SONiC Configuration and Testing

July 11th, 2017

Maggie Sun

Topics

- SONiC Basic Management
  - Public Repo
  - Configuration and Minigraph
- Testbed Requirement, Topology and Deployment
  - Basic Hardware Requirement
  - Software Repo
  - Testbed Topology
  - Testbed Management and deployment
- Test Cases Execution
- What’s Next

SONiC Repos

- SONiC Management Repo
  - https://github.com/azure/sonic-mgmt
- SONiC Build Image Repo
  - https://github.com/Azure/sonic-buildimage
- SONiC Management Quick Introduction
  - Ansible/Ansible-playbook:
    - Use Ansible and Ansible playbook to manage test lab, testbed, test cases and test run
  - Build Ansible Docker
    - User can build a ‘sonic-mgmt’ docker image from sonic-buildimage repo and use it for SONiC test execution

Configuration and Minigraph

- Minigraph
  - SONiC is using ‘Minigraph’ as the entry to configure SONiC box
    - /etc/sonic/minigraph.xml
  - Configuration automatically generated for SNOiC based on Minigraph
    - sonic-cfggen -m /etc/sonic/minigraph.xml
  - Minigraph Auto/Manual Update
    - /etc/sonic/updategraph.conf
  - Detailed Specification of Minigraph
    - https://github.com/Azure/SONiC/wiki/Configuration-and-Minigraph

SONiC Testbed Hardware

- **Basic Hardware**
  - **Management Server**: Regular Linux Server(Ubuntu) for testbed management and test run
    - One network interface routable to management network
  - **Test Server**: One High Performance Linux Server(Ubuntu) for testbed as traffic generator
    - Minimum memory requirement should be 92G for one T1 testbed, we are using 192G
    - At least one 40G or 100G(based on your testbed speed) interface for traffic generation
    - At least 2 Management interface to can access management network to manage server and VMs
  - **Fanout Switch**:
    - At least one ‘Fanout’ Switch to connect all DUT front panel ports and Server(Minimum 34 for 32 Ports DUT)
    - To connect more DUTs to fanout switches, you could have a ‘Root’ fanout and multiple ‘Leaf’ fanout switches
    - The interface speed best match DUTs and Servers
    - We are using Arista7260 64*40G
  - **More Detailed information:**
    - https://github.com/Azure/sonic-mgmt/blob/master/ansible/README.testbed.md

SONiC Testbed Software

- **Basic Software**
  - **Management Server**: Regular Linux Server(Ubuntu) for testbed management and test run
    - We are using Ubuntu 16.04, have basic Python2.7 and Dev packages installed
    - Recommend to have Docker engine installed in this server and build sonic-mgmt-build from sonic-buildimage to have a SONiC management docker with all the correct dependencies built for running all SONiC management through this docker
    - Install Ansible (2.0.0.2) and run ansible playbook directly for OS also works but not is recommended(Not officially support, you are on your own)

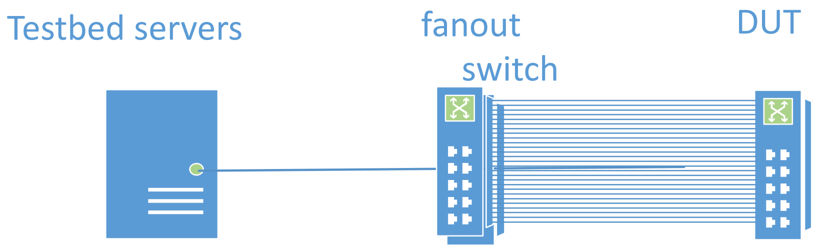
  - **Test Server**: One High Performance Linux Server(Ubuntu) for testbed as traffic generator
    - We are using Ubuntu 16.04
    - Correct drivers for 40G or 100G networks
    - KVM engine to run VMs
    - VMs: we are using Arista vEOS

  - **Fanout Switch**:
    - Any Switches support Lay2 Vlan

Testbed – Physical and Logical

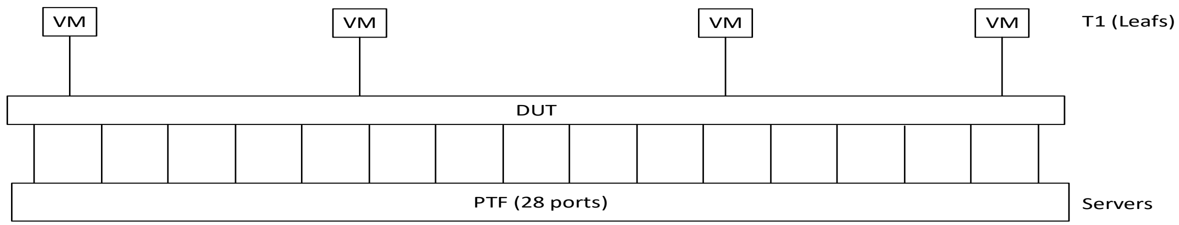
- Testbed Physical Topology

  Physical topology defines/describes how DUT/Server/Switches physical ports cable connections in lab testbed.



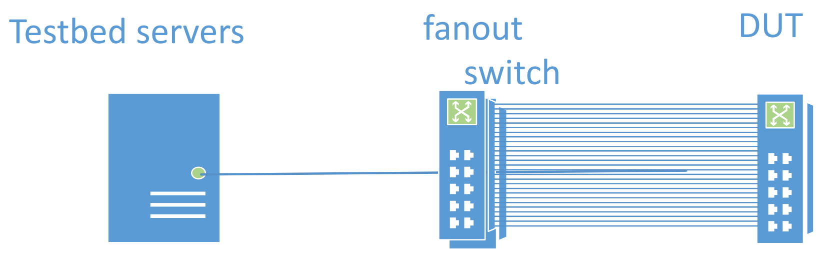
- Testbed Logical Topology

  Logical topology defines how DUT ports connect to VMs in testbed to conduct test



Simplified Testbed Physical Topology

- Very Basic Testbed Physical Topology



- Every DUT port is connected to fanout switch
- Fanout switch connects to testbed servers
- Connections from root fanout switches are 802.1Q trunks
- Any testbed server can access any DUT port by sending a packet with the port vlan tag (fanout switch should have this vlan number enabled on the server trunk)

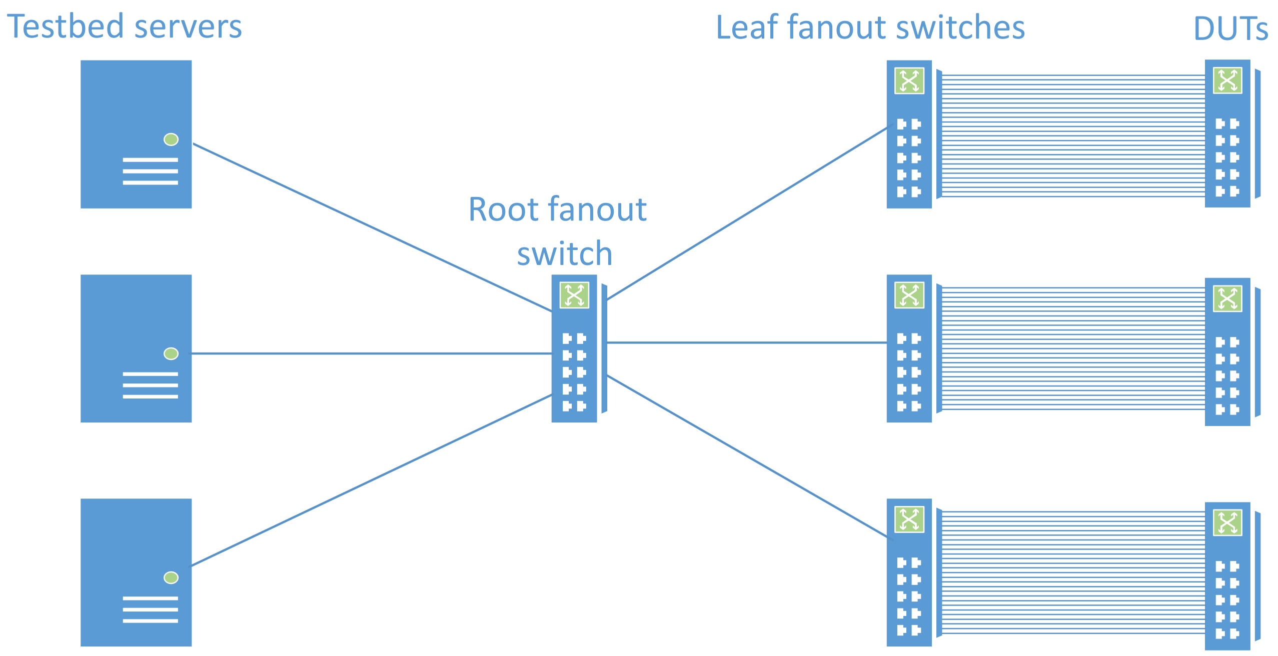
Simplified Physical Testbed Fanout Graph File

- **Fanout Graph File:**
  - **ansible/files/lab_connection_graph.xml**
    - https://github.com/Azure/sonic-mgmt/blob/master/ansible/files/lab_connection_graph.xml

This is the lab graph file for library/conn_graph_facts.py to parse and get all lab fanout switch connections information.

Manually edit this file to Make Fanout Root and Fanout Leaf both point to the only fanout switch.

Testbed Physical Topology



- Every DUT port is connected to one of leaf fanout switches
- Every leaf fanout switch has unique vlan tag for every DUT port
- Root fanout switch connects leaf fanout switches and testbed servers
- Connections from root fanout switches are 802.1Q trunks
- Any testbed server can access any DUT port by sending a packet with the port vlan tag (root fanout switch should have this vlan number enabled on the server trunk)

Physical Testbed Fanout Graph Files

- **Fanout Graph File:**
  - **ansible/files/lab_connection_graph.xml**

    This is the lab graph file for library/conn_graph_facts.py to parse and get all lab fanout switch connections information. Based on ansible_facts from the graph file, you may write Ansible playbooks to deploy fanout switches or run test which requires to know the DUT physical connections to fanout switch
- **Supporting files help to generate fanout graph file**
  - ansible/files/sonic_lab_devices.csv

    Manually created file helps you create lab_connection_graph.xml, list all devices that are physically connected to fanout testbed
  - ansible/files/sonic_lab_links.csv

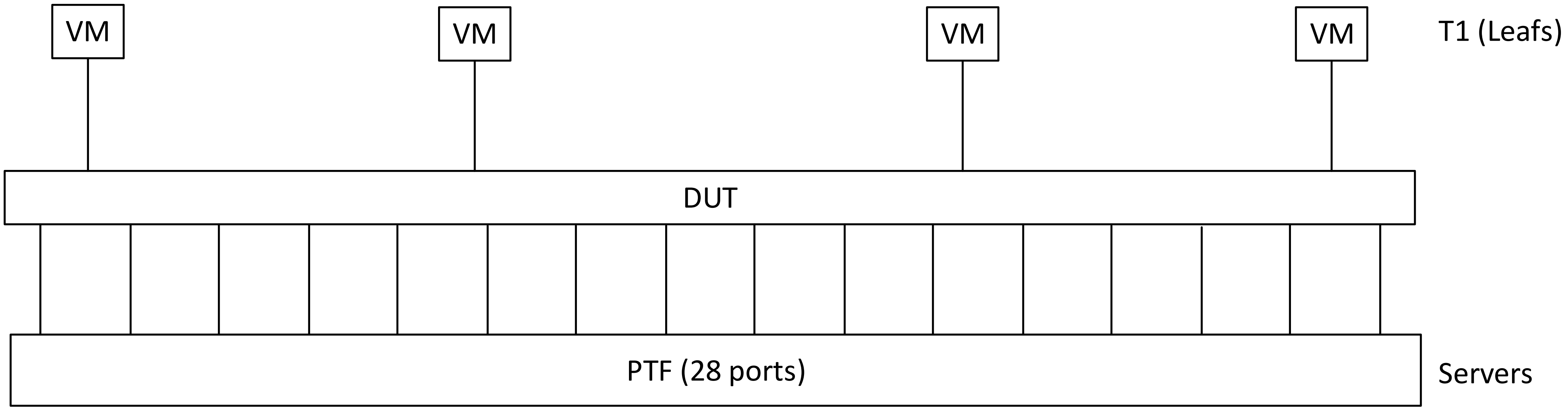
    Manully created file helps you to create lab_connection_graph.xml, list all physical links between DUT, Fanoutleaf and Fanout root switches, servers and vlan configurations for each link
  - ansible/files/creategraph.py

    Python executable helps you generate a lab_connection_graph.xml based on the device file and link file specified above.

Testbed Logical Topology Type

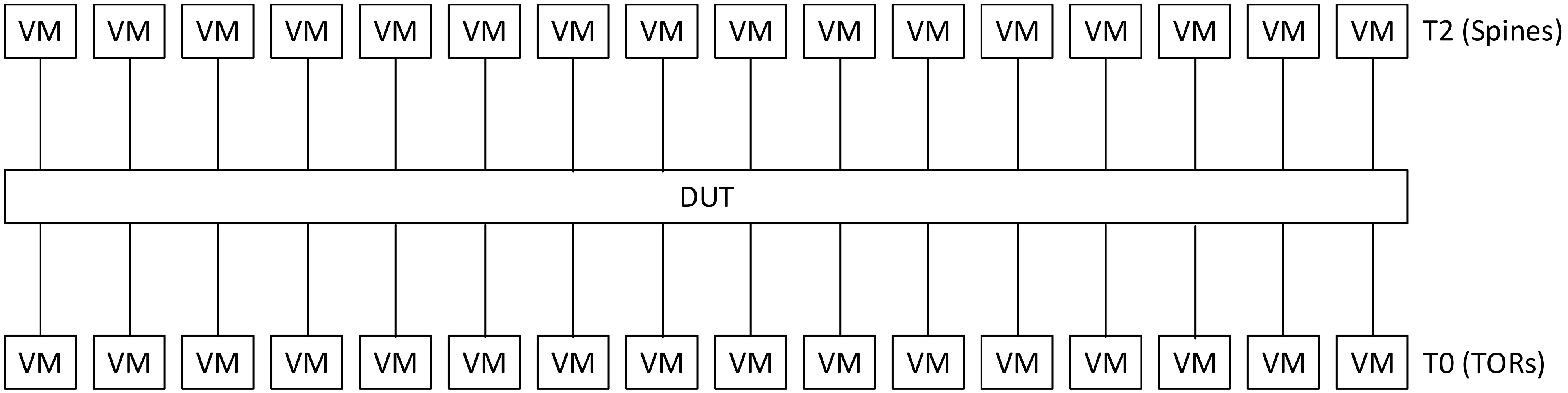
- T0
- T1
- T1-lag
- Ptf32
- Ptf64

Logical Topology T0



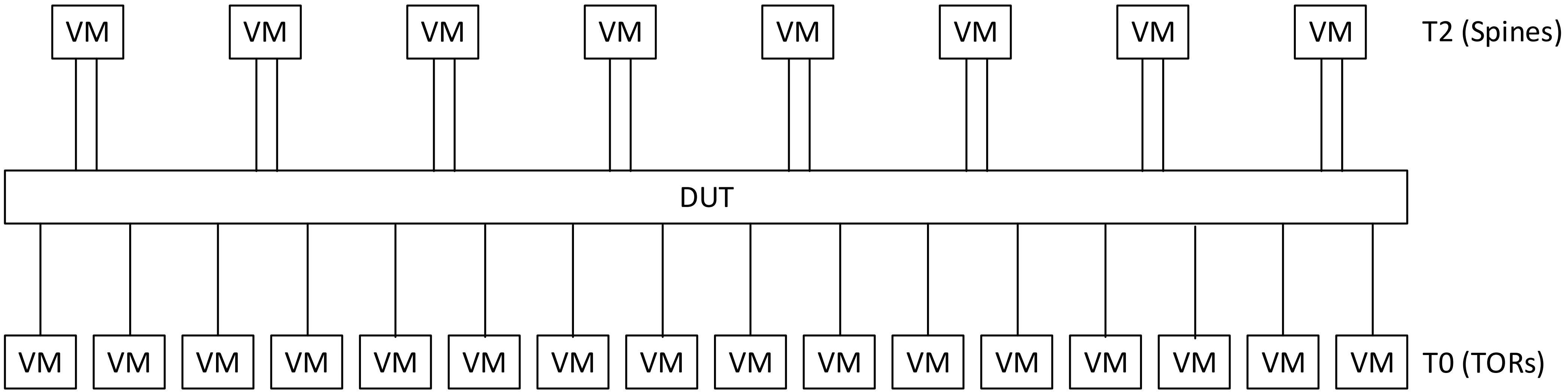
- 4 VMs
- 4 DUT ports are connected to VMs
- PTF container has 4 injected ports and 28 directly connected ports

Logical Topology: T1



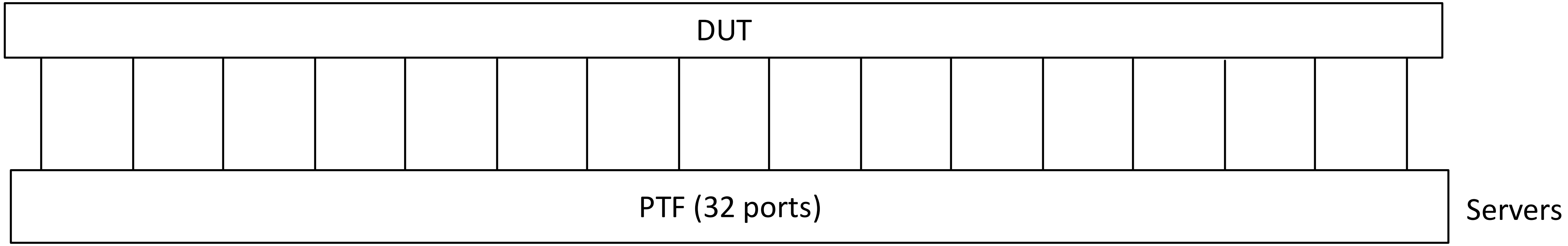
- 32 VMs
- All DUT ports are connected to VMs
- PTF container has injected ports only

Logical topology: T1-lag



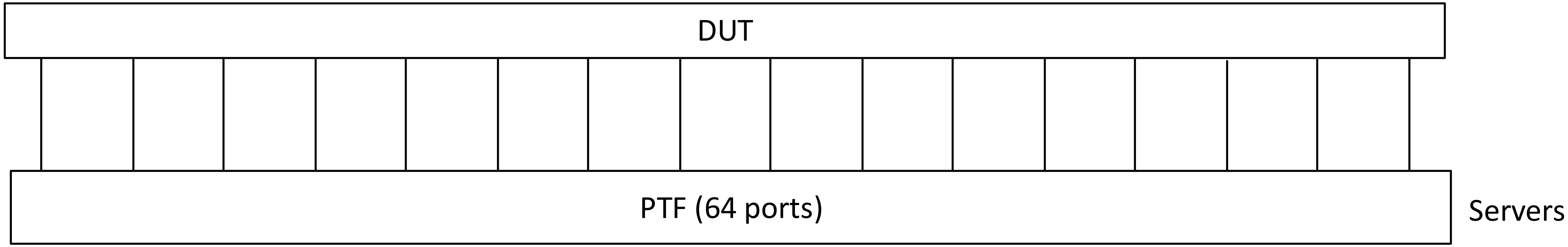
- 24 VMs
- All DUT ports are connected to VMs
- PTF container has injected ports only

Logical topology: ptf32



- 0 VMs
- All DUT ports are directly connected to PTF container
- PTF container has no injected ports

Logical Topology: ptf64



- 0 VMs
- All DUT ports are directly connected to PTF container
- PTF container has no injected ports

Logical Testbed Configuration and Deployment

- Quick Summary
  - Configuration of all testbeds defined in one file: testbed.csv
  - One script to operate all testbeds: testbed-cli.sh
  - Flexible topologies which allows to use vm_set and ptf container as one entity
  - All VM management ip information in one place: veos inventory file
  - ptf container is generalized and used in every topology

Logical Testbed Configuration

- One entry in testbed.csv
- Consist of:
  - physical topology: How ports of VMs and ptf connected to DUT
  - configuration templates for VMs
- Defined in vars/topo_*.yml files
- Current topologies are:
  - t1: 32 VMs + ptf container for injected ports
  - t1-lag: 24 VMs + ptf container for injected ports. 8 VMs has two ports each in LAG
  - ptf32: classic ptf container with 32 ports connected directly to DUT ports
  - ptf64: as ptf32, but with 64 ports
  - t0: 4 VMs + ptf. ptf container has 4 injected ports + 28 directly connected ports

Sample of testbed.csv

uniq-name	testbed-name	topo	ptf_imagename	ptf_mgmt_ip	server	vm_base	DUT	Comment
ptf1-1	ptf1-1	ptf32	docker-ptf	10.0.0.188/24	server_1		str-sw1-8	Jenkins
ptf1-3	ptf1-3	ptf32	docker-ptf	10.0.0.254/24	server_1	VM100	str-sw1-2	User-A
ptf1-4	ptf1-4	ptf32	docker-ptf	10.0.0.185/24	server_1	VM200	str-sw2-4	User-B

- uniq-name - to address row in table
- testbed-name – used in interface names, up to 8 characters
- topo – name of topology
- ptf_imagename – defines ptf image
- ptf_mgmt_ip – ip address for mgmt interface of ptf container
- server – server where the testbed resides
- vm_base – first VM for the testbed. If empty, no VMs are used
- DUT – target dut name
- Comment – any text here

Deployment: testbed-cli.sh

- Maintenance purposes only
  - ./testbed-cli.sh start-vms {server_name} ~./password
    - after a server restarted
  - ./testbed-cli.sh stop-vms {server_name} ~./password
    - before a server restarted
- ./testbed-cli.sh add-topo {topo_name} ~./password
  - create topo with name {topo_name} from testbed.csv
- ./testbed-cli.sh remove-topo {topo_name} ~./password
  - destroy topo with name {topo_name} from testbed.csv
- ./testbed-cli.sh renumber-topo {topo_name} ~./password
  - renumber topo with name {topo_name} from testbed.csv

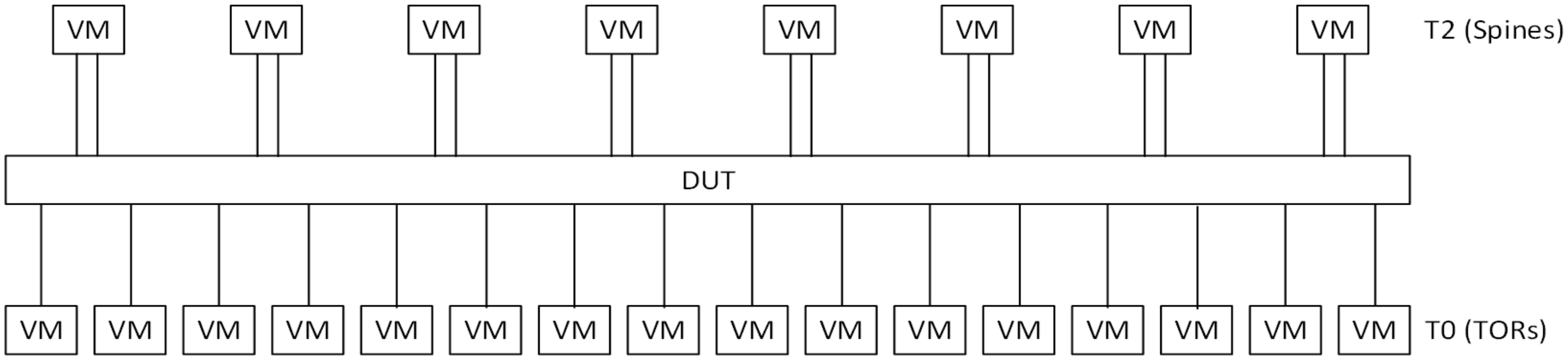
Test cases Execution

- All test cases are in sonic-mgmt repo
  - https://github.com/Azure/sonic-mgmt/blob/master/ansible/roles/test/tasks/sonic.yml
- A testbed needed to be set up before hand. See Testbed for more information. Depending on the test, either a PTF testbed or a VM set testbed might be required.
- SONiC DUT Configuration Minigraph needs to match the testbed specified above.
- To run a test:

  ansible-playbook test_sonic.yml -i lab --limit {DUT_NAME} --tags {**Test Name**} --extra-vars "run_dir=/tmp testbed_type={TESTBED_TYPE} ptf_host={PTF_HOST}"

Test Run Example

- Test case: https://github.com/Azure/sonic-mgmt/blob/master/ansible/roles/test/tasks/fib.yml
- Test case design: https://github.com/Azure/SONiC/wiki/FIB-Scale-Test-Plan
- Test run: ansible-playbook test_sonic.yml -i lab --limit str-msn2700-01 --vault-password-file password.txt --tags fib --extra-vars 'testbed_type=t1 ptf_host=10.250.0.26 ipv6=False'



Next

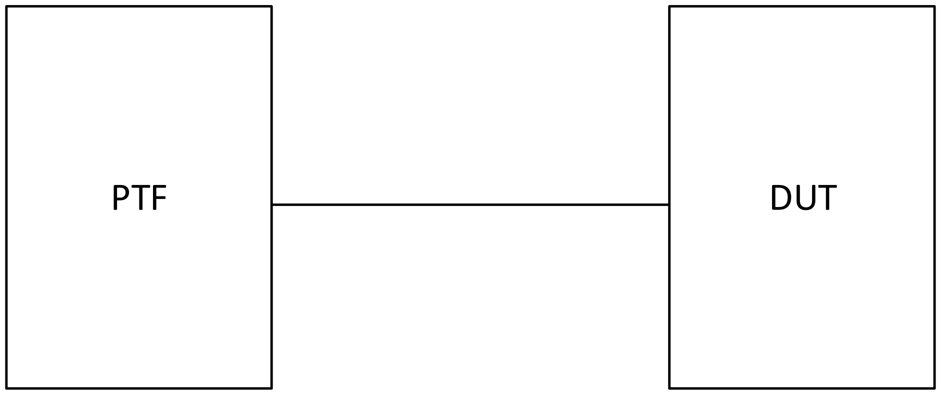
- Add playbook to create configuration minigraph file for each topology
- Tests to be added with new feature

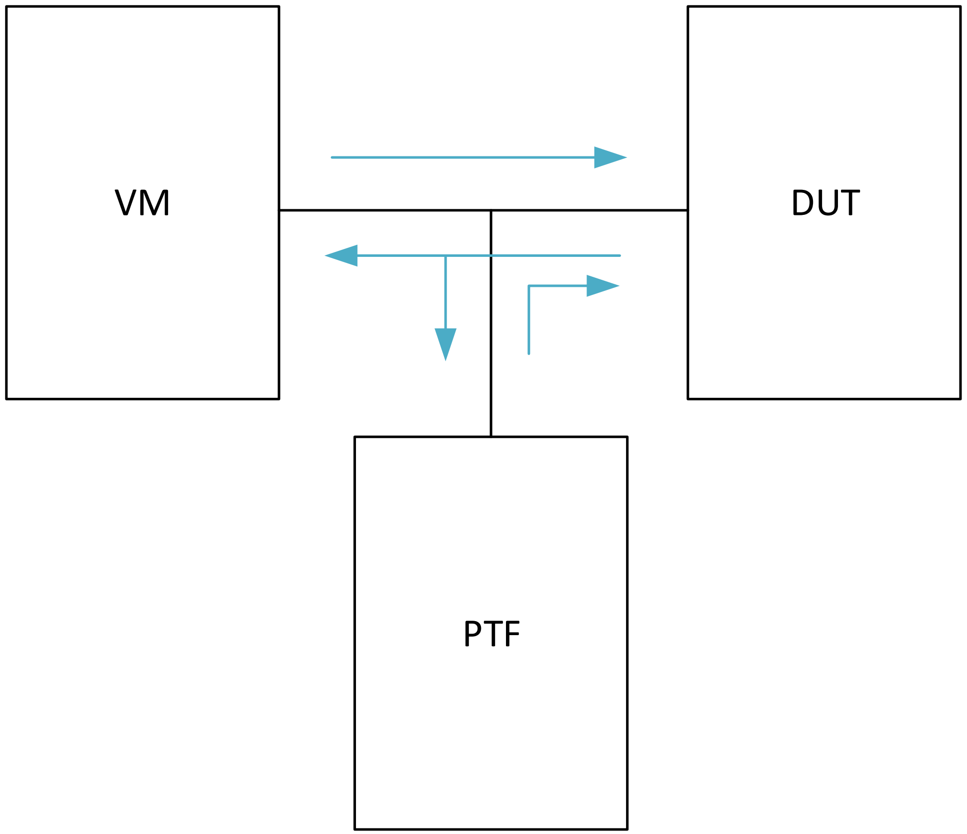
Q & A

- Q & A

- Quick Resource Reference:
  - Wiki: https://github.com/Azure/sonic/wiki
  - Sonic-buildimage: https://github.com/Azure/sonic-buildimage/blob/master/README.md
  - Sonic-Configuration: https://github.com/Azure/SONiC/wiki/Configuration-and-Minigraph
  - Sonic-testing: https://github.com/Azure/SONiC/wiki/Testing-Guide

Direct interface vs injected interface





- Injected interface:
  - capture traffic from DUT to VM
  - Inject traffic to DUT
- Injected interface:
  - VM <–> DUT – BGP traffic
  - PTF <–> DUT – test traffic

testbed.csv Consistency rules

uniq-name	testbed-name	topo	ptf_imagename	ptf_mgmt._ip	server	vm_base	dut	Commen
vms1-1-t1	vms1-1	t1	docker-ptf-sai-brcm	10.0.0.178/24	server_1	VM0100	str-sw1-11	
vms1-1-t1-lag	vms1-1	t1-lag	docker-ptf-sai-mlnx	10.0.0.178/24	server_1	VM0100	str-sw2-4	

Must be strictly checked in code reviews

- uniq-name must be unique
- All testbed records with the same testbed-name must have the same:
  - ptf_ip
  - server
  - vm_base
- testbed-name must be up to 8 characters long
- topo name must be valid (topo registered in veos and topo file presented in vars/topo_*.yml
- ptf_imagename must be valid
- server name must be valid and presented in veos inventory file
- vm_base must not overlap with testbeds from different groups (different test-name)

TODO: check this constraints in testbed-cli.sh