neutron-lan

SDN study environement @ home

github.com/alexanderplatz1999

Last update: May 27th, 2014

Background and Motivation

- My belief
  - SDN = Python and Java defines network
- Too many SDN definitions
  - I have been confused a lot.
  - OpenFlow, OVSDB, Netconf, BGP extensions such as FlowSpec...
  - The latest addition: 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

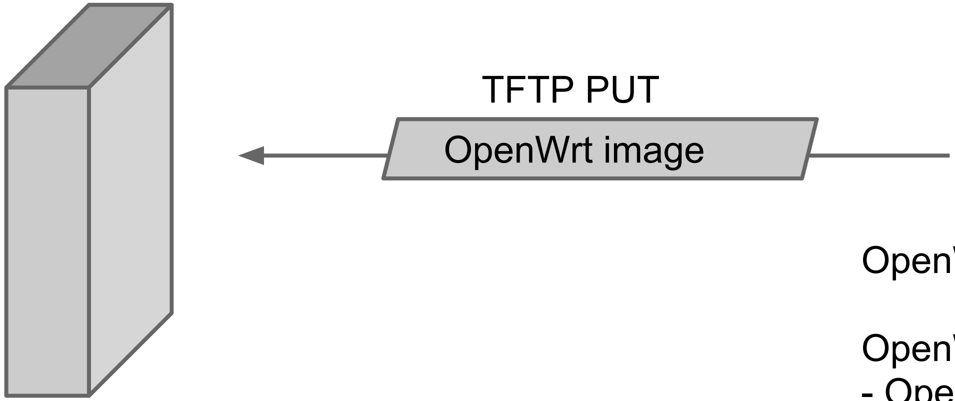
- Cheap routers as ‘Baremetal Switch’
  - OpenWrt routers and Raspberry Pi
  - u-boot for installing new firmware
- Home-made DevOps tool ‘NLAN’ from scratch
  - 100% Python implementation
  - YAML-based state rendering
  - Model-driven service abstraction
- VXLAN-based edge-overlay for network virtualization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

Project neutron-lan
Three major works so far (Dec/2013 ~ May/2014)

- Rebuilding OpenWrt/Raspbian kernel/kernel-modules capabile of OVS 2.0.0, network namespaces(netns), veth(virtual ether) and LXC.
- Configuring VXLAN-based edge-overlay for my home network
- Developing a home-made DevOps tool “NLAN”

Cheap routers as ‘Baremetal Switch’

TFTP PUT



PC

Broadband Router
w/ u-boot boot loader

OpenWrt build environment

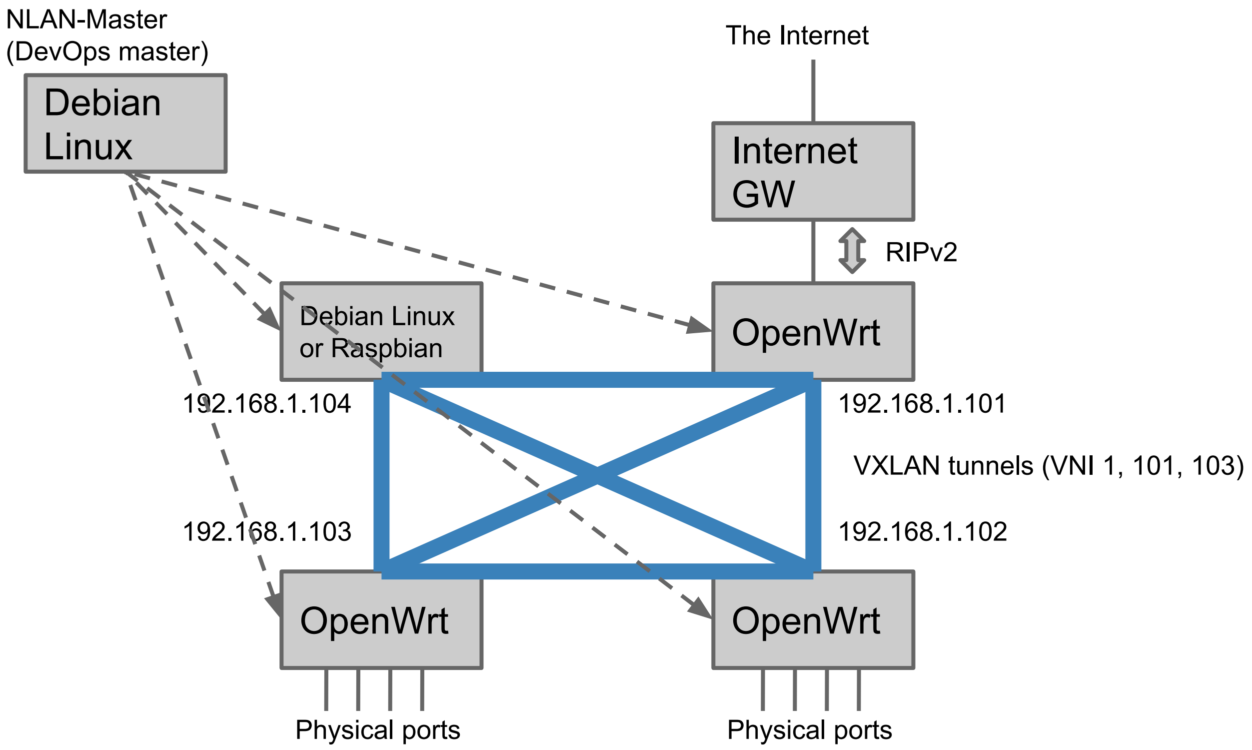
OpenWrt ‘BarrierBreaker’ image w/
- Open vSwitch 2.0.0
- Network namespaces
- Veth

Test bed (cont'd)

OpenWrt routers
(and Home Gateway)

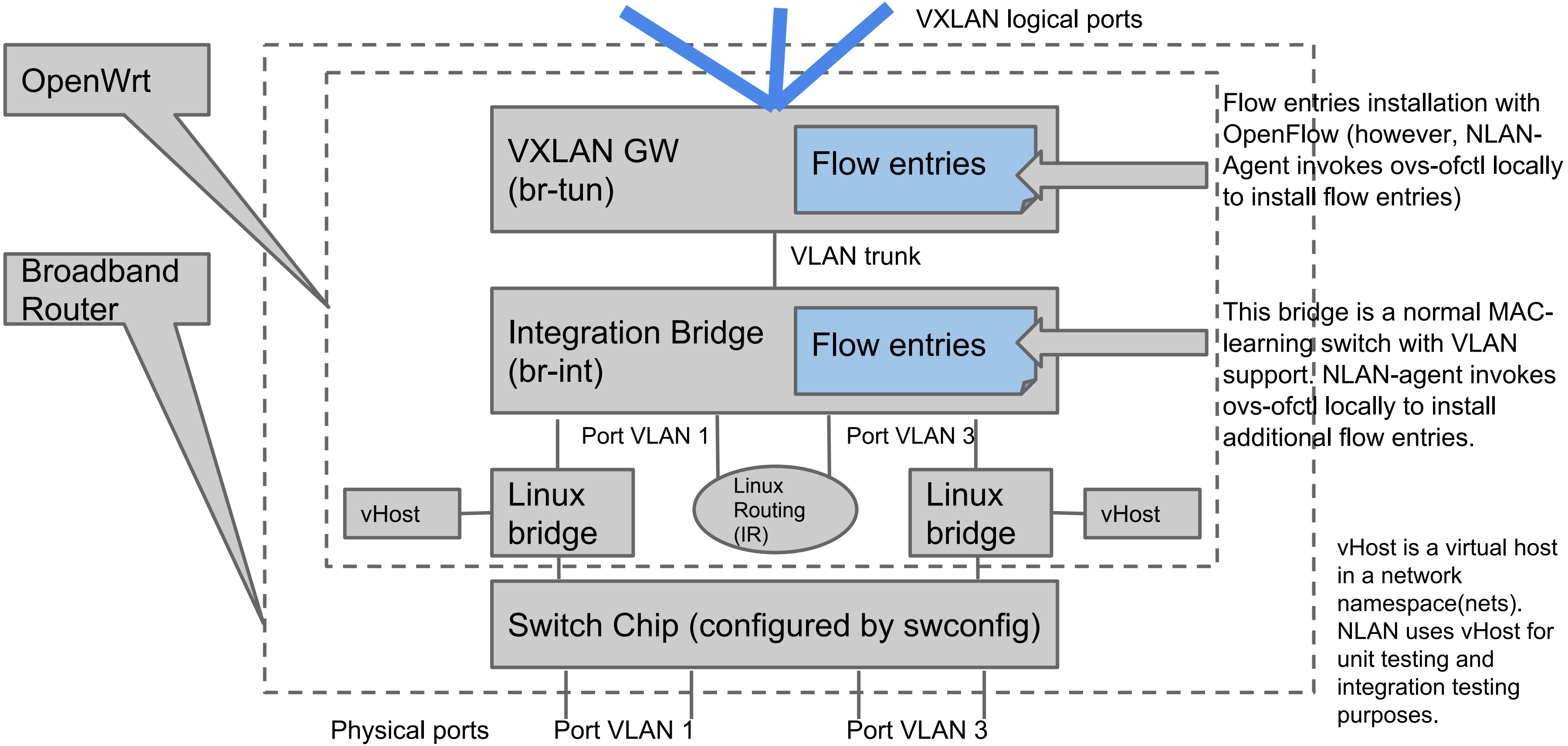
Raspberry Pi

Test bed

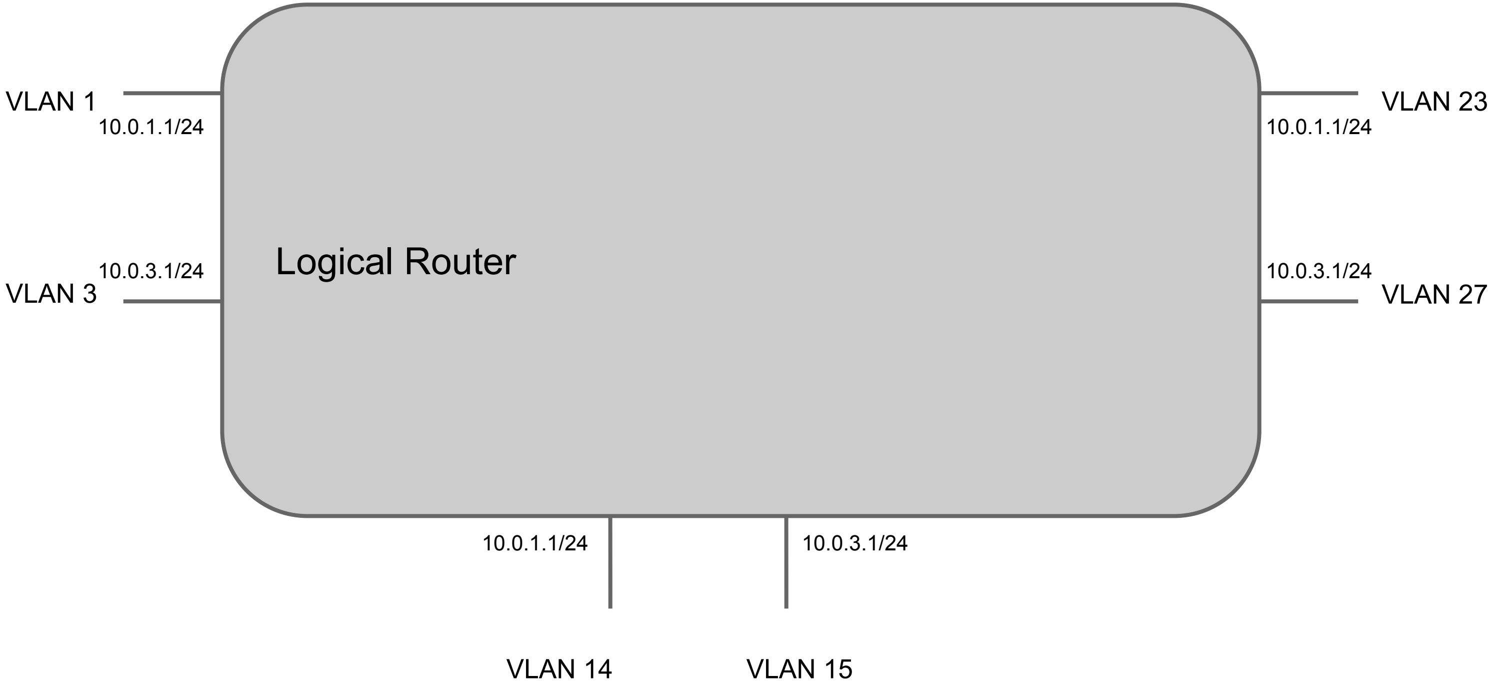


NLAN-Master
(DevOps master)
Debian Linux
The Internet
Internet GW
RIPv2
Debian Linux or Raspbian
OpenWrt
192.168.1.104
192.168.1.101
VXLAN tunnels (VNI 1, 101, 103)
192.168.1.103
192.168.1.102
OpenWrt
OpenWrt
Physical ports
Physical ports

OpenStack-neutron-like bridge configuration



Distributed Virtual Router (Logical view)



Virtual network topologies

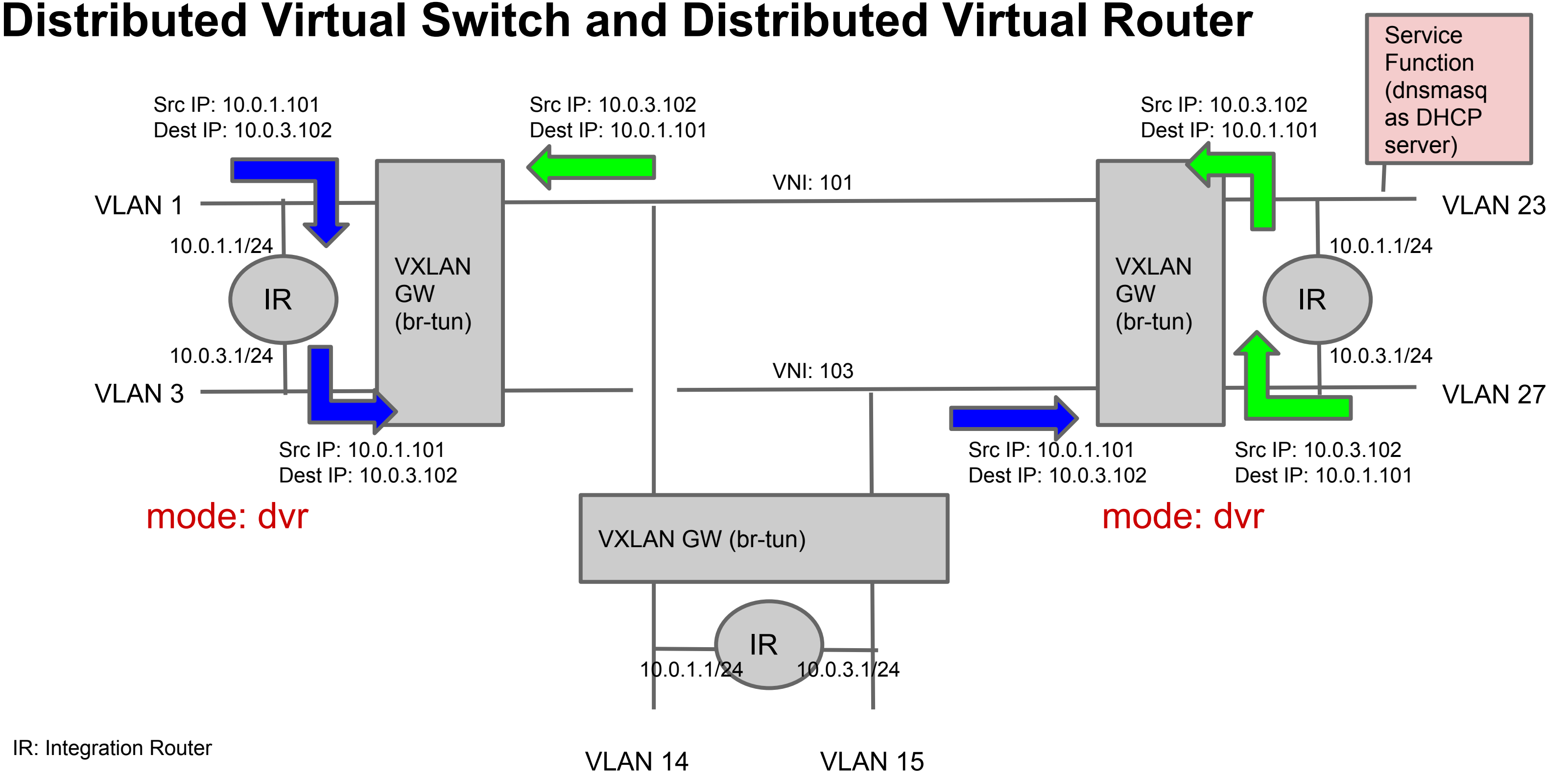
NLAN node operation mode	Virtual Network Topology
dvr	Distributed Virtual Router
hub	Hub & Spoke
spoke	Hub & Spoke
spoke_dvr	Mixture of DVR and Hub & Spoke

<u>NLAN “subnets” state and its parmeters in YAML</u>

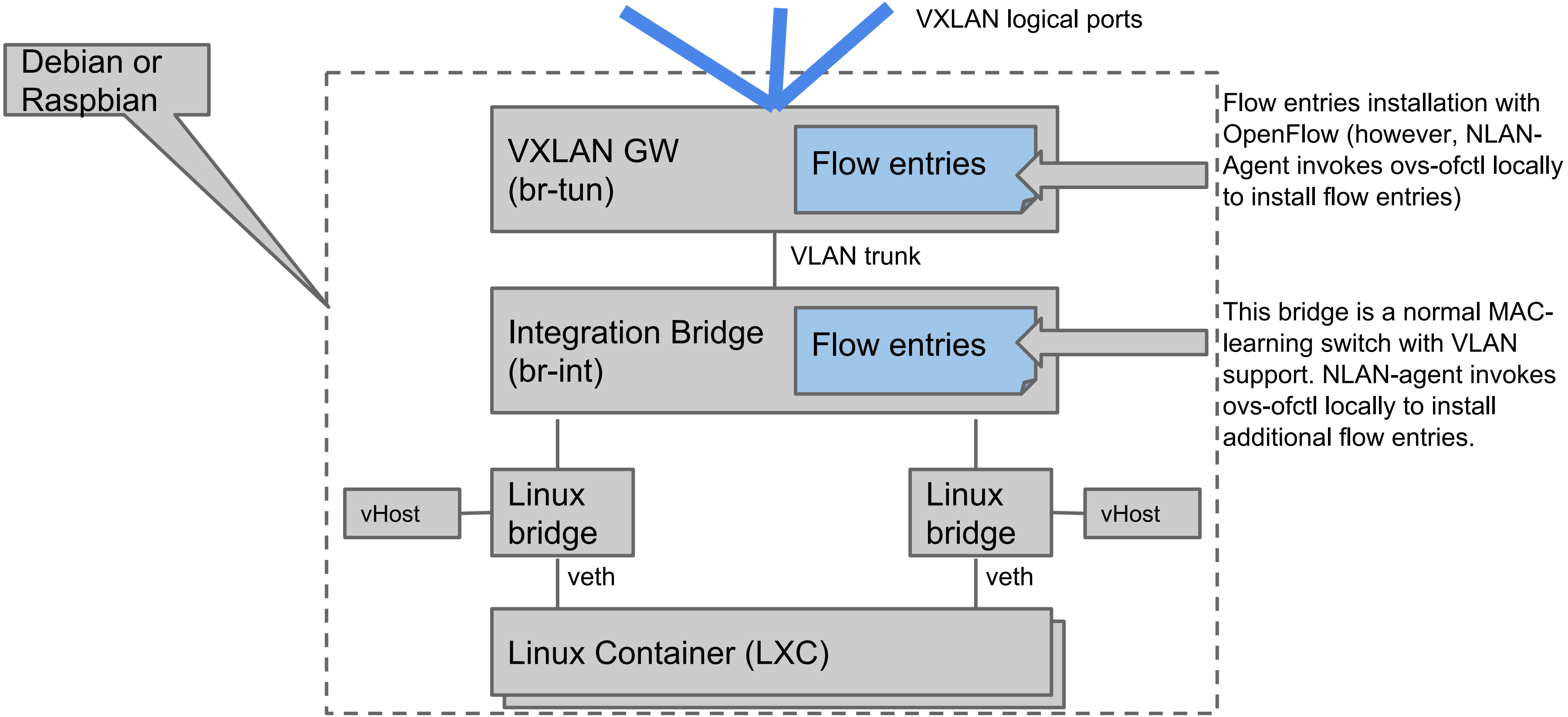
```
subnets:
  - vid: 1
    vni: 1001
    ip_dvr:
      addr: '10.0.1.1/24'
      mode: hub
      dhcp: enabled
    ip_vhost: '10.0.1.101/24'
    ports: [eth0.1]
    peers: <peers>
```

mode can be ‘dvr’, ‘hub’, ‘spoke’ or ‘spoke_dvr’

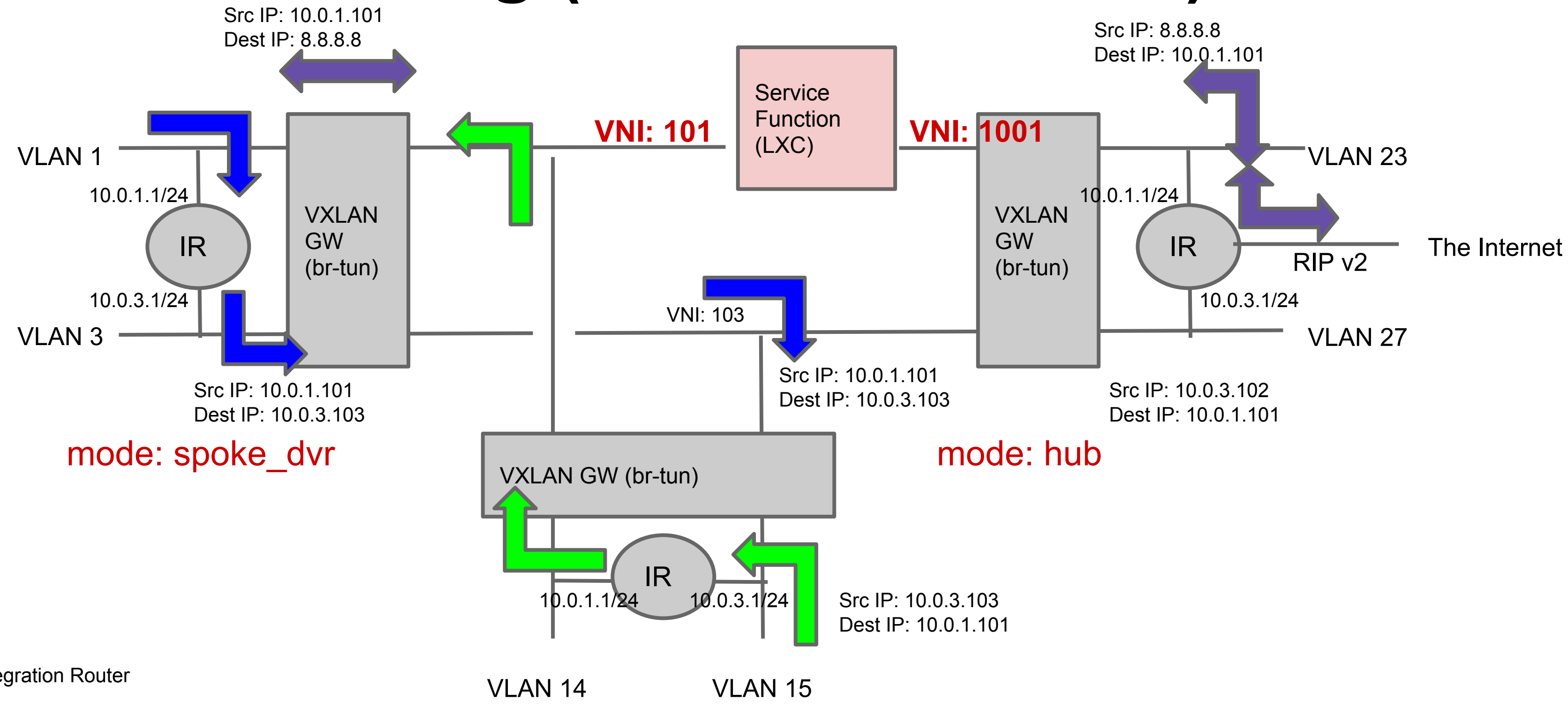
Distributed Virtual Switch and Distributed Virtual Router



Service Function in Linux Container

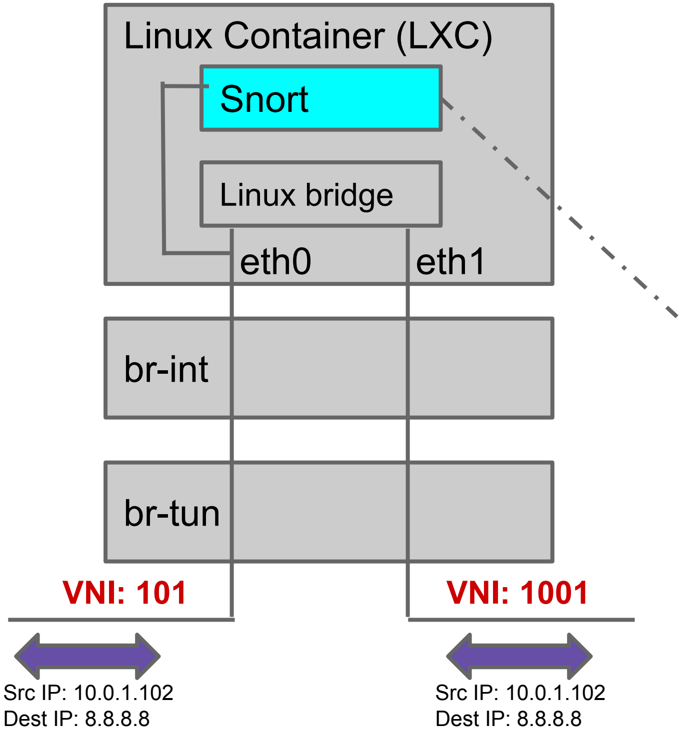


Service Chaining (Service Insertion)



IR: Integration Router

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 -> 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 -> 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 -> 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
=+

Snort is a free and open source IPS/IDS software.

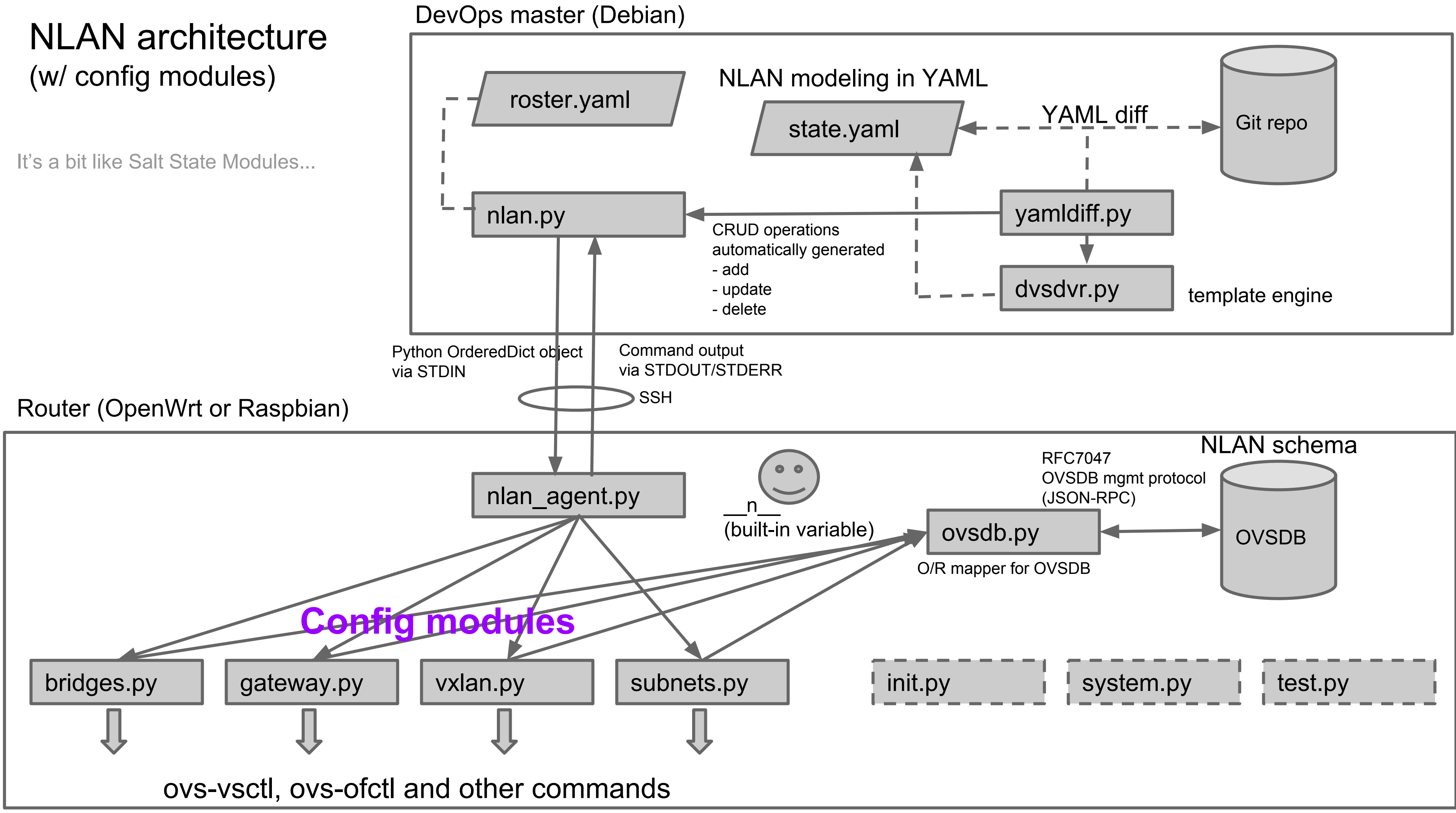
Home-made DevOps tool ‘NLAN’

- 100% Python implementation
- Borrwed a lot of ideas from SaltStack
  - Model-driven approach
  - YAML-based state rendering w/ a simple template engine
  - Imperative/declarative state rendering
- Works with OpenWrt with minimal Python
  - opkg install python-mini
  - opkg install python-json
  - sshd
- OVSDB as a local config mgmt database
- State schema defined in YAML
  - merged with OVSDB schema in JSON

Salt Stack is an open source DevOps framework and applications written in Python.
opkg is a light-weight package management software for OpenWrt.

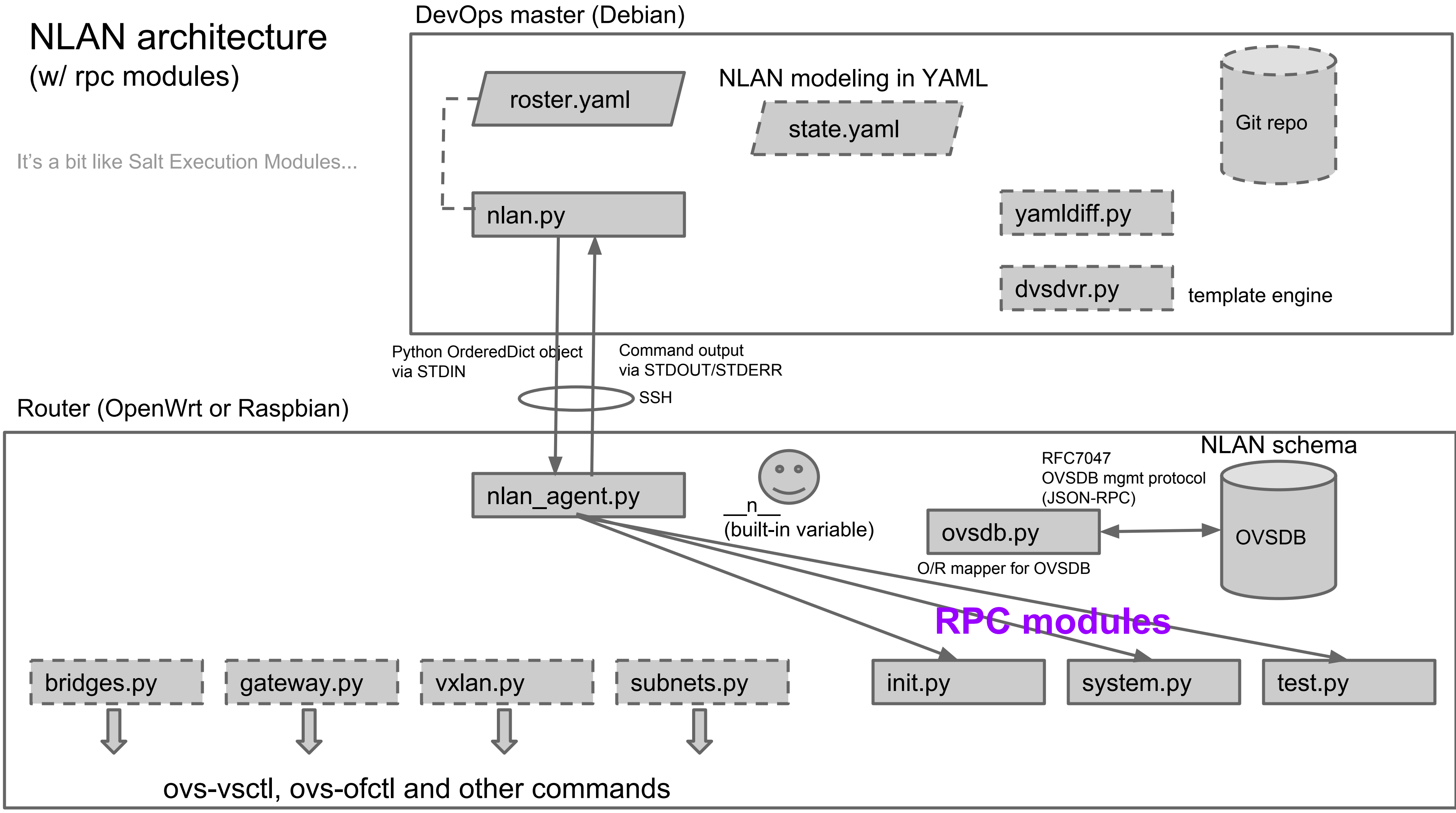
NLAN architecture
(w/ config modules)

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



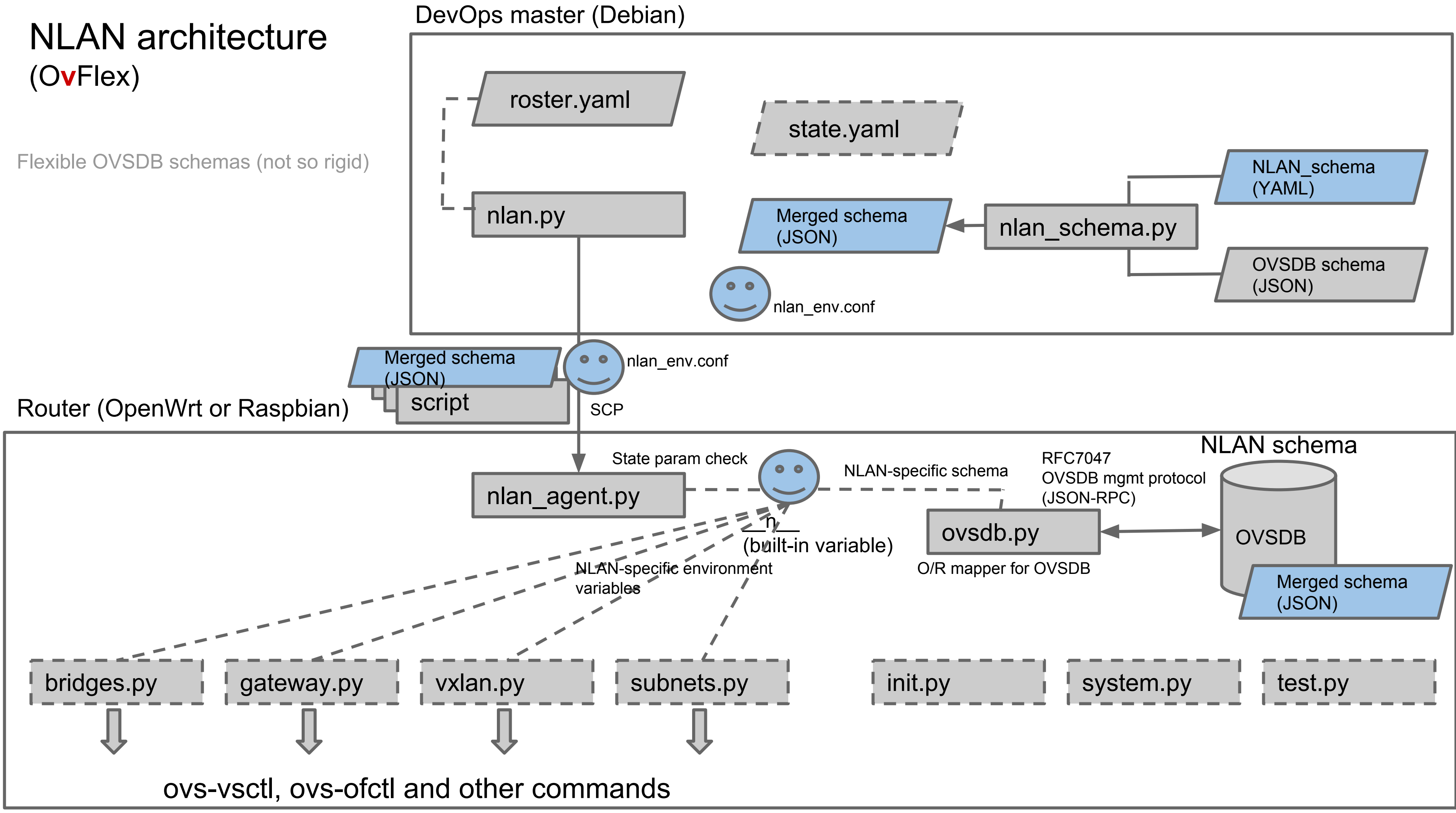
NLAN architecture
(w/ rpc modules)

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

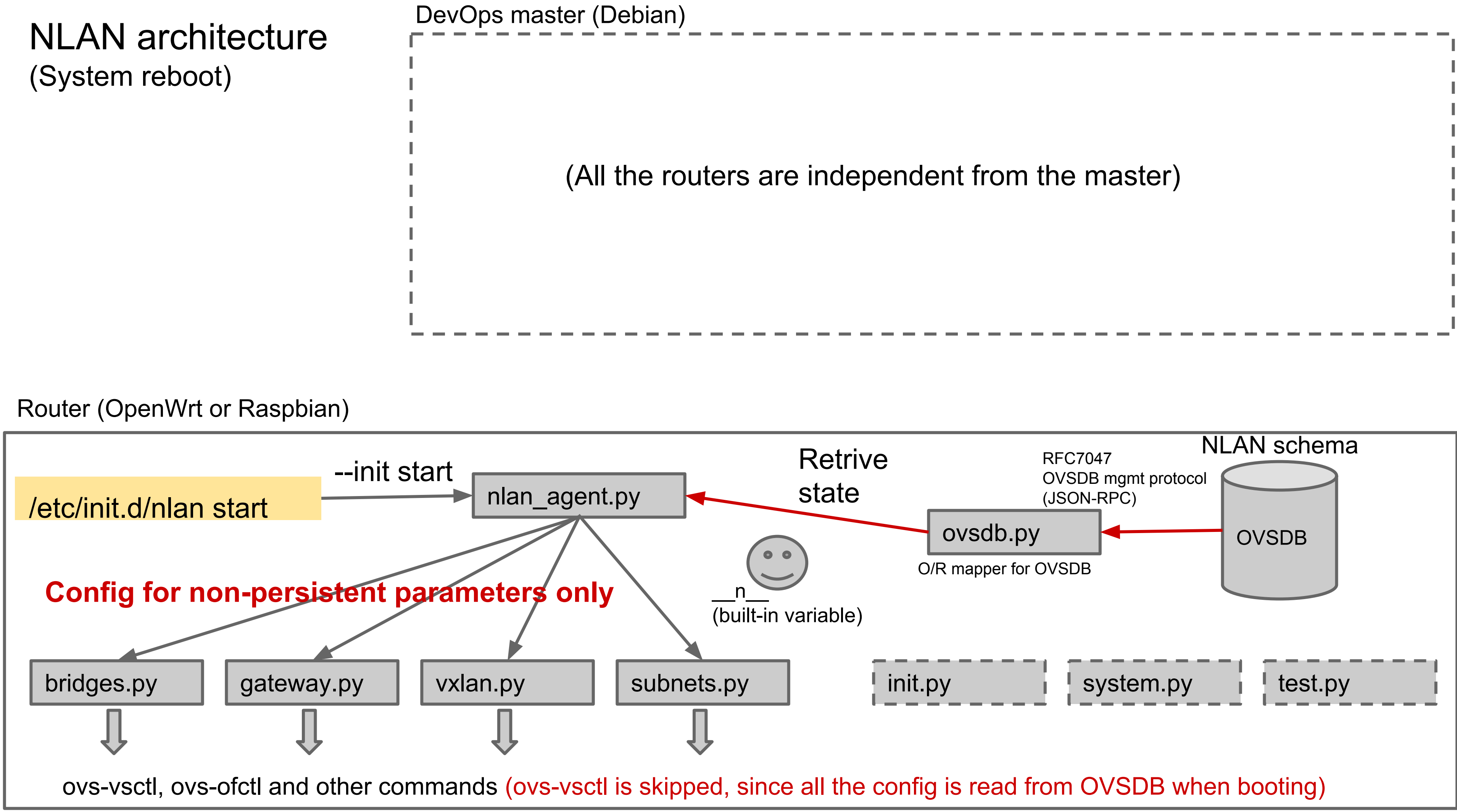


NLAN architecture
(OvFlex)

Flexible OVSDB schemas (not so rigid)

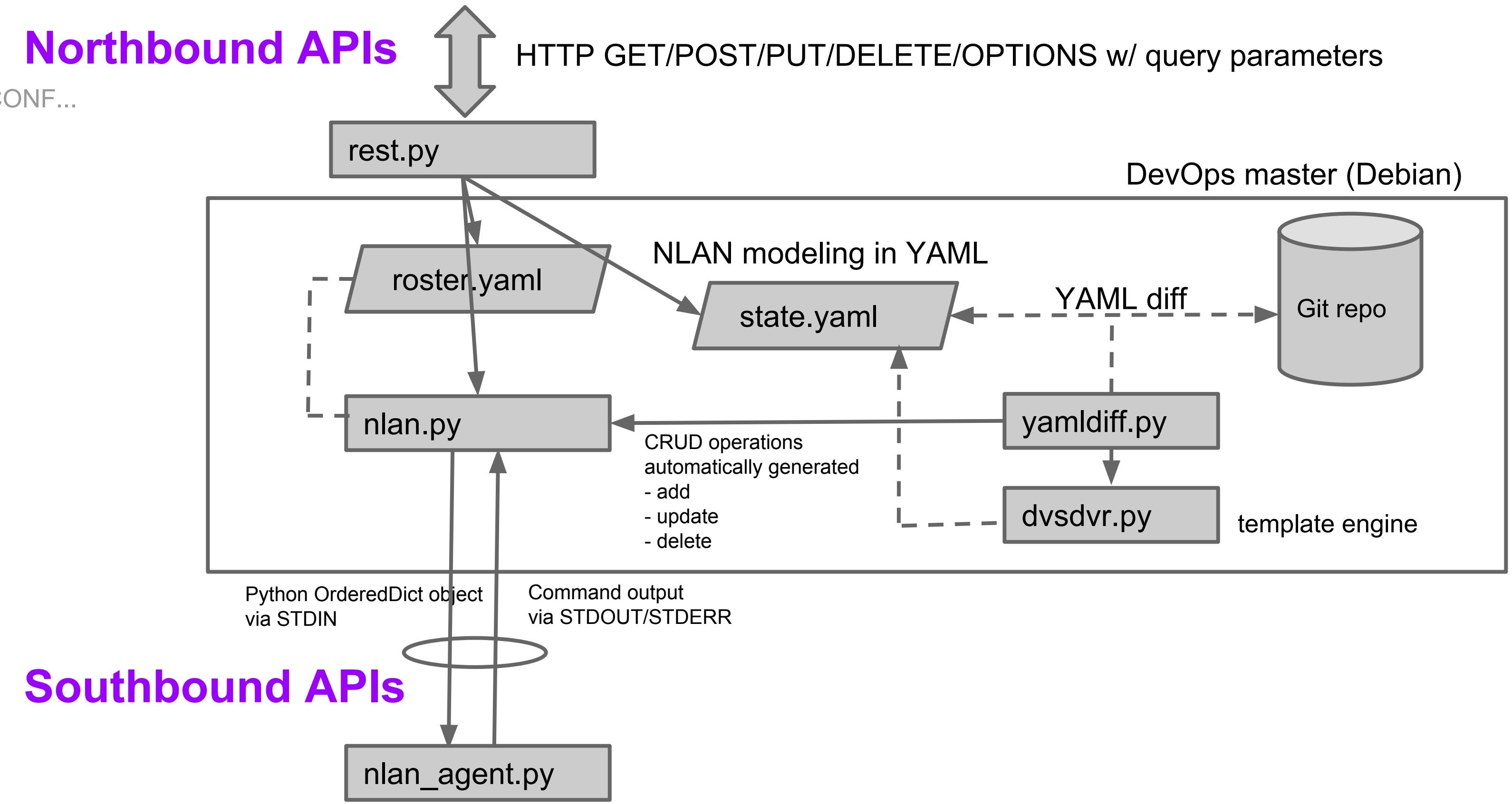


NLAN architecture
(System reboot)



NLAN architecture
(REST APIs)

It’s a bit like RESTCONF...



OvFlex -- Flexible OVSDB schemas

OpFlex:

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

OvFlex:

"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)

NLAN schema in YAML

:

```
NLAN_Subnet:
          columns:
          vni:
          type:
          key: {type: integer, minInteger: 0, maxInteger: 16777215}
          min: 1
          max: 1
          _description: "Virtual network identifier"
          vid:
          type:
          key: {type: integer, minInteger: 0, maxInteger: 4095}
          min: 0
          max: 1
          _description: "VLAN ID"
          ip_dvr:
          type:
          key: {type: string, enum: [set, [addr, mode, dhcp]]}
          value: {type: string, _pattern: {addr: ipv4_prefix, mode: dvr_mode, dhcp:
string}}
          min: 0
          max: 3
          _description: "Distributed Virtual Router setting"
          ip_vhost:
          type:
          key: {type: string, _pattern: ipv4_prefix}
          min: 0
          max: 1
          _description: "Virtual host in a linux network namespace"
          default_gw:
          type:
          key: {type: string, _pattern: ipv4_address}
          min: 0
          max: 1
          _description: "Default GW address for this subnet"
```

:

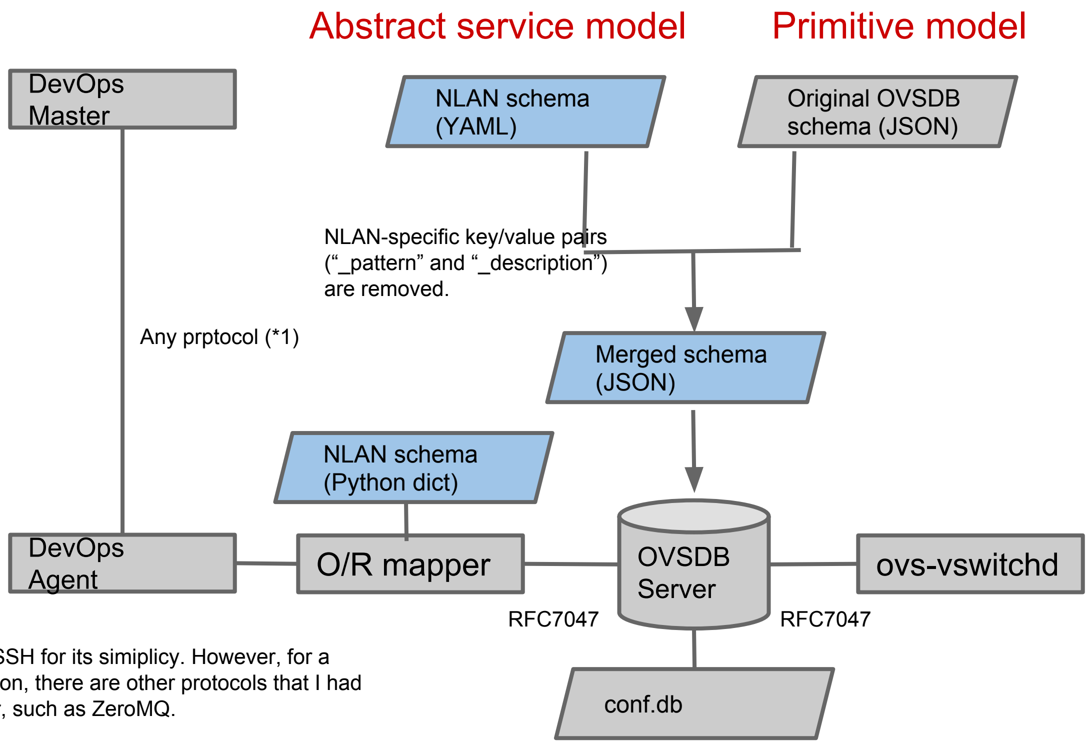
Original OVSDB schema in JSON

```
{"name": "Open_vSwitch",
 "version": "7.4.2",
 "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,
         "maxRows": 1},
```

:

Merging schemas

OVSDB schema can also express more abstract data models.



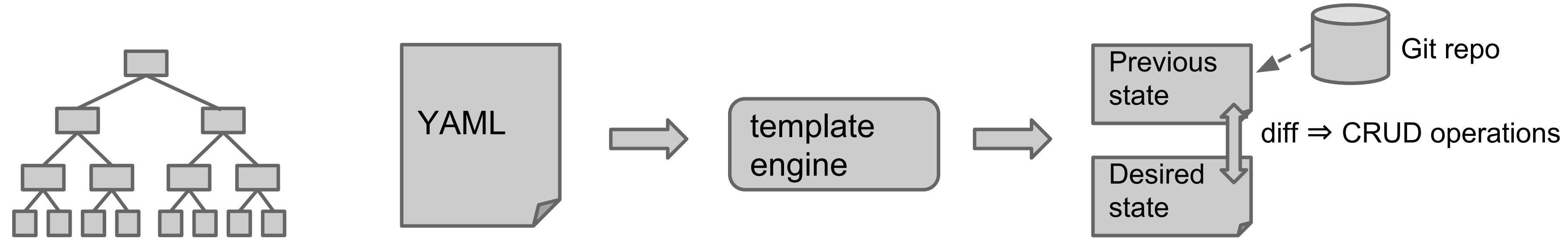
(*1) I just use SSH for its simiplicy. However, for a scalability reason, there are other protocols that I had better consider, such as ZeroMQ.

NLAN states in OVSDB

(ovsdb-client dump Open_vSwitch)

```
NLAN table
_uuid                                bridges                              gateway                              services subnets
                                                                    vxlan
------------------------------------ ------------------------------------ ------------------------------------ -------- ----------------------------------------------
------------------------------------------------------------------ ------------------------------------
f41f2732-c9a0-42ef-a72f-3275ec79c13d 90af6783-f57c-418f-8102-6e02e30d1427 1b3f608c-9a3e-4c4a-9b7c-9805cb8d0d69 []       [3965d784-b309-4d64-b700-396d1ad0a2a4, 53cc4e84
-db7b-4a1e-9d02-df29fa5d9afc, aa54b72e-a22c-47a6-8c0f-c537da6fa8cf] 88d7ad5f-158f-437e-9169-78f79d2e38d5

NLAN_Bridges table
_uuid                                controller ovs_bridges
------------------------------------ ---------- -----------
90af6783-f57c-418f-8102-6e02e30d1427 []         enabled

NLAN_Gateway table
_uuid                                network rip
------------------------------------ ------- -------
1b3f608c-9a3e-4c4a-9b7c-9805cb8d0d69 "eth2"  enabled

NLAN_Service table
_uuid chain name
----- ----- ----

NLAN_Subnet table
_uuid                                default_gw ip_dvr                               ip_vhost             peers                                ports vid vni
------------------------------------ ---------- ------------------------------------ -------------------- ------------------------------------ ----- --- ----
53cc4e84-db7b-4a1e-9d02-df29fa5d9afc []         {addr="10.0.1.1/24", mode=hub}       "10.0.1.101/24"      ["192.168.1.104"]                    []    1   1001
3965d784-b309-4d64-b700-396d1ad0a2a4 []         {addr="10.0.3.1/24", mode=dvr}       "10.0.3.101/24"      ["192.168.1.102", "192.168.1.103"]   []    3   103
aa54b72e-a22c-47a6-8c0f-c537da6fa8cf []         {addr="192.168.100.1/24", mode=dvr}  "192.168.100.101/24" ["192.168.1.102", "192.168.1.103"]   []    2   1

NLAN_VXLAN table
_uuid                                local_ip        remote_ips
------------------------------------ --------------- ---------------------------------------------------
88d7ad5f-158f-437e-9169-78f79d2e38d5 "192.168.1.101" ["192.168.1.102", "192.168.1.103", "192.168.1.104"]
```

Model-driven service abstraction (cont'd)



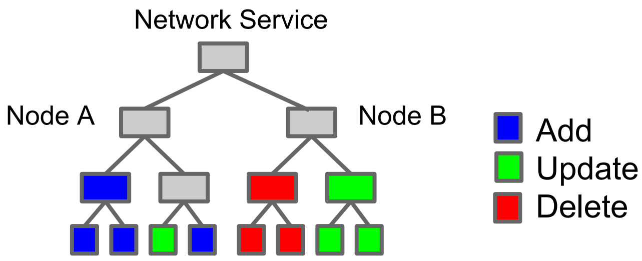
Step1: define network service model

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 automatically.

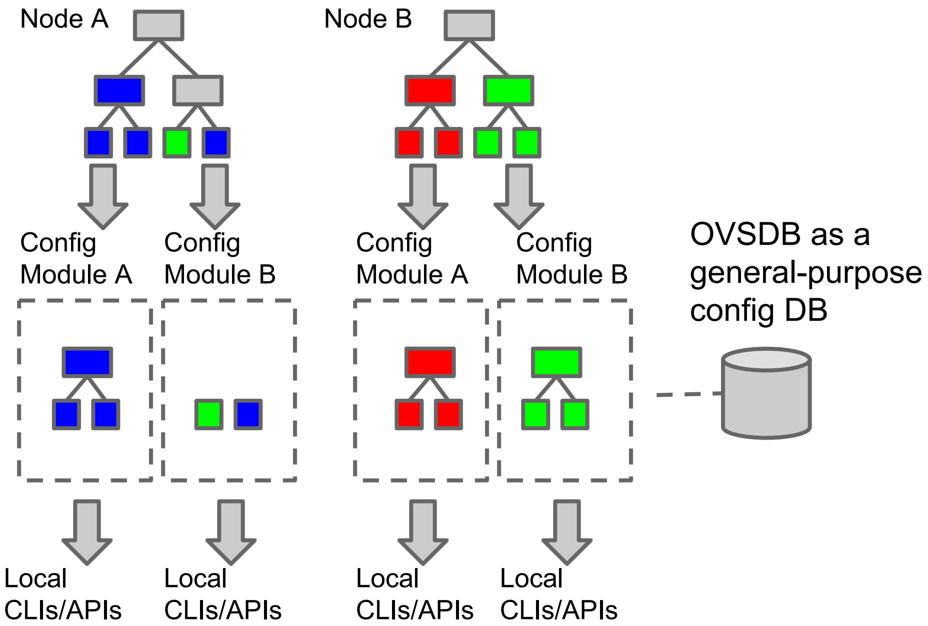
Step4: NLAN-Master (nlan.py and yamldiff.py) generates CRUD operations comparing the desired state with the previous state

Model-driven service abstraction



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

Step6: NLAN-Agent (nlan_agent.py) routes the CRUD operations to corresponding nodes/modules



Template and placeholders example

```
#!template.dvsdvr
openwrt1:
      :
  vxlan:
      local_ip: <local_ip>
      remote_ips: <remote_ips>
  subnets:
      - vid: 1
      vni: 101
      ip_dvr: {addr: '10.0.1.1/24'. mode: dvr}
      ip_vhost: '10.0.1.101/24'
      ports:
      - eth0.1
      peers: <peers>
      :
```

template engine
“template.
dvsdvr”

- generates a local ip address
- generates VXLAN remote ip addresses
- generates broadcast tree per VNI
- automatically resolves dependencies among parameters

```
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: {addr: '10.0.1.1/24', mode: dvr}
      ip_vhost: '10.0.1.101/24'
      ports:
      - eth0.1
      peers: ['192.168.1.102', '192.168.1.104']
      :
```

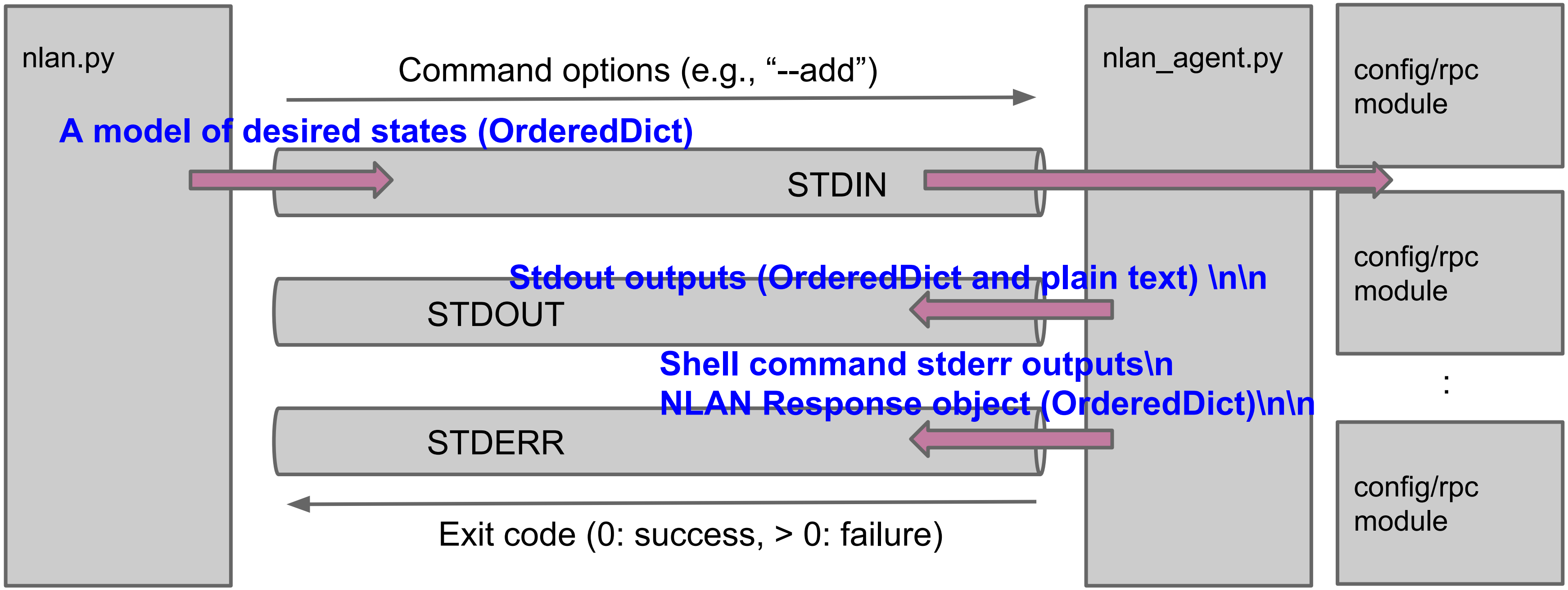
Model of desired state

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

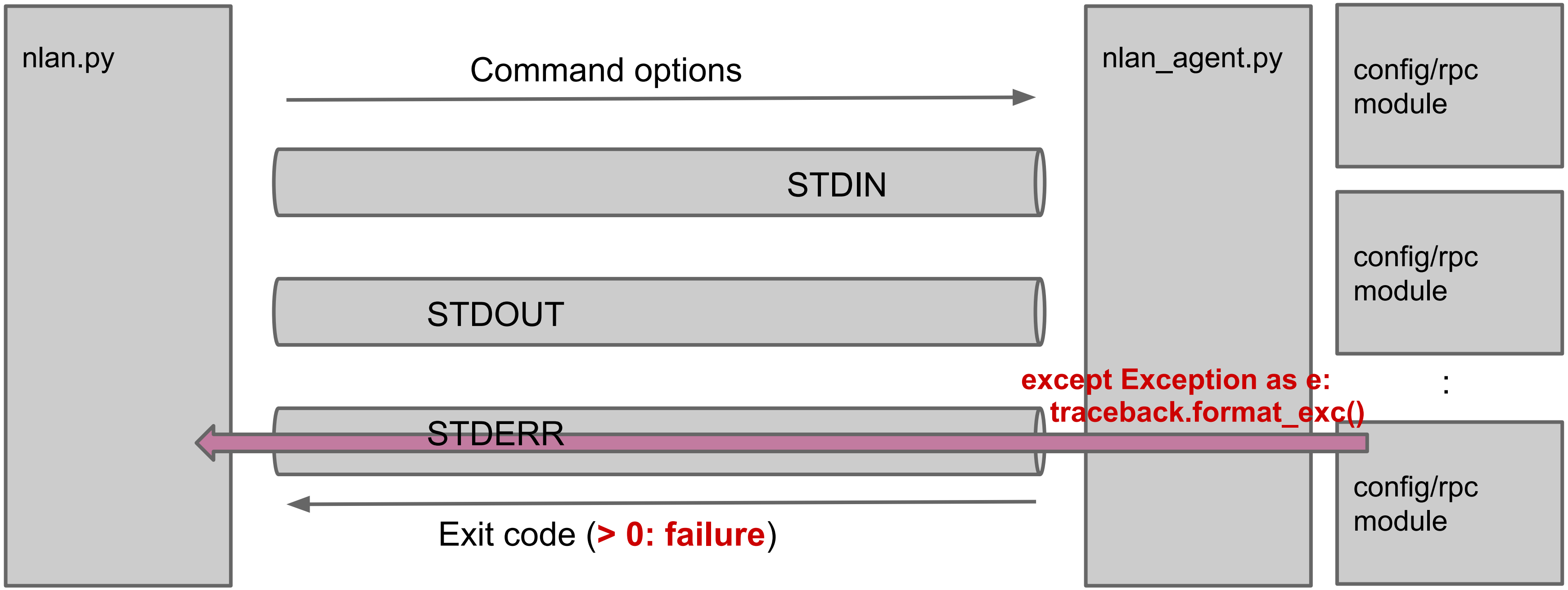
"OrderedDict([('bridges', {'ovs_bridges': 'enabled'}), ('gateway', {'network': 'eth2', 'rip': 'enabled'}), ('vxlan', {'remote_ips': ['192.168.1.103', '192.168.1.102', '192.168.1.104'], 'local_ip': '192.168.1.101'}), ('subnets', [{'peers': ['192.168.1.102', '192.168.1.103'], 'vid': 2, '_index': ['vni', 1], 'ip_vhost': '192.168.100.101/24', 'vni': 1, 'ip_dvr': OrderedDict([('addr', '192.168.100.1/24'), ('mode', 'dvr')])}, {'peers': ['192.168.1.102', '192.168.1.103'], 'vid': 3, '_index': ['vni', 103], 'ip_vhost': '10.0.3.101/24', 'vni': 103, 'ip_dvr': OrderedDict([('addr', '10.0.3.1/24'), ('mode', 'dvr')])}, {'peers': ['192.168.1.104'], 'vid': 1, '_index': ['vni', 1001], 'ip_vhost': '10.0.1.101/24', 'vni': 1001, 'ip_dvr': OrderedDict([('addr', '10.0.1.1/24'), ('mode', 'hub')])}])])"

- Imperative/declarative state representation.
- I don't use JSON, since NLAN is 100% Python implementation.

NLAN Request/Response over SSH

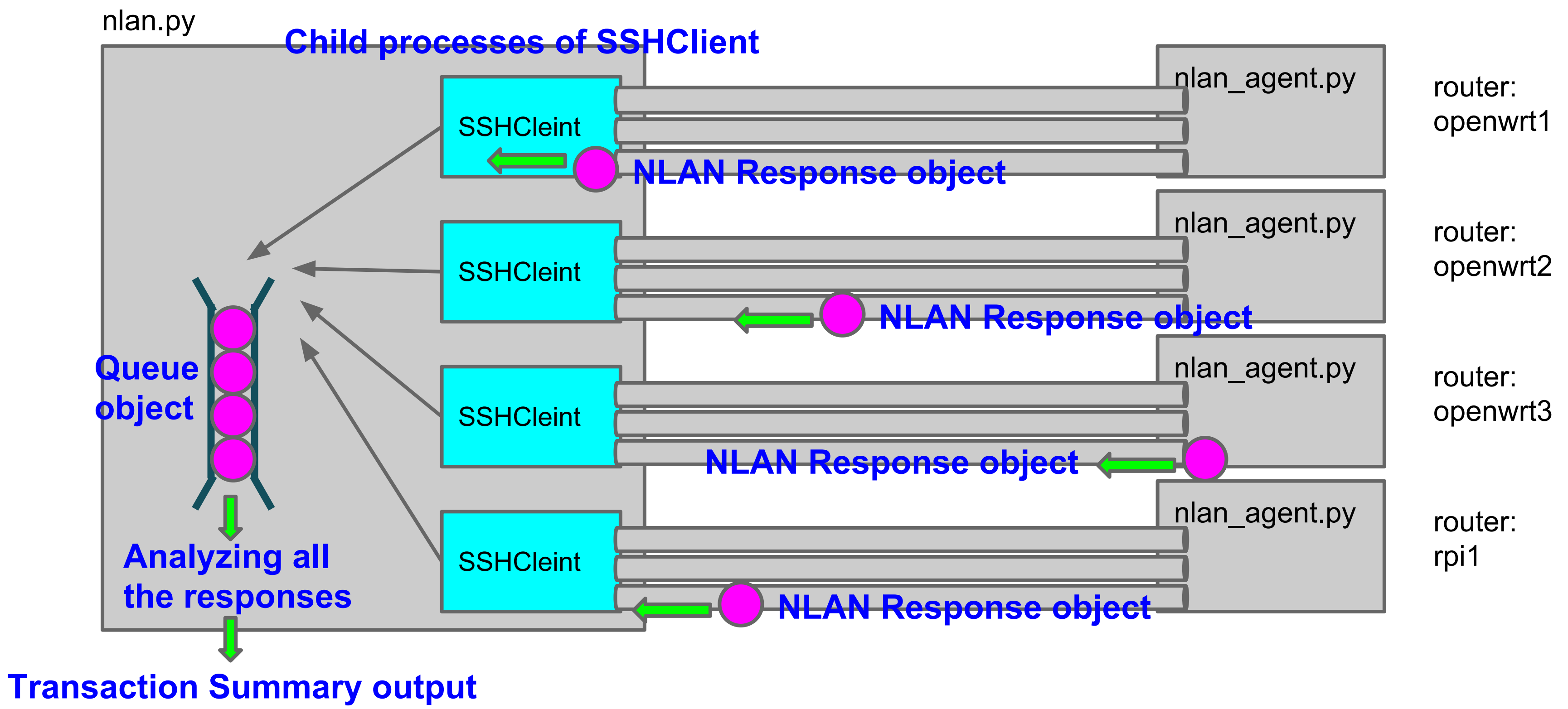


Exception handling



Trouble shooting becomes much easier by sending traceback.format_exc() and raw command outputs (plain text via stdout/stderr) to nlan.py.

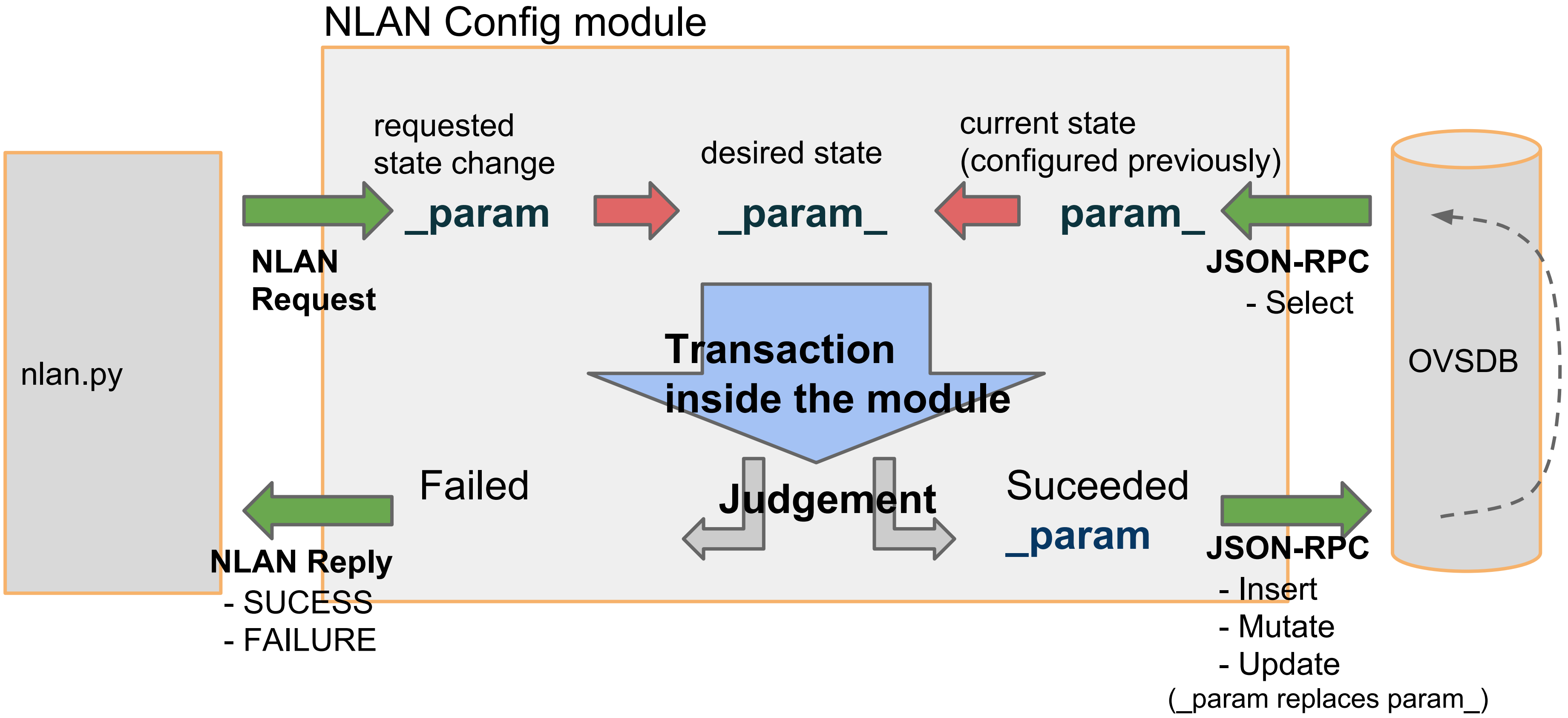
Parallel SSH sessions



Transaction Summary output

```
=+=+= Transaction Summary =+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

Start Time: 2014-04-22 20:48:31.835552

Router          Result    Elapsed Time
---------------------------------------
openwrt1         :-)       2.88(sec)
openwrt3         :-)       2.99(sec)
openwrt2         :-)       3.00(sec)
rpi1             :-)       3.08(sec)
```

CRUD operations inside NLAN config modules



Global variables (__dict__) generated by CRUD.params()

model

```
state -+- param a
       |
       +- param b
       |
       +- param c
       |
       +- param d
```

State parameters (for add/update operations)

			Global variables generated by Model.params()			
State params	Requested by Master	OVSDB	_param (Requested change)	_param_ (Desired state)	param_ (Current state in OVSDB)	Operation
a	1	None	_a=1	_a_=1	a_=None	add
b	2	1	_b=2	_b_=2	b_=1	update
c	None	1	_c=None	_c_=1	c_=1	
d	None	None	_d=None	_d_=None	d_=None	

State parameters (for delete operations)

			Global variables generated by Model.params()			
State params	Requested by Master	OVSDB	_param (Requested change)	_param_ (Desired state)	param_ (Current state in OVSDB)	Operation
a	1	None	_a=1	_a_=***	a_=None	(Never exists)
b	1	1	_b=1	_b_=None	b_=1	delete
c	None	1	_c=None	_c_=1	c_=1	
d	None	None	_d=None	_d_=None	d_=None	

Python conding in NLAN Config modules

nlan_agent.py

```
with oputil.CRUD(...):
        module.add()
```

```
with oputil.CRUD(...):
        module.delete()
```

```
with oputil.CRUD(...):
        module.update()
```

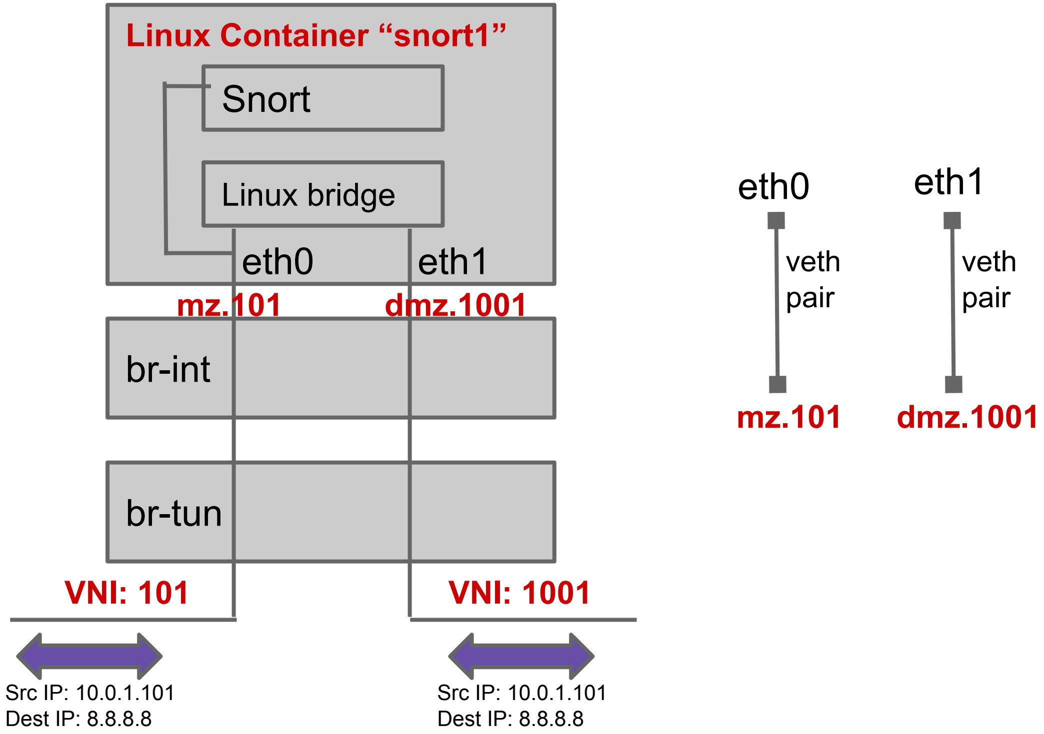
module.py

```
def add():
        if _param1:
                (execute local commands w/ _param1 and other
        _param_(s) as arguments)
        if _param2:
                (execute local commands w/ _param2 and other
        _param_(s) as arguments)
        if _param3:
                (execute local commands w/ _param3 and other
        _param_(s) as arguments)

def delete():
                :

def update():
                :
```

Every NLAN config module MUST implement add(), delete() and update() functions.

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 command usage (nlan.py)

Copy NLAN-agent-side modules to all the target routers (incl. NLAN/OVSDB schema and Linux init.d scripts):

$ nlan.py --scpmod

Initialize states at all the target routers:

$ nlan.py init.run

Ask all the target routers to transit to the desired states

$ nlan.py -G deploy

Rollback to the previous config

$ nlan.py init.run

$ nlan.py -R deploy

Command line CRUD (add/get/update/delete) operations

$ nlan.py -t openwrt1 --add subnets _index=101 vid=1 ip_dvr=mode:dvr,addr:10.0.1.1/24 ip_vhost=10.0.1.101/24

$ nlan.py -t openwrt1 --update subnets _index=101 ip_vhost=10.0.1.109/24

$ nlan.py -t openwrt1 --delete subnets _index=101 ip_vhost=10.0.1.109/24

$ nlan.py -t openwrt1 --get subnets _index=101 vid ip_dvr

Reboot all the target routers:

$ nlan.py system.reboot

NLAN command usage (nlan_agent.py)

Initialize states:

```
$ nlan_agent.py init.run
```

Command line CRUD (add/get/update/delete) operations

```
$ nlan_agent.py --add subnets _index=101 vid=1 ip_dvr=mode:dvr,addr:10.0.1.1/24 ip_vhost=10.0.1.101/24
$ nlan_agent.py --update subnets _index=101 ip_vhost=10.0.1.109/24
$ nlan_agent.py --delete subnets _index=101 ip_vhost=10.0.1.109/24
$ nlan_agent.py --get subnets _index=101 vid ip_dvr
```

Reboot all the target routers:

```
$ nlan.py system.reboot
```

REST APIs

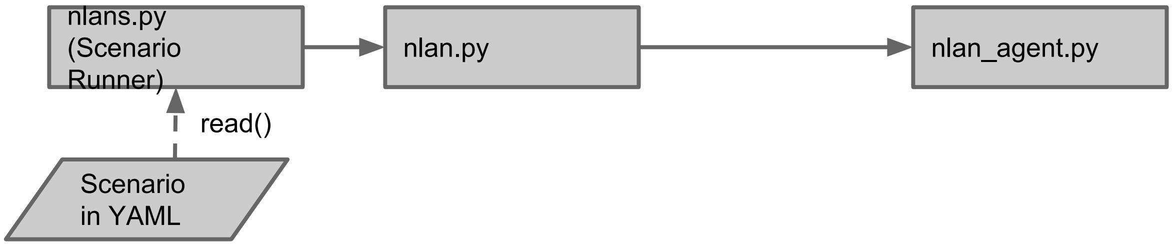
HTTP Method	NLAN CRUD operations	NLAN RPC operations
GET	CRUD: get URL: /<router>/config/<module>/<_index>	URL: /<router>/rpc/<module>/<func>
POST	CRUD: add URL: /<router>/config/<module>/<_index>	-
PUT	CRUD: update URL: /<router>/config/<module>/<_index>	-
DELETE	CRUD: delete URL: /<router>/config/<module>/<_index>	-
OPTIONS	Get NLAN schemas URL: none	-

REST APIs example

HTTP Method	URL	Query parameters
POST	/_ALL/rpc/test/echo	params=Hello!
OPTIONS	(none)	params=subnets
POST	/openwrt1/rpc/init/run	(none)
POST	/openwrt1/config/bridges	ovs_bridges=enabled
POST	/openwrt1/config/vxlan	local_ip=192.168.1.101&remote_ips=192.168.1.102,192.168.56.103
PUT	/openwrt1/config/vxlan	remote_ips=192.168.1.102,192.168.56.104
GET	/openwrt1/config/vxlan	params=remote_ips
POST	/openwrt1/config/subnets/101	vni=101&vid=1&ip_dvr=addr:10.0.1.1/24,mode:dvr
DELETE	/openwrt1/config/subnets/101	params=ip_dvr
POST	/openwrt1/rpc/db/state	(none)

Scenario Runner

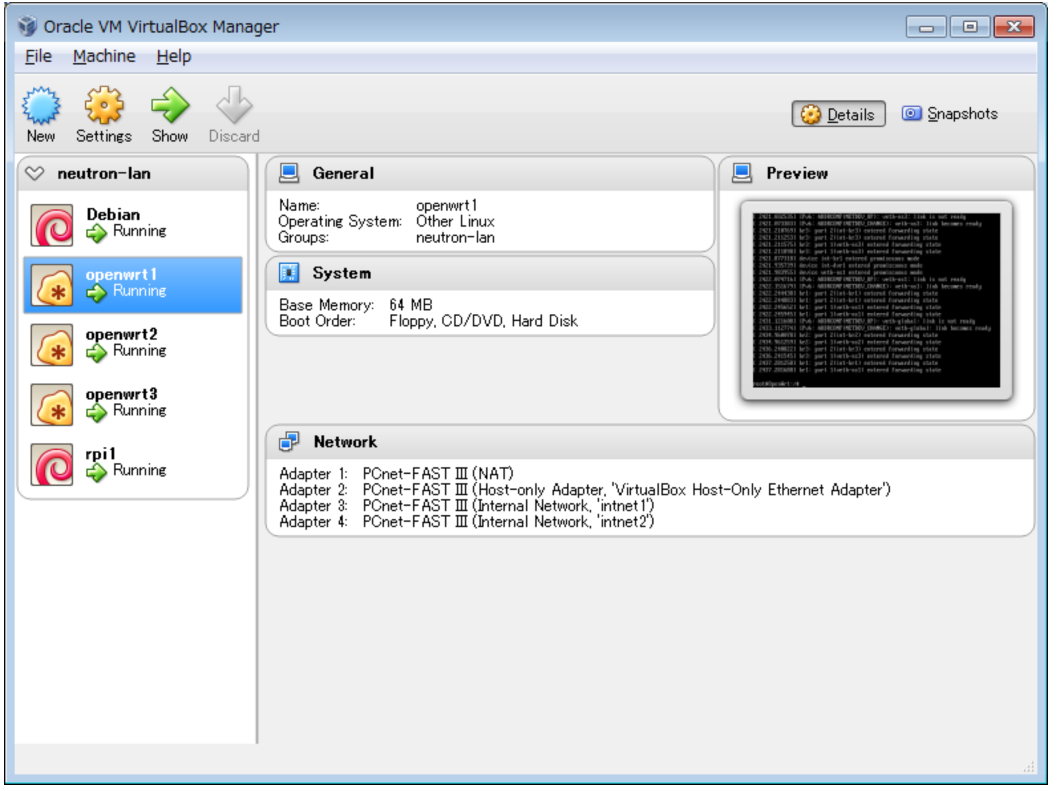
- nlans.py -- reads test scenarios and executes each test
- Test scenarios written in YAML
- Automatic test result confirmation
  - Inspired by Python's "unittest"
    - "assert"
    - "asserRaises"
    - ("assertOutputs" to be supported)



Simple RPC library

```
import rpc

# calls test.kwargs_test(...) at router ‘openwrt1’
rpc = rpc.RPC(module='test', func='kwargs_test', target='openwrt1')
result = rpc(1, b='Hello', d='World!')
print result

# calls test.echo(*args) at all routers on the roster
rpc = rpc.RPC(module='test', func='echo')
results = rpc('Hello World!')
for l in results:
    print l['router'], l['stdout']
```

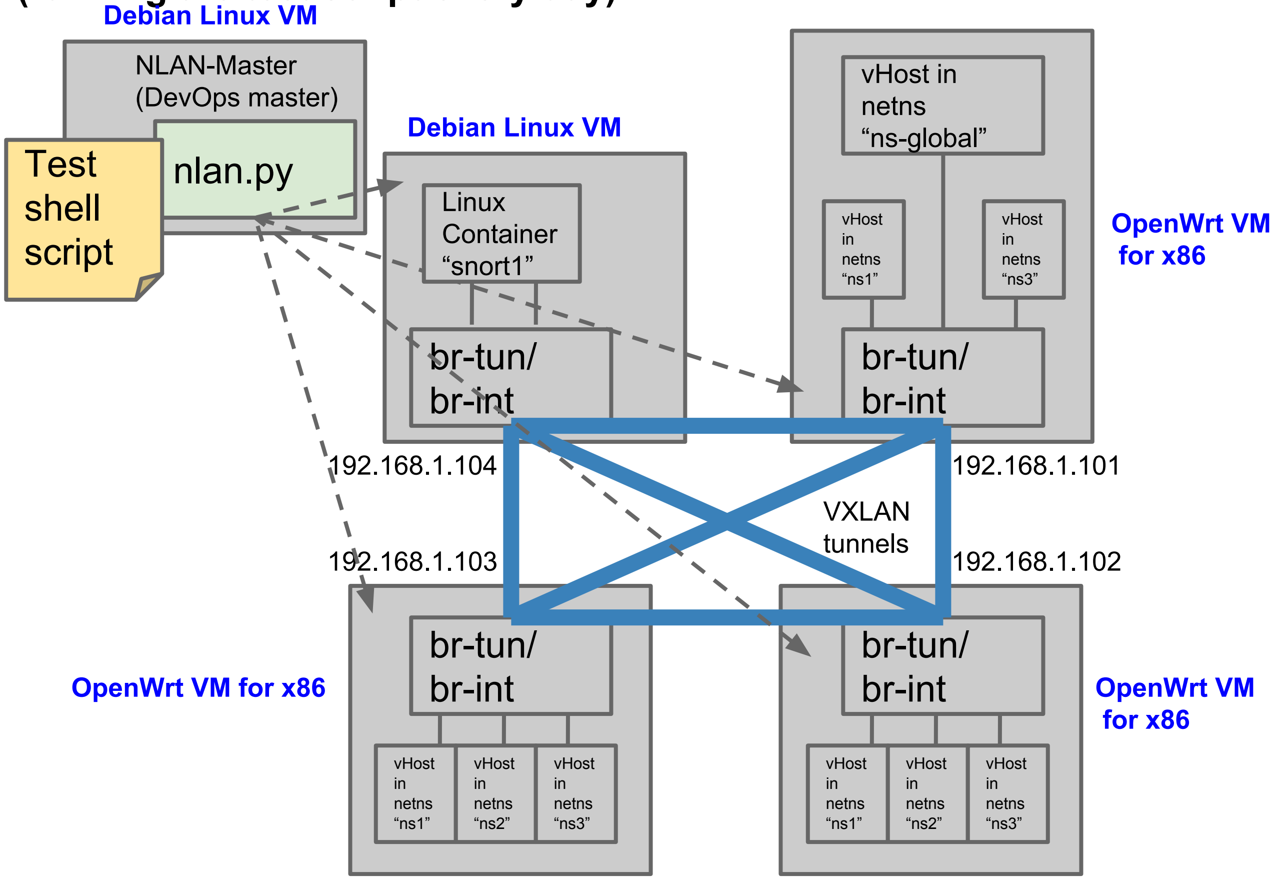
NLAN Software Development environment on VirtualBox



- Five VMs running on one Win7 PC.
  - Two Debian VMs
  - Three OpenWrt VMs
- OpenWrt image for x86
  - I built the kernel with Open vSwtich 2.0.0 and netns/veth/LXC support
  - Very light-weight Linux supporting Open vSwitch 2.0.0 ⇒ An alternative to mininet 2.0.0
- Network adapters setting
  - Internet access: “NAT”
  - Management: “Host-Only”
  - NLAN underlay: “Internal”

Integration Test environment on VirtualBox

(running the test script every day)



- Open vSwitch-based network more realisic than mininet 2.0
  - Every vSwitch with full-fledged(?) Linux
  - netns-based virtual hosts
- Mimics "Beremetal Switch"
- Integration Test scipt running on Debian Linux VM
  - Makes use of "nlan.py"