

# neutron-lan

SDN study environnement @ home

[github.com/alexanderplatz1999](https://github.com/alexanderplatz1999)

Last update: April 18th, 2014

# Background and Motivation

- My belief
  - SDN = software defines network
- Too many SDN definitions
  - I have been confused a lot.
  - OpenFlow, OVSD, Netconf, BGP extensions such as FlowSpec...
  - The latest one: OpFlex (DevOps-like)
- What's the real SDN?
  - Let's develop SDN by myself and examine every definition.
- But, wait! I need a SDN study environment at home.
  - I am a poor guy, so I cannot buy expensive SDN-capable switches from Cisco, Juniper...

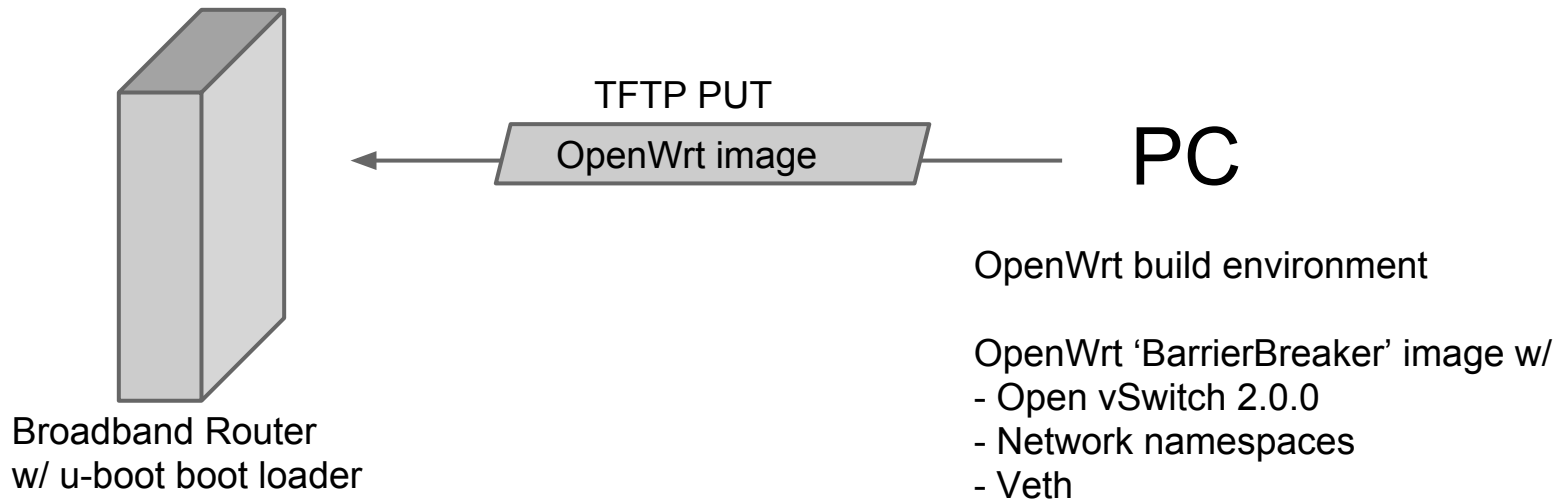
# Strategy

- My budget is less than \$200.
- Switches/routers I purchased in Akihabara, Tokyo
  - Three \$40 broadband routers and one \$40 Raspberry Pi
- And I develop all the SDN software from scratch
  - But reuse existing networking software as much as possible, such as Open vSwitch
- Base knowledge/skills
  - SDN in the past: SIP and IP-PBX
  - OpenFlow, OpenStack neutron and SaltStack
  - Java and Python
  - HTML5 and CSS (a little)
- Let's develop neutron-like SDN for my home network ⇒ let's call it 'neutron-lan'

# Project 'neutron-lan' characteristics

- Cheap routers as 'Baremetal Switch'
  - OpenWrt, Raspberry Pi
  - u-boot for installing new firmware
- Home-made DevOps tool 'NLAN' from scratch
  - 100% Python implementation
  - YAML-based state rendering
  - Model-driven service abstraction
- VXLAN-based edge-overlay for network virtualization
- LXC for Network Functions Virtualization
- Open vSwitch as a programmable switch
- OVSDB as a general-purpose config database

# Cheap routers as 'Baremetal Switch'



# Test bed (cont'd)

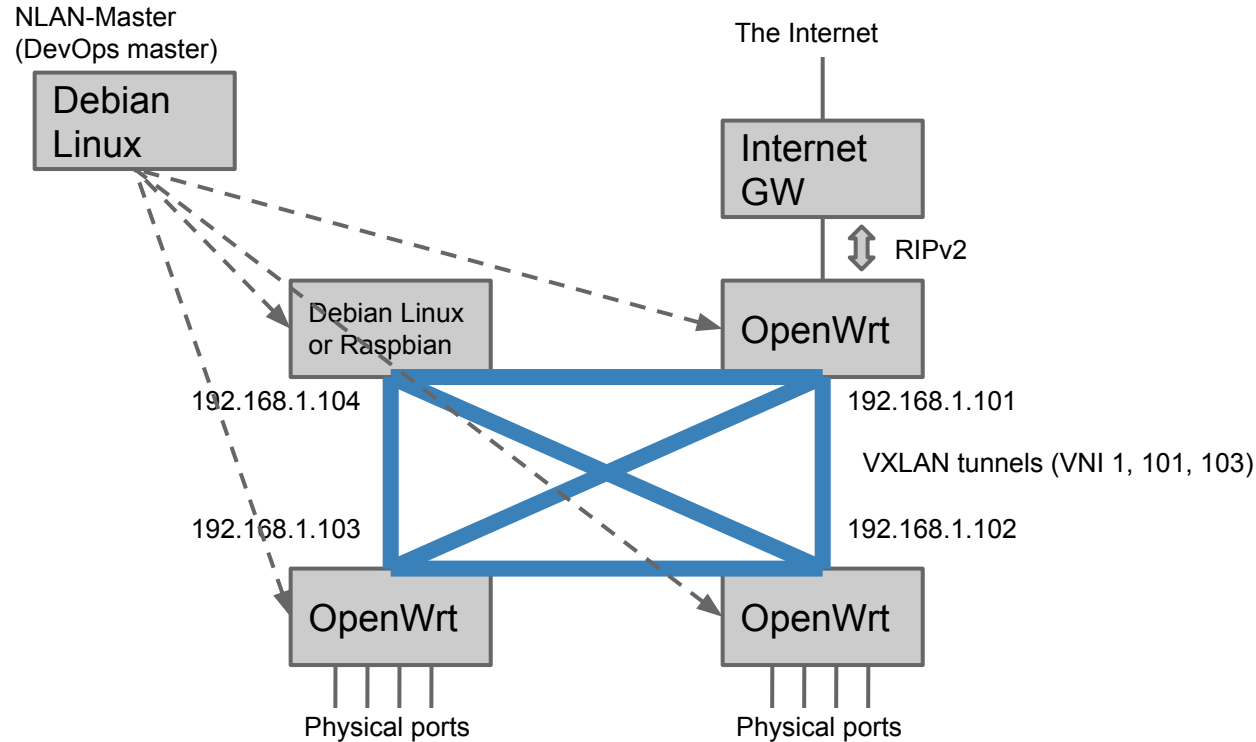


OpenWrt routers  
(and Home Gateway)

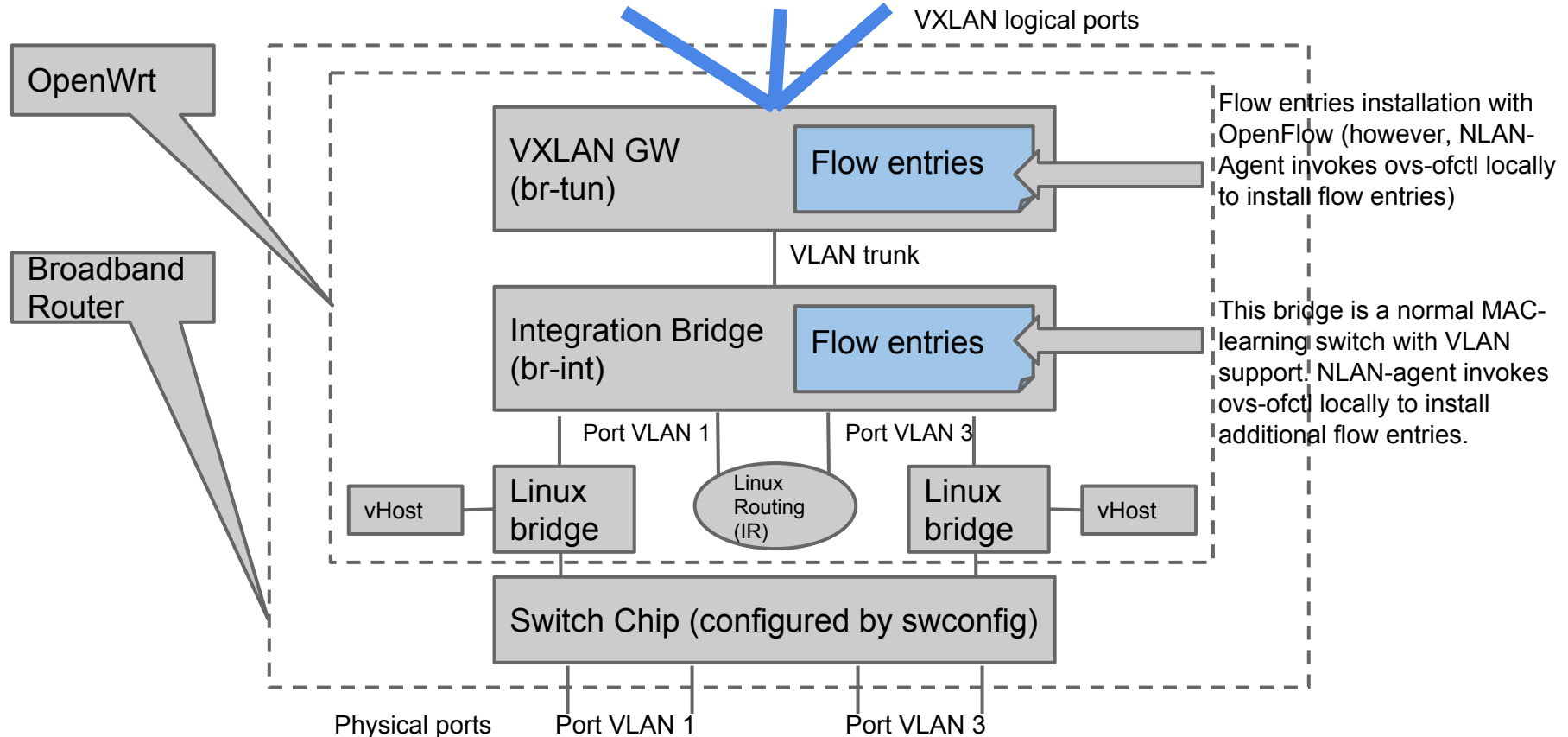


Raspberry Pi

# Test bed

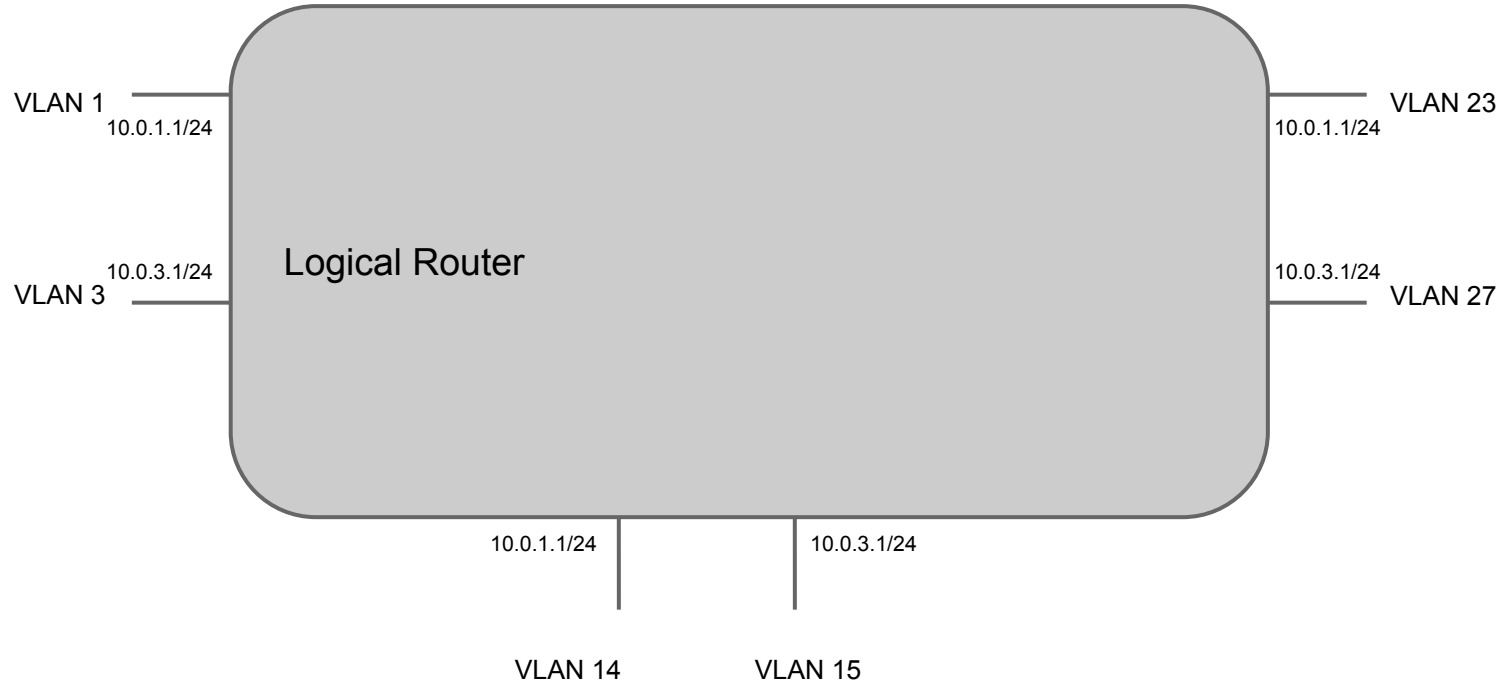


# OpenStack-neutron-like bridge configuration





# Distributed Virtual Router (Logical view)



# Virtual network topologies

NLAN node operation mode	Virtual Network Topology
dvr	Distributed Virtual Router
hub	Hub & Spoke
spoke	Hub & Spoke
spoke_dvr	Mixture of DVR and Hub & Spoke

## NLAN state in YAML

subnets:

- vid: 1

vni: 1001

ip\_dvr: '10.0.1.1/24'

mode: **hub**  mode can be 'dvr', 'hub', 'spoke' or 'spoke\_dvr'

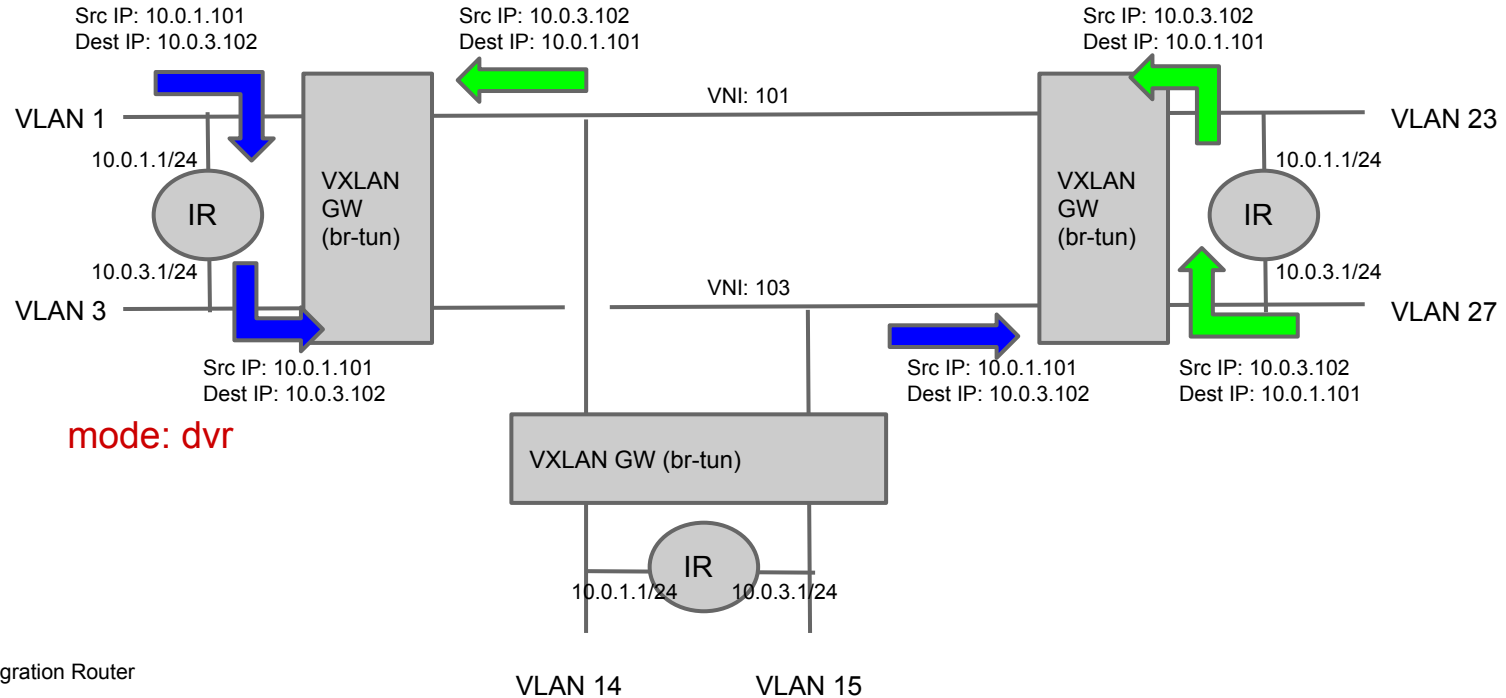
ip\_vhost: '10.0.1.101/24'

ports:

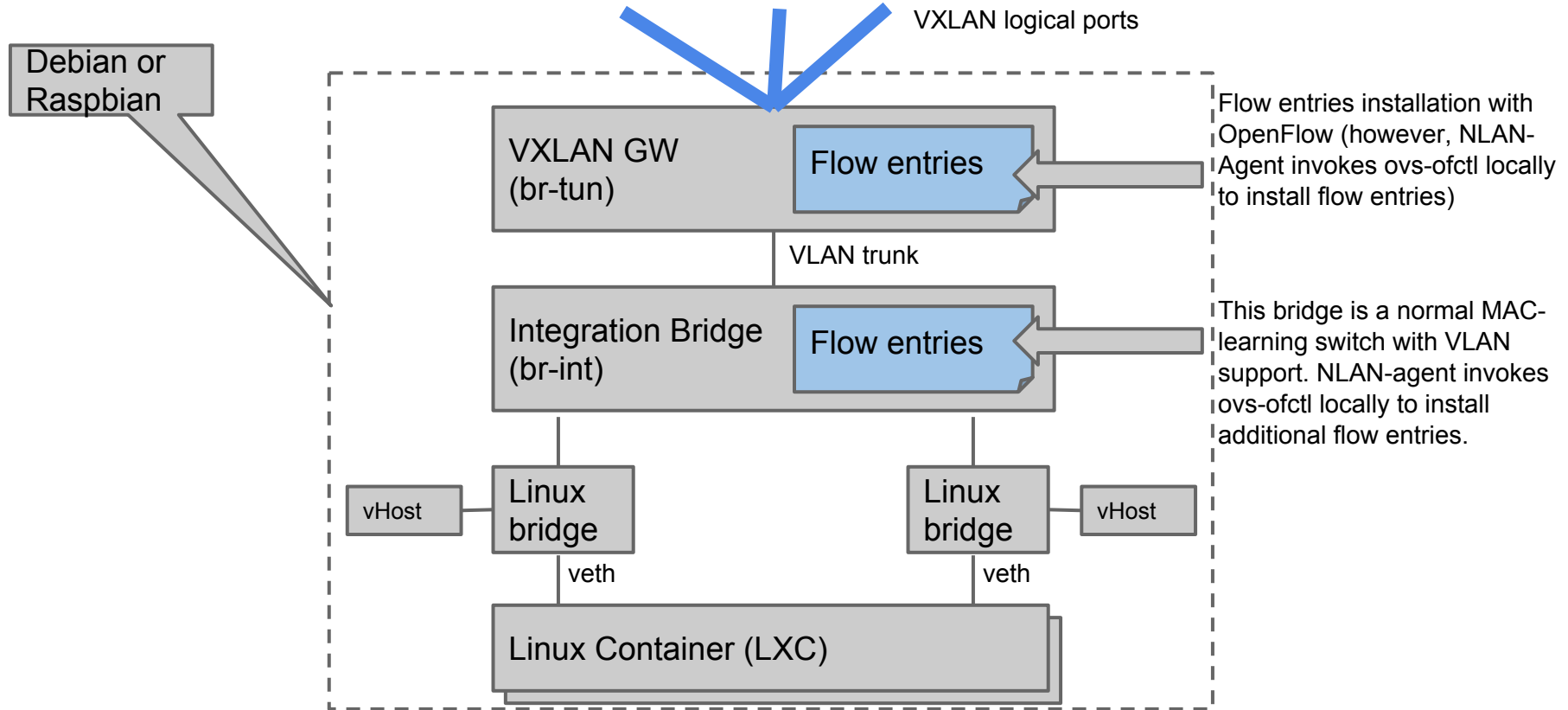
- eth0.1

peers: <peers>

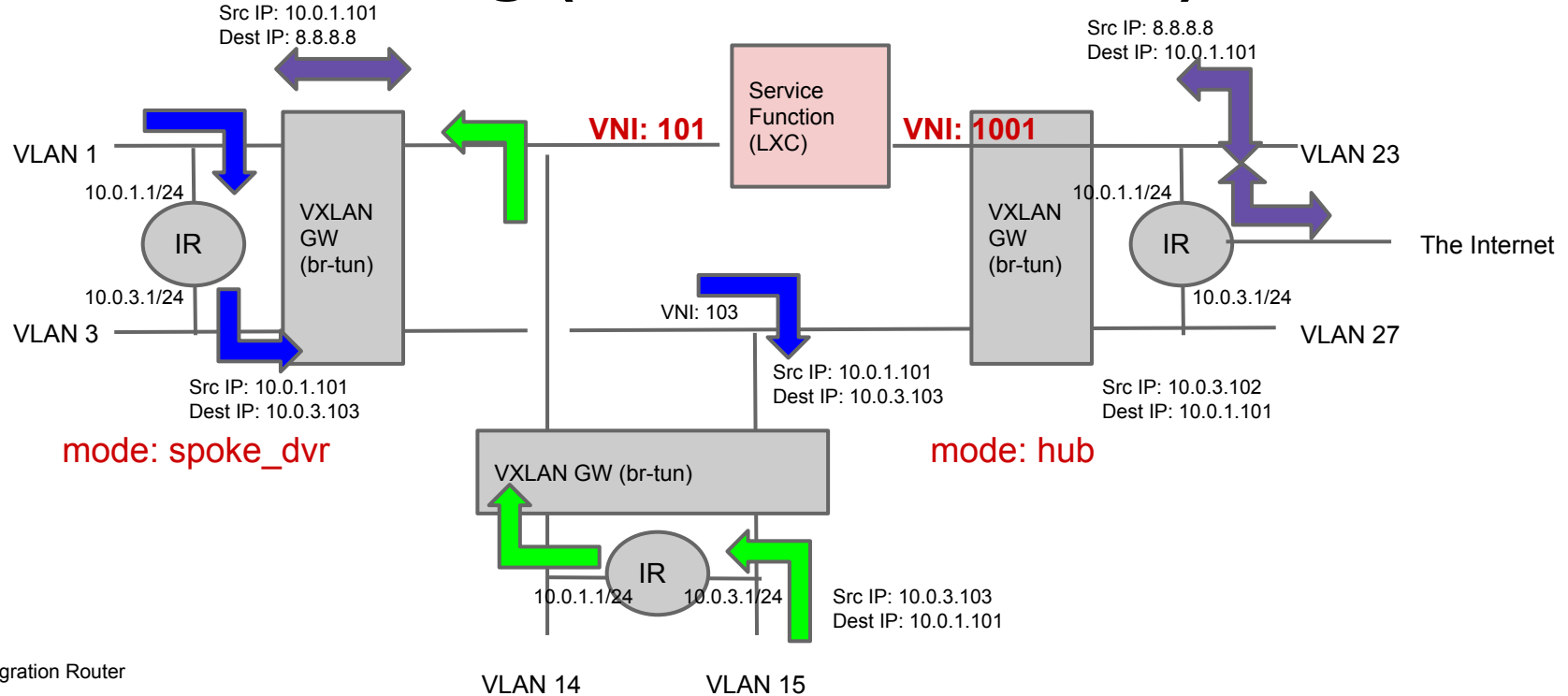
# Distributed Virtual Switch and Distributed Virtual Router



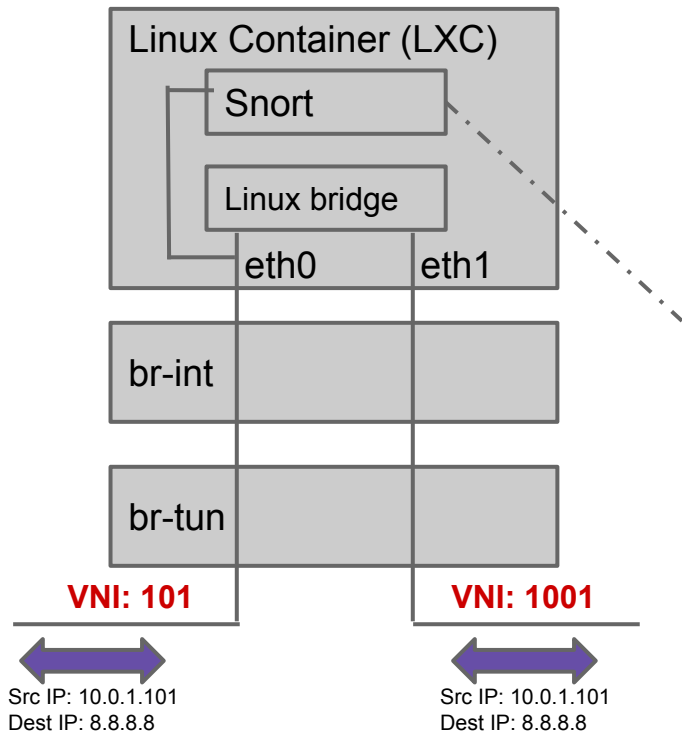
# Service Function in Linux Container



# Service Chaining (Service Insertion)



## Example: Snort (in IPS mode) as Service Function



04/13-23:16:32.859385 10.0.1.102 -> 8.8.8.8

ICMP TTL:64 TOS:0x0 ID:11745 IpLen:20 DgmLen:84 DF

Type:8 Code:0 ID:49496 Seq:25 ECHO

=====

04/13-23:16:32.861970 8.8.8.8 -&gt; 10.0.1.102

ICMP TTL:63 TOS:0x0 ID:38077 IpLen:20 DgmLen:84

Type:0 Code:0 ID:49496 Seq:25 ECHO REPLY

=====

04/13-23:16:33.861151 10.0.1.102 -&gt; 8.8.8.8

ICMP TTL:64 TOS:0x0 ID:11746 IpLen:20 DgmLen:84 DF

Type:8 Code:0 ID:49496 Seq:26 ECHO

=====

04/13-23:16:33.862906 8.8.8.8 -&gt; 10.0.1.102

ICMP TTL:63 TOS:0x0 ID:38078 IpLen:20 DgmLen:84

Type:0 Code:0 ID:49496 Seq:26 ECHO REPLY

=====

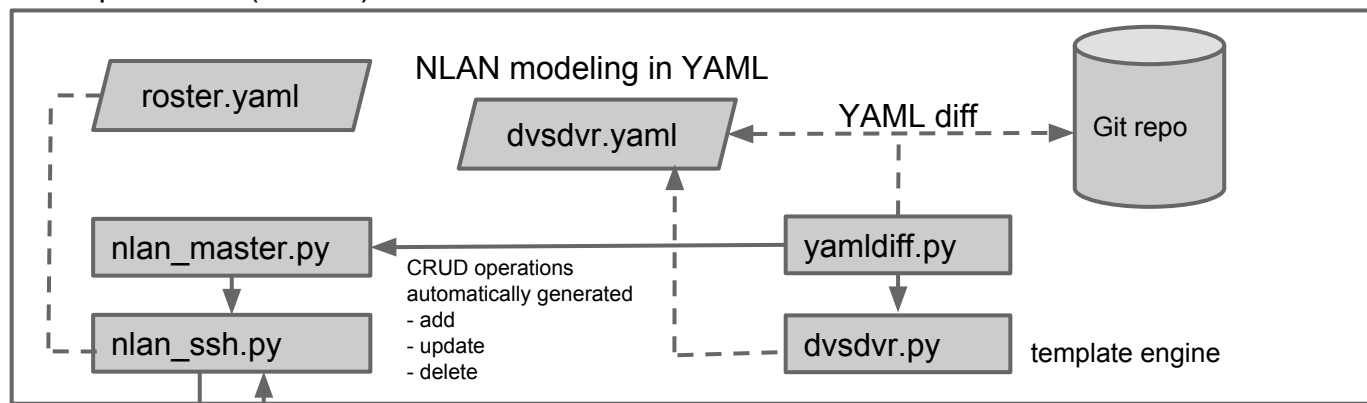
# Home-made DevOps tool 'NLAN'

- 100% Python implementation
- Borrowed a lot of ideas from SaltStack
  - Model-driven procedure
  - YAML-based states w/ a simple template engine
- Works with OpenWrt with minimal Python
  - opkg install python-mini
  - opkg install python-json
  - sshd only
- OVSDDB as a local config mgmt database

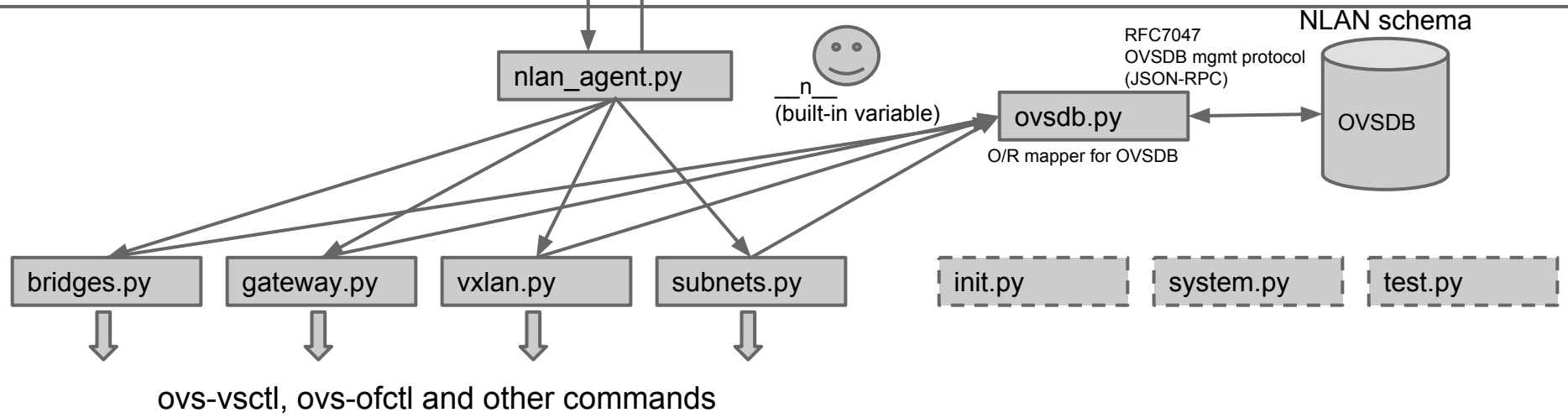
# NLAN architecture (w/ config modules)

It's a bit like Salt State Modules...

DevOps master (Debian)



Router (OpenWrt or Raspbian)

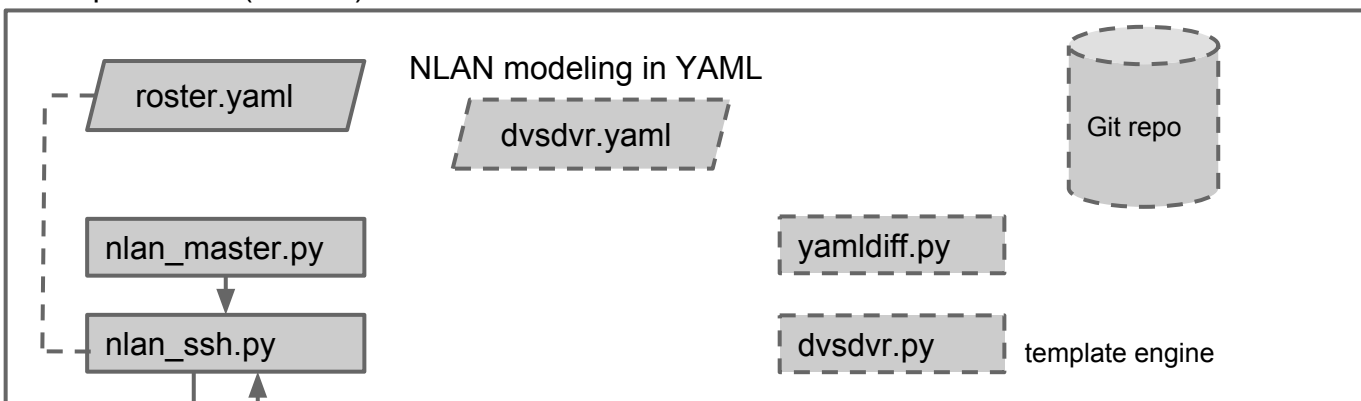




# NLAN architecture (w/ command modules)

It's a bit like Salt Execution Modules...

DevOps master (Debian)

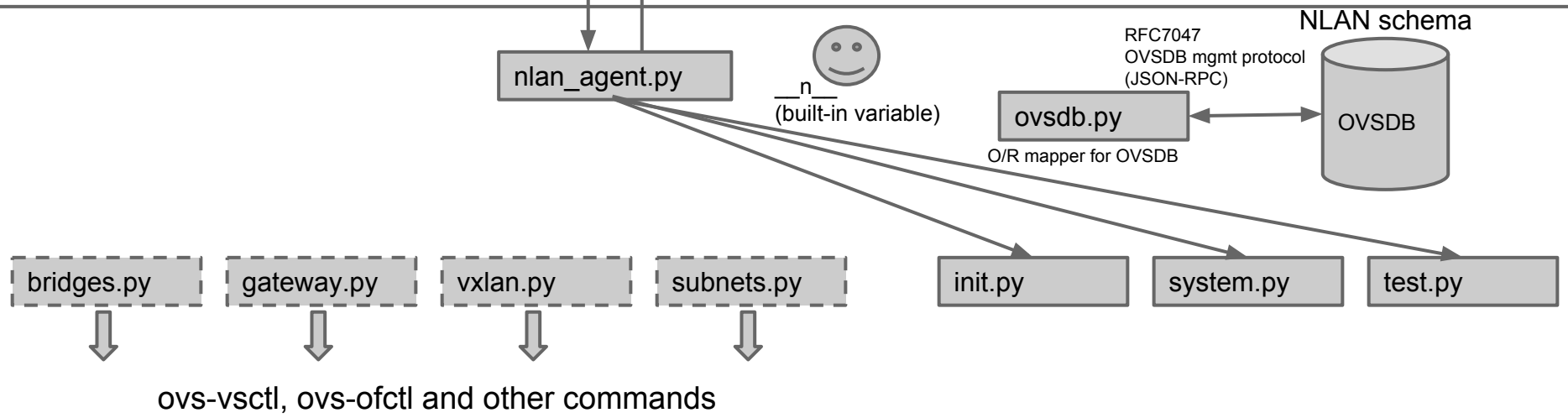


Python OrderedDict object  
via STDIN

Command output  
via STDOUT/STDERR

SSH

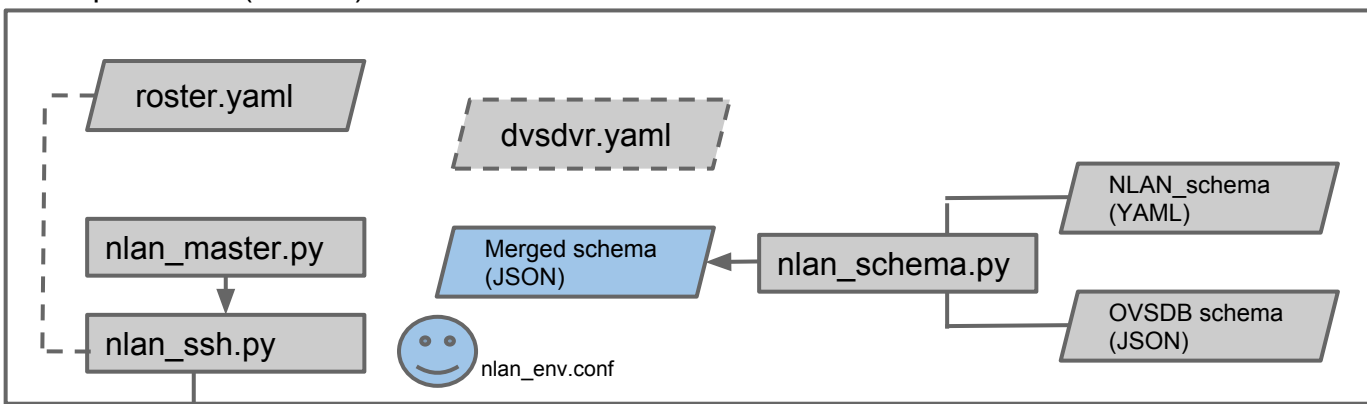
Router (OpenWrt or Raspbian)



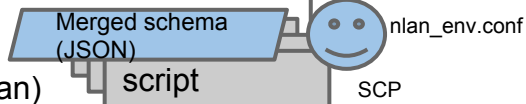
# NLAN architecture (OvFlex)

Flexible OVSDb schemas (not so rigid)

DevOps master (Debian)



Router (OpenWrt or Raspbian)



nlan\_agent.py

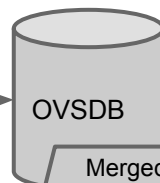


NLAN-specific schema

RFC7047  
OVSDb mgmt protocol  
(JSON-RPC)

NLAN schema

ovsdb.py  
O/R mapper for OVSDb



(built-in variable)

NLAN-specific environment  
variables

bridges.py

gateway.py

vxlan.py

subnets.py

init.py

system.py

test.py

ovs-vsctl, ovs-ofctl and other commands

# OpFlex -- Flexible OVSDB schemas

OpFlex:

<http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html>

OpFlex:

“It uses dynamic, flexible schemas for interaction with devices, effectively increasing the network to a higher common denominator feature set. The Open vSwitch Database (OVSDB) management protocol allows configuration of high-level abstract data models as well as basic primitives such as ports and bridges, and can support SDN geeks’ innovations”

# OVSDB schema (Open\_vSwitch database)

```
{
  "name": "Open_vSwitch",
  "version": "7.4.1",
  "cksum": "951746691 20389",
  "tables": {
    "NLAN": {
      "columns": {
        "bridges": {
          "type": {"key": {"type": "uuid",
            "refTable": "NLAN_Bridges"},
            "min": 0, "max": 1}},
        "services": {
          "type": {"key": {"type": "uuid",
            "refTable": "NLAN_Service"},
            "min": 0, "max": "unlimited"}},
        "gateway": {
          "type": {"key": {"type": "uuid",
            "refTable": "NLAN_Gateway"},
            "min": 0, "max": 1}},
        "vxlan": {
          "type": {"key": {"type": "uuid",
            "refTable": "NLAN_VXLAN"},
            "min": 0, "max": 1}},
        "subnets": {
          "type": {"key": {"type": "uuid",
            "refTable": "NLAN_Subnet"},
            "min": 0, "max": "unlimited"}}},
      "isRoot": true,
      :
```

“Basic primitives”

```
cksum: 951746691 20389
name: Open_vSwitch
tables:
```

Bridge:

columns:

controller:

type:

key: {refTable: Controller, type: uuid}

max: unlimited

min: 0

datapath\_id:

ephemeral: true

type: {key: string, max: 1, min: 0}

datapath\_type: {type: string}

external\_ids:

type: {key: string, max: unlimited, min: 0, value: string}

fail\_mode:

type:

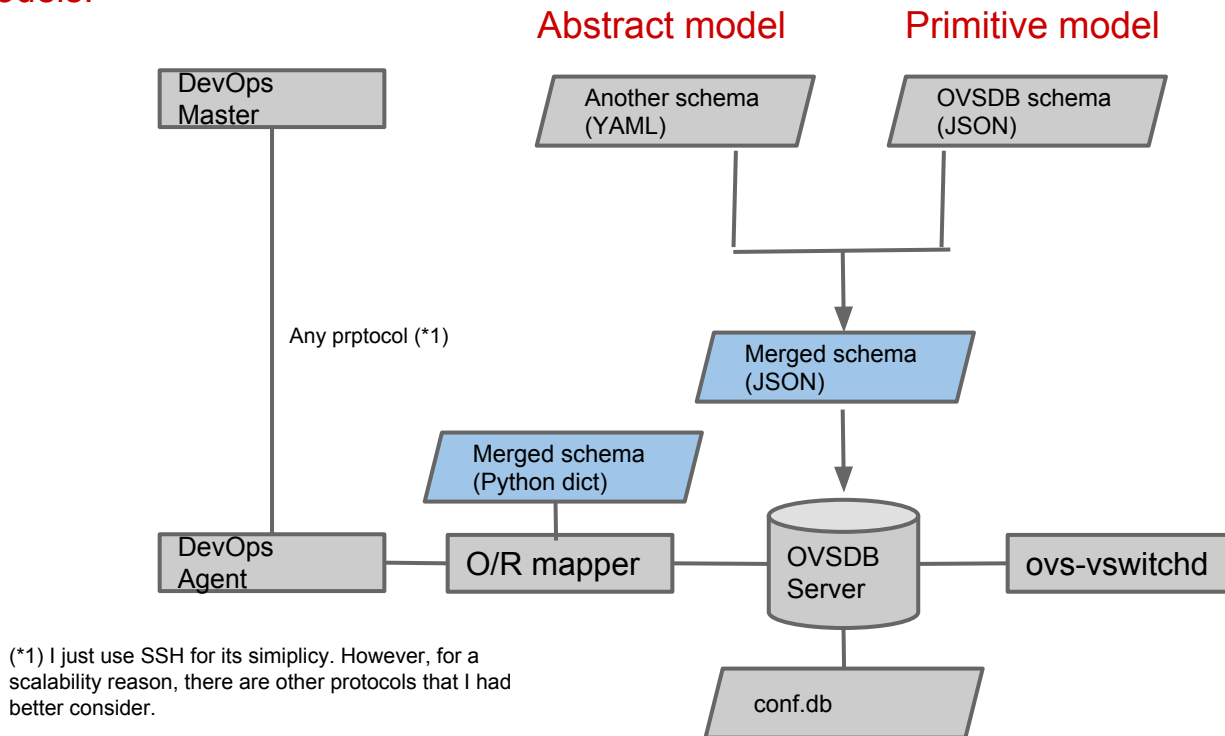
:

Also possible to convert  
it into JSON format by  
using some libraries.

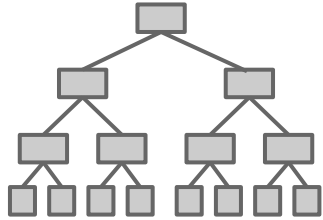
# Merging schemas

OVSDB schema can also express more abstract data models.

```
:
NLAN_Service:
  columns:
  chain:
    type:
      key: {type: string}
      min: 0
      max: unlimited
  name:
    type:
      key: {type: string}
      min: 1
      max: 1
  indexes:
    - [name]
```



# Model-driven service abstraction (cont'd)



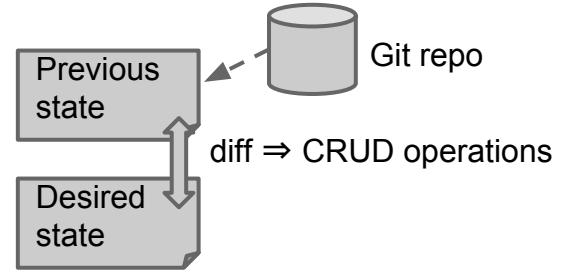
Step1: define network service model



Step2: write the mode as "desired state" in YAML format w/ some placeholders for a template engine

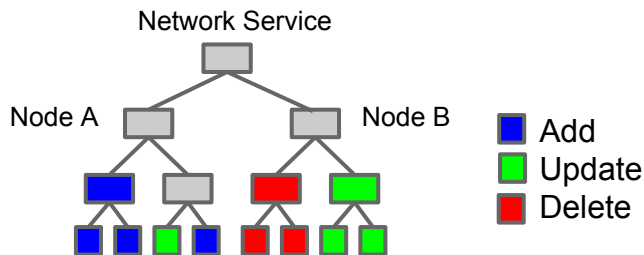


Step3: write a template engine to fill out the placeholders.



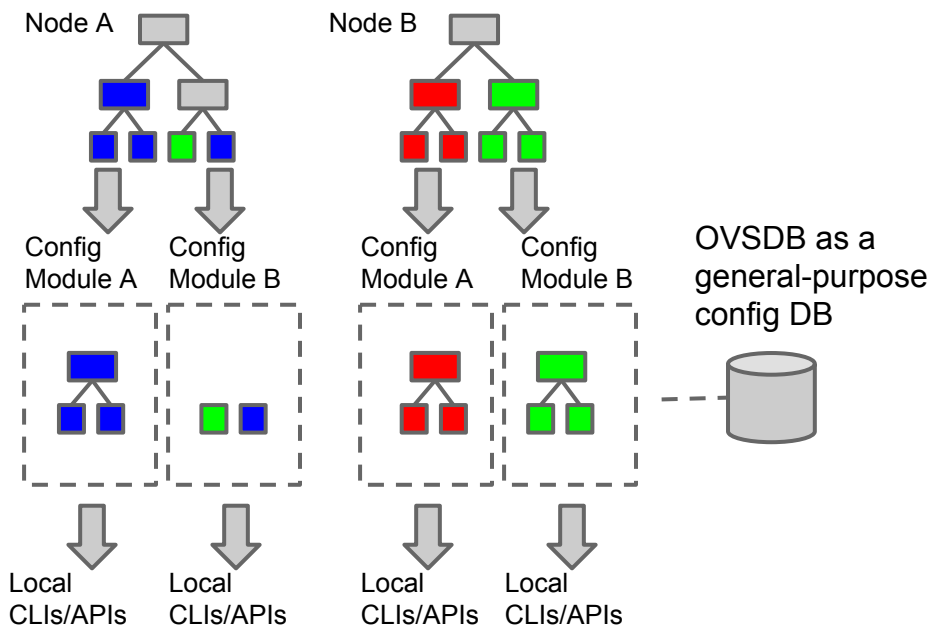
Step4: NLAN-Master generates CRUD operations comparing the desired state with the previous state

# Model-driven service abstraction



Step5: Now CRUD-operations (= diff of the previous and the desired states) are in the form of Python OrderedDict object

Step6: NLAN-Agent routes the CRUD operations to corresponding nodes/modules



# Template and placeholders example

#!/template.dvsvdvr

openwrt1:

:

vxlan:

local\_ip: <local\_ip>

remote\_ips: <remote\_ips>

subnets:

- vid: 1

vni: 101

ip\_dvr: '10.0.1.1/24'

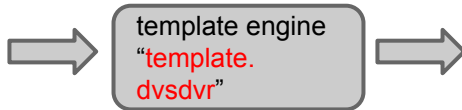
ip\_vhost: '10.0.1.101/24'

ports:

- eth0.1

peers: <peers>

:



- generates a local ip address
- generates VXLAN remote ip addresses
- generates broadcast tree per VNI

#!/template.dvsvdvr

openwrt1:

:

vxlan:

local\_ip: '192.168.1.101'

remote\_ips: ['192.168.1.102', '192.168.1.103', '192.168.1.104']

subnets:

- vid: 1

vni: 101

ip\_dvr: '10.0.1.1/24'

ip\_vhost: '10.0.1.101/24'

ports:

- eth0.1

peers: ['192.168.1.102', '192.168.1.104']

:



# Python OrderedDict

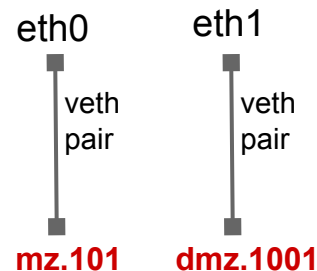
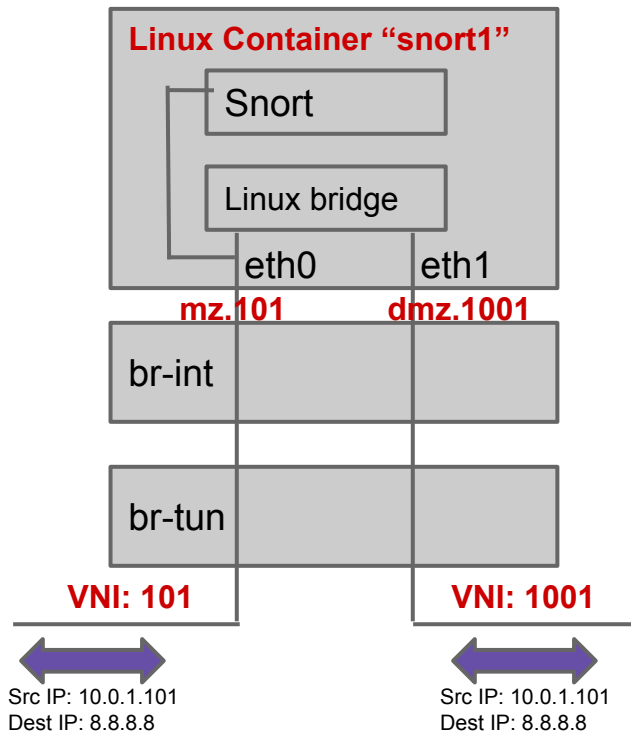
- NLAN-Master sends Python OrderedDict to NLAN-Agent via ssh STDIN.
- To be exact, string form of an OrderedDict object

```
"OrderedDict([('bridges', {'ovs_bridges': 'enabled'}), ('gateway', {'network': 'eth0.2', 'rip': 'enabled'}), ('vxlan', {'remote_ips': ['192.168.1.103', '192.168.1.102', '192.168.1.104'], 'local_ip': '192.168.1.101'}), ('subnets', {'(vni', 1): {'ip_vhost': '192.168.100.101/24', 'ip_dvr': '192.168.100.1/24', 'peers': ['192.168.1.102', '192.168.1.103'], 'vid': 2, 'vni': 1}, ('vni', 103): {'peers': ['192.168.1.102', '192.168.1.103', '192.168.1.104'], 'vid': 3, 'ip_vhost': '10.0.3.101/24', 'vni': 103, 'ip_dvr': '10.0.3.1/24', 'ports': ['eth0.3']}, ('vni', 101): {'peers': ['192.168.1.102', '192.168.1.103', '192.168.1.104'], 'vid': 1, 'ip_vhost': '10.0.1.101/24', 'vni': 101, 'ip_dvr': '10.0.1.1/24', 'ports': ['eth0.1']}}))]"
```

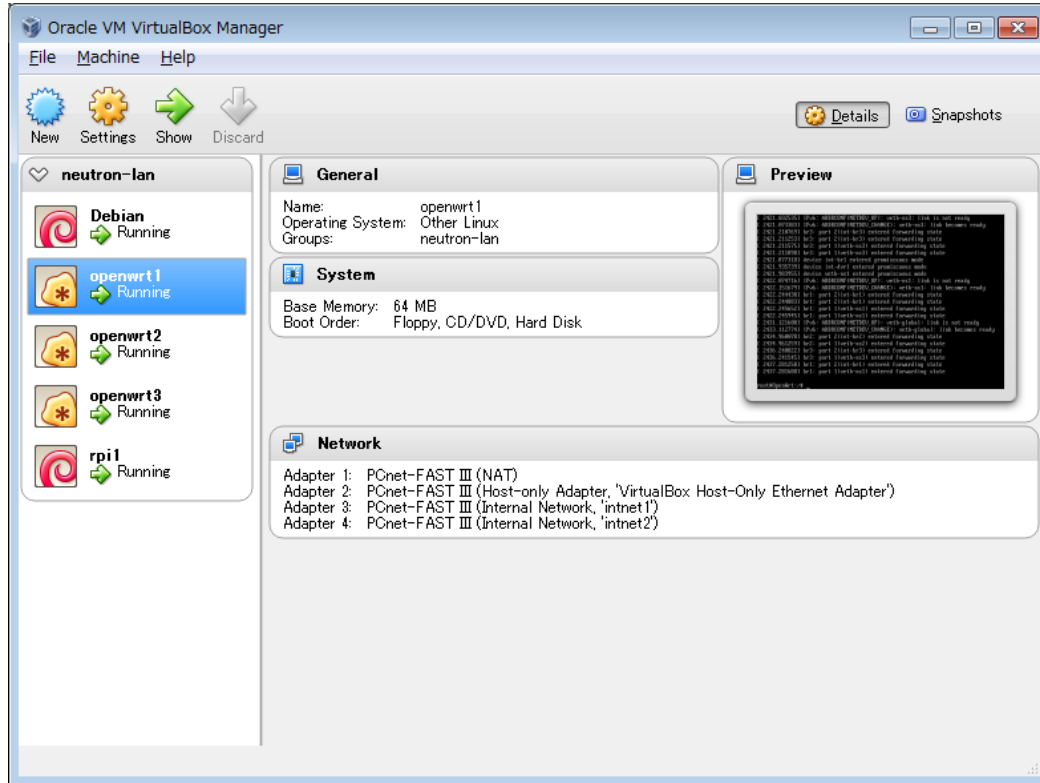
- I don't use JSON, since NLAN is 100% Python implementation.

# Service Function Chaining in YAML

```
rpi1:
  bridges:
    ovs_bridges: enabled
  services: # Service Functions
    - name: snort1
      chain: [mz.101, dmz.1001]
  vxlan:
    local_ip: <local_ip>
    remote_ips: <remote_ips>
  subnets:
    - vid: 111
      vni: 1001
      peers: <peers>
      ports: <sfports>
    - vid: 1
      vni: 101
      peers: <peers>
      ports: <sfports>
```



# NLAN Software Development on VirtualBox



- OpenWrt image for x86
  - Very light-weight Linux supporting Open vSwitch 2.0.0
- Network adapters
  - Internet access: “NAT”
  - Management: “Host-Only”
  - NLAN underlay: “Internal”