SONiC MGMT Repository

Complete Guide

Table of Contents

Introduction

This repository contains code for SONiC testbed deployment and setup, SONiC testing, and test report processing.

Ansible is the main tool powering all the tasks for SONiC testing. The tasks include:

- Deploy and setup testbed
- Interact with various devices in testbed in Ansible playbooks (legacy) and PyTest scripts.

Originally, all the tests were written in Ansible playbooks. In 2019, PyTest was first introduced to replace the Ansible playbook-based tests. Since then, new tests have all been written using PyTest scripts. Existing Ansible playbook tests have also been converted to PyTest gradually. Currently, only PyTest-based new tests are accepted.

Although testing functionality is now fulfilled using PyTest, Ansible is still the core tool powering SONiC testing. Under the hood, all the PyTest-based test scripts use Ansible to interact with various devices in the testbed. Other than testing, Ansible playbooks are now mainly used for testbed deployment and configuration.

Repository Structure

The SONiC MGMT repository comprises of a total of five folders:

- **ansible**
- **docs**
- **spytest**
- **test_reporting**
- **tests**

sonic-mgmt/ansible

This folder contains all the files and scripts required to deploy SONiC devices and MGMT testing topologies (which will be discussed later in this document), as well as legacy support to run tests via Ansible playbooks. Ansible in the SONiC management repository serves three purposes:

1. Deploy SONiC (via Ansible playbooks and XML files)
2. Deploy management testing topologies
3. SONiC testing (legacy)

The **ansible** folder contains scripts specifically designed to automate the deployment of management testing topologies. It does this by creating a Docker container which acts as the central control through which Ansible playbooks and minigraph (XML) configurations are deployed to the SONiC testbed via automated shell scripts. In e the case of virtual test beds, it also holds scripts that set up topology components as KVM virtual machines and create the necessary topology network through Linux bridging.

sonic-mgmt/docs

This folder contains all documentation concerning the management repository. It holds various guides and instructions, API documentation, information on how to communicate with topology components (eg. DUT hosts, PTF hosts, etc.), information on standard testbeds, and test plan documents corresponding to all test cases created for the testing of SONiC.

sonic-mgmt/spytest

SPyTest is a test automation framework designed to validate SONiC. It utilizes PyTest as its foundation and leverages various open-source Python packages for tasks such as device access, CLI output parsing, and traffic generation.

** Currently, SPyTest acts as a separate testing framework inside the SONiC management framework being an alternative to testing through PyTest + Ansible. For example, SPyTest offers an alternative method to deploy testbeds using its own testbed files, while Ansible uses playbooks and minigraphs. Similarly, Ansible and SPyTest both offer their own methods of manipulating topology components (DUTs, neighbors, T-Gens). As of the writing of this document, most of the SONiC community is focused on carrying out testing via PyTest + Ansible, although some tests have been written using SPyTest which are available in the **sonic-mgmt/spytest/tests/** folder. **

The components of SPyTest include:

- **Framework:**

  This forms the core of the automation framework, providing the necessary infrastructure and functionalities to author test scripts, and execute and generate test reports.

- **TGen APIs:**

  Traffic Generator APIs enable the generation and control of network traffic for testing purposes. It allows users to configure and manipulate traffic patterns, perform packet-level operations, and measure network performance.

- **Feature APIs:**

  The Feature APIs provide a set of functions and methods that allow testers to interact with specific features and functionalities of SONiC. This component simplifies the testing process by providing a higher-level abstraction layer for validating individual features.

- **Utility APIs:**

  The Utility APIs offer a collection of utility functions that assist in various testing operations.

- **TextFSM Templates:**

  TextFSM is a framework for parsing and extracting structured data from unstructured text outputs, such as command line outputs. SPyTest utilizes TextFSM templates to extract relevant information from the CLI outputs of devices under test.

- **Test Scripts:**

  Test scripts are the actual test cases written using the SPyTest framework. These scripts combine the functionalities provided by the components mentioned above to define the test scenarios and validate the behavior of SONiC.

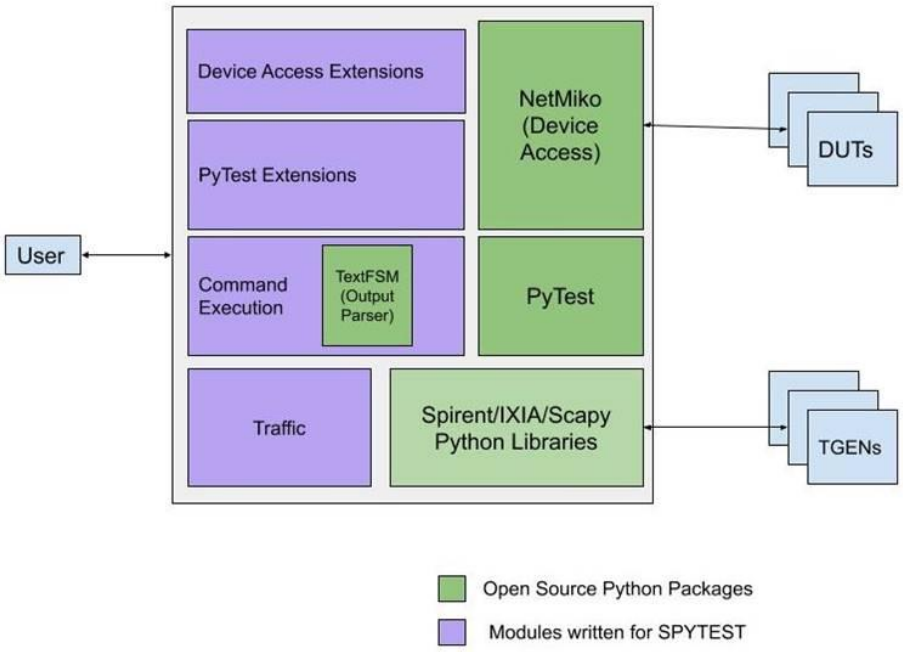

Fig 1: SPyTest Top-Level Design

***** This statement is as per my initial understanding, and may or may not be correct.***

sonic-mgmt/test_reporting

This component of the management repository holds relevant scripts and tools used for parsing, uploading, and processing JUnit XML test reports generated by PyTest. Processed test data is also uploaded to Kusto for querying.

JUnit XML is a standard format for reporting test results used by many testing frameworks and tools, including PyTest and SPyTest. It is an XML-based format that stores information about the execution of a test suite, including the name of each test, whether it passed or failed, and any error messages that were generated. The **test_reporting** component of the management repository holds scripts that parse these generated test reports and extract the relevant data.

The extracted data from JUnit XML test reports can also be uploaded to a specified Kusto or Azure Data Explorer (ADX) cluster, which are Microsoft database querying platforms.

sonic-mgmt/tests

This directory of the management repository holds all the infrastructure code and test scripts used for PyTest + Ansible-based testing.

Most of the infrastructure code held in this directory is situated in **sonic-mgmt/tests/common**. Here, code is written allowing test scripts to manipulate topology components via Ansible connections. It does this by implementing each component as an Ansible host class. The base class is defined in **…/common/devices/base.py** which is inherited by separate devices such as DUT hosts, PTF hosts, and other device classes. These device classes are instantiated as objects along with other standard Ansible variables (eg. rand_one_dut_hostname) via fixtures defined in **sonic-mgmt/tests/conftest.py**. Device objects have a set of attributes and methods that PyTest test scripts can use by importing the relevant device object.

The “common” directory also holds other resources such as helpers, fixtures, plugins, and other platform-related codes. Details of the directory can be found below.

Directory	Subdirectory	Description
sonic-mgmt/tests/common/	**cache/**	Code for caching repeatedly used facts from topology devices in a testing session.
	connections/	Classes for various console connection objects to topology devices (eg. SSH, Telnet, etc.).
	devices/	Classes for device objects of various topology device hosts (Ansible) in a test suite (eg. DUT Hosts, PTF Hosts, SONiC/Arista/Cisco Neighbors, etc.). A base Ansible host class is defined in **./base.py** and all other host classes inherit this class.
	dualtor/	Common functions and PyTest fixtures used for Dual TOR testing. Test cases that use these functions and fixtures can be found at **sonic-mgmt/tests/dualtor_io/** and **sonic-mgmt/tests/dualtor/** directories.
	fixtures/	PyTest fixtures and utility functions to initialize and test configurations in various topology components (eg.

		DUT, PTF, etc.).
	helpers/	Common helper functions to be used in various setups/teardowns and other functionalities of test cases (eg. build_icmp_packet function)
	ixia/	Necessary fixtures and helper functions for running test cases with Ixia devices and IxNetwork.
	multibranch/	Classes and fixtures to initialize SONiC auto-techsupport (command that records device state information for debugging purposes) and then tear it down after testing.
	pkt_filter/	Code for “filter packet in buffer” feature for finding packets inside PTF buffer. The code contains a class for the FilterPktBuffer object used to find packets.
	platform/	Utility functions for platform-specific tasks such as PMON daemon check, processes, SSH, interfaces, fanout switch, and transceiver management.
	plugins/	Code for common utilities used by test case codes such as the **loganalyzer**, **dut_monitor**, **ptfadapter**, etc.
	pytest_argus/	Contains a document outlining instructions on how to install and embed the PyTest Argus plugin into the sonic-mgmt Docker container.
	snappi_tests/	Contains helper functions, modules, and fixtures required to run **snappi**-based tests. Test scripts written in snappi, an auto-generated Python SDK, can be executed against any traffic generator conforming to Open Traffic Generator API (eg. Ixia).
	storage_backend/	Contains a fixture providing the ability to skip test cases in the test

		module the topology is of storage backend type.
	system_utils/	Utilities for interacting with Docker on the DUT host.
	templates/	**.j2** templates for ACL rule and PFC Storm configurations for various device types.

Other than infrastructure code, this directory also holds PyTest test scripts themselves which are organised into folders corresponding to the feature/domain they are testing. Each domain folder typically contains Python code files that fall under one of the following three categories:

1. **“conftest.py” files:**

   Files containing PyTest fixtures used for the setup/teardown of the test cases for that feature/domain.

2. **Helper files:**

   Files containing common helper functions that are repetitively used by test cases for that feature/domain.

3. **Test files:**

   Files containing the actual test cases themselves. As per the standard functionality of PyTest, each test case function name must start with “test_” (eg. “test_ospf_neighborship”).

Testbed Overview

The SONiC management framework automates the setup of SONiC testbeds on which to run PyTest test cases. This section describes the ideology behind testbeds employed by the management framework.

Physical Topologies

In a data center, typically there are rows of racks. Each rack has servers installed. Different tiers of network devices forming a modified clos network serve the traffic between devices in and between racks. The network device tiers are:

- Tier0
- Tier1
- Tier2

These tiers of devices can be further described by their respective upstream and downstream devices.

Tiers	Downstream Devices	Upstream Devices
T0	Servers	T1 devices
T1	T0 devices	T2 devices
T2	T1 devices	Regional Gateways (T3 devices)

The purpose of tests in the SONiC management framework is to test features and functions of SONiC running as either T0, T1, or T2 network devices. These specific devices are referred to as the **Devices Under Test (DUTs)**. Therefore, testbeds must simulate the topologies of T0, T1, and T2 topologies, which requires **neighboring devices** and **traffic generators (PTF)** alongside DUTs. To increase efficiency, these components are implemented as KVM VMs (neighbors) and Docker containers (PTF). To meet these requirements, testbed topologies are designed as below:

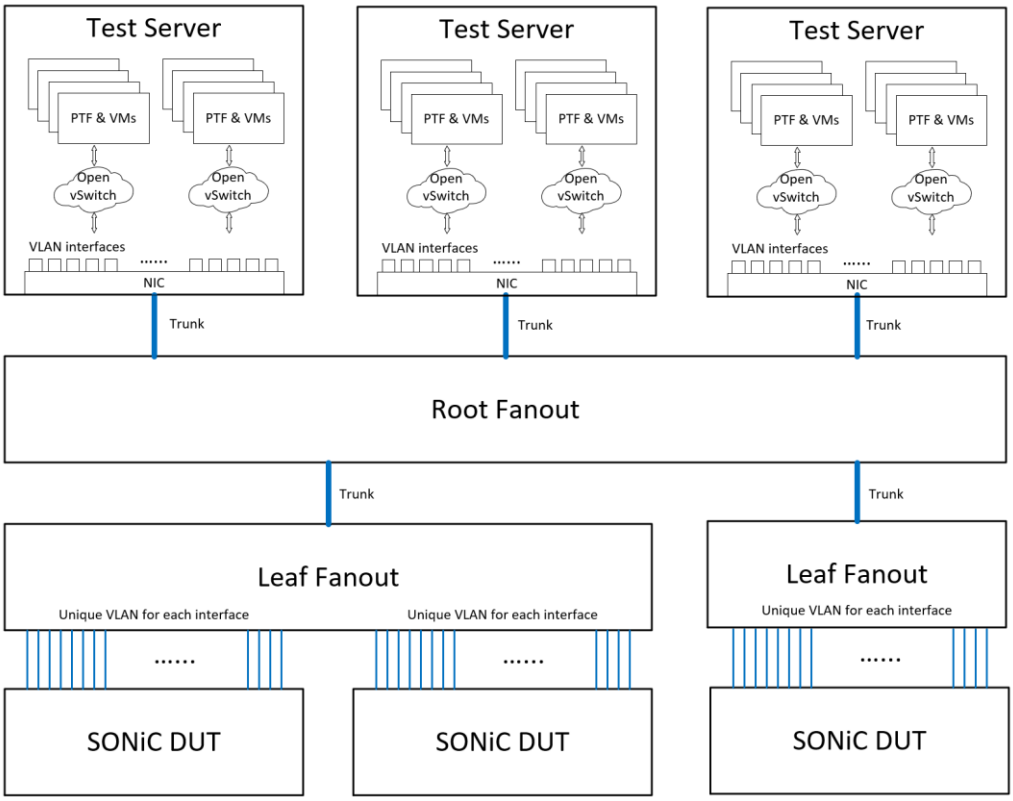

Fig. 2: General testbed topology

The management framework also allows for SONiC DUTs to be implemented as virtual switches (virtual testbeds), in which case the DUTs are also situated inside test servers as KVM VMs. This allows for the entire testbed to be deployed within servers, eliminating the need for any other hardware.

Logical Topologies

Mainly 4 types of testbed topologies can be simulated based on the physical topology design described above:

- T0
- T1
- T2
- PTF

Each of the topologies mentioned above has several variations to them. Details of the logical topologies are defined in **sonic-mgmt/ansible/vars/topo_*.yml** files. Each topology type aims to test SONiC devices (DUTs) as T0/T1/T2 network devices.

Further details on logical topologies and their variations can be found at **sonic-mgmt/docs/testbed/README.testbed.Overview.md**. Furthermore, instructions on how to deploy various testbeds are available in the **sonic-mgmt/docs/testbed/** directory.

Contributing Tests & Test Plans

One can contribute test cases and their corresponding test plans to the SONiC management repository by following the conventions outlined in this section. Contributing to the testing of the management repository consists of two steps:

1. **Adding a Test Plan in sonic-mgmt/docs/testplans/**:

   This is a document outlining the intended test, the testbed required to run the test, and the steps involved in the test that will be implemented in the test code. Typically, test plan documents are added as **.md** files and follow the following basic structure:

   - OVERVIEW
     - Scope of the test
     - Testbed(s) compatible with the test
     - Relevant SONiC CLI commands for the feature being tested (on the DUT)
   - TEST STRUCTURE
     - Setup configurations required before the test is run
     - Testbed setup (eg. T0, T1, etc.)
   - TEST CASES
     - Test objectives
     - Test steps

2. **Adding Test Code in sonic-mgmt/tests/*/**:

   Corresponding to the test plan, test case code is added to the relevant feature folder (eg. bgp, ospf, acl, etc.). Test code is added in the form of a **.py** file containing test case functions written using PyTest. Setup and teardown phases of each test are handled using PyTest fixtures, which are typically located inside a separate **conftest.py** file inside the same directory. Additionally, any repeated helper functions may be added separately to a **helper file** (eg. bgp_helpers.py) located in the same directory. Further details of PyTest fixtures and the setup/teardown phase of tests are given below.

Writing SONiC Test Code using PyTest

PyTest is a popular testing framework in Python that provides a simple and efficient way to write test cases. It handles test cases as Python functions, the names of which **must start with "test_*"** (eg. test_ospf_neighborship). This is because PyTest automatically detects all functions starting with "test_*" and treats/runs them as test case functions.

Other than test cases, all tests require a "setup" and "teardown" phase. During the setup phase of any test, initially required configurations are made on various topology components (eg. DUT, PTF, neighbors, etc.). Once a test is completed, there must be a teardown phase where all changes made in the topology during the test are reversed and the topology is brought back to its original state, ready for the next test in the PyTest

session. Typically, the teardown phase of most SONiC tests involves running the **config reload** command to bring the DUT and neighboring devices back to their original states.

PyTest provides a fixture mechanism to handle the setup and teardown phases of tests. Fixtures are functions that PyTest runs before each test to set up any necessary state and runs after each test to clean up. A fixture typically follows the structure shown below.

```python
import pytest

@pytest.fixture(scope="module", autouse=TRUE)
def setup_teardown():
    print("Setting up...")
    yield
    print("Tearing down...")
```

In this example, the **setup_teardown** fixture is defined using the **@pytest.fixture** decorator. The **yield** keyword is used to pause the execution of the fixture, so that any code before the "yield" statement is executed before the test (setup), and any code after the "yield" statement is executed after the test (teardown).

Fixtures can also be scoped to different levels. This can be done using the **scope** parameter in the **@pytest.fixture** decorator. These options determine when the setup and teardown code of a fixture is executed. The available options are:

- **function:**

  This is the **default** scope. In this case, the setup and teardown code of the fixture is executed before and after each test function that uses the fixture.

- **module:**

  In this case, the setup and teardown code of the fixture is executed once before any test functions in the module that uses the fixture, and once after all test functions in the module.

- **session:**

  In this case, the setup and teardown code of the fixture is executed once before any test functions in the test session that uses the fixture, and once after all test functions in the session.

- **class:**

  This scope is used when the fixture is used as a class-level fixture. In this case, the setup and teardown code of the fixture is executed once before any test methods in the class that uses the fixture, and once after all test methods in the class.

Typically, in order for a test case function to use a fixture, the fixture name must be explicitly called as a parameter to the test case function. An alternative to this is the **autouse** parameter in the fixture decorator. The **autouse** parameter in PyTest fixtures is an optional parameter that can be used (set to **TRUE**) to automatically use a fixture in all test

functions within a test module or class, without having to declare it as an argument in each test function explicitly.

In SONiC testing, PyTest fixtures are conventionally written separately inside a **conftest.py** file, which is a special file that PyTest looks for in directories. This file **must** be named "conftest.py" for it to automatically be included in the testing session without having to be explicitly imported by every test case file.