ovsdbapp Documentation

Release 2.8.1.dev2

OpenStack Foundation

Sep 06, 2024

Contents

A library for creating OVSDB applications

The ovdsbapp library is useful for creating applications that communicate via Open_vSwitchs OVSDB protocol (https://tools.ietf.org/html/rfc7047). It wraps the Python ovs and adds an event loop and friendly transactions.

- Free software: Apache license
- Source: https://opendev.org/openstack/ovsdbapp/
- Bugs: https://bugs.launchpad.net/ovsdbapp

Features:

- An thread-based event loop for using ovs.db.Idl
- Transaction support
- Native OVSDB communication

1 Installation

At the command line:

```
$ pip install ovsdbapp
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv ovsdbapp
$ pip install ovsdbapp
```

2 Library User Guide

This document describes OVSDBapp concepts and best practices to enable writing effective OVSDBapp-based applications.

2.1 Overview

Overview

OVSDBapp is a library to make it easier to write applications that interact with an Open vSwitch database server. It allows the user to separate support for a particular OVSDB schema and the backend method of communication with the OVSDB server.

OVSDBapp Concepts

API
: The interface that an application will use for reading or modifying entries in the OVS database. Whatever backend communication method is used, as long as user code only accesses methods in this API, no user code should need to be changed when swapping between backends.

Backend
: The Backend handles the communication with Open vSwitch. Originally, there were two OVSDBapp backends: 1) one that used the ovs-vsctl CLI utility to interact with the OVS database and 2) one that maintains a persistent connection to an OVSDB server using the python-ovs library. Currently, only the python-ovs backend is being maintained.

Command
: OVSDBapp uses the Command Pattern to isolate individual units of work that will be run as part of an OVSDB transaction.

Event
: OVSDB provides the ability to monitor database changes as they happen. OVSDBapp backends each implement the `RowEvent` and `RowEventHandler` to handle delivering these events to user code.

API Implementations:
: The backend-specific implementation of an OVSDBapp API. Only this code should need to be implemented to support a new backend. All other user code should be backend-agnostic.

Schema
: The OVSDB database schema for which the API is implemented. In current ovsdbapp code, the schema and API are intrinsically linked in a 1:1 manner, but ultimately they are independent. User code could easily define an API specific to their application that encompasses multiple OVSDB schemas as long as the Backend supported it.

Transaction
: An OVSDB transaction consisting of one or more Commands.

Virtual Environment
: OVSDBapp supports running OVS and OVN services in a virtual environment. This is primarily used for testing.

2.2 Tutorial

Tutorial

Open vSwitch Environment Setup

This tutorial will use the Open vSwitch sandbox environment from the OVS source tree. For the sake of simplicity, we will build OVS without SSL support. You will need git, C development tools, automake, autoconf, and libtool. See the Installing Open vSwitch instructions for build requirements and more detailed build instructions.

```
git clone https://github.com/openvswitch/ovs
cd ovs
./boot.sh
./configure --disable-ssl --enable-shared
export OVS_SRCDIR=`pwd`
make -j $(nproc) sandbox
```

Backend Setup

While the original ovs-vsctl -based backend required no setup, other backends may. For example, the python-ovs IDL backend maintains a constant connection to ovsdb-server and requires an IDL class to be instantiated and passed to an OVSDBapp IDL backend Connection object.

```
import os
from ovs.db import idl as ovs_idl
from ovsdbapp.backend.ovs_idl import connection
from ovsdbapp.schema.open_vswitch import impl_idl

src_dir = os.getenv("OVS_SRCDIR")
run_dir = os.getenv("OVS_RUNDIR", "/var/run/openvswitch")
schema_file = os.path.join(src_dir, "vswitchd", "vswitch.ovsschema")
db_sock = os.path.join(run_dir, "db.sock")
remote = f"unix:{db_sock}"

schema_helper = ovs_idl.SchemaHelper(schema_file)
```

(continues on next page)

(continued from previous page)

```
schema_helper.register_all()
idl = ovs_idl.Idl(remote, schema_helper)
conn = connection.Connection(idl=idl, timeout=60)

api = impl_idl.OvsdbIdl(conn)
```

Using the API

Each API definition varies based on the schemas it supports and what the app requires. There is built-in support for many common OVS and OVN-related schemas, but it is possible that the APIs defined for these are not optimized for a given apps use cases. It may often make sense for apps to define APIs separate from those that are in ovsdbapp.

With that said, any api that inherits from ovsdbapp.api.API will at least have methods defined for the standard generic OVSDB DB operations found described in the ovs-vsctl manpage under Database Commands.

- list
- find
- get
- set
- add
- remove
- clear
- create
- destroy

They are all prefixed with db_ (e.g. list becomes db_list) and have an interface similar to that used by ovs-vsctl, ovn-nbctl, ovn-sbctl, etc. db_list() and db_find() return results as lists of dicts with each dict representing a row, with keys as the column names. Later versions added db_list_rows() and db_find_rows() to return lists of RowView objects.

API commands that interact with the OVSDB server typically return an instance of a subclass of ovsdbapp.api.Command. These objects hold the state of a request that will be sent to an OVSDB server as part of a transaction. They can be thought of as the equivalent of queries in SQL.

For a Command to be sent to the OVSDB server, it must be attached to a transaction, and committed. For single commands, this can be done with execute():

```
results = api.db_list("Open_vSwitch").execute(check_error=True)
```

This implicitly creates a transaction, adds the Command returned by db_list() to that transaction, calls commit() on the transaction, and returns the result that is stored on the Command object. It is the equivalent of:

```
txn = api.create_transaction(check_error=True)
list_cmd = api.db_list("Open_vSwitch")
```

(continues on next page)

(continued from previous page)

```
txn.add(list_cmd)
txn.commit()
results = list_cmd.result
```

That API also defines transaction(), a context manager, that makes multi-command transactions easier.

```
with api.transaction(check_error=True) as txn:
    br_cmd = txn.add(api.db_create("Bridge", name="test-br"))
    txn.add(api.db_add("Open_vSwitch", ".", "bridges", br_cmd))
```

There are some things to note with the above code. First, is that Transaction.add() returns the Command object that is passed to it. In the case of the db_create() command, the row it will create can be referenced in other commands in the same transaction. Second, if a table is defined as having at most one row, like the Open_vSwitch table, instead of passing its UUID, . can be passed. Lastly, note that we are creating a Bridge row and adding it to the Open_vSwitch rows bridges field. The Bridge table is not set as a root table in the Open_vSwitch schema. What this means is that if no row in a root table references this Bridge, ovsdb-server will automatically clean up this row. The Open_vSwitch table is a root table, so referencing the bridge in that row prevents the bridge that was just created from being immediately removed.

3 Contributing

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

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

If you already have a good understanding of how the system works and your OpenStack accounts are set up, you can skip to the development workflow section of this documentation to learn how changes to OpenStack should be submitted for review via the Gerrit tool:

> https://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/ovsdbapp