congress Documentation

Release

OpenStack Foundation

May 01, 2015

Contents

Contents:

CHAPTER 1

Congress Introduction and Installation

1.1 1. What is Congress

Congress is an open policy framework for the cloud. With Congress, a cloud operator can declare, monitor, enforce, and audit "policy" in a heterogeneous cloud environment. Congress get inputs from a cloud's various cloud services; for example in Openstack, Congress fetches information about VMs from Nova, and network state from Neutron, etc. Congress then feeds input data from those services into its policy engine where Congress verifies that the cloud's actual state abides by the cloud operator's policies. Congress is designed to work with **any policy** and **any cloud service**.

1.2 2. Why is Policy Important

The cloud is a collection of autonomous services that constantly change the state of the cloud, and it can be challenging for the cloud operator to know whether the cloud is even configured correctly. For example,

- The services are often independent from each other, and do not support transactional consistency across services, so a cloud management system can change one service (create a VM) without also making a necessary change to another service (attach the VM to a network). This can lead to incorrect behavior.
- Other times, we have seen a cloud operator allocate cloud resources and then forget to clean them up when the resources are no longer in use, effectively leaving garbage around the system and wasting resources.
- The desired cloud state can also change over time. For example, if a security vulnerability appears in Linux version X, then all machines with version X that were ok in the past are now in an undesirable state. A version number policy would detect all the machines in that undesirable state. This is a trivial example, but the more complex the policy, the more helpful a policy system becomes.

Congress's job is to help people manage that plethora of state across all cloud services with a susinct policy language.

1.3 3. Using Congress

Setting up Congress involves writing policies and configuring Congress to fetch input data from the cloud services. The cloud operator writes policy in the Congress policy language, which receives input from the cloud services in the form of tables. The language itself resembles datalog. For more detail about the policy language and data format see *Policy*.

To add a service as an input data source, the cloud operator configures a Congress "driver", and the driver queries the service. Congress already has drivers for several types of service, but if a cloud operator needs to use an unsupported service, she can write a new driver without much effort, and probably contribute the driver to the Congress project so that no one else needs to write the same driver.

Finally, when using Congress, the cloud operator must choose what Congress should do with the policy it has been given:

- **monitoring**: detect violations of policy and provide a list of those violations
- **proactive enforcement**: prevent violations before they happen (functionality that requires other services to consult with Congress before making changes)
- **reactive enforcement**: correct violations after they happen (a manual process that Congress tries to simplify)

In the future, Congress will also help the cloud operator audit policy (analyze the history of policy and policy violations).

Congress is free software and is licensed with Apache.

- Free software: Apache license

1.4 4. Installing Congress

There are 2 ways to install Congress.

- As part of devstack. Get Congress running alongside other OpenStack services like Nova and Neutron, all on a single machine. This is a great way to try out Congress for the first time.
- Standalone. Get Congress running all by itself. Congress works well with other OpenStack services but can be deployed without them.

1.4.1 4.1 Devstack-install

The contrib/devstack/ directory contains the files necessary to integrate Congress with devstack.

To install, make sure you have *git* installed. Then:

```
$ git clone https://git.openstack.org/openstack-dev/devstack
 (Or set env variable DEVSTACKDIR to the location to your devstack code)

$ wget http://git.openstack.org/cgit/openstack/congress/plain/contrib/devstack/prepare_devstack.sh

$ chmod u+x prepare_devstack.sh

$ ./prepare_devstack.sh
```

Configure ENABLED_SERVICES in the devstack/localrc file (make sure to include congress):

```
ENABLED_SERVICES=congress,g-api,g-reg,key,n-api,n-crt,n-obj,n-cpu,n-sch,n-cauth,horizon,mysql,rabbit,
```

Run devstack as normal. Note: the default data source configuration assumes the admin password is 'password':

```
$ ./stack.sh
```

1.4.2 4.2 Standalone-install

Install the following software, if you haven't already.

- python 2.7: https://www.python.org/download/releases/2.7/
- pip: https://pip.pypa.io/en/latest/installing.html

- java: http://java.com (any reasonably current version should work) On Ubuntu: apt-get install default-jre
- Additionally:

```
$ apt-get install git gcc python-dev libxml2 libxslt1-dev libzip-dev mysql-server python-mysqldb
```

Clone Congress:

```
$ git clone https://github.com/openstack/congress.git
$ cd congress
```

Install Source code:

```
$ sudo python setup.py install
```

Configure congress:

```
(Assume you put config files in /etc/congress)

$ sudo mkdir -p /etc/congress
$ sudo mkdir -p /etc/congress/snapshot
$ sudo cp etc/api-paste.ini /etc/congress
$ sudo cp etc/policy.json /etc/congress
$ sudo cp etc/congress.conf.sample /etc/congress/congress.conf

Uncomment policy_path and add drivers in /etc/congress/congress.conf [DEFAULT] section:

drivers = congress.datasources.neutronv2_driver.NeutronV2Driver,congress.datasources.glancev2_driver

Modify [keystone_authtoken] and [database] according to your environment.

For setting congress with "noauth":
  Add the following line to [DEFAULT] section in /etc/congress/congress.conf

  auth_strategy = noauth

  Also, might want to delete/comment [keystone_authtoken] section in
  /etc/congress/congress.conf
```

Create database:

```
$ mysql -u root -p
$ mysql> CREATE DATABASE congress;
$ mysql> GRANT ALL PRIVILEGES ON congress.* TO 'congress'@'localhost' \
         IDENTIFIED BY 'CONGRESS_DBPASS';
$ mysql> GRANT ALL PRIVILEGES ON congress.* TO 'congress'@'%' \
         IDENTIFIED BY 'CONGRESS_DBPASS';

(Configure congress.conf with db information)

Push down schema
$ sudo congress-db-manage --config-file /etc/congress/congress.conf upgrade head
```

Setup congress accounts:

```
(You should change parameters according to your environment)

$ ADMIN_ROLE=$(openstack role list | awk "/ admin / { print \$2 }")
$ SERVICE_TENANT=$(openstack project list | awk "/ admin / { print \$2 }")
$ CONGRESS_USER=$(openstack user create --password password --project admin \
  --email "congress@example.com" congress)
```

```
$ openstack role add $ADMIN_ROLE --user $CONGRESS_USER --project \
  $SERVICE_TENANT
$ CONGRESS_SERVICE=$(openstack service create congress --type "policy" \
  --description "Congress Service")
$ openstack endpoint create $CONGRESS_SERVICE \
  --region RegionOne \
  --publicurl http://127.0.0.1:1789/ \
  --adminurl http://127.0.0.1:1789/ \
  --internalurl http://127.0.0.1:1789/
```

Configure datasource drivers:

```
First make sure you have congress client (project python-congressclient) installed.
Run this command for every service that congress will poll for data:

$ openstack congress datasource create $SERVICE "$SERVICE" \
  --config username=$OS_USERNAME \
  --config tenant_name=$OS_TENANT_NAME \
  --config password=$OS_PASSWORD \
  --config auth_url=http://$SERVICE_HOST:5000/v2.0

Please note that the service name $SERVICE should match the id of the datasource driver,
e.g. "neutronv2" for Neutron and "glancev2" for Glance. $OS_USERNAME, $OS_TENANT_NAME,
$OS_PASSWORD and $SERVICE_HOST are used to configure the related datasource driver
so that congress knows how to talk with the service.
```

Start congress:

```
$ sudo /usr/local/bin/congress-server --debug
```

Install test harness:

```
$ pip install 'tox<1.7'
```

Run unit tests:

```
$ tox -epy27
```

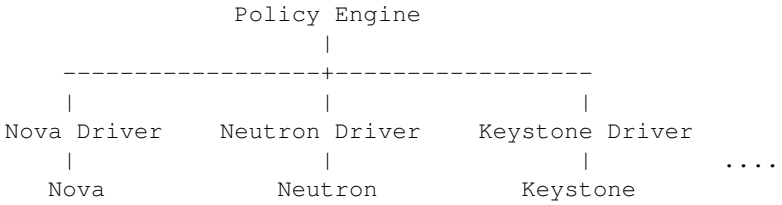
Read the HTML documentation:

```
$ make docs
Open doc/html/index.html in a browser
```

CHAPTER 2

Architecture

Congress consists of the Congress policy engine and a driver for any number of other cloud services that act as sources of information about the cloud:



2.1 1. Cloud Services, Drivers, and State

A service is anything that manages cloud state. For example, OpenStack components like Nova, Neutron, Cinder, Swift, Heat, and Keystone are all services. Software like ActiveDirectory, inventory management systems, anti-virus scanners, intrusion detection systems, and relational databases are also services.

Congress uses a driver to connect each service to the policy engine. A driver fetches cloud state from its respective cloud service, and then feeds that state to the policy engine in the form of tables. A table is a collection of rows; each row is a collection of columns; each row-column entry stores simple data like numbers or strings.

For example, the Nova driver periodically makes API calls to Nova to fetch the list of virtual machines in the cloud, and the properties associated with each VM. The Nova driver then populates a table in the policy engine with the Nova state. For example, the Nova driver populates a table like this::

```
-------------------------------------------
| VM id | Name | Status | Power State | ... |
-------------------------------------------
| 12345 | foo  | ACTIVE | Running     | ... |
| ...   |      |        |             |     |
-------------------------------------------
```

The state for each service will be unique to that service. For Neutron, the existing logical networks, subnets, and ports make up that state. For Nova, the existing VMs along with their disk and memory space make up that state. For an anti-virus scanner, the results of all its most recent scans are the state. The *Services* section describes services and drivers in more detail.

2.2 2. Policy

A Congress policy defines all those states of the cloud that are permitted: all those combinations of service tables that are possible when the cloud is behaving as intended. Since listing the permitted states explicitly is an insurmountable task, policy authors describe the permitted states implicitly by writing a collection of if-then statements that are always true when the cloud is behaving as intended.

More precisely, Congress uses Datalog as its policy language. Datalog is a declarative language and is similar in many ways to SQL, Prolog, and first-order logic. Datalog has been the subject of research and development for the past 50 years, which means there is a wealth of tools, algorithms, and deployment experience surrounding it. The *Policy* section describes policies in more detail.

2.3 3. Capabilities

Once Congress is given a policy, it has three capabilities:

- monitoring the cloud for policy violations
- preventing violations before they occur
- correcting violations after the fact

In the future, Congress will also record the history of policy and its violations for the purpose of audit. The *Monitoring and Enforcement* section describes these capabilities in more detail.

2.4 4. Congress Server and API

Congress runs as a standalone server process and presents a RESTful API for clients; drivers run as part of the server. Instructions for installing and starting the Congress server can be found in the *Readme* file.

The API allows clients to perform the following operations:

- insert and delete policy statements
- check for policy violations
- ask hypothetical questions: if the cloud were to undergo these changes, would that cause any policy violations?
- execute actions

The *API* section describes the API in more detail.

CHAPTER 3

Cloud Services

3.1 1. Congress Works With All Services

Congress will work with any cloud service, as long as Congress can represent the service's state in *table* format. A table is a collection of rows, where each row is a collection of columns, and each row-column entry contains a string or a number.

For example, Neutron contains a mapping between IP addresses and the ports they are assigned to; neutron represents this state as the following table.:

```
====================================== ==========
ID                                     IP
====================================== ==========
"66dafde0-a49c-11e3-be40-425861b86ab6" "10.0.0.1"
"66dafde0-a49c-11e3-be40-425861b86ab6" "10.0.0.2"
"73e31d4c-a49c-11e3-be40-425861b86ab6" "10.0.0.3"
====================================== ==========
```

3.2 2. Drivers

To plug a new service into Congress, you write a small piece of code, called a *driver*, that queries the new service (usually through API calls) and translates the service state into tables of data. Out of the box Congress includes drivers for a number of common services (see below).

For example, the driver for Neutron invokes the Neutron API calls that list networks, ports, security groups, and routers. The driver translates each of the JSON objects that the API calls return into tables (where in Python a table is a list of tuples). The Neutron driver is implemented here:

```
congress/datasources/neutronv2_driver.py
```

Once the driver is available, you install it into Congress, you configure it (such as with an IP address/port/username), and you write policy that references the tables populated by that driver.

3.2.1 2.1 Driver installation

To install a new driver, you must add its location to the Congress configuration file and restart the server. Congress has a single configuration parameter (called *drivers*) that is a list of all the installed drivers. To install a new driver, simply add to this list and restart.

For example, to install the Neutron driver, you add the following to the list of drivers in the configuration file:

```
congress.datasources.neutronv2_driver.NeutronV2Driver
```

If you have Nova and Neutron installed, you configure Congress as:

```
drivers = congress.datasources.neutronv2_driver.NeutronV2Driver,congress.datasources.nova_driver.Nova
```

3.2.2 2.2 Driver configuration and writing policy

Once the driver code is in place, you can use it to create a *datasource* whose data is available to Congress policies. To create a datasource, you use the API and provide a unique name (the name you will use in policy to refer to the service), the name of the datasource driver you want to use, and additional connection details needed by your service (such as an IP and a username/password).

For example, using the Congress CLI, you can create a datasource named 'neutron' using the 'neutronv2' driver:

```
$ openstack congress datasource create <driver_name> <datasource_name>
   --config username=<username>
   --config password=<password>
   --config tenant_name=<tenant>
   --config auth_url=<url_authentication>

$ openstack congress datasource create neutronv2 neutron_test
  --config username=neutron
  --config password=password
  --config tenant_name=cloudservices
  --config auth_url=http://10.10.10.10:5000/v2.0
```

And if you had a second instance of Neutron running to manage your production network, you could create a second datasource (named say 'neutron_prod') using the neutronv2 driver so that you could write policy over both instances of Neutron.

When you write policy, you would use the name 'neutron:ports' to reference the 'ports' table generated by the 'neutron' datasource, and you use 'neutron:networks' to reference the 'networks' table generated by the 'neutron' datasource. Similarly, you use 'neutron_prod:ports' and 'neutron_prod:networks' to reference the tables populated by the 'neutron_prod' datasource. (More details about writing policy can be found in the *Policy* section.)

3.3 3. Currently Supported Drivers

Congress currently has drivers for each of the following services. Each driver has a differing degree of coverage for the available API calls.

- OpenStack Ceilometer
- OpenStack Cinder
- OpenStack Glance (v2)
- OpenStack Ironic
- OpenStack Keystone
- OpenStack Murano
- OpenStack Neutron (v2)
- OpenStack Nova
- OpenStack Swift

- Cloud Foundry
- Plexxi
- vCenter

Using the API or CLI, you can review the list of tables and columns that a driver supports. Roughly, you can think of each table as a collection of objects (like networks or servers), and the columns of that table as the attributes of those objects (like name, status, or ID). The value of each row-column entry is a (Python) string or number. If the attribute as returned by the API call is a complex object, that object is flattened into its own table (or tables).

For example:

```
$ openstack congress datasource schema show nova
+--------------+-------------------------------------------------+
| table        | columns                                         |
+--------------+-------------------------------------------------+
| flavors      | {'name': 'id', 'description': 'None'},          |
|              | {'name': 'name', 'description': 'None'},        |
|              | {'name': 'vcpus', 'description': 'None'},       |
|              | {'name': 'ram', 'description': 'None'},         |
|              | {'name': 'disk', 'description': 'None'},        |
|              | {'name': 'ephemeral', 'description': 'None'},   |
|              | {'name': 'rxtx_factor', 'description': 'None'}  |
|              |                                                 |
| hosts        | {'name': 'host_name', 'description': 'None'},   |
|              | {'name': 'service', 'description': 'None'},     |
|              | {'name': 'zone', 'description': 'None'}         |
|              |                                                 |
| floating_IPs | {'name': 'fixed_ip', 'description': 'None'},    |
|              | {'name': 'id', 'description': 'None'},          |
|              | {'name': 'ip', 'description': 'None'},          |
|              | {'name': 'host_id', 'description': 'None'},     |
|              | {'name': 'pool', 'description': 'None'}         |
|              |                                                 |
| servers      | {'name': 'id', 'description': 'None'},          |
|              | {'name': 'name', 'description': 'None'},        |
|              | {'name': 'host_id', 'description': 'None'},     |
|              | {'name': 'status', 'description': 'None'},      |
|              | {'name': 'tenant_id', 'description': 'None'},   |
|              | {'name': 'user_id', 'description': 'None'},     |
|              | {'name': 'image_id', 'description': 'None'},    |
|              | {'name': 'flavor_id', 'description': 'None'}    |
|              |                                                 |
+--------------+-------------------------------------------------+
```

3.4 4. Writing a Datasource Driver

This section is a tutorial for those of you interested in writing your own datasource driver. It can be safely skipped otherwise.

3.4.1 4.1 Implementing a Datasource Driver

All the Datasource drivers extend the code found in:

`congress/datasources/datasource_driver.py`

Typically, you will create a subclass of DataSourceDriver; each instance of that class will correspond to a different service using that driver.

The following steps detail how to implement a datasource driver.

1. Create a new Python module and include 1 static method

   `d6service(name, keys, inbox, datapath, args)`

   When a service is created, Congress calls `d6service` on the appropriate driver module to construct an instance of DataSourceDriver tailored for that service.

   `name`, `keys`, `inbox`, and `datapath` are all arguments that should be passed unaltered to the constructor of the DataSourceDriver subclass.

2. Create a subclass of :code'DataSourceDriver'.

   ```
   from congress.datasources.datasource_driver import DataSourceDriver
   ```

   ```
   class MyDriver(DataSourceDriver)
   ```

3. Implement the constructor `MyDriver.__init__()`

   ```
   def __init__(name, keys, inbox, datapath, args)
   ```

   You must call the DataSourceDriver's constructor.

   ```
   super(NeutronDriver, self).__init__(name, keys, inbox=inbox,
   datapath=datapath, poll_time=poll_time, creds
   ```

4. Implement the function `MyDriver.update_from_datasource()`

   ```
   def update_from_datasource(self)
   ```

   This function is called to update `self.state` to reflect the new state of the service. `self.state` is a dictionary that maps a tablename (as a string) to a set of tuples (to a collection of tables). Each tuple element must be either a number or string. This function implements the polling logic for the service.

5. By convention, it is useful for debugging purposes to include a `main` that calls update_from_datasource, and prints out the raw API results along with the tables that were generated.

3.4.2 4.2 Converting API results into Tables

Since Congress requires the state of each dataservice to be represented as tables, we must convert the results of each API call (which may be comprised of dictionaries, lists, dictionaries embedded within lists, etc.) into tables.

4.2.1 Convenience translators

Congress provides a translation method to make the translation from API results into tables convenient. The translation method takes a description of the API data structure, and converts objects of that structure into rows of one or more tables (depending on the data structure). For example, this is a partial snippet from the Neutron driver:

```
networks_translator = {
    'translation-type': 'HDICT',
    'table-name': 'networks',
    'selector-type': 'DICT_SELECTOR',
    'field-translators':
        ({'fieldname': 'id', 'translator': value_trans},
         {'fieldname': 'name', 'translator': value_trans},
```

```
{'fieldname': 'tenant_id', 'translator': value_trans},
{'fieldname': 'subnets', 'col': 'subnet_group_id',
 'translator': {'translation-type': 'LIST',
                'table-name': 'networks.subnets',
                'id-col': 'subnet_group_id',
                'val-col': 'subnet',
                'translator': value_trans}})}
```

This networks_translator describes a python dictionary data structure that contains four keys: id, name, tenant_id, and subnets. The value for the subnets key is a list of subnet_group_ids each of which is a number. For example:

{ "id": 1234, "name": "Network Foo", "tenant_id": 5678, "subnets": [100, 101] }

Given the networks_translator description, the translator creates two tables. The first table is named "networks" with a column for name, subnets, tenant_id, and id. The second table will be named "networks.subnet" and will contain two columns, one containing the subnet_group_id, and the second containing an ID that associates the row in the network to the rows in the networks.subnets table.

To use the translation methods, the driver defines a translator such as networks_translator and then passes the API response objects to translate_objs() which is defined in congress/datasources/datasource_driver.py See congress/datasources/neutron_driver.py as an example.

4.2.2 Custom data conversion

The convenience translators may be insufficient in some cases, for example, the data source may provide data in an unusual format, the convenience translators may be inefficient, or the fixed translation method may result in an unsuitable table schema. In such cases, a driver may need to implement its own translation. In those cases, we have a few recommendations.

Recommendation 1: Row = object. Typically an API call will return a collection of objects (e.g. networks, virtual machines, disks). Conceptually it is convenient to represent each object with a row in a table. The columns of that row are the attributes of each object. For example, a table of all virtual machines will have columns for memory, disk, flavor, and image.

Table: virtual_machine

ID	Memory	Disk	Flavor	Image
66dafde0-a49c-11e3-be40-425861b86ab6	256GB	1TB	1	83e31d4c-a49c-11e3-be40-425861b86ab6
73e31d4c-a49c-11e3-be40-425861b86ab6	10GB	2TB	2	93e31d4c-a49c-11e3-be40-425861b86ab6

Recommendation 2. Avoid wide tables. Wide tables (i.e. tables with many columns) are hard to use for a policy-writer. Breaking such tables up into smaller ones is often a good idea. In the above example, we could create 4 tables with 2 columns instead of 1 table with 5 columns.

Table: virtual_machine.memory

ID	Memory
66dafde0-a49c-11e3-be40-425861b86ab6	256GB
73e31d4c-a49c-11e3-be40-425861b86ab6	10GB

Table: virtual_machine.disk

ID	Disk
66dafde0-a49c-11e3-be40-425861b86ab6	1TB
73e31d4c-a49c-11e3-be40-425861b86ab6	2TB

Table: virtual_machine.flavor

ID	Flavor
66dafde0-a49c-11e3-be40-425861b86ab6	1
73e31d4c-a49c-11e3-be40-425861b86ab6	2

Table: virtual_machine.image

ID	Image
66dafde0-a49c-11e3-be40-425861b86ab6	83e31d4c-a49c-11e3-be40-425861b86ab6
73e31d4c-a49c-11e3-be40-425861b86ab6	93e31d4c-a49c-11e3-be40-425861b86ab6

Recommendation 3. Try these design patterns. Below we give a few design patterns. Notice that when an object has an attribute whose value is a structured object itself (e.g. a list of dictionaries), we must recursively flatten that subobject into tables.

- A List of dictionary converted to tuples

  Original data:

```
[{'key1':'value1','key2':'value2'},
 {'key1':'value3','key2':'value4'}
]
```

  Tuple:

```
[('value1', 'value2'),
 ('value3', 'value4')
]
```

- List of Dictionary with a nested List

  Original data:

```
[{'key1':'value1','key2':['v1','v2']},
 {'key1':'value2','key2':['v3','v4']}
]
```

  Tuple:

```
[('value1', 'uuid1'),
 ('value1', 'uuid2'),
 ('value2', 'uuid3'),
 ('value2', 'uuid4')
]

[('uuid1', 'v1'),
 ('uuid2', 'v2'),
 ('uuid3', 'v3'),
 ('uuid4', 'v4')
]
```

  Note : uuid* are congress generated uuids

- List of Dictionary with a nested dictionary

  Original data:

```
[{'key1':'value1','key2':{'k1':'v1'}},
 {'key1':'value2','key2':{'k1':'v2'}}
]
```

  Tuple:

```
[('value1', 'uuid1'),
 ('value2', 'uuid2')
]

[('uuid1', 'k1', 'v1'),
 ('uuid2', 'k1', 'v2'),
]
```

Note : uuid* are congress generated uuids

3.4.3 4.3 Writing a Datasource driver test

Once you've written a driver, you'll want to add a unit test for it. To help, this section describes how the unit test for the Glance driver works. Here are the relevant files.

- Driver code: congress/datasources/glance_v2driver.py
- Test code: congress/tests/datasources/test_glancev2_driver.py (appearing in full at the end of this section)

The test code has two methods: setUp() and test_update_from_datasource().

4.3.1 Glance setup

We begin our description with the setUp() method of the test.

```
def setUp(self):
```

First the test creates a fake (actually a mock) Keystone. Most clients talk to Keystone, so having a fake one seems to be necessary to make the Glance client work properly.

```
self.keystone_client_p = mock.patch(
    "keystoneclient.v2_0.client.Client")
self.keystone_client_p.start()
```

Next the test creates a fake Glance client. Glance is an OpenStack service that stores (among other things) operating system Images that you can use to create a new VM. The Glance datasource driver makes a call to <glance-client>.images.list() to retrieve the list of those images, and then turns that list of images into tables. The test creates a fake Glance client so it can control the return value of <glance-client>.images.list().

```
self.glance_client_p = mock.patch("glanceclient.v2.client.Client")
self.glance_client_p.start()
```

Next the test instantiates the GlanceV2Driver class, which contains the code for the Glance driver. Passing 'poll_time' as 0 is probably unnecessary here, but it tells the driver not to poll automatically. Passing 'client' is important because it tells the GlanceV2Driver class to use a mocked version of the Glance client instead of creating its own.

```
args = helper.datasource_openstack_args()
args['poll_time'] = 0
args['client'] = mock.MagicMock()
self.driver = glancev2_driver.GlanceV2Driver(args=args)
```

Next the test defines which value it wants <glance-client>.images.list() to return. The test itself will check if the Glance driver code properly translates this return value into tables. So this is the actual input to the test. Either you can write this by hand, or you can run the heat-client and print out the results.

```
self.mock_images = {'images': [
    {u'checksum': u'9e486c3bf76219a6a37add392e425b36',
     u'container_format': u'bare',
     u'created_at': u'2014-10-01T20:28:08Z',
     ...
```

4.3.2 Glance test

test_update_from_datasource() is the actual test, where we have the datasource driver grab the list of Glance images and translate them to tables. The test runs the update_from_datasource() method like normal except it ensures the return value of <glance-client>.images.list() is self.mock_images.

```
def test_update_from_datasource(self):
```

The first thing the method does is set the return value of self.driver.glance.images.list() to self.mock_images['images']. Then it calls update_from_datasource() in the usual way, which translates self.mock_images['images'] into tables and stores the result into the driver's self.state dictionary.

```
with mock.patch.object(self.driver.glance.images, "list") as img_list:
    img_list.return_value = self.mock_images['images']
    self.driver.update_from_datasource()
```

Next the test defines the tables that update_from_datasource() should construct. Actually, the test defines the expected value of Glance's self.state when update_from_datasource() finishes. Remember that self.state is a dictionary mapping a table name to the set of tuples that belong to the table. For Glance, there's just one table: 'images', and so the expected self.state is a dictionary with one key 'images' and one value: a set of tuples.

```
expected = {'images': set([
    (u'6934941f-3eef-43f7-9198-9b3c188e4aab',
     u'active',
     u'cirros-0.3.2-x86_64-uec',
     u'ami',
     u'2014-10-01T20:28:06Z',
     u'2014-10-01T20:28:07Z',
     u'ami',
     u'4dfdcf14a20940799d89c7a5e7345978',
     'False',
     0,
     0,
     u'4eada48c2843d2a262c814ddc92ecf2c',
     25165824,
     u'/v2/images/6934941f-3eef-43f7-9198-9b3c188e4aab/file',
     u'15ed89b8-588d-47ad-8ee0-207ed8010569',
     u'c244d5c7-1c83-414c-a90d-af7cea1dd3b5',
     u'/v2/schemas/image',
     u'public'),
     ...
```

At this point in the test, update_from_datasource() has already been run, so all it does is check that the driver's self.state has the expected value.

```
self.assertEqual(self.driver.state, expected)
```

4.3.3 Glance test code in full

```
import mock

from congress.datasources import glancev2_driver
from congress.tests import base
from congress.tests import helper


class TestGlanceV2Driver(base.TestCase):

    def setUp(self):
        super(TestGlanceV2Driver, self).setUp()
        self.keystone_client_p = mock.patch(
            "keystoneclient.v2_0.client.Client")
        self.keystone_client_p.start()
        self.glance_client_p = mock.patch("glanceclient.v2.client.Client")
        self.glance_client_p.start()

        args = helper.datasource_openstack_args()
        args['poll_time'] = 0
        args['client'] = mock.MagicMock()
        self.driver = glancev2_driver.GlanceV2Driver(args=args)

        self.mock_images = {'images': [
            {u'checksum': u'9e486c3bf76219a6a37add392e425b36',
             u'container_format': u'bare',
             u'created_at': u'2014-10-01T20:28:08Z',
             u'disk_format': u'qcow2',
             u'file': u'/v2/images/c42736e7-8b09-4906-abd2-d6dc8673c297/file',
             u'id': u'c42736e7-8b09-4906-abd2-d6dc8673c297',
             u'min_disk': 0,
             u'min_ram': 0,
             u'name': u'Fedora-x86_64-20-20140618-sda',
             u'owner': u'4dfdcf14a20940799d89c7a5e7345978',
             u'protected': False,
             u'schema': u'/v2/schemas/image',
             u'size': 209649664,
             u'status': u'active',
             u'tags': ['type=xen2', 'type=xen'],
             u'updated_at': u'2014-10-01T20:28:09Z',
             u'visibility': u'public'},
            {u'checksum': u'4eada48c2843d2a262c814ddc92ecf2c',
             u'container_format': u'ami',
             u'created_at': u'2014-10-01T20:28:06Z',
             u'disk_format': u'ami',
             u'file': u'/v2/images/6934941f-3eef-43f7-9198-9b3c188e4aab/file',
             u'id': u'6934941f-3eef-43f7-9198-9b3c188e4aab',
             u'kernel_id': u'15ed89b8-588d-47ad-8ee0-207ed8010569',
             u'min_disk': 0,
             u'min_ram': 0,
             u'name': u'cirros-0.3.2-x86_64-uec',
             u'owner': u'4dfdcf14a20940799d89c7a5e7345978',
             u'protected': False,
             u'ramdisk_id': u'c244d5c7-1c83-414c-a90d-af7cea1dd3b5',
             u'schema': u'/v2/schemas/image',
             u'size': 25165824,
             u'status': u'active',
```

```
            u'tags': [],
            u'updated_at': u'2014-10-01T20:28:07Z',
            u'visibility': u'public'}]}

    def test_update_from_datasource(self):
        with mock.patch.object(self.driver.glance.images, "list") as img_list:
            img_list.return_value = self.mock_images['images']
            self.driver.update_from_datasource()
        expected = {'images': set([
            (u'6934941f-3eef-43f7-9198-9b3c188e4aab',
             u'active',
             u'cirros-0.3.2-x86_64-uec',
             u'ami',
             u'2014-10-01T20:28:06Z',
             u'2014-10-01T20:28:07Z',
             u'ami',
             u'4dfdcf14a20940799d89c7a5e7345978',
             'False',
             0,
             0,
             u'4eada48c2843d2a262c814ddc92ecf2c',
             25165824,
             u'/v2/images/6934941f-3eef-43f7-9198-9b3c188e4aab/file',
             u'15ed89b8-588d-47ad-8ee0-207ed8010569',
             u'c244d5c7-1c83-414c-a90d-af7cea1dd3b5',
             u'/v2/schemas/image',
             u'public'),
            (u'c42736e7-8b09-4906-abd2-d6dc8673c297',
             u'active',
             u'Fedora-x86_64-20-20140618-sda',
             u'bare',
             u'2014-10-01T20:28:08Z',
             u'2014-10-01T20:28:09Z',
             u'qcow2',
             u'4dfdcf14a20940799d89c7a5e7345978',
             'False',
             0,
             0,
             u'9e486c3bf76219a6a37add392e425b36',
             209649664,
             u'/v2/images/c42736e7-8b09-4906-abd2-d6dc8673c297/file',
             'None',
             'None',
             u'/v2/schemas/image',
             u'public')]),
            'tags': set([
                (u'c42736e7-8b09-4906-abd2-d6dc8673c297', 'type=xen'),
                (u'c42736e7-8b09-4906-abd2-d6dc8673c297', 'type=xen2')])}
        self.assertEqual(self.driver.state, expected)
```

CHAPTER 4

Policy

4.1 1. What Does a Policy Look Like

A policy describes how services (either individually or as a whole) ought to behave. More specifically, a policy describes which **states** of the cloud are permitted and which are not. Or a policy describes which **actions** to take in each state of the cloud, in order to transition the cloud to one of those permitted states. For example For example, a policy might simply state that the minimum password length on all systems is eight characters, or a policy might state that if the minimum password length on some system is less than 8 that the minimum length should be reset to 8.

In both cases, the policy relies on knowing the state of the cloud. The state of the cloud is the amalgamation of the states of all the services running in the cloud. In Congress, the state of each service is represented as a collection of tables (see *Cloud Services*). The policy language determines whether any violation exists given the content of the state tables.

For example, one desirable policy is that each Neutron port has at most one IP address. That means that the following table mapping port id to ip address with the schema "port(id, ip)" is permitted by the policy.

ID	IP
"66dafde0-a49c-11e3-be40-425861b86ab6"	"10.0.0.1"
"73e31d4c-e89b-12d3-a456-426655440000"	"10.0.0.3"

Whereas, the following table is a violation.

ID	IP
"66dafde0-a49c-11e3-be40-425861b86ab6"	"10.0.0.1"
"66dafde0-a49c-11e3-be40-425861b86ab6"	"10.0.0.2"
"73e31d4c-e89b-12d3-a456-426655440000"	"10.0.0.3"

This is the policy written in Congress's policy language:

error(port_id, ip1, ip2) :- port(port_id, ip1), port(port_id, ip2), not equal(ip1, ip2);

Note that the policy above does not mention specific table content; instead it describes the general condition of tables. The policy says that for every row in the port table, no two rows should have the same ID and different IPs.

This example verifies a single table within Neutron, but a policy can use many tables as well. Those tables might all come from the same cloud service (e.g. all the tables might be Neutron tables), or the tables may come from different cloud services (e.g. some tables from Neutron, others from Nova).

For example, if we have the following table schemas from Nova, Neutron, and ActiveDirectory, we could write a policy that says every network connected to a VM must either be public or owned by someone in the same group as the VM owner.:

```
error(vm, network) :-
  nova:virtual_machine(vm)
  nova:network(vm, network)
  nova:owner(vm, vm_owner)
  neutron:owner(network, network_owner)
  not neutron:public_network(network)
  not same_group(vm_owner, network_owner)

same_group(user1, user2) :-
  ad:group(user1, group)
  ad:group(user2, group)
```

And if one of these errors occurs, the right solution is to disconnect the offending network (as opposed to deleting the VM, changing the owner, or any of the other feasible options):

```
execute[neutron:disconnectNetwork(vm, network)] :-
  error(vm, network)
```

The language Congress supports for expressing policy is called Datalog, a declarative language derived from SQL and first-order logic that has been the subject of research and development for decades.

4.2 2. Datalog Policy Language

As a policy writer, your goal is to define the contents of the *error* table, and in so doing to describe exactly those conditions that must be true when policy is being obeyed.

As a policy writer, you can also describe which actions Congress should take when policy is being violated by using the *execute* operator and thinking of the action to be executed as if it were a table itself.

Either when defining policy directly or describing the conditions under which actions should be executed to eliminate policy violations, it is often useful to use higher-level concepts than the cloud services provide natively. Datalog allows us to do this by defining new tables (higher-level concepts) in terms of existing tables (lower-level concepts) by writing *rules*. For example, OpenStack does not tell us directly which VMs are connected to the internet; rather, it provides a collection of lower-level API calls from which we can derive that information. Using Datalog we can define a table that lists all of the VMs connected to the internet in terms of the tables that Nova/Neutron support directly. As another example, if Keystone stores some collection of user groups and Active Directory stores a collection of user groups, we might want to create a new table that represents all the groups from either Keystone or Active Directory.

Datalog has a collection of core features for manipulating tables, and it has a collection of more advanced features that become important when you go beyond toy examples.

4.2.1 2.1 Core Datalog Features

Since Datalog is entirely concerned with tables, it's not surprising that Datalog allows us to represent concrete tables directly in the language.

Concrete tables. Suppose we want to use Datalog to represent a Neutron table that lists which ports have been assigned which IPs, such as the one shown below.

Table: neutron:port_ip

ID	IP
"66dafde0-a49c-11e3-be40-425861b86ab6"	"10.0.0.1"
"66dafde0-a49c-11e3-be40-425861b86ab6"	"10.0.0.2"
"73e31d4c-e89b-12d3-a456-426655440000"	"10.0.0.3"

To represent this table, we write the following Datalog:

```
neutron:port_ip("66dafde0-a49c-11e3-be40-425861b86ab6", "10.0.0.1")
neutron:port_ip("66dafde0-a49c-11e3-be40-425861b86ab6", "10.0.0.2")
neutron:port_ip("73e31d4c-e89b-12d3-a456-426655440000", "10.0.0.3")
```

Each of the Datalog statements above is called a *ground atom* (or *ground fact*). A ground atom takes the form `<tablename>(arg1, ..., argn)`, where each `argi` is either a double-quoted Python string or a Python number.

Basic rules The real power of Datalog is that it allows you to write recipes for constructing new tables out of existing tables, regardless which rows are in those existing tables.

To create a new table out of an existing table, we write Datalog *rules*. A *rule* is a simple if-then statement, where the *if* part is called the *head* and the *then* part is called the *body*. The head is always a single Datalog atom. The body is an AND of several possibly negated Datalog atoms. OR is accomplished by writing multiple rules with the same table in the head.

Suppose we want to create a new table `has_ip` that is just a list of the Neutron ports that have been assigned at least one IP address. We want our table to work regardless what IDs and IPs appear in the neutron:port_ip table so we use variables in place of strings/numbers. Variables have the same meaning as in algebra: they are placeholders for any value. (Syntactically, a variable is any symbol other than a number or a string.):

```
has_ip(x) :- neutron:port_ip(x, y)
```

This rule says that a port *x* belongs to the *has_ip* table if there exists some IP *y* such that row <*x,y*> belongs to the *neutron:port* table. Conceptually, this rule says to look at all of the ground atoms for the neutron:port_ip table, and for each one assign *x* to the port UUID and *y* to the IP. Then create a row in the *has_ip* table for *x*. This rule when applied to the neutron:port_ip table shown above would generate the following table:

```
has_ip("66dafde0-a49c-11e3-be40-425861b86ab6")
has_ip("73e31d4c-e89b-12d3-a456-426655440000")
```

Notice here that there are only 2 rows in *has_ip* despite there being 3 rows in *neutron:port_ip*. That happens because one of the ports in neutron:port_ip has been assigned 2 distinct IPs.

AND operator As a slightly more complex example, we could define a table *same_ip* that lists all the pairs of ports that are assigned the same IP.:

```
same_ip(port1, port2) :- neutron:port_ip(port1, ip), neutron:port_ip(port2, ip)
```

This rule says that the row <port1, port2> must be included in the *same_ip* table if there exists some *ip* where both <*port1, ip*> and <*port2, ip*> are rows in the *neutron:port* table (where notice that *ip* is the same in the two rows). Notice here the variable *ip* appears in two different places in the body, thereby requiring the value assigned to that variable be the same in both cases. This is called a *join* in the realm of relational databases and SQL.

NOT operator As another example, suppose we want a list of all the ports that have NOT been assigned any IP address. We can use the *not* operator to check if a row fails to belong to a table.

```
no_ip(port) :- neutron:port(port), not has_ip(port)
```

There are special restrictions that you must be aware of when using *not*. See the next section for details.

OR operator. Some examples require an OR, which in Datalog means writing multiple rules with the same table in the head. Imagine we have two tables representing group membership information from two different services: Keystone and Active Directory. We can create a new table *group* that says a person is a member of a group if she is a member of that group either according to Keystone or according to Active Directory. In Datalog we create this table by writing two rules.:

```
group(user, grp) :- ad:group(user, grp)
group(user, grp) :- keystone:group(user, grp)
```

These rules happen to have only one atom in each of their bodies, but there is no requirement for that.

4.2.2 2.2 Extended Datalog Features

In addition writing basic rules with and/or/not, the version of Datalog used by Congress includes the features described in this section.

Builtins. Often we want to write rules that are conditioned on things that are difficult or impossible to define within Datalog. For example, we might want to create a table that lists all of the virtual machines that have at least 100 GB of memory. To write that rule, we would need a way to check if the memory of a given machine is greater-than 100 or not. Basic arithmetic, string manipulation, etc. are operations that are built into Datalog, but they look as though they are just ordinary tables. Below the *gt* is a builtin table implementing greater-than:

```
plenty_of_memory(vm) :- nova:virtual_machine.memory(vm, mem), gt(mem, 100)
```

In a later section we include the list of available builtins.

Column references. Some tables have 5+ columns, and when tables have that many columns writing rules can be awkward. Typically when we write a rule, we only want 1 or 2 columns, but if there are 10 columns, then we end up needing to invent variable names to fill all the unneeded columns.

For example, Neutron's *ports* table has 10 columns. If you want to create a table that includes just the port IDs (as we used above), you would write the following rule:

```
port(id) :-
  neutron:ports(id, tenant_id, name, network_id, mac_address, admin_state_up,
                status, device_owner, fixed_ips, security_groups)
```

To simplify such rules, we can write rules that reference only those columns that we care about by using the column's name. Since the name of the first column of the *neutron:ports* table is "ID", we can write the rule above as follows:

```
port(x) :- neutron:ports(id=x)
```

You can only use these column references for tables provided by cloud services (since Congress only knows the column names for the cloud service tables). Column references like these are translated automatically to the version without column-references, which is something you may notice from time to time.

Table hierarchy. The tables in the body of rules can either be the original cloud-service tables or tables that are defined by other rules (with some limitations, described in the next section). We can think of a Datalog policy as a hierarchy of tables, where each table is defined in terms of the tables at a lower level in the hierarchy. At the bottom of that hierarchy are the original cloud-service tables representing the state of the cloud.

Order irrelevance. One noteworthy feature of Datalog is that the order in which rules appear is irrelevant. The rows that belong to a table are the minimal ones required by the rules if we were to compute their contents starting with the cloud-service tables (whose contents are given to us) and working our way up the hierarchy of tables. For more details, search the web for the term *stratified Datalog semantics*.

Execute modal. To write a policy that tells Congress the conditions under which it should execute a certain action, we write rules that utilize the *execute* modal in the head of the rule.

For example, to dictate that Congress should ask Nova to pause() all of the servers whose state is ACTIVE, we would write the following policy statement:

```
execute[nova:servers.pause(x)] :- nova:servers(id=x, status="ACTIVE")
```

We discuss this modal operator in greater detail in Section 3.

Grammar. Here is the grammar for Datalog policies:

```
<policy> ::= <rule>*
<rule> ::= <head> COLONMINUS <literal> (COMMA <literal>)*
<head> ::= <atom>
<head> ::= EXECUTE[<atom>]
<literal> ::= <atom>
<literal> ::= NOT <atom>
<atom> ::= TABLENAME LPAREN <arg> (COMMA <arg>)* RPAREN
<arg> ::= <term>
<arg> ::= COLUMNNAME=<term>
<term> ::= INTEGER | FLOAT | STRING | VARIABLE
```

4.2.3 2.3 Datalog Syntax Restrictions

There are a number of syntactic restrictions on Datalog that are, for the most part, common sense.

Head Safety: every variable in the head of a rule must appear in the body.

Head Safety is natural because if a variable appears in the head of the rule but not the body, we have not given a prescription for which strings/numbers to use for that variable when adding rows to the table in the head.

Body Safety: every variable occurring in a negated atom or in the input of a built-in table must appear in a non-negated, non-builtin atom in the body.

Body Safety is important for ensuring that the sizes of our tables are always finite. There are always infinitely many rows that DO NOT belong to a table, and there are often infinitely many rows that DO belong to a builtin (like equal). Body safety ensures that the number of rows belonging to the table in the head is always finite.

No recursion: You are not allowed to define a table in terms of itself.

A classic example starts with a table that tells us which network nodes are directly adjacent to which other nodes (by a single network hop). Then you want to write a policy about which nodes are connected to which other nodes (by any number of hops). Expressing such a policy requires recursion, which is not allowed.

Modal safety: The *execute* modal may only appear in the heads of rules.

The Datalog language is we have is called a condition-action language, meaning that action-execution depends on conditions on the state of the cloud. But it is not an event-condition-action language, which would enable action-execution to depend on the conditions of the cloud plus the action that was just executed. An event-condition-action language would allow the *execute* modal to appear in the body of rules.

Schema consistency: Every time a rule references one of the cloud service tables, the rule must use the same (number of) columns that the cloud service provides for that table.

This restriction catches mistakes in rules that use the wrong number of columns or the wrong column names.

4.2.4 2.4 Datalog builtins

Here is a list of the currently supported builtins. A builtin that has N inputs means that the leftmost N columns are the inputs, and the remaining columns (if any) are the outputs. If a builtin has no outputs, , starting with arithmetic.

Arithmetic Builtin	Inputs	Description
lt(x, y)	2	True if x < y
lteq(x, y)	2	True if x <= y
gt(x, y)	2	True if x > y
gteq(x, y)	2	True if x >= y
max(x, y, z)	2	z = max(x, y)
plus(x, y, z)	2	z = x + y
minus(x, y, z)	2	z = x - y
mul(x, y, z)	2	z = x * y
div(x, y, z)	2	z = x / y
float(x, y)	1	y = float(x)
int(x, y)	1	y = int(x)

Next are the string builtins.

String Builtin	Inputs	Description
concat(x, y, z)	2	z = concatenate(x, y)
len(x, y)	1	y = number of characters in x

Last are the builtins for manipulating dates and times. These builtins are based on the Python DateTime object.

Datetime Builtin	Inputs	Description
now(x)	0	The current date-time
unpack_date(x, year, month, day)	1	Extract year/month/day
unpack_time(x, hours, minutes, secs)	1	Extract hours/minutes/seconds
unpack_datetime(x, y, m, d, h, i, s)	1	Extract date and time
pack_time(hours, minutes, seconds, x)	3	Create date-time with date
pack_date(year, month, day, x)	3	Create date-time with time
pack_datetime(y, m, d, h, i, s, x)	6	Create date-time with date/time
extract_date(x, date)	1	Extract date obj from date-time
extract_time(x, time)	1	Extract time obj from date-time
datetime_to_seconds(x, secs)	1	secs from 1900 to date-time x
datetime_plus(x, y, z)	2	z = x + y
datetime_minus(x, y, z)	2	z = x - y
datetime_lt(x, y)	2	True if x is before y
datetime_lteq(x, y)	2	True if x is no later than y
datetime_gt(x, y)	2	True if x is later than y
datetime_gteq(x, y)	2	True if x is no earlier than y
datetime_equal(x, y)	2	True if x == y

4.3 3. Multiple Policies

One of the goals of Congress is for several different people in an organization to collaboratively define a single, overarching policy that governs a cloud. The example, the compute admin might some tables that are good building blocks for writing policy about compute. Similarly the network and storage admins might create tables that help define policy about networking and storage, respectively. Using those building blocks, the cloud administrator might then write policy about compute, storage, and networking.

To make it easier for several people to collaborate (or for a single person to write more modular policies) Congress allows you organize your Datalog statements using policy modules. Each policy module is simply a collection of Datalog statements. You create and delete policy modules using the API, and the you insert/delete Datalog statements into a particular policy module also using the API.

The rules you insert into one policy module can reference tables defined in other policy modules. To do that, you prefix the name of the table with the name of the policy and separate the policy module and table name with a colon.

For example, if the policy module *compute* has a table that lists all the servers that have not been properly secured *insecure(server)* and the policy module *network* has a table of all devices connected to the internet *connected_to_internet*, then as a cloud administrator, you might write a policy that says there is an error whenever a server is insecure and connected to the internet.

    error(x) :- compute:insecure(x), network:connected_to_internet(x)

Notice that this is exactly the same syntax you use to reference tables exported directly by cloud services:

```
has_ip(x) :- neutron:port_ip(x, y)
```

In fact, the tables exported by cloud services are stored in a policy module with the same name as the service.

While the term *policy module* is accurate, we usually abbreviate it to *policy*, and say that Congress supports multiple policies. Note, however, that supporting multiple policies is not the same thing as supporting multi-tenancy. Currently, all of the policies are visible to everyone using the system, and everyone using the system has the same view of the tables the cloud services export. For true multi-tenancy, you would expect different tenants to have different sets of policies and potentially a different view of the data exported by cloud services.

See section *API* for details about creating, deleting, and populating policies.

4.3.1 3.1 Syntactic Restrictions for Multiple Policies

There are a couple of additional syntactic restrictions imposed when using multiple policies.

No recursion across policies. Just as there is no recursion permitted within a single policy, there is no recursion permitted across policies.

For example, the following is prohibited:

```
# Not permitted because of recursion
Module compute:  p(x) :- storage:q(x)
Module storage:  q(x) :- compute:p(x)
```

No policy name may be referenced in the head of a rule. A rule may not mention any policy in the head (unless the head uses the modal *execute*).

This restriction prohibits one policy from changing the tables defined within another policy. The following example is prohibited (in all policy modules, including 'compute'):

```
# Not permitted because 'compute' is in the head
compute:p(x) :- q(x)
```

The following rule is permitted, because it utilizes *execute* in the head of the rule:

```
# Permitted because of execute[]
execute[nova:pause(x)] :- nova:servers(id=x, status="ACTIVE")
```

Congress will stop you from inserting rules that violate these restrictions.

CHAPTER 5

Monitoring and Enforcement

Congress is given two inputs: the other cloud services in the datacenter and a policy describing the desired state of those services. Congress does two things with those inputs: monitoring and enforcement. *Monitoring* means passively comparing the actual state of the other cloud services and the desired state (i.e. policy) and flagging mismatches. *Enforcement* means actively working to ensure that the actual state of the other cloud services is also a desired state (i.e. that the other services obey policy).

5.1 1. Monitoring

Recall from *Policy* that policy violations are represented with the table *error*. To ask Congress for a list of all policy violations, we simply ask it for the contents of the *error* table.

For example, recall our policy from *Policy*: each Neutron port has at most one IP address. For that policy, the *error* table is has 1 row for each Neutron port that has more than 1 IP address. Each of those rows specify the UUID for the port, and two different IP addresses. So if we had the following mapping of Neutron ports to IP addresses:

ID	IP
“66dafde0-a49c-11e3-be40-425861b86ab6”	“10.0.0.1”
“66dafde0-a49c-11e3-be40-425861b86ab6”	“10.0.0.2”
“73e31d4c-e89b-12d3-a456-426655440000”	“10.0.0.3”
“73e31d4c-e89b-12d3-a456-426655440000”	“10.0.0.4”
“8caead95-67d5-4f45-b01b-4082cddce425”	“10.0.0.5”

the *error* table would be something like the one shown below.

ID	IP 1	IP 2
“66dafde0-a49c-11e3-be40-425861b86ab6”	“10.0.0.1”	“10.0.0.2”
“73e31d4c-e89b-12d3-a456-426655440000”	“10.0.0.3”	“10.0.0.4”

The API would return this table as the following collection of Datalog facts (encoded as a string):

```
error("66dafde0-a49c-11e3-be40-425861b86ab6", "10.0.0.1", "10.0.0.2")
error("73e31d4c-e89b-12d3-a456-426655440000", "10.0.0.3", "10.0.0.4")
```

It is the responsibility of the client to periodically ask the server for the contents of the error table.

5.2 2. Proactive Enforcement

Often we want policy to be enforced, not just monitored. *Proactive enforcement* is the term we use to mean preventing policy violations before they occur. Proactive enforcement requires having enforcement points in the cloud that stop

changes before they happen. Cloud services like Nova, Neutron, and Cinder are good examples of enforcement points. For example, Nova could refuse to provision a VM that would cause a policy violation, thereby proactively enforcing policy.

To enable other cloud services like Nova to check if a proposed change in the cloud state would violate policy, the cloud service can consult Congress using its `simulate()` functionality. The idea for `simulate()` is that we ask Congress to answer a query after having temporarily made some changes to data and policies. Simulation allows us to explore the effects of proposed changes. Typically simulation is used to ask: if I made these changes, would there be any new policy violations? For example, provisioning a new VM might add rows to several of Nova's tables. After receiving an API call that requests a new VM be provisioned, Nova could ask Congress if adding those rows would create any new policy violations. If new violations arise, Nova could refuse to provision the VM, thereby proactively enforcing the policy.

In this writeup we assume you are using the python-client.

Suppose you want to know the policy violations after making the following changes.

1. insert a row into the *nova:servers* table with ID uuid1, 2TB of disk, and 10GB of memory
2. delete the row from *neutron:security_groups* with the ID "uuid2" and name "alice_default_group"

(Here we assume the nova:servers table has columns ID, disk-size, and memory and that neutron:security groups has columns ID, and name.)

To do a simulation from the command line, you use the following command:

```
$ openstack congress policy simulate <policy-name> <query> <change-sequence> <action-policy-name>
```

- <policy-name>: the name of the policy in which to run the query
- <query>: a string representing the query you would like to run after applying the change sequence
- <change-sequence>: a string codifying a sequence of insertions and deletions of data and rules. Insertions are denoted by '+' and deletions by '-'
- <action-policy-name>: the name of another policy of type 'action' describing the effects of any actions occuring in <change-sequence>. Actions are not necessary and are explained later. Without actions, this argument can be anything (and will in the future be optional).

For our nova:servers and neutron:security_groups example, we would run the following command to find all of the policy violations after inserting a row into nova:servers and then deleting a row out of neutron:security_groups:

```
$ openstack congress policy simulate classification
    'error(x)'
    'nova:servers+("uuid1", "2TB", "10 GB")
     neutron:security_groups-("uuid2", "alice_default_group")'
    null
```

More examples

Suppose the table 'p' is a collection of key-value pairs: p(key, value). Let's begin by creating a policy and adding some key/value pairs for 'p':

```
$ openstack congress policy create alice
$ openstack congress policy rule create alice 'p(101, 0)'
$ openstack congress policy rule create alice 'p(202, "abc")'
$ openstack congress policy rule create alice 'p(302, 9)'
```

Let's also add a statement that says there's an error if a single key has multiple values or if any key is assigned 9:

```
$ openstack congress policy rule create classification
    'error(x) :- p(x, val1), p(x, val2), not eq(val1, val2)'
$ openstack congress policy rule create classification 'error(x) :- p(x, 9)'
```

Each of the following is an example of a simulation query you might want to run.

1. **Basic usage**. Simulate adding the value 5 to key 101 and ask for the contents of p:

```
$ openstack congress policy simulate classification 'p(x,y)' 'p+(101, 5)' null
p(101, 0)
p(101, 5)
p(202, "abc")
p(302, 9)
```

2. **Error table**. Simulate adding the value 5 to key 101 and ask for the contents of error:

```
$ openstack congress policy simulate classification 'error(x)' 'p+(101, 5)' null
error(101)
error(302)
```

3. **Inserts and Deletes**. Simulate adding the value 5 to key 101 and deleting 0 and ask for the contents of error:

```
$ openstack congress policy simulate classification 'error(x)'
    'p+(101, 5) p-(101, 0)' null
error(302)
```

4. **Error changes**. Simulate changing the value of key 101 to 9 and query the **change** in the error table:

```
$ openstack congress policy simulate classification 'error(x)'
    'p+(101, 9) p-(101, 0)' null --delta
error+(101)
```

6. **Multiple error changes**. Simulate changing 101:9, 202:9, 302:1 and query the *change* in the error table:

```
$ openstack congress policy simulate classification 'error(x)'
    'p+(101, 9) p-(101, 0) p+(202, 9) p-(202, "abc") p+(302, 1) p-(302, 9)'
    null --delta
error+(202)
error+(101)
error-(302)
```

7. **Order matters**. Simulate changing 101:9, 202:9, 302:1, and finally 101:15 (in that order). Then query the *change* in the error table:

```
$ openstack congress policy simulate classification 'error(x)'
    'p+(101, 9) p-(101, 0) p+(202, 9) p-(202, "abc") p+(302, 1) p-(302, 9)
     p+(101, 15) p-(101, 9)' null --delta
error+(202)
error-(302)
```

8. **Tracing**. Simulate changing 101:9 and query the *change* in the error table, while asking for a debug trace of the computation:

```
$ openstack congress policy simulate classification 'error(x)'
    'p+(101, 9) p-(101, 0)' null --delta --trace
error+(101)
RT    : ** Simulate: Querying error(x)
Clas  : Call: error(x)
Clas  : | Call: p(x, 9)
Clas  : | Exit: p(302, 9)
Clas  : Exit: error(302)
Clas  : Redo: error(302)
Clas  : | Redo: p(302, 9)
Clas  : | Fail: p(x, 9)
Clas  : Fail: error(x)
```

```
Clas  : Found answer [error(302)]
RT    : Original result of error(x) is [error(302)]
RT    : ** Simulate: Applying sequence [set(101, 9)]
Action: Call: action(x)
...
```

9. **Changing rules**. Simulate adding 101: 5 (which results in 101 having 2 values) and deleting the rule that says each key must have at most 1 value. Then query the error table:

```
$ openstack congress policy simulate classification 'error(x)'
    'p+(101, 5)   error-(x) :- p(x, val1), p(x, val2), not eq(val1, val2)'
    null
error(302)
```

The syntax for inserting/deleting rules is a bit awkward since we just afix a + or - to the head of the rule. Ideally we would afix the +/- to the rule as a whole. This syntactic sugar will be added in a future release.

There is also currently the limitation that you can only insert/delete rules from the policy you are querying. And you cannot insert/delete action description rules.

5.2.1 2.1 Simulation with Actions

The downside to the simulation functionality just described is that the cloud service wanting to prevent policy violations would need to compute the proposed changes in terms of the *tables* that Congress uses to represent its internal state. Ideally a cloud service would have no idea which tables Congress uses to represent its internals. But even if each cloud service knew which tables Congress was using, it would still need convert each API call into a collection of changes on its internal tables.

For example, an API call for Nova to provision a new VM might change several tables. An API call to Heat to provision a new app might change tables in several different cloud services. Translating each API call exposed by a cloud service into the collection of Congress table changes is sometimes impractical.

In the key/value examples above, it the caller needed to know the current state of the key/value store in order to accurately describe the changes she wanted to make. Setting the key 101 to value 9 meant knowing that its current value was 0 so that during the simulation we could say to delete the assignment of 101 to 0 and add the assignment of 101 to 9.

It would be preferable if an external cloud service could simply ask Congress if the API call it is about to execute is permitted by the policy. To do that, we must tell Congress what each of those actions do in terms of the cloud-service tables. Each of these *action descriptions* describe which rows are inserted/deleted from which tables if the action were to be executed in the current state of the cloud. Those action descriptions are written in Datalog and are stored in a policy of type 'action'.

Action description policy statements are regular Datalog rules with one main exception: they use + and - to adorn the table in the head of a rule to indicate whether they are describing how to *insert* table rows or to *delete* table rows, respectively.

For example in the key-value store, we can define an action 'set(key, value)' that deletes the current value assigned to 'key' and adds 'value' in its place. To describe this action, we write two things: a declaration to Congress that *set* is indeed an action using the reserved table name *action* and rules that describe which table rows *set* inserts and which rows it deletes:

```
action("set")
p+(x,y) :- set(x,y)
p-(x,oldy) :- set(x,y), p(x,oldy)
```

Note: Insertion takes precedence over deletion, which means that if a row is both inserted and deleted by an action, the row will be inserted.

To insert these rows, we create a policy of type 'action' and then insert these rules into that policy:

```
$ openstack congress policy create aliceactions --kind 'action'
$ openstack congress policy rule create action 'action("set")'
$ openstack congress policy rule create action 'p+(x,y) :- set(x,y)'
$ openstack congress policy rule create action 'p-(x,oldy) :- set(x,y), p(x,oldy)'
```

Below we illustrate how to use *set* to simplify the simulation queries shown previously.

1. **Inserts and Deletes**. Set key 101 to value 5 and ask for the contents of error:

   ```
   $ openstack congress policy simulate classification 'error(x)' 'set(101, 5)' null
   error(302)
   ```

2. **Multiple error changes**. Simulate changing 101:9, 202:9, 302:1 and query the *change* in the error table:

   ```
   $ openstack congress policy simulate classification 'error(x)'
       'set(101, 9) set(202, 9) set(302, 9)' null --delta
   error+(202)
   error+(101)
   error-(302)
   ```

3. **Order matters**. Simulate changing 101:9, 202:9, 302:1, and finally 101:15 (in that order). Then query the *change* in the error table:

   ```
   $ openstack congress policy simulate classification 'error(x)'
       'set(101, 9) set(202, 9) set(302, 1) set(101, 15)' null --delta
   error+(202)
   error-(302)
   ```

4. **Mixing actions and state-changes**. Simulate changing 101:9 and adding value 7 for key 202. Then query the *change* in the error table:

   ```
   $ openstack congress policy simulate classification 'error(x)'
       'set(101, 9) p+(202, 7)' null --delta
   error+(202)
   error+(101)
   ```

5.3 3. Manual Reactive Enforcement

Not all policies can be enforced proactively on all clouds, which means that sometimes the cloud will violate policy. Once policy violations happen, Congress can take action to transition the cloud back into one of the states permitted by policy. We call this *reactive enforcement*. Currently, to reactively enforce policy, Congress relies on people to tell it which actions to execute and when to execute them, hence we call it *manual* reactive enforcement.

Of course, Congress tries to make it easy for people to tell it how to react to policy violations. People write policy statements that look almost the same as standard Datalog rules, except the rules use the modal *execute* in the head. For more information about the Datalog language and how to write these rules, see *Policy*.

Take a simple example that is easy and relatively safe to try out. The policy we want is that no server should have an ACTIVE status. The policy we write tells Congress how to react when this policy is violated: it says to ask Nova to execute `pause()` every time it sees a server with ACTIVE status:

```
execute[nova:servers.pause(x)] :- nova:servers(id=x, status="ACTIVE")
```

The way this works is that everytime Congress gets new data about the state of the cloud, it figures out whether that new data causes any new rows to be added to the `nova:servers.pause(x)` table. (While policy writers know that

nova:servers.pause isn't a table in the usual sense, the Datalog implementation treats it like a normal table and computes all the rows that belong to it in the usual way.) If there are new rows added to the `nova:servers.pause(x)` table, Congress asks Nova to execute `servers.pause` for every row that was newly created. The arguments passed to `servers.pause` are the columns in each row.

For example, if two servers have their status set to ACTIVE, Congress receives the following data (in actuality the data comes in with all the columns set, but here we use column references for the sake of pedagogy):

```
nova:servers(id="66dafde0-a49c-11e3-be40-425861b86ab6", status="ACTIVE")
nova:servers(id="73e31d4c-a49c-11e3-be40-425861b86ab6", status="ACTIVE")
```

Congress will then ask Nova to execute the following commands:

```
servers.pause("66dafde0-a49c-11e3-be40-425861b86ab6")
servers.pause("73e31d4c-a49c-11e3-be40-425861b86ab6")
```

Congress will not wait for a response from Nova. Nor will it change the status of the two servers that it asked Nova to pause in its `nova:servers` table. Congress will simply execute the pause() actions and wait for new data to arrive, just like always. Eventually Nova executes the pause() requests, the status of those servers change, and Congress receives another data update.

nova:servers(id="66dafde0-a49c-11e3-be40-425861b86ab6", status="PAUSED")
nova:servers(id="73e31d4c-a49c-11e3-be40-425861b86ab6", status="PAUSED")

At this point, Congress updates the status of those servers in its `nova:servers` table to PAUSED. But this time, Congress will find that no new rows were **added** to the `nova:servers.pause(x)` table and so will execute no actions. (Two rows were deleted, but Congress ignores deletions.)

In short, Congress executes actions exactly when new rows are inserted into a table augmented with the *execute* modal.

CHAPTER 6

API

The design document for the API can be found below. This document contains the API as of the current release:

`https://docs.google.com/document/d/14hM7-GSm3CcyohPT2Q7GalyrQRohVcx77hxEx4AO4Bk/edit#`

There are two top-level concepts in today's API: Policies and Data-sources.

- Policies have *rules* that describe the permitted states of the cloud, along with *tables* representing abstractions of the cloud state.
- Data-sources have *tables* representing the current state of the cloud.
- The *tables* of both policies and data-sources have rows that describe their contents.

6.1 1. Policy (/v1/)

You can create and delete policies. Two policies are provided by the system, and you are not permitted to delete them: *classification* and *action*. A policy has the following fields:

- name: a unique name that is human-readable
- abbreviation: a shorter name that appears in traces
- description: an explanation of this policy's purpose
- kind: either *nonrecursive* or *action*. The default is *nonrecursive* and unless you are writing action descriptions for use with `simulate` you should always use the default.

Op	URL	Result
GET	.../policies	List policies
GET	.../policies/<policy-id>	Read policy properties
POST	.../policies/<policy-id>	Create new policy
DELETE	.../policies/<policy-id>	Delete policy

You can also utilize the simulation API call, which answers hypothetical questions: if we were to change the state of the cloud in this way, what would the answer to this query be? See *Monitoring and Enforcement* for more details and examples:

```
POST .../policies/<policy-id>
  ?action=simulate
  &query=<query>                    # string query like: 'error(x)'
  &sequence=<sequence>              # changes to state like: 'p+(1) p-(2)'
  &action_policy=<action_policy>    # name of a policy: 'action'
  [&delta=true]                     # return just change in <query>
  [&trace=true]                     # also return explanation of result
```

6.2 2. Policy Rules (/v1/policies/<policy-id>/...)

Each policy is a collection of rules. Congress supports the usual CRUD operations for changing that collection. A rule has the following fields:

- ID: a unique identifier
- name: a human-friendly identifier
- rule: a string representing the actual rule as described in *Policy*

Op	URL	Result
GET	.../rules	List policy rules
POST	.../rules	Create policy rule
GET	.../rules/<rule-id>	Read policy rule
DELETE	.../rules/<rule-id>	Delete policy rule

6.3 3. Policy Tables (/v1/policies/<policy-id>/...)

All the tables mentioned in the rules of a policy can be queried via the API. They have only an ID field.

Op	URL	Result
GET	.../tables	List tables
GET	.../tables/<table-id>	Read table properties

6.4 4. Policy Table Rows (/v1/policies/<policy-id>/tables/<table-id>/...)

Rules are used to instruct Congress how to create new tables from existing tables. Congress allows you to query the actual contents of tables at any point in time. Congress will also provide a trace of how it computed a table, to help policy authors understand why certain rows belong to the table and others do not.

Op	URL	Result
GET	.../rows	List rows
GET	.../rows?trace=true	List rows with explanation (use 'printf' to display)

6.5 5. Drivers (/v1/system/)

A driver is a piece of code that once instantiated and configured interacts with a specific cloud service like Nova or Neutron. A driver has the following fields.

- ID: a human-friendly unique identifier
- description: an explanation of which type of cloud service this driver interacts with

Op	URL	Result
GET	.../drivers	List drivers
GET	.../drivers/<driver-id>	Read driver properties

6.6 6. Data sources (/v1/)

A data source is an instantiated and configured driver that interacts with a particular instance of a cloud service (like Nova or Neutron). You can construct multiple datasources using the same driver. For example, if you have two instances of Neutron running, one in production and one in test and you want to write policy over both of them, you would create two datasources using the Neutron driver and give them different names and configuration options. For example, you might call one datasource 'neutron_prod' and the other 'neutron_test' and configure them with different IP addresses.

A datasource has the following fields.

- ID: a unique identifier
- name: a human-friendly unique that is unique across datasources and policies
- driver: the name of the driver code that this datasource is running
- config: a dictionary capturing the configuration of this datasource
- description: an explanation of the purpose of this datasource
- enabled: whether or not this datasource is functioning (which is always True)

Op	URL	Result
GET	.../data-sources	List data sources
POST	.../data-sources	Create data source
DELETE	.../data-sources/<ds-id>	Delete data source
GET	.../data-sources/<ds-id>/schema	Show schema (tables and table-columns)
GET	.../data-sources/<ds-id>/status	Show data source status

6.7 7. Data source Tables (/v1/data-sources/<ds-id>/...)

Each data source maintains a collection of tables (very similar to a Policy). The list of available tables for each data source is available via the API. A table just has an ID field.

Op	URL	Result
GET	.../tables	List data sources
GET	.../tables/<table-id>	Read data source properties

6.8 8. Data source Table Rows (/v1/data-sources/<ds-id>/tables/<table-id>/...)

The contents of each data source table (the rows of each table) can be queried via the API as well. A row has just a Data field, which is a list of values.

Op	URL	Result
GET	.../rows	List rows

CHAPTER 7

Contributing

The Congress wiki page is the authoritative starting point.

> https://wiki.openstack.org/wiki/Congress

If you would like to contribute to the development of any OpenStack project including Congress, you must follow the steps in this page:

> http://docs.openstack.org/infra/manual/developers.html

Once those steps have been completed, changes to OpenStack should be submitted for review via the Gerrit tool, following the workflow documented at:

> http://docs.openstack.org/infra/manual/developers.html#development-workflow

Pull requests submitted through GitHub will be ignored.

Bugs should be filed on Launchpad, not GitHub:

> https://bugs.launchpad.net/congress

CHAPTER 8

Code Overview

This page gives a brief overview of the code structure that implements Congress.

8.1 1. External information

The main source of information is the Congress wiki. There are two separate codebases that implement Congress: the server and the python client bindings.

- wiki: https://wiki.openstack.org/wiki/Congress
- server: https://git.openstack.org/cgit/openstack/congress
- client: https://git.openstack.org/cgit/openstack/python-congressclient

The structure of the client code is the same as that for other recent OpenStack python clients. The bulk of the Congress code is contained within the server. The remainder of this page describes the layout of the server code.

8.2 2. Server directory structure

Here are the most important components of the code, described by how they are laid out in the repository.

- `congress/harness.py`: instantiates message bus and installs datasource drivers and policy engine onto the bus
- `congress/datalog`: implementation of Datalog policy language
- `congress/policy_engines`: entities running on the message bus that understand policy languages
- `congress/datasources`: datasource drivers: thin wrappers/adapters for integrating services like Nova, Neutron
- `congress/dse`: message bus that the policy engine and datasources use to communicate
- `congress/api`: API data models (entry points into the system from the API)
- `contrib`: code for integrating into other services, e.g. devstack, horizon, tempest

8.3 3. Datalog

First is a description of the files and folders in congress/datalog. These files implement Datalog: the language Congress uses for describing policies.

- `congress/datalog/Congress.g`: Antlr3 grammar defining the syntax of Datalog. `make` uses Congress.g to generate CongressLexer.py and CongressParser.py, which contain the code used to convert strings into Python datastructures.
- `congress/datalog/compile.py`:
  - Convert policy strings into Python datastructures that represent those strings.
  - Includes datastructures for individual policy statements.
  - Also includes additional syntax checks that are not handled by the grammar.
- `congress/datalog/unify.py`: unification routines used at the heart of the policy reasoning algorithms.

Second is a brief overview of the fundamental datastructures used to represent individual policy statements.

- `congress/datalog/compile.py:Rule`: represents a single rule of the form `head1, ..., headn :- body1, ..., bodym`. Each headi and bodyi are Literals.
- `congress/datalog/compile.py:Literal`: represents a possibly negated atom of the form `[not] table(arg1, ..., argn)`. Each argi is a term.
- `congress/datalog/compile.py:Term`: represents an argument to a Literal. Is either a Variable or an ObjectConstant.
- `congress/datalog/compile.py:ObjectConstant`: special kind of Term that represents a fixed string or number.
- `congress/datalog/compile.py:Variable`: special kind of Term that is a placeholder used in a rule to represent an ObjectConstant.

Third is an overview of the datastructures used to represent entire policies. There are several different kinds of policies that you can choose from when creating a new policy. Each makes different tradeoffs in terms of time/space or in terms of the kind of policy statements that are permitted. Internally these are called 'theories'.

- `congress/datalog/nonrecursive.py:NonrecursiveRuleTheory`: represents an arbitrary collection of rules (without recursion). No precomputation of table contents is performed. Small memory footprint, but query time can be large. (A Prolog implementation of rules.) This is the default datastructure used when creating a new policy.
- `congress/datalog/ruleset.py:Ruleset`: represents a collection of rules, with indexing for faster query evaluation. Used by NonrecursiveRuleTheory.
- `congress/datalog/factset.py:FactSet`: represents a collection of non-negated Literals without variables, e.g. `p(1, "alice")`. Designed for minimal memory overhead.
- `congress/datalog/materialized.py:MaterializedViewTheory`: represents an arbitrary collection of rules (even allows recursion). Contents of all tables are computed and stored each time policy changes. Large memory footprint, but query time is small when asking for the contents of any table. Not actively maintained.
- `congress/datalog/database.py:Database`: represents a collection of non-negated Literals without variables, e.g. `p(1, "alice"`. Similar to a FactSet but with additional overhead. Used by the MaterializedViewTheory internally. Not actively maintained.

8.4 4. Policy engines

The congress/policy_engines directory contains implementations and wrappers for policy engines. At the time of writing, there are 2 policy engines in this directory: the domain-agnostic policy engine (agnostic.py) and the skeleton of a policy engine specialized for VM-placement (vm_placement.py). We detail only the domain-agnostic policy engine.

8.4.1 4.1 Domain-agnostic policy engine

Source code found in `congress/policy_engines/agnostic.py`.

- class `Runtime` is the top-level class for the policy engine. It implements the creation/deletion of (different kinds of) policies, the insertion/deletion of policy statements, and all the other functionality built on top of the Datalog implementation.
- class `DseRuntime` inherits from `Runtime` to make it run on the DSE message bus. It handles publishing/subscribing to the tables exported by the datasources.

Below we give a list of the top-level entry points to the domain-agnostic Runtime class—the top-level class for the domain agnostic policy engine.

- `create_policy`, `delete_policy`: implement multiple policies
- `select`: ask for the answer to a standard database query (e.g. the contents of a table) for a specified policy
- `insert`, `delete`: insert or delete a single policy statement into a specified policy
- `update`: batch of inserts/deletes into multiple policies
- `simulate`: apply a sequence of updates (temporarily), answer a query, and roll-back the updates.
- `TriggerRegistry`: central datastructure for triggers (the mechanism used to implement manual-reactive-enforcement rules). See `initialize_tables` and `_update_obj` to see how and when triggers are executed.

CHAPTER 9

Release Notes

9.1 Kilo

Main features

- Datalog: basic rules, column references, multiple policies, action-execution rules
- Monitoring: check for policy violations by asking for the rows of the `error` table
- Proactive enforcement: prevent violations by asking Congress before making changes using the `simulate` API call
- Manual reactive enforcement: correct violations by writing Datalog statements that say which actions to execute to eliminate violations
- Datasource drivers for Ceilometer, Cinder, CloudFoundry, Glance, Ironic, Keystone, Murano, Neutron, Nova, Plexxi, Swift, vCenter

Known issues

- `GET /v1/policies/<policy-name>/rules` fails to return 404 if the policy name is not found. There are similar issues for other `/v1/policies/<policy-name>/rules` API calls.
- Within a policy, you may not use both `execute[<table>(<args>)]` and `<table>(<args>)` in the heads of rules.

Tutorials:

CHAPTER 10

Congress Tutorial - Tenant Sharing Policy

10.1 Overview

This tutorial illustrates how to create a Congress monitoring policy that detects when one Openstack tenant shares a network with another Openstack tenant, and then flags that sharing as a policy violation.

Data Source Tables

- Neutron networks: list of each network and its owner tenant.
- Neutron ports: list of each port and its owner tenant.
- Nova servers: list of each server, and its owner tenant.

Detailed Policy Description

This policy collects the owner information for each server, any ports that server is connected to, and each network those ports are part of. It then verifies that the owner (tenant_id) is the same for the server, ports, and networks. If the tenant_id does not match, the policy will insert the server's name to the Congress error table.

10.2 Setting up Devstack

The first step is to install and configure Devstack + Congress:

1. Install Devstack and Congress using the directions in the following README. When asked for a password, type "password" without the quotes.

   https://github.com/openstack/congress/blob/master/README.rst#41-devstack-install

2. The Devstack installation script will automatically create a data source instance of the neutronv2 driver. If you are not using Devstack, you will need to create the data source:

   ```
   $ AUTH_URL=`keystone endpoint-get --service=identity | grep "publicURL" | awk '{print $4}'`
   $ openstack congress datasource create neutronv2 neutronv2 --config username=admin --config tena
   ```

3. Change auth_strategy from "keystone" to "noauth" in /etc/congress/congress.conf

4. Restart congress-server:

   ```
   $ screen -x stack
   switch to congress window <Ctrl-A> ' <congress-window-number> <Enter>
   <Ctrl-C>
   <Up>
   ```

```
<Enter>
<Ctrl-A> d
```

10.3 Setting up an Openstack VM and network

At this point, Devstack and Congress are running and ready to accept API calls. Now you can setup the Openstack environment, including a network and subnet owned by the "admin" tenant, a port owned by the "demo" tenant, and a VM owned by the "demo" tenant.

5. Change to the congress directory:

```
$ cd /opt/stack/congress
```

6. Login as the admin tenant:

```
$ source ~/devstack/openrc admin admin
```

7. Create a network called "network-admin". Note this is owned by the admin tenant:

```
$ neutron net-create network-admin
Created a new network:
+-----------------------+--------------------------------------+
| Field                 | Value                                |
+-----------------------+--------------------------------------+
| admin_state_up        | True                                 |
| id                    | a4130b34-81b4-46df-af3a-f133b277592e |
| name                  | network-admin                        |
| port_security_enabled | True                                 |
| shared                | False                                |
| status                | ACTIVE                               |
| subnets               |                                      |
| tenant_id             | 7320f8345acb489e8296ddb3b1ad1262     |
+-----------------------+--------------------------------------+
```

8. Create a subnet called "subnet-admin". Note this is owned by the admin tenant:

```
$ neutron subnet-create network-admin 2.2.2.0/24 --name subnet-admin
Created a new subnet:
+-------------------+------------------------------------------+
| Field             | Value                                    |
+-------------------+------------------------------------------+
| allocation_pools  | {"start": "2.2.2.2", "end": "2.2.2.254"} |
| cidr              | 2.2.2.0/24                               |
| dns_nameservers   |                                          |
| enable_dhcp       | True                                     |
| gateway_ip        | 2.2.2.1                                  |
| host_routes       |                                          |
| id                | 6ff5faa3-1752-4b4f-b744-2e0744cb9208     |
| ip_version        | 4                                        |
| ipv6_address_mode |                                          |
| ipv6_ra_mode      |                                          |
| name              | subnet-admin                             |
| network_id        | a4130b34-81b4-46df-af3a-f133b277592e     |
| tenant_id         | 7320f8345acb489e8296ddb3b1ad1262         |
+-------------------+------------------------------------------+
```

9. Create port owned by the demo tenant:

```
$ source ~/devstack/openrc admin demo
$ neutron port-create network-admin | tee port-create.log
Created a new port:
+-----------------------+-----------------------------------------------------------------------
| Field                 | Value
+-----------------------+-----------------------------------------------------------------------
| admin_state_up        | True
| allowed_address_pairs |
| binding:host_id       |
| binding:profile       | {}
| binding:vif_details   | {}
| binding:vif_type      | unbound
| binding:vnic_type     | normal
| device_id             |
| device_owner          |
| fixed_ips             | {"subnet_id": "6ff5faa3-1752-4b4f-b744-2e0744cb9208", "ip_address": "2
| id                    | 066c5cfc-949e-4d56-ad76-15528c68c8b8
| mac_address           | fa:16:3e:e9:f8:2a
| name                  |
| network_id            | a4130b34-81b4-46df-af3a-f133b277592e
| security_groups       | dd74db4f-fe35-4a51-b920-313fd36837f2
| status                | DOWN
| tenant_id             | 81084a94769c4ce0accb6968c397a085
+-----------------------+-----------------------------------------------------------------------

$ PORT_ID=`grep " id " port-create.log | awk '{print $4}'`
```

10. Create vm named "vm-demo" with the newly created port. The vm is owned by the demo tenant:

```
$ nova boot --image cirros-0.3.2-x86_64-uec --flavor 1 vm-demo --nic port-id=$PORT_ID
+--------------------------------------+-------------------------------------------------------
| Property                             | Value
+--------------------------------------+-------------------------------------------------------
| OS-DCF:diskConfig                    | MANUAL
| OS-EXT-AZ:availability_zone          | nova
| OS-EXT-SRV-ATTR:host                 | Ubuntu1204Server
| OS-EXT-SRV-ATTR:hypervisor_hostname  | Ubuntu1204Server
| OS-EXT-SRV-ATTR:instance_name        | instance-00000001
| OS-EXT-STS:power_state               | 0
| OS-EXT-STS:task_state                | networking
| OS-EXT-STS:vm_state                  | building
| OS-SRV-USG:launched_at               | -
| OS-SRV-USG:terminated_at             | -
| accessIPv4                           |
| accessIPv6                           |
| adminPass                            | js6ZnNjX82rQ
| config_drive                         |
| created                              | 2014-08-15T00:08:11Z
| flavor                               | m1.tiny (1)
| hostId                               | 930764f06a4a5ffb8e433b24efce63fd5096ddaee5e62b439169fbd
| id                                   | 19b6049e-fe69-416a-b6f1-c02afaf54a34
| image                                | cirros-0.3.2-x86_64-uec (e8dc8305-c9de-42a8-b3d1-6b1bc9
| key_name                             | -
| metadata                             | {}
| name                                 | vm-demo
| os-extended-volumes:volumes_attached | []
| progress                             | 0
| security_groups                      | default
```

```
| status                                | BUILD
| tenant_id                             | 81084a94769c4ce0accb6968c397a085
| updated                               | 2014-08-15T00:08:12Z
| user_id                               | 3d6c6119e5c94c258a26ab246cdcac12
+---------------------------------------+----------------------------------------------------------
```

11. Get tenant ids:

```
$ keystone tenant-list | tee tenant-list.log
+----------------------------------+--------------------+---------+
|                id                |        name        | enabled |
+----------------------------------+--------------------+---------+
| 7320f8345acb489e8296ddb3b1ad1262 |       admin        |   True  |
| 81084a94769c4ce0accb6968c397a085 |        demo        |   True  |
| 315d4a5892ed4da1bdf717845e8959df | invisible_to_admin |   True  |
| b590e27c87fa40c18c850954dca4c879 |      service       |   True  |
+----------------------------------+--------------------+---------+

$ ADMIN_ID=`grep " admin " tenant-list.log | awk '{print $2}'`
$ DEMO_ID=`grep " demo " tenant-list.log | awk '{print $2}'`
```

10.4 Creating a Congress Policy

At this point, demo's vm exists and its port is connected to an network belonging to admin. This is a violation of the policy. Now you will add the congress policy to detect the violation.

12. Add a rule that detects when a VM is connected to a port belonging to a different group:

```
CongressClient:
$ openstack congress policy rule create classification "error(name2) :- neutronv2:ports(a, tenan
+---------+-------------------------------------------------------------------------------+
| Field   | Value                                                                         |
+---------+-------------------------------------------------------------------------------+
| comment | None                                                                          |
| id      | c235f3a6-44cc-4222-8201-80188f9601ce                                          |
| name    | None                                                                          |
| rule    | error(name2) :-                                                               |
|         |     neutronv2:ports(a, tenant_id, c, network_id, e, f, g, device_id, i),      |
|         |     nova:servers(device_id, name2, c2, d2, tenant_id2, f2, g2, h2),           |
|         |     neutronv2:networks(network_id, tenant_id3, c3, d3, e3, f3),               |
|         |     not same_group(tenant_id, tenant_id2)                                     |
+---------+-------------------------------------------------------------------------------+
```

or:

```
$ curl -X POST localhost:1789/v1/policies/<classification-id>/rules -d '{"rule": "error(name2) :
{"comment": null, "id": "783ff249-6a52-4691-baf7-3cdfb8f9d200", "rule": "error(name2) :- \n    n
```

13. Add a rule that detects when a port is connected to a network belonging to a different group:

```
CongressClient:
$ openstack congress policy rule create classification "error(name2) :- neutronv2:ports(a, tenan
+---------+-------------------------------------------------------------------------------+
| Field   | Value                                                                         |
+---------+-------------------------------------------------------------------------------+
| comment | None                                                                          |
| id      | f7369e20-8b1b-4315-9b68-68197d740521                                          |
```

```
| name     | None                                                                    |
| rule     | error(name2) :-                                                         |
|          |     neutronv2:ports(a, tenant_id, c, network_id, e, f, g, device_id, i), |
|          |     nova:servers(device_id, name2, c2, d2, tenant_id2, f2, g2, h2),     |
|          |     neutronv2:networks(network_id, tenant_id3, c3, d3, e3, f3),         |
|          |     not same_group(tenant_id2, tenant_id3)                              |
+----------+-------------------------------------------------------------------------+
```

or:

```
$ curl -X POST localhost:1789/v1/policies/<classification-id>/rules -d '{"rule": "error(name2) :
{"comment": null, "id": "f7708411-a0fc-4ee8-99e6-0f4be7e980ff", "rule": "error(name2) :- \n    n
```

14. Define a table mapping a tenant_id to any other tenant in the same group:

```
CongressClient:
$ openstack congress policy rule create classification "same_group(x, y) :- group(x, g), group(y
+----------+--------------------------------------+
| Field    | Value                                |
+----------+--------------------------------------+
| comment  | None                                 |
| id       | a3d0cfcb-d013-4578-ac60-3e8cefb4ab35 |
| name     | None                                 |
| rule     | same_group(x, y) :-                  |
|          |     group(x, g),                    |
|          |     group(y, g)                      |
+----------+--------------------------------------+
```

or:

```
$ curl -X POST localhost:1789/v1/policies/<classification-id>/rules -d '{"rule": "same_group(x,
{"comment": null, "id": "e919d62e-b9af-4b50-a22c-c266379417b8", "rule": "same_group(x, y) :- \n
```

15. Create a table mapping tenant_id to a group name. admin and demo are in two separate groups called "IT" and "Marketing" respectively. In practice, this "group" table would receive group membership information from a system like Keystone or ActiveDirectory. In this tutorial, we'll populate the group table with membership information manually:

```
CongressClient:
$ openstack congress policy rule create classification "group(\"$ADMIN_ID\", \"IT\") :- true"
+----------+------------------------------------------------------+
| Field    | Value                                                |
+----------+------------------------------------------------------+
| comment  | None                                                 |
| id       | 97a6aeb0-0c9d-493b-8b0c-77691c1c3547                 |
| name     | None                                                 |
| rule     | group("14a3eb4f5b234b578ff905a4bec71605", "IT") :-   |
|          |     true()                                           |
+----------+------------------------------------------------------+
```

or:

```
$ curl -X POST localhost:1789/v1/policies/<classification-id>/rules -d "{\"rule\": \"group(\\\"$
{"comment": null, "id": "4a51b768-1458-4c68-881f-1cf2f1edb344", "rule": "group(\"14a3eb4f5b234b5
```

Then:

```
CongressClient:
$ openstack congress policy rule create classification "group(\"$DEMO_ID\", \"Marketing\") :- tr
+----------+------------------------------------------------------------+
```

```
| Field   | Value                                                          |
+---------+----------------------------------------------------------------+
| comment | None                                                           |
| id      | 67c0d86d-f7cf-4db1-9efa-4d46960a3905                           |
| name    | None                                                           |
| rule    | group("8f08a89de9c945d4ac7f945f1d93b676", "Marketing") :-      |
|         |     true()                                                     |
+---------+----------------------------------------------------------------+
```

or:

```
$ curl -X POST localhost:1789/v1/policies/<classification-id>/rules -d "{\"rule\": \"group(\\\"$
{"comment": null, "id": "e6b57c8f-ffd2-4acf-839c-83284519ae3c", "rule": "group(\"8f08a89de9c945d
```

10.5 Listing Policy Violations

Finally, we can print the error table to see if there are any violations (which there are).

16. List the errors. You should see one entry for "vm-demo":

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables/error/rows
{
  "results": [
    {
      "data": [
        "vm-demo"
      ]
    }
  ]
}
```

10.6 Fix the Policy Violation

17. To fix the policy violation, we'll remove the demo's port from admin's network:

```
$ neutron port-delete $PORT_ID
Deleted port: 066c5cfc-949e-4d56-ad76-15528c68c8b8
```

10.7 Relisting Policy Violations

18. Now, when print the error table it will be empty because there are no violations:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables/error/rows
[]
```

CHAPTER 11

Troubleshooting

So you've installed Congress with devstack as per the README, and now something is not behaving the way you think it should. Let's say you're using the policy that follows (from the tutorial), but the *error* table does not contain the rows you expect. In this document, we describe how to figure out what the problem is and hopefully how to fix it. At the end we also detail a collection of problems and tips for how to fix them.

```
  error(name2) :-
    neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m, n, device_id, p),
    nova:servers(device_id, name2, c2, d2, tenant_id2, f2, g2, h2),
    neutron:networks(a3, b3, c3, d3, e3, f3, tenant_id3, h3, i3, j3, network_id, l3),
    not same_group(tenant_id, tenant_id2)

  error(name2) :-
    neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m, n, device_id, p),
    nova:servers(device_id, name2, c2, d2, tenant_id2, f2, g2, h2),
    neutron:networks(a3, b3, c3, d3, e3, f3, tenant_id3, h3, i3, j3, network_id, l3),
    not same_group(tenant_id2, tenant_id3)

  same_group(x, y) :-
      group(x, g),
      group(y, g)

group("7320f8345acb489e8296ddb3b1ad1262", "IT") :- true()
group("81084a94769c4ce0accb6968c397a085", "Marketing") :- true()
```

11.1 Policy-engine troubleshooting

Make sure the policy engine knows about the rules you think it knows about. It is possible that the policy engine rejected a rule because of a syntax error. Let's assume you're using the *classification* policy.

Check: Ensure the policy engine has the right rules:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/rules
```

For example:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/rules
{
  "results": [
    {
      "comment": "None",
      "id": "2be98841-953d-44f0-91f6-32c5f4dd4f83",
```

```
      "rule": "group(\"d0a7ff9e5d5b4130a586a7af1c855c3e\", \"IT\") :- true()"
    },
    {
      "comment": "None",
      "id": "c01067ef-10e4-498f-8aaa-0c1cce1272a3",
      "rule": "group(\"e793326db18847e1908e791daa69a5a3\", \"Marketing\") :- true()"
    },
    {
      "comment": "None",
      "id": "3c6e48ee-2783-4b5e-94ff-aab00ccffd42",
      "rule": "error(name2) :- neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m,
    },
    {
      "comment": "None",
      "id": "36264def-d917-4f39-a6b0-4aaf12d4b349",
      "rule": "error(name2) :- neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m,
    },
    {
      "comment": "None",
      "id": "31922bb0-711c-43da-9f11-cef69828a2c4",
      "rule": "same_group(x, y) :- group(x, g), group(y, g)"
    }
  ]
}
```


It is also possible that you might have a typo in one of the table names that appear in the rules. To eliminate that possibility, ask the list of tables that occur in the rules and compare those to the ones the datasources export. You might also look for near-duplicates, such as *same_group* and *samegroup*, in case tables are spelled differently in different rules.

Check: Ensure there are no typos in any of the table names by asking for the list of tables occurring in the rules:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables
```

For example:


```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables
{
  "results": [
    {
      "id": "nova:servers"
    },
    {
      "id": "neutron:ports"
    },
    {
      "id": "group"
    },
    {
      "id": "neutron:networks"
    },
    {
      "id": "error"
    },
    {
      "id": "same_group"
    },
    {
      "id": "true"
```

```
    }
  ]
}
```

Next we want to check that tables have the rows we would expect. A good place to start is with the tables exported by external datasources like Nova and Neutron. If these tables are empty, that points to a problem with the datasources (see below for troubleshooting datasources). If they are not empty, it points to a problem with the rules:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables/<table-id>/rows
```

For example, below are the rows in the *neutron:ports* table. There are 2 rows (each of which represents a port), and each row has 16 columns:

```
 $ curl -X GET localhost:1789/v1/policies/<classification-id>/tables/neutron:ports/rows
{
  "results": [
    {
      "data": [
        "795a4e6f-7cc8-4052-ae43-80d4c3ad233a",
        "5f1f9b53-46b2-480f-b653-606f4aaf61fd",
        "1955273c-242d-46a6-8063-7dc9c20cbba9",
        "None",
        "ACTIVE",
        "",
        "True",
        "37eee894-a65f-414d-bd8c-a9363293000a",
        "e793326db18847e1908e791daa69a5a3",
        "None",
        "network:router_interface",
        "fa:16:3e:28:ab:0b",
        "4b7e5f9c-9ba8-4c94-a7d0-e5811207d26c",
        "882911e9-e3cf-4682-bb18-4bf8c559e22d",
        "c62efe5d-d070-4dff-8d9d-3df8ac08b0ec",
        "None"
      ]
    },
    {
      "data": [
        "ebeb4ee6-14be-4ba2-a723-fd62f220b6b9",
        "f999de49-753e-40c9-9eed-d01ad76bc6c3",
        "ad058c04-05be-4f56-a76f-7f3b42f36f79",
        "None",
        "ACTIVE",
        "",
        "True",
        "07ecce19-d7a4-4c79-924e-1692713e53a7",
        "e793326db18847e1908e791daa69a5a3",
        "None",
        "compute:None",
        "fa:16:3e:3c:0d:13",
        "af179309-65f7-4662-a087-e583d6a8bc21",
        "149f4271-41ca-4b1a-875b-77909debbeac",
        "bc333f0f-b665-4e2b-97db-a4dd985cb5c8",
        "None"
      ]
    }
  ]
}
```

After checking the tables exported by datasources like Nova and Neutron, it is useful to check the contents of the other tables that build upon those tables.

In our running example, we should check the rows of the *group* table. Here we see what we expect: that there are two users, each of which belongs to a different group:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables/group/rows
{
  "results": [
    {
      "data": [
        "d0a7ff9e5d5b4130a586a7af1c855c3e",
        "IT"
      ]
    },
    {
      "data": [
        "e793326db18847e1908e791daa69a5a3",
        "Marketing"
      ]
    }
  ]
}
```

Once you have found a table that contains the wrong rows, it may be obvious looking at the rules for that table what the problem is. But if there are many rules or if some of the rules are long, it can be difficult to pinpoint the problem. When this happens, you can ask for a trace that describes how the rows of that table were computed:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables/<table-id>/rows?trace=true
```

The trace is similar to a function-call trace. It uses the following annotations:

```
* Call Q: a query for all the rows that match Q
* Exit Q: successfully discovered row Q
* Fail Q: failure to find a row matching Q (in the current context)
* Redo Q: attempt to find another row matching Q
```

In our example, we know the contents of the *error* table is empty, but all of the tables used to construct *error* look reasonable. So we ask for a trace showing why the *error* table is empty. The trace is returned as a string and be quite large.:

```
$ curl -X GET localhost:1789/v1/policies/<classification-id>/tables/error/rows?trace=true
{
  "results": [],
  "trace": "Clas  : Call: error(x0)\nClas  : | Call: neutron:ports(a, b, c, d, e, f, g, network_id, t
}
```

We can print the trace using 'printf <trace>' (without the quotes):

```
$ printf "Clas  : Call: error(x0)...
Clas  : Call: error(x0)
Clas  : | Call: neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m, n, device_id, p
Clas  : | Exit: neutron:ports("795a4e6f-7cc8-4052-ae43-80d4c3ad233a", "5f1f9b53-46b2-480f-b653-606f4a
Clas  : | Call: nova:servers("c62efe5d-d070-4dff-8d9d-3df8ac08b0ec", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Fail: nova:servers("c62efe5d-d070-4dff-8d9d-3df8ac08b0ec", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Redo: neutron:ports("795a4e6f-7cc8-4052-ae43-80d4c3ad233a", "5f1f9b53-46b2-480f-b653-606f4a
Clas  : | Exit: neutron:ports("ebeb4ee6-14be-4ba2-a723-fd62f220b6b9", "f999de49-753e-40c9-9eed-d01ad
Clas  : | Call: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Exit: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", "vm-demo", "c5dd62237226c4f2eadc
Clas  : | Call: neutron:networks(a3, b3, c3, d3, e3, tenant_id3, f3, g3, h3, "07ecce19-d7a4-4c79-924e
```

```
Clas  : | Fail: neutron:networks(a3, b3, c3, d3, e3, tenant_id3, f3, g3, h3, "07ecce19-d7a4-4c79-924e
Clas  : | Redo: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", "vm-demo", "c5dd62237226c4f2eadc
Clas  : | Fail: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Redo: neutron:ports("ebeb4ee6-14be-4ba2-a723-fd62f220b6b9", "f999de49-753e-40c9-9eed-d01ad
Clas  : | Fail: neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m, n, device_id, p

Clas  : | Call: neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m, n, device_id, p
Clas  : | Exit: neutron:ports("795a4e6f-7cc8-4052-ae43-80d4c3ad233a", "5f1f9b53-46b2-480f-b653-606f4a
Clas  : | Call: nova:servers("c62efe5d-d070-4dff-8d9d-3df8ac08b0ec", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Fail: nova:servers("c62efe5d-d070-4dff-8d9d-3df8ac08b0ec", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Redo: neutron:ports("795a4e6f-7cc8-4052-ae43-80d4c3ad233a", "5f1f9b53-46b2-480f-b653-606f4a
Clas  : | Exit: neutron:ports("ebeb4ee6-14be-4ba2-a723-fd62f220b6b9", "f999de49-753e-40c9-9eed-d01ad
Clas  : | Call: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Exit: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", "vm-demo", "c5dd62237226c4f2eadc
Clas  : | Call: neutron:networks(a3, b3, c3, d3, e3, tenant_id3, f3, g3, h3, "07ecce19-d7a4-4c79-924e
Clas  : | Fail: neutron:networks(a3, b3, c3, d3, e3, tenant_id3, f3, g3, h3, "07ecce19-d7a4-4c79-924e
Clas  : | Redo: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", "vm-demo", "c5dd62237226c4f2eadc
Clas  : | Fail: nova:servers("bc333f0f-b665-4e2b-97db-a4dd985cb5c8", x0, c2, d2, tenant_id2, f2, g2,
Clas  : | Redo: neutron:ports("ebeb4ee6-14be-4ba2-a723-fd62f220b6b9", "f999de49-753e-40c9-9eed-d01ad
Clas  : | Fail: neutron:ports(a, b, c, d, e, f, g, network_id, tenant_id, j, k, l, m, n, device_id, p
Clas  : Fail: error(x0)
```

Recall that there are 2 rules defining *error*. The part of the trace occurring before the line break is from one of the rules; the part of the trace after the line break is from the other. (The line break does not appear in the trace–we inserted it for the sake of pedagogy.)

Both rules join the tables neutron:ports, nova:servers, and neutron:networks. The trace shows the join being computed one row at a time. In this case, we see that there is some port (from neutron:ports) connected to a VM (from nova:servers) for which there is no record of the port's network (from neutron:networks). In this case, there is a row missing from neutron:networks: the one with ID 07ecce19-d7a4-4c79-924e-1692713e53a7.

At this point, it seems clear that the problem is with the Neutron datasource, not the rules.

11.2 Datasource troubleshooting

At this point, you believe the problem is with one of the datasources. The first thing to consider is whether Congress can properly connect to the associated cloud service. The best way to do that is to examine the tables that the problematic datasource is exporting. If the tables being exported by a service is empty, the datasource driver is not properly connecting to the datasource.

Check: Ensure each (relevant) datasource is exporting the tables the documentation says should be exported:

```
$ curl -X GET localhost:1789/v1/data-sources/<ds-id>/tables
```

To fix connection problems, do both of the following.

- Ensure the datasource component is enabled in devstack (if you're using devstack)
- Fix the configuration of the datasource by asking to see its current configuration, and if it is wrong, delete that datasource and create a new one with the proper configuration. Don't forget that datasources sometimes return different information for different username/password combinations.

Below are examples of how to list datasource configuration, delete an existing datasource, and create a new datasource:

```
# show list and configuration options for each
$ curl -X GET localhost:1789/v1/data-sources

# delete old datasource
```

```
$ curl -X DELETE http://127.0.0.1:1789/v1/data-sources/<ds-id>

# create new datasource
$ curl -X POST localhost:1789/v1/data-sources -d
 '{"config": {"username": "admin",
              "tenant_name": "admin",
              "password": "password",
              "auth_url": "http://127.0.0.1:5000/v2"},
   "driver": "neutronv2",
   "name": "neutronv2"}'
```

For example, below we see that the *neutron* datasource is exporting all the right tables:

```
$ curl -X GET localhost:1789/v1/data-sources/<neutron-id>/tables
{
  "results": [
    {
      "id": "ports.binding_capabilities"
    },
    {
      "id": "routers"
    },
    {
      "id": "ports.extra_dhcp_opts"
    },
    {
      "id": "ports.fixed_ips"
    },
    {
      "id": "ports"
    },
    {
      "id": "ports.fixed_ips_groups"
    },
    {
      "id": "ports.security_groups"
    },
    {
      "id": "networks.subnets"
    },
    {
      "id": "networks"
    },
    {
      "id": "security_groups"
    },
    {
      "id": "ports.address_pairs"
    }
  ]
}
```

Once the datasource is properly configured and is returning the proper list of tables, the next potential problem is that the rows of one of the tables are incorrect.

Check: Ensure the rows of each of the tables exported by the datasource are correct:

```
$ curl -X GET localhost:1789/v1/data-sources/<ds-id>/tables/<table-name>/rows
```

To check that the rows are correct, you'll need to look at the datasource's schema to see what each column means and compare that to the current contents of the actual datasource.

For example, we can look at the rows of the *networks* table in the *neutron* service. In this example, there are two rows. Each row is the value of the *data* key:

```
$ curl -X GET localhost:1789/v1/data-sources/<neutron-id>/tables/networks/rows
{
  "results": [
    {
      "data": [
        "ACTIVE",
        "public",
        "faf2c578-2893-11e4-b1e3-fa163ebf1676",
        "None",
        "True",
        "d0a7ff9e5d5b4130a586a7af1c855c3e",
        "None",
        "True",
        "False",
        "0d31bf61-c749-4791-8cf2-345f624bad8d",
        "None"
      ]
    },
    {
      "data": [
        "ACTIVE",
        "private",
        "faf31ec4-2893-11e4-b1e3-fa163ebf1676",
        "None",
        "True",
        "e793326db18847e1908e791daa69a5a3",
        "None",
        "False",
        "False",
        "37eee894-a65f-414d-bd8c-a9363293000a",
        "None"
      ]
    }
  ]
}
```

Compare these rows to the schema for this datasource:

```
$ curl -X GET localhost:1789/v1/data-sources/<neutron-id>/schema

{'tables': [
  ...
  {'columns': [
    {'description': 'None', 'name': 'status'},
    {'description': 'None', 'name': 'name'},
    {'description': 'None', 'name': 'subnet_group_id'},
    {'description': 'None', 'name': 'provider_physical_network'},
    {'description': 'None', 'name': 'admin_state_up'},
    {'description': 'None', 'name': 'tenant_id'},
    {'description': 'None', 'name': 'provider_network_type'},
    {'description': 'None', 'name': 'router_external'},
    {'description': 'None', 'name': 'shared'},
    {'description': 'None', 'name': 'id'},
```

```
    {'description': 'None', 'name': 'provider_segmentation_id'}],
   'table_id': 'networks'},
  ...]}
```

The schema says the 1st column is the network's status, which in both the rows above, has the value "ACTIVE". The schema says the 10th column is the network's ID, which in the two rows above are 0d31bf61-c749-4791-8cf2-345f624bad8d and 37eee894-a65f-414d-bd8c-a9363293000a. Notice that the missing network from our earlier analysis of the policy trace is missing from here as well: 07ecce19-d7a4-4c79-924e-1692713e53a7.

This points to a problem in the configuration of the datasource, in particular using a username/password combination that does not return all the networks.

11.3 Message bus troubleshooting

One thing that sometimes happens is that the datasource has the right rows, but the policy engine does not. For example, the *networks* table of the *neutron* service is not identical to the *neutron:networks* table. Typically, this means that the policy engine simply hasn't received and processed the update from the datasource on the message bus. Waiting several seconds should fix the problem.

Check: Compare the policy engine's version of a table to the datasource's version. Remember that the policy engine's name for table T in datasource D is D:T, e.g. the *networks* table for service *neutron* is named *neutron:networks*:

```
curl -X GET localhost:1789/v1/policies/classification/tables/<ds-name>:<table-name>/rows
curl -X GET localhost:1789/v1/data-sources/<ds-id>/tables/<table-name>/rows
```

Warning: In the current datasource drivers for Neutron and Nova, a single API call can generate several different tables. Each table is sent independently on the message bus, which can lead to inconsistencies between tables (e.g. table *neutron:ports* might be out of sync with *neutron:ports.security_groups*). This kind of data skew is an artifact of our implementation and will be addressed in a future release. The best solution currently is to wait until all the messages from the latest polling reach the policy engine.

A similar problem can arise when two datasources are out of sync with each other. This happens because the two datasources are polled independently. If something changes one of the datasources in between when those datasources are polled, the local cache Congress has will be out of sync. In a future release, we will provide machinery for mitigating the impact of these kinds of synchronization problems.

11.4 Production troubleshooting

Another class of problems arises most often in production deployments. Here we give a couple of problems encountered in production deployments along with tips for solving them.

1. Log file too big

Symptom: slower than normal performance, log size not changing

Solution: set up logrotate (a Linux service). In the directory */etc/logrotate.d*, include a file *congress* and add an entry such as the one shown below. (Here we're assuming the congress log is in /var/log/congress):

```
/var/log/congress
{
  rotate 7
  daily
  missingok
  notifempty
  delaycompress
```

```
    compress
    endscript
}
```

CHAPTER 12

Indices and tables

- *genindex*
- *modindex*
- *search*