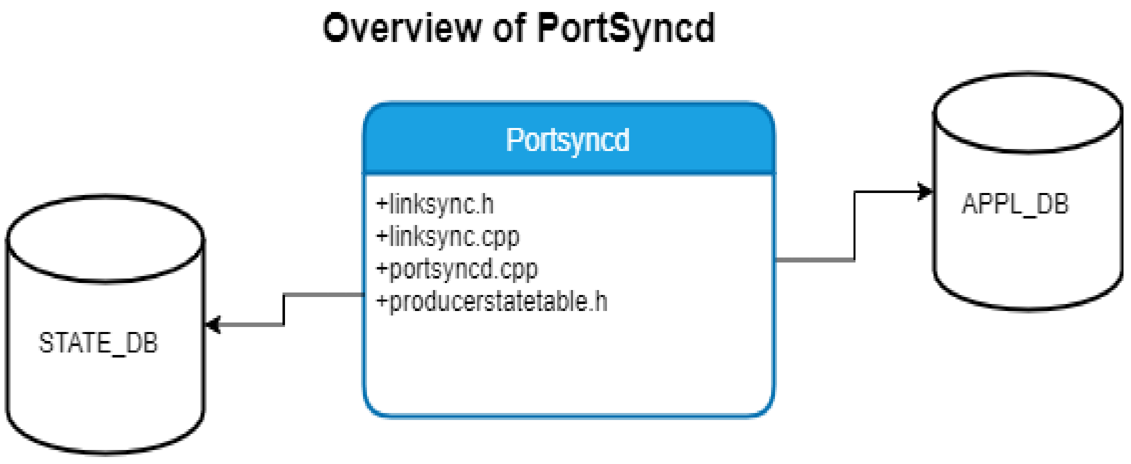
The above processes clearly act as state producers as they inject information into the publisher-subscriber pipeline represented by the redis-engine. But obviously, there must be another set of processes acting as subscribers willing to consume and redistribute all this incoming state. This is precisely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

- **Orchagent:** The most critical component in the Swss subsystem. Orchagent contains logic to extract all the relevant state injected by syncd daemons, process and message this information accordingly, and finally push it towards its south-bound interface. This south-bound interface is yet again another database within the redis-engine (ASIC_DB), so as we can see, Orchagent operates both as a consumer (for example for state coming from APPL_DB), and also as a producer (for state being pushed into ASIC_DB).

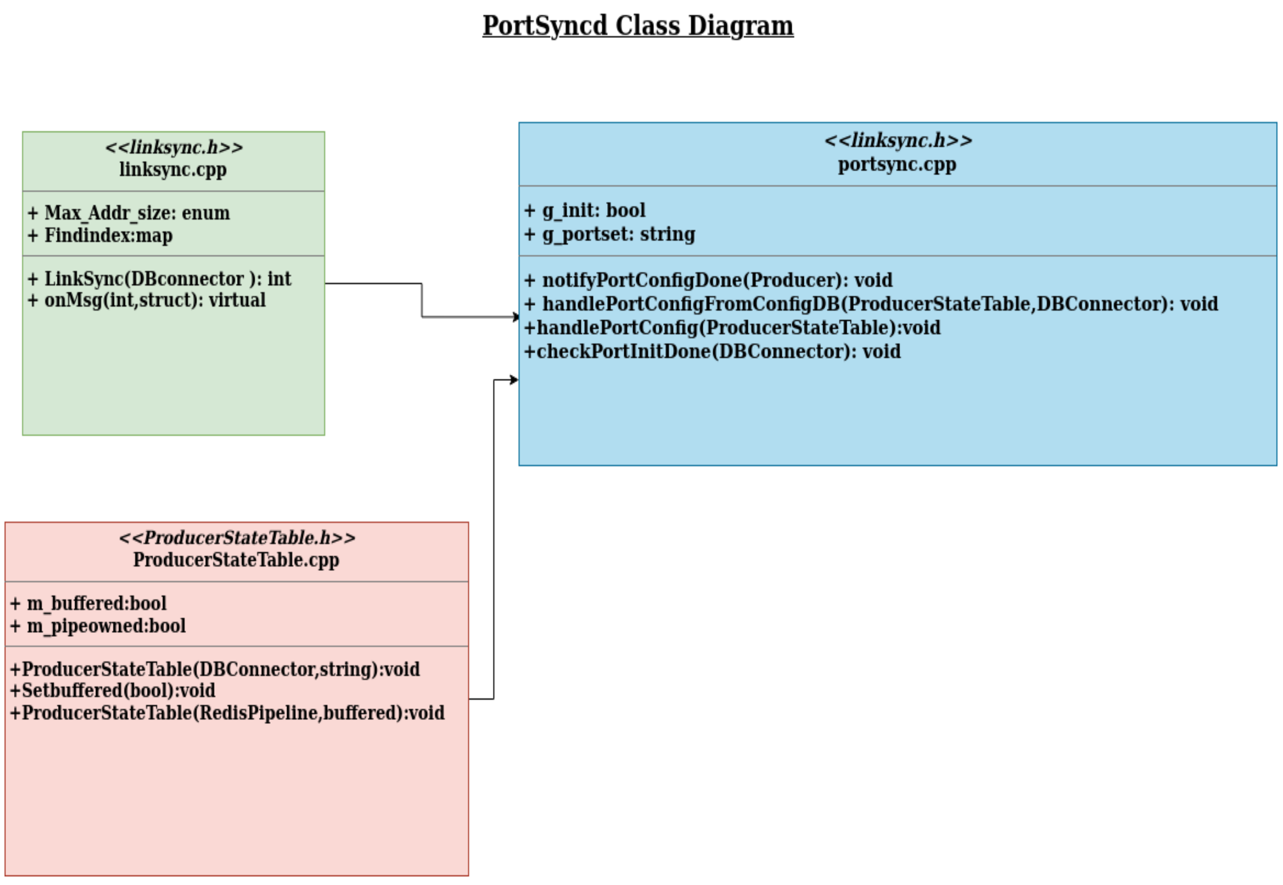
- **IntfMgrd:** Reacts to state arriving from APPL_DB, CONFIG_DB and STATE_DB to configure interfaces in the linux kernel. This step is only accomplished if there is no conflicting or inconsistent state within any of the databases being monitored.

- **VlanMgrd:** Reacts to state arriving from APPL_DB, CONFIG_DB and STATE_DB to configure vlan-interfaces in the linux kernel. As in IntfMgrd's case, this step will be only attempted if there is no dependent state/conditions being unmet.

SWSS – PortSyncd



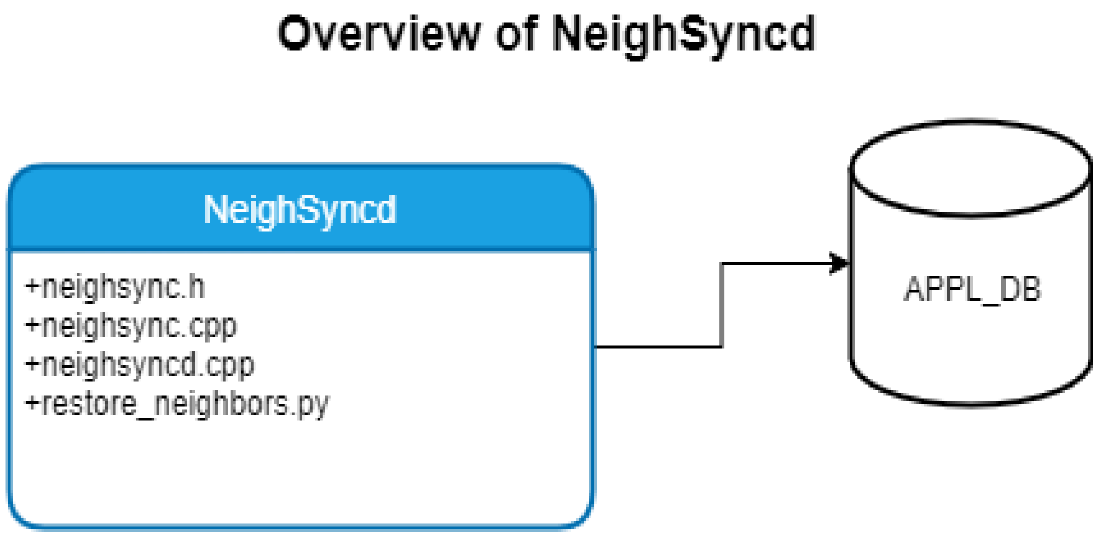
SWSS – PortSyncd Class Diagram


PortSyncd Class Diagram
<<linksync.h>>
linksync.cpp
+ Max_Addr_size: enum
+ Findindex:map
+ LinkSync(DBconnector): int
+ onMsg(int,struct): virtual
<<linksync.h>>
portsync.cpp
+ g_init: bool
+ g_portset: string
+ notifyPortConfigDone(Producer): void
+ handlePortConfigFromConfigDB(ProducerStateTable,DBConnector): void
+handlePortConfig(ProducerStateTable):void
+checkPortInitDone(DBConnector): void
<<ProducerStateTable.h>>
ProducerStateTable.cpp
+ m_buffered:bool
+ m_pipeowned:bool
+ProducerStateTable(DBConnector,string):void
+Setbuffered(bool):void
+ProducerStateTable(RedisPipeline,buffered):void

PortSynd Functionalities:

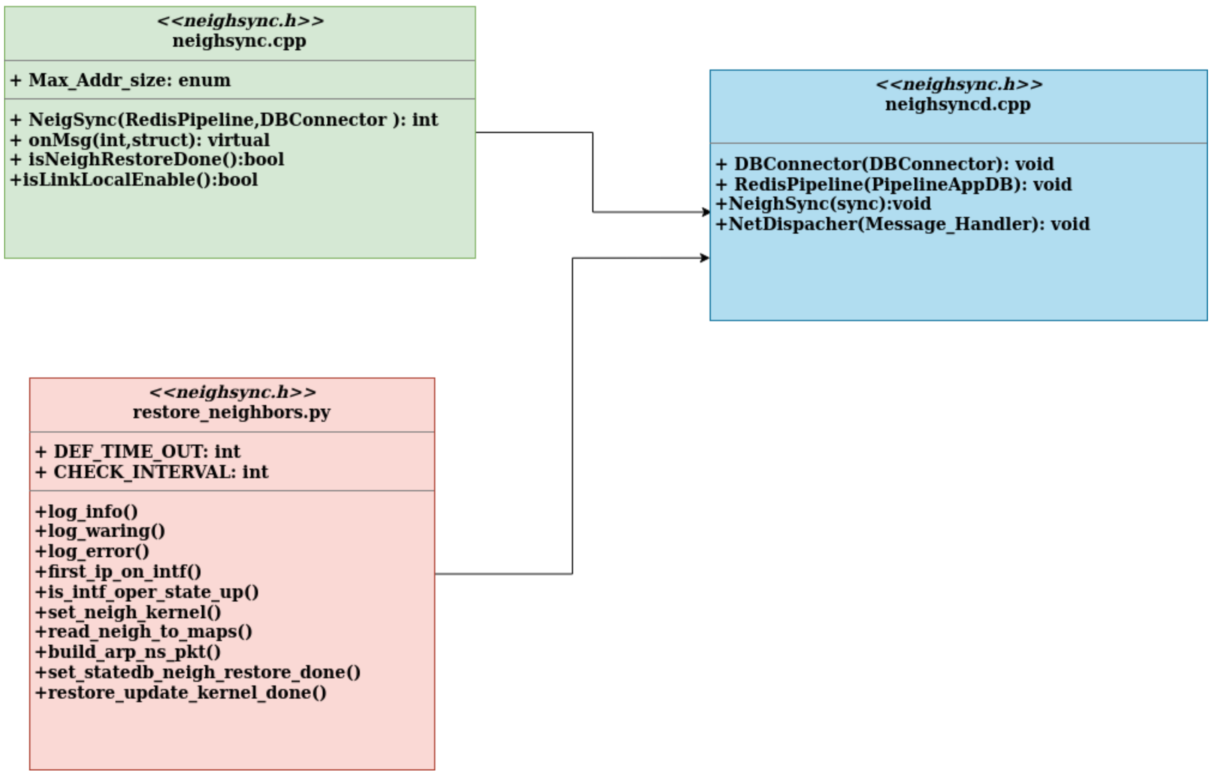
- Responsible for physical port related information and its synchronization with Redis Database.
- It contains a ProducerStateTable which manages records of available ports and creates their state tables.
- This ProducerStateTable is used by the LinKSyncd file which manages the connectivity of available ports with Database for establishing a stable and synchronized link.
- Both these files are used by *PortSyncd.cpp* to check, handle and notify port speeds,lanes and mtu information to APPL_DB and STATE_DB.

SWSS – NeighSyncd



SWSS – NeighSyncd Class Diagram

NeighSyncd Class Diagram



NeighSyncd Functionalities:

- Responsible for neighbors related activities and their record management in the data-plane for L2-rewrite purposes.
- It contains *neighsyn.cpp* to discover and synchronize all the neighbors and maintain a neighbors_table.
- Once all the neighbors are synchronized it creates a *neighsyncd.cpp* file. If the system is warmstart it reads the neighbor table to cache the map and update the neighbor_table.
- The *restore_neigbors.py* is used for restoring the neighbor table into kernel during system warm reboot The script is started by supervisord in swss docker when the docker is started. It does not do anything in case neither system nor swss warm restart is enabled. In case swss warm restart enabled only, it sets the stateDB flag so neighsyncd can continue the reconciliation process.

- In case system warm reboot is enabled, it will try to restore the neighbor table into kernel through netlink API calls and update the neighbor table in kernel by sending arp/ns request so all neighbor entries, then it sets the stateDB flag for neighsyncd to continue the reconciliation process.