
VOLTTRON Documentation

Release 8.0 Release Candidate

The VOLTTRON Community

Nov 11, 2020

Introduction

1	Features	3
2	Indices and tables	203
	Index	205



VOLTTRON™ is an open-source platform for distributed sensing and control. The platform provides services for collecting and storing data from buildings and devices and provides an environment for developing applications that interact with that data.

- a secure message bus allowing connectivity between modules on individual platforms and between platform instances in large scale deployments
- a flexible agent framework allowing users to adapt the platform to their unique use-cases
- a configurable driver framework for collecting data from and sending control signals to buildings and devices
- automatic data capture and retrieval through our historian framework
- an extensible web framework allowing users and services to securely connect to the platform from anywhere

VOLTTRON™ is open source and publicly available from [GitHub](#). The project is supported by the U.S. Department of Energy and receives ongoing updates from a team of core developers at PNNL. The VOLTTRON team encourages and appreciates community involvement including issues and pull requests on Github, meetings at our bi-weekly office-hours and on Slack. To be invited to office-hours or slack, please [send the team an email](#).

1.1 What is VOLTTRON?

VOLTTRON™ is a software platform on which software modules called “agents” and device driver modules connect to a message bus to interact. Users may configure included drivers for industry standard device communication protocols such as BACnet or Modbus, or develop and configure their own. Additionally, agents can be installed or developed to perform a vast variety of tasks.

1.1.1 Design Philosophy

VOLTTRON was designed by Pacific Northwest National Laboratory to service building efficiency, building-grid integration and transactive controls systems. These systems are working to improve energy efficiency and resiliency in critical infrastructure. To this end, VOLTTRON was built with the following pillars in mind:

- Cost-Effectiveness - Open source software (free to users) and can be hosted on inexpensive computing resources
- Scalability - Can be used in one building or a fleet of buildings

- Interoperability - Enables interaction/connection with various systems and subsystems, in and out of the energy sector
- Security - Underpinned with a robust security foundation to combat today's cyber vulnerabilities and attacks

1.1.2 Basic Components

- Message bus - The VOLTTRON message bus uses [message queueing software](#) to exchange messages between agents and drivers installed on the platform. VOLTTRON messages are exchanged using a publish/subscriber paradigm, or messages can be routed to specific agents through the bus using remote procedure calls.
- Agents - Agents are software modules which autonomously perform a set of desired functions on behalf of a user. VOLTTRON agents are often used to collect data, send control signals to devices, implement control algorithms or perform simulations.
- Drivers - Drivers can be installed on the platform and configured to communicate with industrial or Internet of Things devices. Drivers provide a set of pre-defined functions which can be mapped to device communication methods to read or set values on the device.
- Historians - Historians are special purpose agents which are used to subscribe to sources broadcasting on the message bus and store their messages for later use.
- Web Framework - The VOLTTRON web framework

1.2 How Does it Work?

The VOLTTRON platform is built around the concept of software agents. Software agents perform autonomous functions on behalf of a user. The VOLTTRON platform was created to allow a suite of agents installed by a user to work together to achieve the user's goals.

1.2.1 Major Components

The platform comprises several components that allow agents to operate and connect to the platform.

- The Message Bus is central to the platform. All other VOLTTRON components communicate through it using VOLTTRON Interconnect Protocol (VIP). VIP implements the publish/subscribe paradigm over a variety of topics or directed communication using Remote Procedure Calls.
- Agents on the platform extend the base agent which provides a VIP connection to the message bus and an agent lifecycle. Agents subscribe to topics which allow it to read. The agent lifecycle is controlled by the Agent Instantiation and Packaging (AIP) component which launches agents in an agent execution environment.
- The Master Driver Agent can be configured with a number of driver configurations and will spawn corresponding driver instances. Each driver instance provides functions for collecting device data and setting values on the device. These functions implement device protocol or remote communication endpoint interfaces. Driver data is published to the message bus or if requested by an agent will be delivered in an RPC response.
- Agents can control devices by interacting with the Actuator Agent to schedule and send commands.
- The Historian framework subscribes to data published on the messages bus and stores it to a database or file, or sends it to another location.

1.2.2 Usability Components

Usability components exist to enhance the base capabilities of the platform for deployments.

- VOLTTTRON Control is the command line interface to controlling a platform instance. VOLTTTRON Control can be used to operate agents, configure drivers, get status and health details, etc.
- Data collection, command and control can be achieved in large deployments by connecting multiple platform instances.
- VOLTTTRON Central is an agent which can be installed on a platform to provide a single management interface to multiple VOLTTTRON platform instances.
- JSON, static and websocket endpoints can be registered to agents via the Web Framework and platform web server. This allows remote agent communication as well as for agents to serve web pages.

1.3 Installing the Platform

VOLTTTRON is written in Python 3.6+ and runs on Linux Operating Systems. For users unfamiliar with those technologies, the following resources are recommended:

- [Python 3.6 Tutorial](#)
- [Linux Tutorial](#)

This guide will specify commands to use to successfully install the platform on supported Linux distributions, but a working knowledge of Linux will be helpful for troubleshooting and may improve your ability to get more out of your deployment.

Note: Volttron version 7.0rc1 is currently tested for Ubuntu versions 18.04 and 18.10 as well as Linux Mint version 19.3. Version 6.x is tested for Ubuntu versions 16.04 and 18.04 as well as Linux Mint version 19.1.

1.3.1 Step 1 - Install prerequisites

The following packages will need to be installed on the system:

- git
- build-essential
- python3.6-dev
- python3.6-venv
- openssl
- libssl-dev
- libevent-dev

On **Debian-based systems**, these can all be installed with the following command:

```
sudo apt-get update
sudo apt-get install build-essential python3-dev python3-venv openssl libssl-dev_
↪ libevent-dev git
```

On Ubuntu-based systems, available packages allow you to specify the Python3 version, 3.6 or greater is required (Debian itself does not provide those packages).

```
sudo apt-get install build-essential python3.6-dev python3.6-venv openssl libssl-dev ↵  
↪ libevent-dev git
```

On arm-based systems (including, but not limited to, Raspbian), you must also install libffi-dev, you can do this with:

```
sudo apt-get install libffi-dev
```

Note: On arm-based systems, the available apt package repositories for Raspbian versions older than buster (10) do not seem to be able to be fully satisfied. While it may be possible to resolve these dependencies by building from source, the only recommended usage pattern for VOLTTRON 7 and beyond is on raspberry pi OS 10 or newer.

On **Redhat or CENTOS systems**, these can all be installed with the following command:

```
sudo yum update  
sudo yum install make automake gcc gcc-c++ kernel-devel python3-devel openssl openssl-  
↪ devel libevent-devel git
```

Warning: Python 3.6 or greater is required, please ensure you have installed a supported version with `python3 --version`

If you have an agent which requires the pyodbc package, install the following additional requirements:

- freetds-bin
- unixodbc-dev

On **Debian-based systems** these can be installed with the following command:

```
sudo apt-get install freetds-bin unixodbc-dev
```

On **Redhat or CentOS systems**, these can be installed from the Extra Packages for Enterprise Linux (EPEL) repository:

```
sudo yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.  
↪ rpm  
sudo yum install freetds unixODBC-devel
```

Note: The above command to install the EPEL repository is for Centos/Redhat 8. Change the number to match your OS version. EPEL packages are included in Fedora repositories, so installing EPEL is not required on Fedora.

It may be possible to deploy VOLTTRON on a system not listed above but may involve some troubleshooting and dependency management on the part of the user.

1.3.2 Step 2 - Clone VOLTTRON code

Repository Structure

There are several options for using the VOLTTRON code depending on whether you require the most stable version of the code or want the latest updates as they happen. In order of decreasing stability and increasing currency:

- *Master* - Most stable release branch, current major release is 7.0. This branch is default.

- *develop* - contains the latest *finished* features as they are developed. When all features are stable, this branch will be merged into *Master*.

Note: This branch can be cloned by those wanting to work from the latest version of the platform but should not be used in deployments.

- Features are developed on “feature” branches or developers’ forks of the main repository. It is not recommended to clone these branches except for exploring a new feature.

Note: VOLTTRON versions 6.0 and newer support two message buses - ZMQ and RabbitMQ.

```
git clone https://github.com/VOLTTRON/volttron --branch <branch name>
```

1.3.3 Step 3 - Setup virtual environment

The *bootstrap.py* script in the VOLTTRON root directory will create a [virtual environment](#) and install the package’s Python dependencies. Options exist for upgrading or rebuilding existing environments, and for adding additional dependencies for optional drivers and agents included in the repository.

Note: The `--help` option for *bootstrap.py* can be specified to display all available optional parameters.

Steps for ZeroMQ

Run the following command to install all required packages:

```
cd <volttron clone directory>
python3 bootstrap.py
```

Then activate the Python virtual environment:

```
source env/bin/activate
```

Proceed to step 4.

Note: You can deactivate the environment at any time by running *deactivate*.

Steps for RabbitMQ

Step 1 - Install Erlang packages

For RabbitMQ based VOLTTRON, some of the RabbitMQ specific software packages have to be installed.

On Debian based systems and CentOS 6/7

If you are running a Debian or CentOS system, you can install the RabbitMQ dependencies by running the “rabbit_dependencies.sh” script, passing in the OS name and appropriate distribution as parameters. The following are supported:

- *debian bionic* (for Ubuntu 18.04)
- *debian xenial* (for Ubuntu 16.04 or Linux Mint 18.04)
- *debian stretch* (for Debian Stretch)
- *debian buster* (for Debian Buster)
- *raspbian buster* (for Raspbian/Raspberry Pi OS Buster)

Example command:

```
./scripts/rabbit_dependencies.sh debian xenial
```

Alternatively

You can download and install Erlang from [Erlang Solutions](<https://www.erlang-solutions.com/resources/download.html>). Please include OTP/components - ssl, public_key, asn1, and crypto. Also lock your version of Erlang using the [yum-plugin-versionlock](<https://access.redhat.com/solutions/98873>)

Note:

Currently VOLTTRON only officially supports specific versions of Erlang for each operating system:

- 1:22.1.8.1-1 for Debian
 - 1:21.2.6+dfsg-1 for Raspbian
 - Specific Erlang 21.x versions correspond to CentOS versions 6, 7, and 8, these can be found [here](#)
-

Step 2 - Configure hostname

Make sure that your hostname is correctly configured in /etc/hosts. See (<<https://stackoverflow.com/questions/24797947/os-x-and-rabbitmq-error-epmd-error-for-host-xxx-address-cannot-connect-to-host>>). If you are testing with VMs make please make sure to provide unique host names for each of the VMs you are using.

The hostname should be resolvable to a valid IP when running on bridged mode. RabbitMQ checks for this during initial boot. Without this (for example, when running on a VM in NAT mode) RabbitMQ start-up would fail with the error “unable to connect to empd (port 4369) on <hostname>.”

Note: RabbitMQ startup error would show up in the VM’s syslog (/var/log/messages) file and not in RabbitMQ logs (/var/log/rabbitmq/rabbitmq@hostname.log)

Step 3 - Bootstrap the environment

```
cd volttron
python3 bootstrap.py --rabbitmq [optional install directory. defaults to <user_home>/
↳rabbitmq_server]
```

This will build the platform and create a virtual Python environment and dependencies for RabbitMQ. It also installs RabbitMQ server as the current user. If an install path is provided, that path should exist and the user should have write permissions. RabbitMQ will be installed under *<install dir>/rabbitmq_server-3.7.7*. The rest of the documentation refers to the directory *<install dir>/rabbitmq_server-3.7.7* as *\$RABBITMQ_HOME*.

Note: There are many additional *options for bootstrap.py* for including dependencies, altering the environment, etc.

You can check if the RabbitMQ server is installed by checking its status:

```
service rabbitmq status
```

Note: The *RABBITMQ_HOME* environment variable can be set in *~/.bashrc*. If doing so, it needs to be set to the RabbitMQ installation directory (default path is *<user_home>/rabbitmq_server/rabbitmq_server-3.7.7*)

```
echo 'export RABBITMQ_HOME=$HOME/rabbitmq_server/rabbitmq_server-3.7.7'|sudo tee --
↳append ~/.bashrc
source ~/.bashrc
$RABBITMQ_HOME/sbin/rabbitmqctl status
```

Step 4 - Activate the environment

```
source env/bin/activate
```

Note: You can deactivate the environment at any time by running *deactivate*.

Step 5 - Configure RabbitMQ setup for VOLTRON

```
vcfg --rabbitmq single [optional path to rabbitmq_config.yml]
```

Refer to *[examples/configurations/rabbitmq/rabbitmq_config.yml]*(*examples/configurations/rabbitmq/rabbitmq_config.yml*) for a sample configuration file. At a minimum you will need to provide the host name and a unique common-name (under *certificate-data*) in the configuration file.

Note: common-name must be unique and the general convention is to use *<volttron instance name>-root-ca*.

Running the above command without the optional configuration file parameter will cause the user to be prompted for all the required data in the command prompt. “vcfg” will use that data to generate a *rabbitmq_config.yml* file in the *VOLTRON_HOME* directory.

Note: If the above configuration file is being used as a basis for creating your own configuration file, be sure to update it with the hostname of the deployment (this should be the fully qualified domain name of the system).

This script creates a new virtual host and creates SSL certificates needed for this VOLTTRON instance. These certificates get created under the subdirectory “certificates” in your VOLTTRON home (typically in ~/.volttron). It then creates the main VIP exchange named “volttron” to route message between the platform and agents and alternate exchange to capture unroutable messages.

Note: We configure the RabbitMQ instance for a single volttron_home and volttron_instance. This script will confirm with the user the volttron_home to be configured. The VOLTTRON instance name will be read from volttron_home/config if available, if not the user will be prompted for VOLTTRON instance name. To run the scripts without any prompts, save the the VOLTTRON instance name in volttron_home/config file and pass the VOLTTRON home directory as a command line argument. For example: `vcfg -vhome /home/vdev/new_vhome -rabbitmq single`

The Following are the example inputs for `vcfg -rabbitmq single` command. Since no config file is passed the script prompts for necessary details.

```
Your VOLTTRON_HOME currently set to: /home/vdev/new_vhome2

Is this the volttron you are attempting to setup? [Y]:
Creating rmq config yml
RabbitMQ server home: [/home/vdev/rabbitmq_server/rabbitmq_server-3.7.7]:
Fully qualified domain name of the system: [cs_cbox.pnl.gov]:

Enable SSL Authentication: [Y]:

Please enter the following details for root CA certificates
Country: [US]:
State: Washington
Location: Richland
Organization: PNNL
Organization Unit: Volttron-Team
Common Name: [volttron1-root-ca]:
Do you want to use default values for RabbitMQ home, ports, and virtual host: [Y]: N
Name of the virtual host under which RabbitMQ VOLTTRON will be running: [volttron]:
AMQP port for RabbitMQ: [5672]:
http port for the RabbitMQ management plugin: [15672]:
AMQPS (SSL) port RabbitMQ address: [5671]:
https port for the RabbitMQ management plugin: [15671]:
INFO:rmq_setup.py:Starting rabbitmq server
Warning: PID file not written; -detached was passed.
INFO:rmq_setup.py:**Started rmq server at /home/vdev/rabbitmq_server/rabbitmq_server-
↳3.7.7
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1):_
↳localhost
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1):_
↳localhost
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1):_
↳localhost
INFO:rmq_setup.py:
Checking for CA certificate

INFO:rmq_setup.py:
Root CA (/home/vdev/new_vhome2/certificates/certs/volttron1-root-ca.crt) NOT Found._
↳Creating root ca for volttron instance
```

(continues on next page)

(continued from previous page)

```

Created CA cert
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1):_
↪localhost
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1):_
↪localhost
INFO:rmq_setup.pyc:**Stopped rmq server
Warning: PID file not written; -detached was passed.
INFO:rmq_setup.pyc:**Started rmq server at /home/vdev/rabbitmq_server/rabbitmq_server-
↪3.7.7
INFO:rmq_setup.pyc:

#####

Setup complete for volttron home /home/vdev/new_vhome2 with instance name=volttron1
Notes:

- Please set environment variable `VOLTTTRON_HOME` to `/home/vdev/new_vhome2` before_
↪starting volttron

- On production environments, restrict write access to
  /home/vdev/new_vhome2/certificates/certs/volttron1-root-ca.crt to only admin user.
↪ For example: sudo chown root /home/vdev/new_vhome2/certificates/certs/volttron1-
↪root-ca.crt

- A new admin user was created with user name: volttron1-admin and password=default_
↪passwd.
  You could change this user's password by logging into <https://cs_cbox.pnl.
↪gov:15671/> Please update /home/vdev/new_vhome2/rabbitmq_config.yml if you change_
↪password

#####

```

1.3.4 Test the VOLTTTRON Deployment

We are now ready to start VOLTTTRON instance. If configured with RabbitMQ message bus a config file would have been generated in `$VOLTTTRON_HOME/config` with the entry `message-bus=rmq`. If you need to revert back to ZeroMQ based VOLTTTRON, you will have to either remove the `message-bus` parameter or set it to the default “zmq” in `$VOLTTTRON_HOME/config`.

The following command starts volttron process in the background:

```
volttron -vv -l volttron.log&
```

This enters the virtual Python environment and then starts the platform in debug (vv) mode with a log file named `volttron.log`. Alternatively you can use the utility script `start-volttron` script that does the same.

```
./start-volttron
```

To stop the platform, use the `vct` command:

```
volttron-ctl shutdown --platform
```

or use the included `stop-volttron` script:

```
./stop-volttron
```

Warning: If you plan on running VOLTTRON in the background and detaching it from the terminal with the `disown` command be sure to redirect `stderr` and `stdout` to `/dev/null`. Some libraries which VOLTTRON relies on output directly to `stdout` and `stderr`. This will cause problems if those file descriptors are not redirected to `/dev/null`

```
#To start the platform in the background and redirect stderr and stdout
#to /dev/null
volttron -vv -l volttron.log > /dev/null 2>&1&
```

Installing and Running Agents

VOLTTRON platform comes with several built in services and example agents out of the box. To install a agent use the script `install-agent.py`

```
python scripts/install-agent.py -s <top most folder of the agent> [-c <config file.>]
↳ Might be optional for some agents>
```

For example, we can use the command to install and start the Listener Agent - a simple agent that periodically publishes heartbeat message and listens to everything on the message bus. Install and start the Listener agent using the following command:

```
python scripts/install-agent.py -s examples/ListenerAgent --start
```

Check `volttron.log` to ensure that the listener agent is publishing heartbeat messages.

```
tail volttron.log
```

```
2016-10-17 18:17:52,245 (listeneragent-3.2 11367) listener.agent INFO: Peer: 'pubsub',
↳ Sender: 'listeneragent-3.2_1', Bus: u'', Topic: 'heartbeat/listeneragent-3.2_1',
↳ Headers: {'Date': '2016-10-18T01:17:52.239724+00:00', 'max_compatible_version': u'',
↳ 'min_compatible_version': '3.0'}, Message: {'status': 'GOOD', 'last_updated':
↳ '2016-10-18T01:17:47.232972+00:00', 'context': 'hello'}
```

You can also use the `volttron-ctl` (or `vctl`) command to start, stop or check the status of an agent

```
(volttron)volttron@volttron1:~/git/rmq_volttron$ vctl status
AGENT                IDENTITY                TAG                STATUS                HEALTH
6 listeneragent-3.2    listeneragent-3.2_1      running [13125]    GOOD
f master_driveragent-3.2 platform.driver          master_driver
```

```
vctl stop <agent id>
```

Note: The default working directory is `~/volttron`. The default directory for creation of agent packages is `~/volttron/packaged`

1.3.5 Next Steps

There are several walk-throughs and detailed explanations of platform features to explore additional aspects of the platform:

- Agent Framework
- Driver Framework
- Demonstration of the management UI
- RabbitMQ setup with Federation and Shovel plugins

1.4 Definition of Terms

This page lays out a common terminology for discussing the components and underlying technologies used by the platform. The first section discusses capabilities and industry standards that VOLTTRON conforms to while the latter is specific to the VOLTTRON domain.

1.4.1 Industry Terms

Agent Software which acts on behalf of a user to perform a set of tasks.

BACNet Building Automation and Control network that leverages ASHRAE, ANSI, and IOS 16484-5 standard protocols

DNP3 (Distributed Network Protocol 3) Communications protocol used to coordinate processes in distributed automation systems

JSON (JavaScript Object Notation) JavaScript object notation is a text-based, human-readable, open data interchange format, similar to XML but less verbose

IEEE 2030.5 Utilities communication standard for managing energy demand and load (previously Smart Energy Profile version 2, SEP2)

JSON-RPC (JSON-Remote Procedure Call) JSON-encoded Remote Procedure Call

Modbus Communications protocol for talking with industrial electronic devices

PLC (Programmable Logic Controller) Computer used in industrial applications to manage processes of groups of industrial devices

Python Virtual Environment The *Python-VE* library allows users to create a virtualized copy of the local environment. A virtual environment allows the user to isolate the dependencies for a project which helps prevent conflicts between dependencies across projects.

Publish/Subscribe A message delivery pattern where senders (publishers) and receivers (subscribers) do not communicate directly nor necessarily have knowledge of each other, but instead exchange messages through an intermediary based on a mutual class or topic.

Note:

The Publish/Subscribe paradigm is often notated as `pub/sub` in VOLTTRON documentation.

RabbitMQ Open-Source message brokering system used by VOLTTRON for sending messages between services on the platform.

Remote Procedure Call Protocol used to request services of another computer located elsewhere on the network or on a different network.

SSH (Secure Shell) Secure Shell is a network protocol providing encryption and authentication of data using public-key cryptography.

SSL (Secure Sockets Layer) Secure Sockets Layer is a technology for encryption and authentication of network traffic based on a chain of trust.

TLS (Transport Layer Security) Transport Layer Security is the successor to SSL.

ZeroMQ (also ØMQ) A library used for inter-process and inter-computer communication.

1.4.2 VOLTTRON Terms

Activated Environment An activated environment is the environment a VOLTTRON instance is run in. The bootstrap process creates the environment from the shell.

AIP (Agent Instantiation and Packaging) This is the module responsible for creating agent wheels, the agent execution environment and running agents. Found in the VOLTTRON repository in the *volttron/platform* directory.

Agent Framework Framework which provides connectivity to the VOLTTRON platform and subsystems for software agents.

Bootstrap the Environment The process by which an operating environment (activated environment) is produced. From the VOLTTRON_ROOT directory, executing *python bootstrap.py* will start the bootstrap process.

Config Store Agent data store used by the platform for storing configuration files and automating the management of agent configuration

Driver Module that implements communication paradigms of a device to provide an interface to devices for the VOLTTRON platform.

Driver Framework Framework for implementing communication between the VOLTTRON platform and devices on the network (or a remote network)

Historian Historians in VOLTTRON are special purpose agents for automatically collecting data from the platform message bus and storing in a persistent data store.

VOLTTRON Central VOLTTRON Central (VC) is a special purpose agent for managing multiple platforms in a distributed VOLTTRON deployment

VOLTTRON_HOME The location for a specific VOLTTRON_INSTANCE to store its specific information. There can be many VOLTTRON_HOMEs on a single computing resource such as a VM, machine, etc. Each *VOLTTRON_HOME* will correspond to a single instance of VOLTTRON.

VOLTTRON_INSTANCE A single volttron process executing instructions on a computing resource. For each VOLTTRON_INSTANCE, there WILL BE only one VOLTTRON_HOME associated with it. For a VOLTTRON_INSTANCE to participate outside its computing resource, it must be bound to an external IP address.

VOLTTRON_ROOT The cloned directory from Github. When executing the command:

```
git clone https://github.com/VOLTTRON/volttron.git

the top level volttron folder is the VOLTTRON_ROOT.
```

VIP VOLTTRON Interconnect Protocol is a secure routing protocol that facilitates communications between agents, controllers, services, and the supervisory VOLTTRON_INSTANCE.

Web Framework Framework used by VOLTTTRON agents to implement web services with HTTP and HTTPS

1.5 License

Copyright 2019, Battelle Memorial Institute.

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

The patent license grant shall only be applicable to the following patent and patent application (Battelle IPID 17008-E), as assigned to the Battelle Memorial Institute, as used in conjunction with this Work: • US Patent No. 9,094,385, issued 7/28/15 • USPTO Patent App. No. 14/746,577, filed 6/22/15, published as US 2016-0006569.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.5.1 Terms

This material was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the United States Department of Energy, nor Battelle, nor any of their employees, nor any jurisdiction or organization that has cooperated in the development of these materials, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness or any information, apparatus, product, software, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY operated by BATTELLE for the UNITED STATES DEPARTMENT OF ENERGY under Contract DE-AC05-76RL01830

1.6 Join the Community

The VOLTTTRON project is transitioning into the Eclipse Foundation as Eclipse VOLTTTRON. Current resources will still be used during this time. Please watch this space!

The Eclipse VOLTTTRON team aims to work with users and contributors to continuously improve the platform with features requested by the community as well as architectural features that improve robustness, security, and scalability. Contributing back to the project, which is encouraged but not required, enhances its capabilities for the whole community. To learn more, check out *Contributing* and *Documentation*.

1.6.1 Slack Channel

voltttron-community.slack.com is where the VOLTTTRON™ community at large can ask questions and meet with others using VOLTTTRON™. To be added to Slack please email the VOLTTTRON team at voltttron@pnnl.gov.

1.6.2 Mailing List

Join the mailing list by emailing volttron@pnnl.gov.

1.6.3 Stack Overflow

The VOLTTRON community supports questions being asked and answered through Stack Overflow. The questions tagged with the *volttron* tag can be found at <http://stackoverflow.com/questions/tagged/volttron>.

1.6.4 Office Hours

PNNL hosts office hours every other week on Fridays at 11 AM (PST). These meetings are designed to be very informal where VOLTTRON developers can answer specific questions about the inner workings of VOLTTRON. These meetings are also available for topical discussions of different aspects of the VOLTTRON platform. Currently the office hours are available through a Zoom meeting. To be invited to the link meeting, contact the volttron team via email: mailto:volttron@pnnl.gov

Meetings are recorded and can be reviewed [here](#).

1.7 Setting Up a Development Environment

An example development environment used by the VOLTTRON team would consist of a Linux VM running on the host development machine on which an IDE would be running. The guides can be used to set up a development environment.

1.7.1 Forking the Repository

The first step to editing the repository is to fork it into your own user space. Creating a fork makes a copy of the repository in your GitHub for you to make any changes you may require for your use-case. This allows you to make changes without impacting the core VOLTTRON repository.

Forking is done by pointing your favorite web browser to <http://github.com/VOLTTRON/volttron> and then clicking “Fork” on the upper right of the screen. (Note: You must have a GitHub account to fork the repository. If you don’t have one, we encourage you to [sign up](#).)

Note: After making changes to your repository, you may wish to contribute your changes back to the Core VOLTTRON repository. Instructions for contributing code may be found [here](#).

Cloning ‘YOUR’ VOLTTRON forked repository

The next step in the process is to copy your forked repository onto your computer to work on. This will create an identical copy of the GitHub repository on your local machine. To do this you need to know the address of your repository. The URL to your repository address will be `https://github.com/<YOUR USERNAME>/volttron.git`. From a terminal execute the following commands:

```
# Here, we are assuming you are doing develop work in a folder called `git`. If you'd
↳ rather use something else, that's OK.
mkdir -p ~/git
cd ~/git
git clone -b develop https://github.com/<YOUR USERNAME>/volttron.git
cd volttron
```

Note: VOLTTRON uses develop as its main development branch rather than the standard *main* branch (the default).

Adding and Committing files

Now that you have your repository cloned, it's time to start doing some modifications. Using a simple text editor you can create or modify any file in the volttron directory. After making a modification or creating a file it is time to move it to the stage for review before committing to the local repository. For this example let's assume we have made a change to *README.md* in the root of the volttron directory and added a new file called *foo.py*. To get those files in the staging area (preparing for committing to the local repository) we would execute the following commands:

```
git add foo.py
git add README.md

# Alternatively in one command
git add foo.py README.md
```

After adding the files to the stage you can review the staged files by executing:

```
git status
```

Finally, in order to commit to the local repository we need to think of what change we actually did and be able to document it. We do that with a commit message (the *-m* parameter) such as the following.

```
git commit -m "Added new foo.py and updated copyright of README.md"
```

Pushing to the remote repository

The next step is to share our changes with the world through GitHub. We can do this by pushing the commits from your local repository out to your GitHub repository. This is done by the following command:

```
git push
```

1.7.2 Installing a Linux Virtual Machine

VOLTTRON requires a Linux system to run. For Windows users this will require a virtual machine (VM).

This section describes the steps necessary to install VOLTTRON using Oracle VirtualBox software. Virtual Box is free and can be downloaded from <https://www.virtualbox.org/wiki/Downloads>.



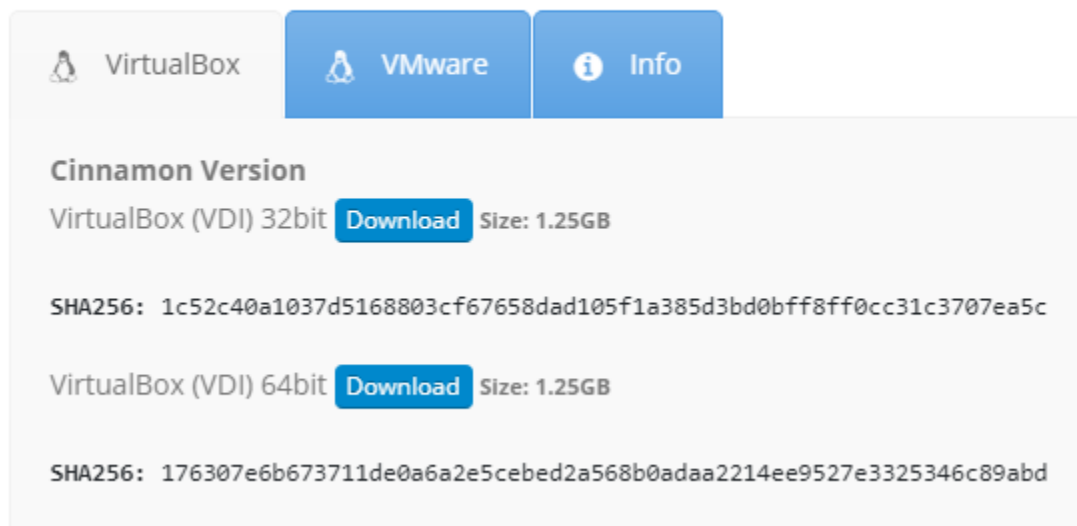
The screenshot shows the VirtualBox website. On the left is a sidebar with links: About, Screenshots, Downloads, Documentation (with sub-links for End-user docs and Technical docs), Contribute, and Community. The main content area has a large 'VirtualBox' header, a search bar, and links for Login and Preferences. Below the header is a 'Download VirtualBox' section with a paragraph stating that links to binaries and source code will be found there. This is followed by a 'VirtualBox binaries' section with a paragraph about agreeing to terms and conditions, and a link to 'VirtualBox 5.1 builds'. Then, a 'VirtualBox 5.2.12 platform packages' section lists links for Windows hosts, OS X hosts, Linux distributions, and Solaris hosts.

After installing VirtualBox download a virtual box appliance from <https://www.osboxes.org/linux-mint/> extract the VDI from the downloaded archive, **or** download a system installation disk. VOLTTRON version 7.0.x has been tested using Ubuntu 18.04, 18.10; Linux Mint 19; VOLTTRON version 6.0.x has been tested with Ubuntu 16.04, 18.04. However, any modern apt based Linux distribution should work out of the box. Linux Mint 19.3 with the Xfce desktop is used as an example, however platform setup in Ubuntu should be identical.

Note: A 32-bit version of Linux should be used when running VOLTTRON on a system with limited hardware (less than 2 GB of RAM).

Adding a VDI Image to VirtualBox Environment

Linux Mint 18.3 Sylvia





The screenshot shows the download page for Linux Mint 18.3 Sylvia on the VirtualBox website. It features tabs for VirtualBox, VMware, and Info. Under the 'Cinnamon Version' section, there are two download options: 'VirtualBox (VDI) 32bit' and 'VirtualBox (VDI) 64bit', both with a 'Download' button and a size of 1.25GB. Below each download option is its corresponding SHA256 hash.


Version	Download Link	Size	SHA256 Hash
VirtualBox (VDI) 32bit	Download	1.25GB	1c52c40a1037d5168803cf67658dad105f1a385d3bd0bff8ff0cc31c3707ea5c
VirtualBox (VDI) 64bit	Download	1.25GB	176307e6b673711de0a6a2e5cebed2a568b0adaa2214ee9527e3325346c89abd

The below info holds the VM's preset username and password.

Linux Mint 18.3 Sylvia

 VirtualBox

 VMware

 Info

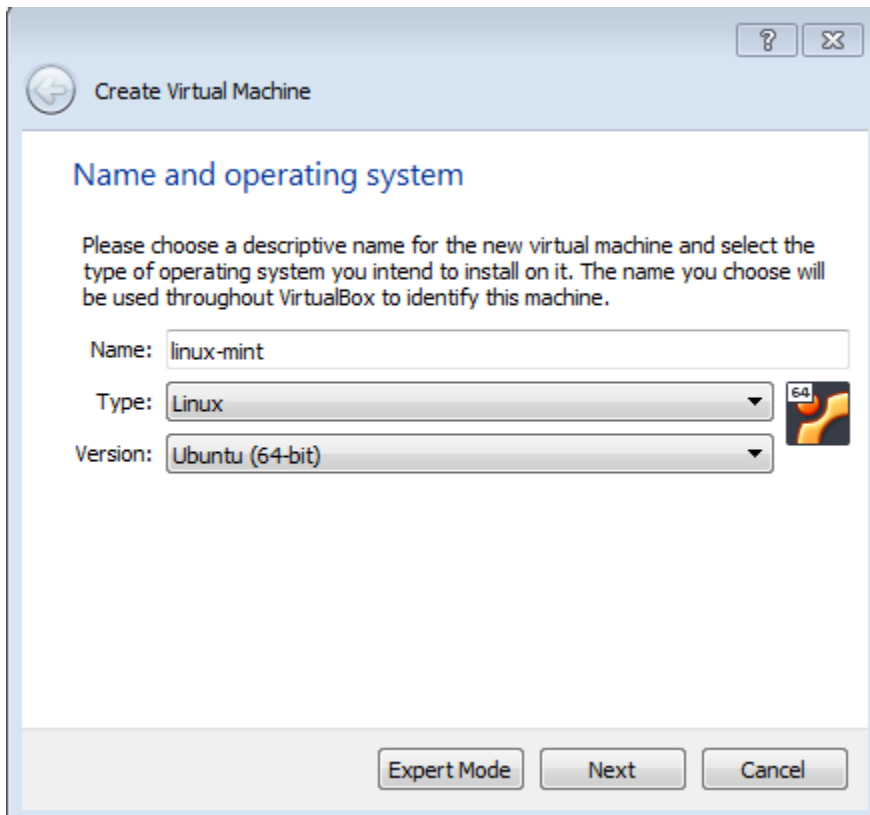
Username: osboxes

Password: osboxes.org

VB Guest Additions & VMware Tools: Not Installed

VMware Compatibility: Version 10+

Create a new VirtualBox Image.



Create Virtual Machine

Name and operating system

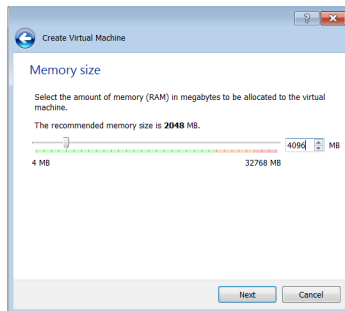
Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name:

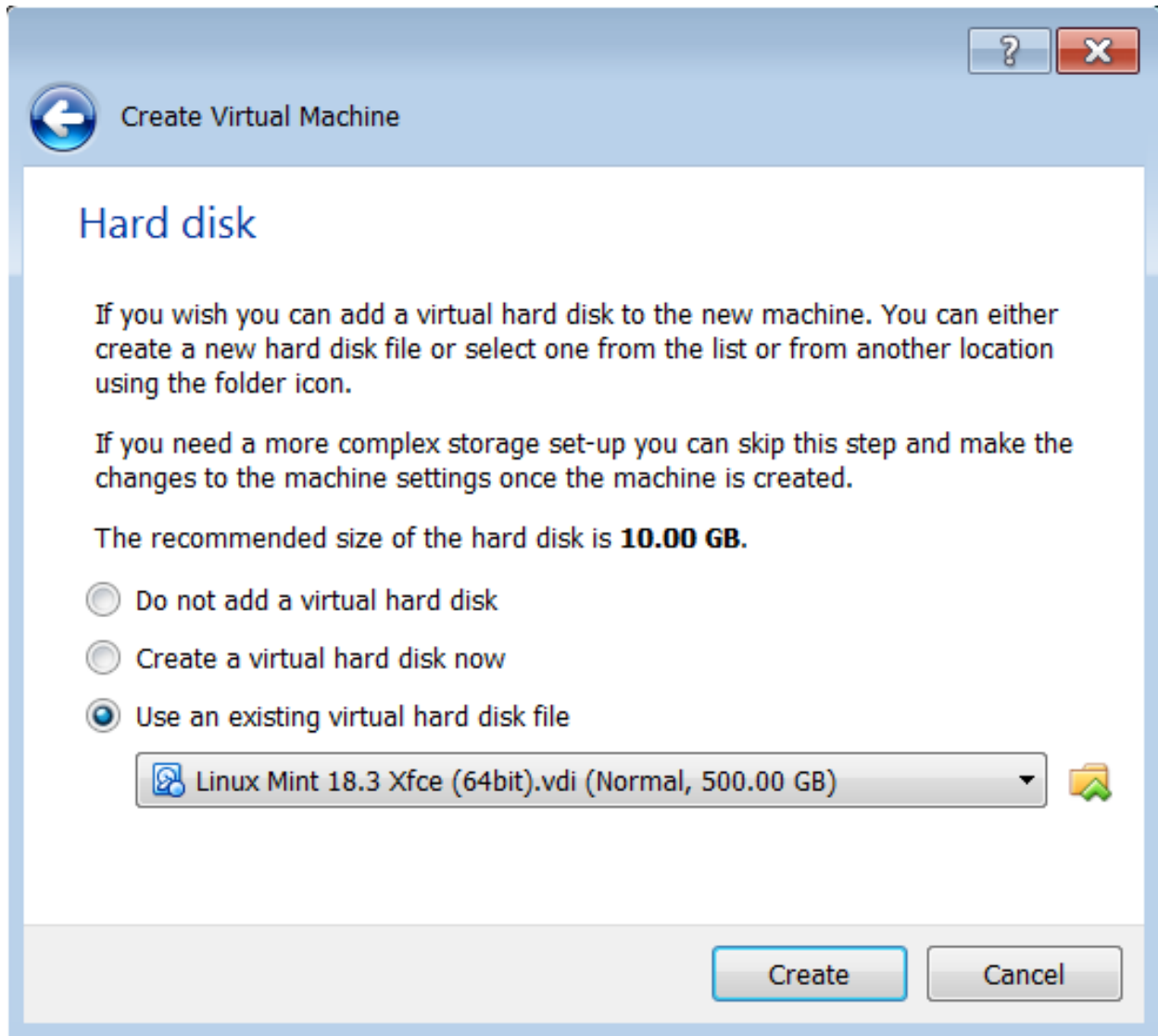
Type:

Version:

Select the amount of RAM for the VM. The recommended minimum is shown in the image below:

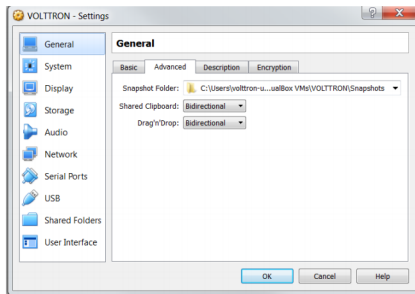


Specify the hard drive image using the extracted VDI file.



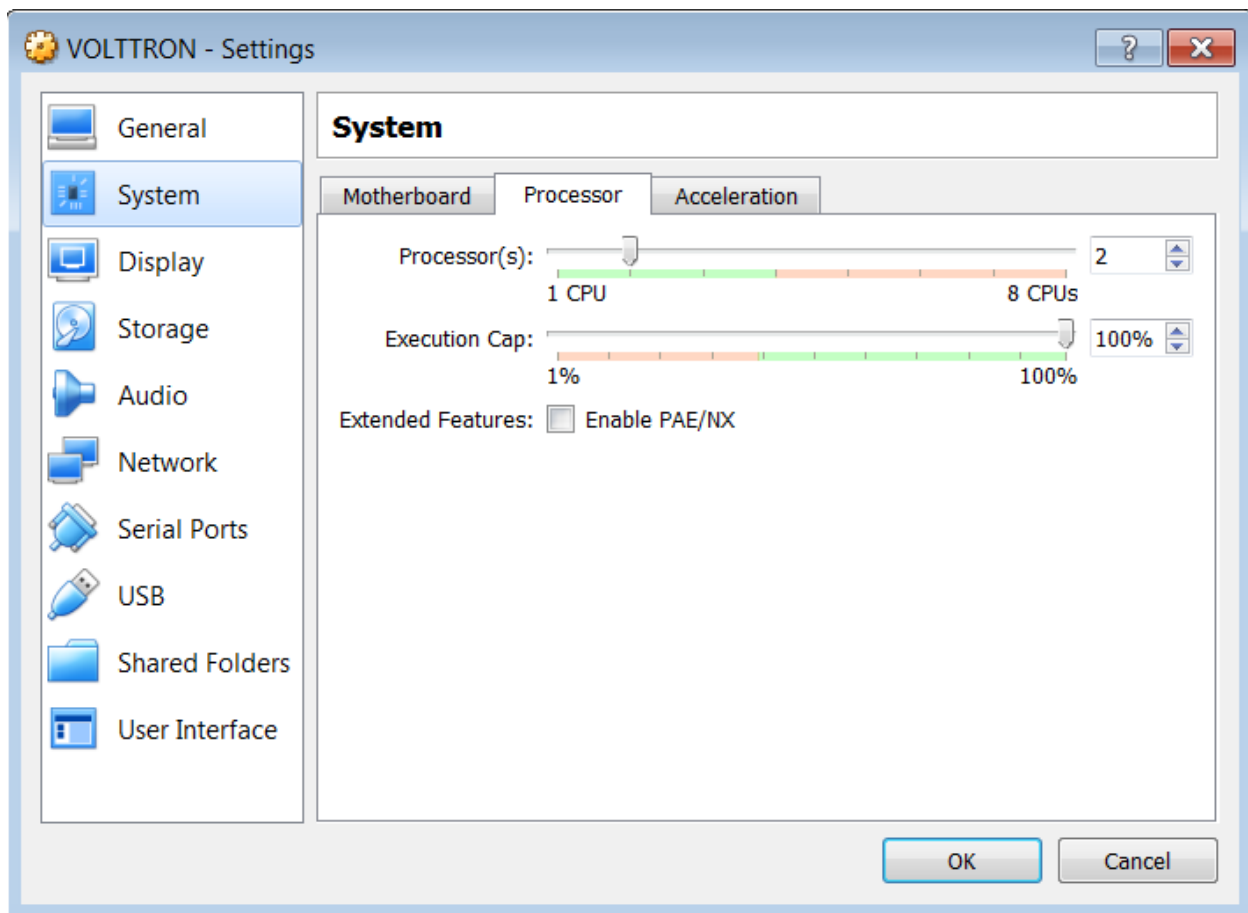
With the newly created VM selected, choose Machine from the VirtualBox menu in the top left corner of the VirtualBox window; from the drop down menu, choose Settings.

To enable bidirectional copy and paste, select the General tab in the VirtualBox Settings. Enable Shared Clipboard and Drag'n'Drop as Bidirectional.



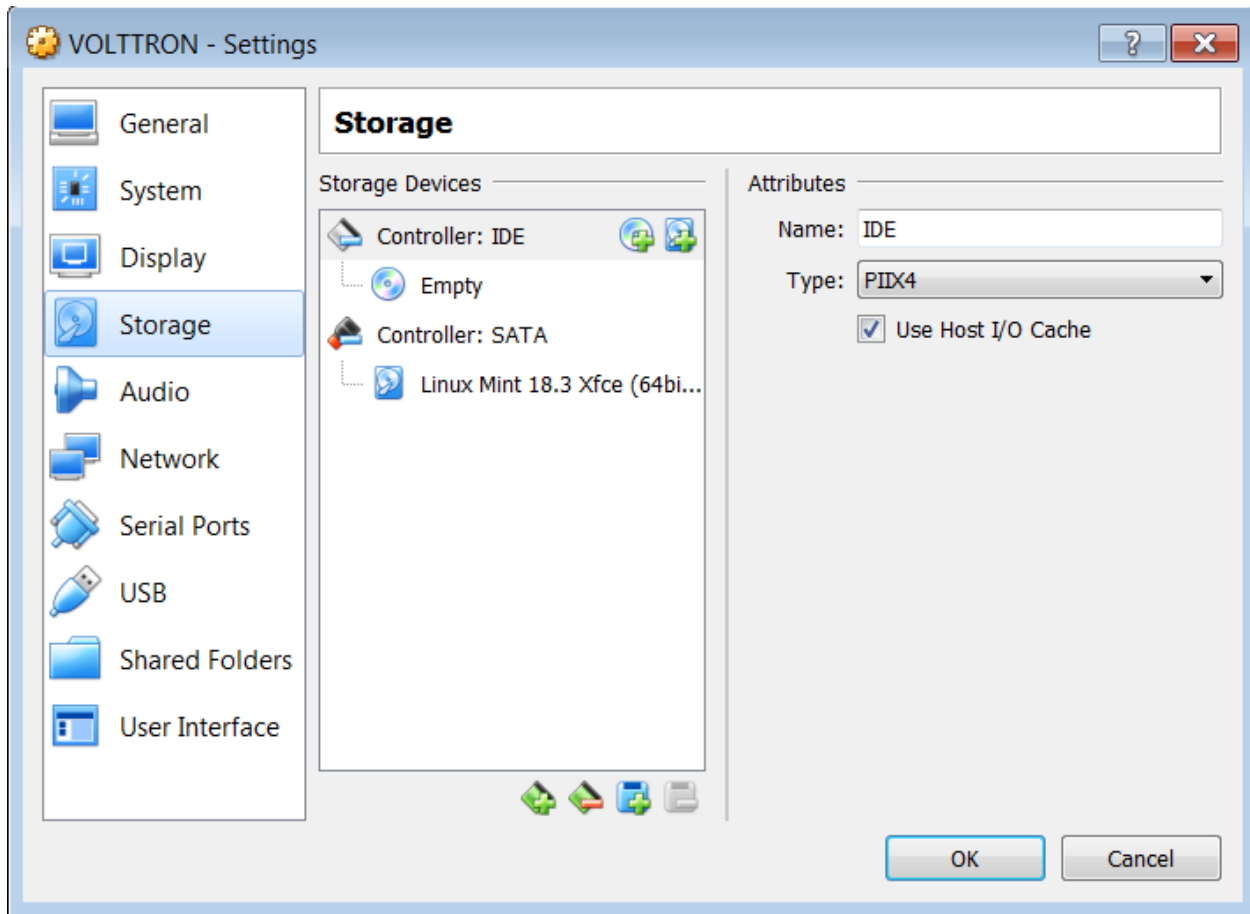
Note: Currently, this feature only works under certain circumstances (e.g. copying / pasting text).

Go to System Settings. In the processor tab, set the number of processors to two.



Go to Storage Settings. Confirm that the Linux Mint VDI is attached to Controller: SATA.

Danger: Do **NOT** mount the Linux Mint iso for Controller: IDE. **Will result in errors.**



Start the machine by saving these changes and clicking the “Start” arrow located on the upper left hand corner of the main VirtualBox window.

1.7.3 Pycharm Development Environment

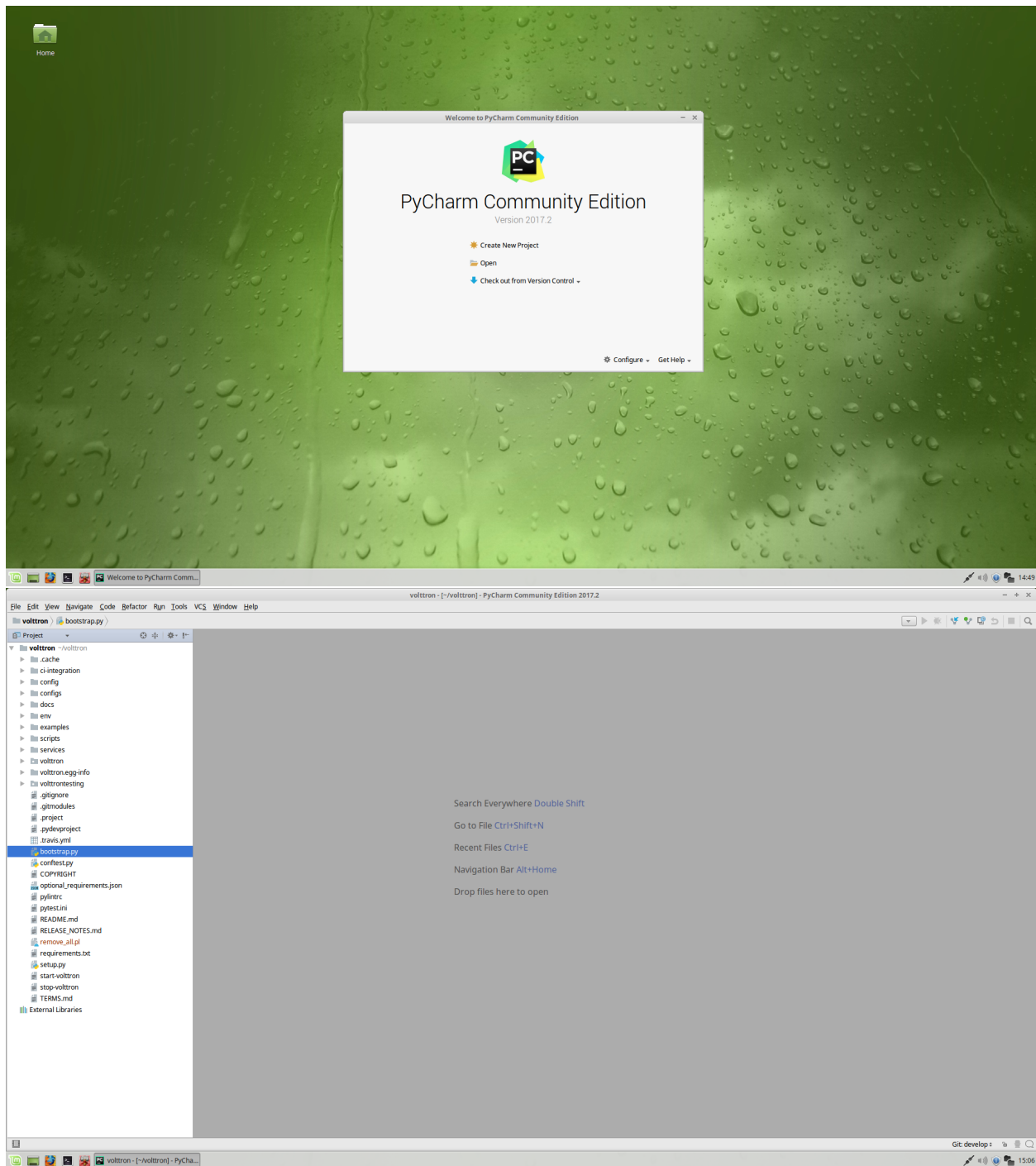
Pycharm is an IDE dedicated to developing python projects. It provides coding assistance and easy access to debugging tools as well as integration with py.test. It is a popular tool for working with VOLTTRON. JetBrains provides a free community version that can be downloaded from <https://www.jetbrains.com/pycharm/>

Open Pycharm and Load VOLTTRON

When launching Pycharm for the first time we have to tell it where to find the VOLTTRON source code. If you have already cloned the repo then point Pycharm to the cloned project. Pycharm also has options to access remote repositories.

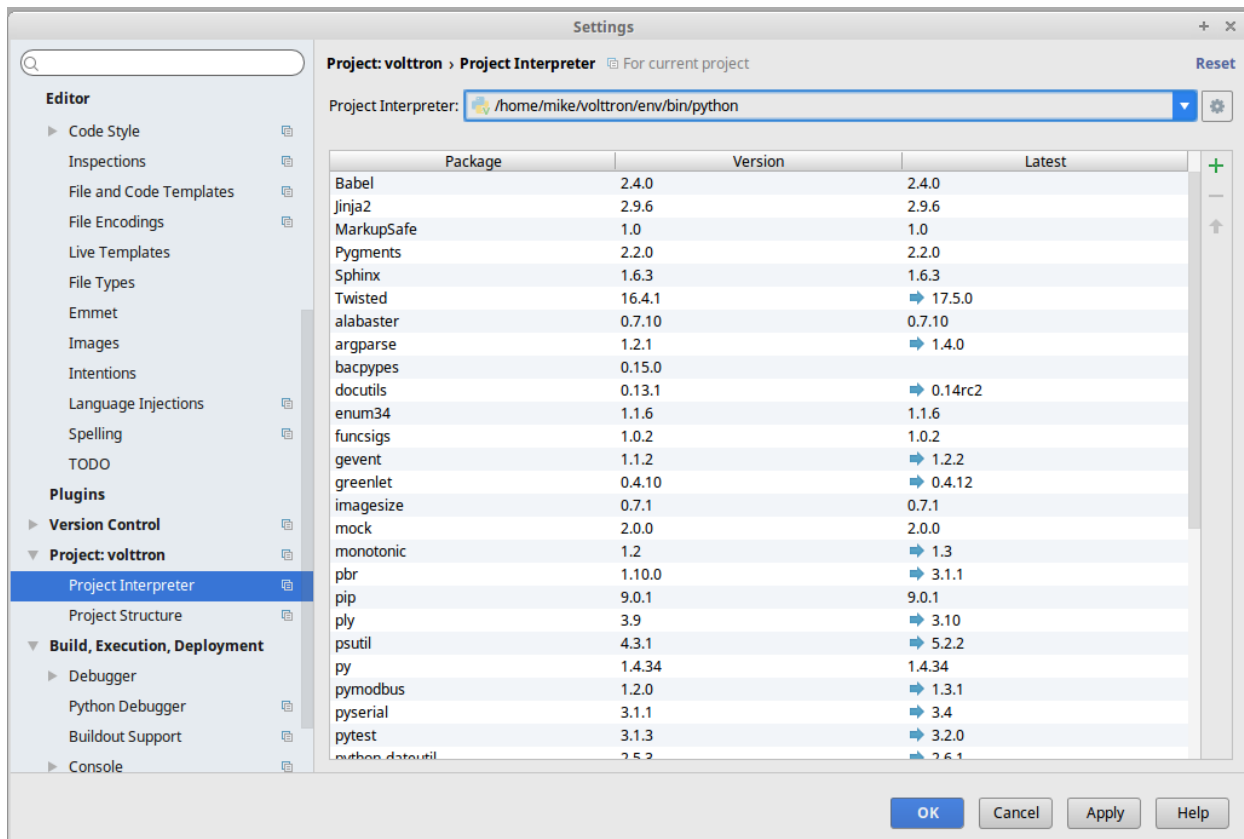
Subsequent instances of Pycharm will automatically load the VOLTTRON project.

Note: When getting started make sure to search for *gevent* in the settings and ensure that support for it is enabled.



Set the Project Interpreter

This step should be completed after running the bootstrap script in the VOLTTRON source directory. Pycharm needs to know which python environment it should use when running and debugging code. This also tells Pycharm where to find python dependencies. Settings menu can be found under the File option in Pycharm.



Running the VOLTTRON Process

If you are not interested in running the VOLTTRON process itself in Pycharm then this step can be skipped.

In **Run > Edit Configurations** create a configuration that has `<your source dir>/env/bin/volttron` in the script field, `-vv` in the script parameters field (to turn on verbose logging), and set the working directory to the top level source directory.

VOLTTRON can then be run from the Run menu.

The screenshot shows the 'Configuration' tab for a tool named 'volttron'. The 'Name' field is set to 'volttron'. There are checkboxes for 'Share' and 'Single instance only', both of which are unchecked. The 'Script' field is set to '/home/mike/volttron/env/bin/volttron'. The 'Script parameters' field is set to '-vv'. Under the 'Environment' section, the 'Environment variables' field is set to 'PYTHONUNBUFFERED=1'. The 'Python interpreter' is set to 'Project Default (Python 2.7.6 virtualenv at ~/volttron/env)'. The 'Interpreter options' field is empty. The 'Working directory' is set to '/home/mike/volttron'. There are four checkboxes at the bottom: 'Add content roots to PYTHONPATH' (checked), 'Add source roots to PYTHONPATH' (checked), 'Emulate terminal in output console' (unchecked), and 'Show command line afterwards' (unchecked). At the bottom, there is a section 'Before launch: Activate tool window' with a dropdown menu. Below this, there are icons for adding, removing, and moving configurations.

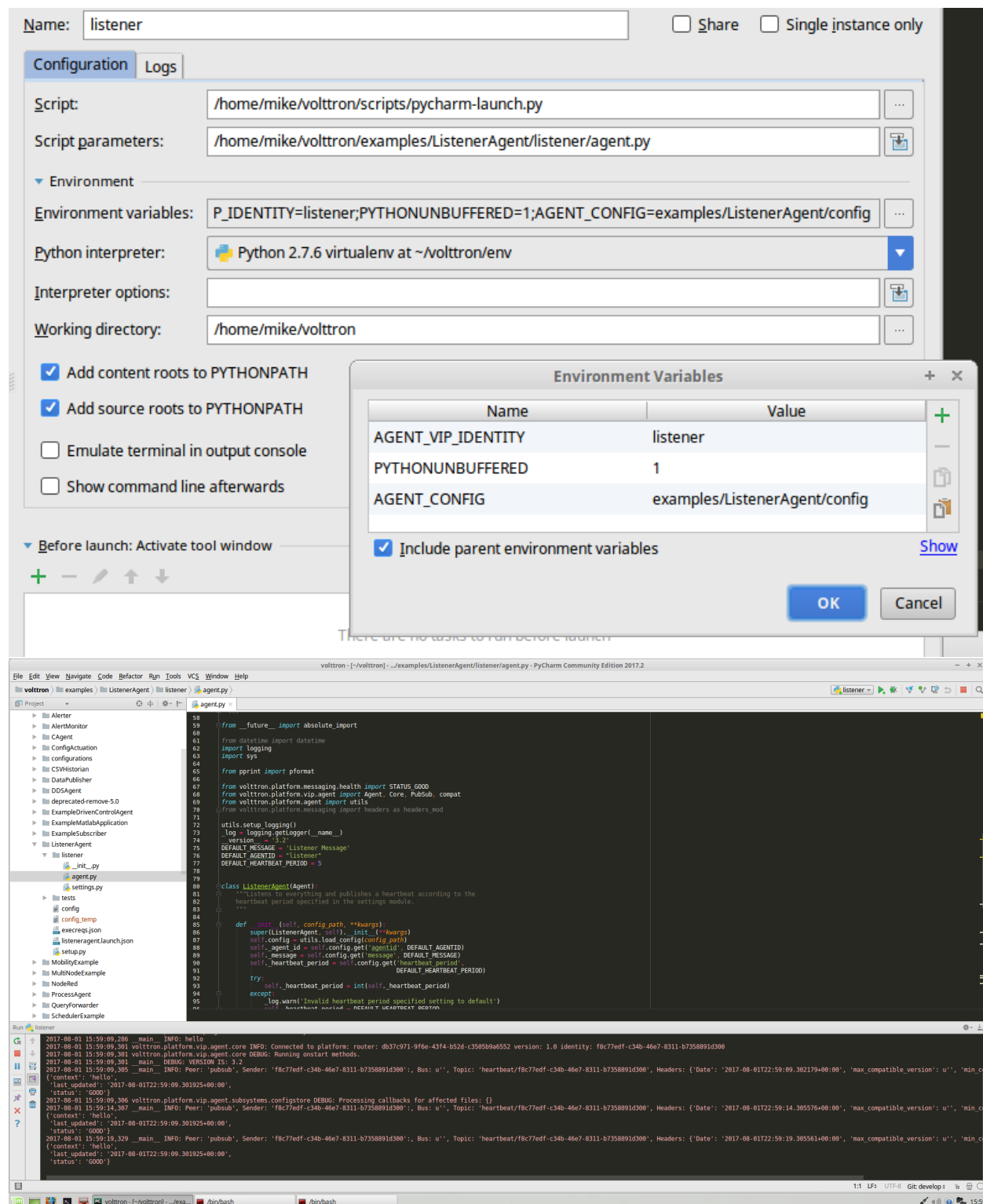
Running an Agent

Running an agent is configured similarly to running VOLTTRON proper. In **Run > Edit Configurations** add a configuration and give it the same name as your agent. The script should be the path to *scripts/pycharm-launch.py* and the script parameter must be the path to your agent's *agent.py* file.

In the Environment Variables field add the variable *AGENT_CONFIG* that has the path to the agent's configuration file as its value, as well as *AGENT_VIP_IDENTITY*, which must be unique on the platform.

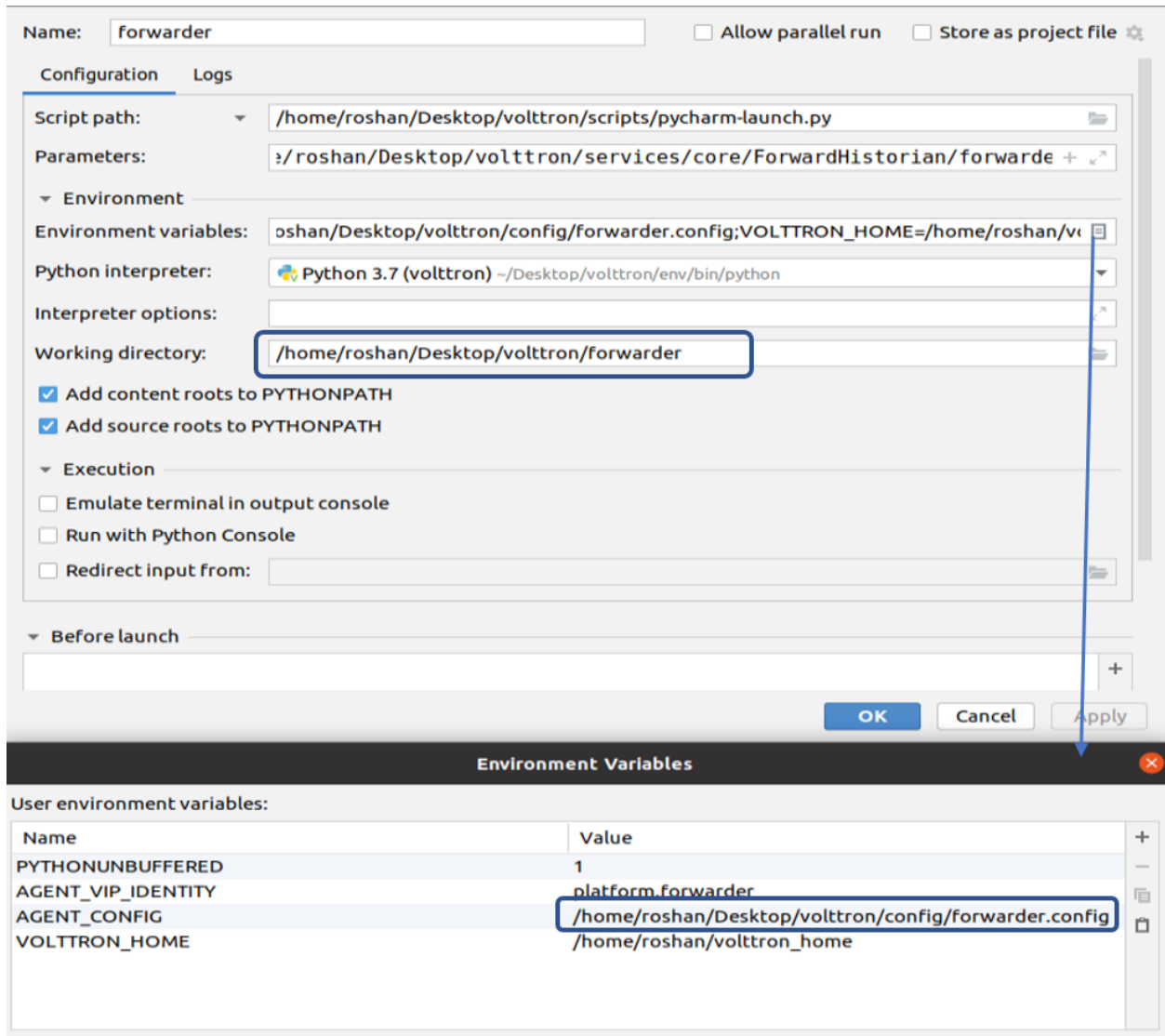
A good place to keep configuration files is in a directory called *config* in top level source directory; git will ignore changes to these files.

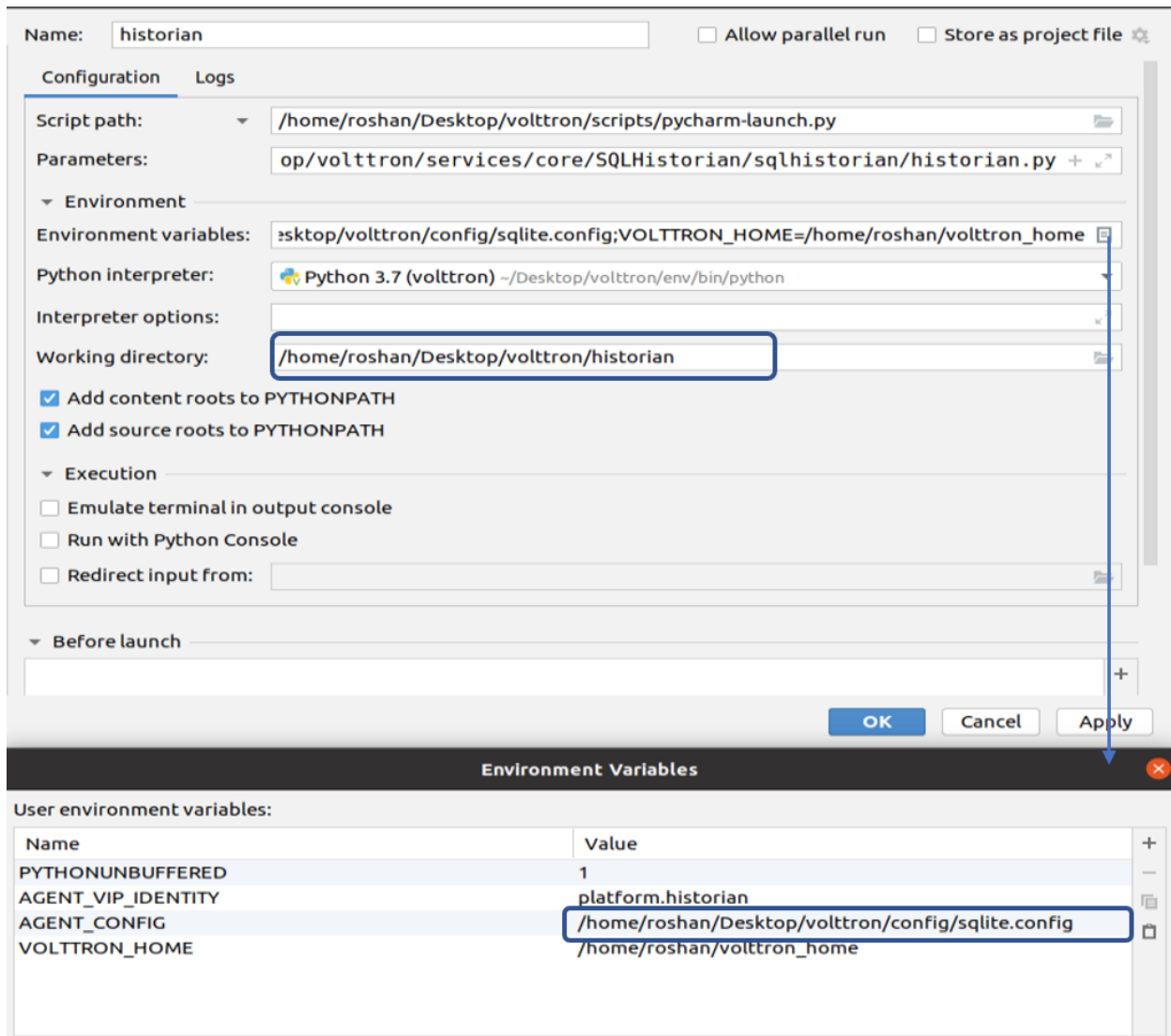
Note: There is an issue with imports in Pycharm when there is a secondary file (i.e. not *agent.py* but another module within the same package). When that happens right click on the directory in the file tree and select **Mark Directory As -> Source Root**



Note: There will be issues if two agents create a file with the same name in the same working directory. For instance: SQLHistorian agent and Forwarder agent both create a backup.sqlite directory on the same working directory. When that happens both the agents attempt to use the same backup db and eventually lock the db. To avoid this situation, create different working directories for each agent and add the absolute path for the config file. The best way to go

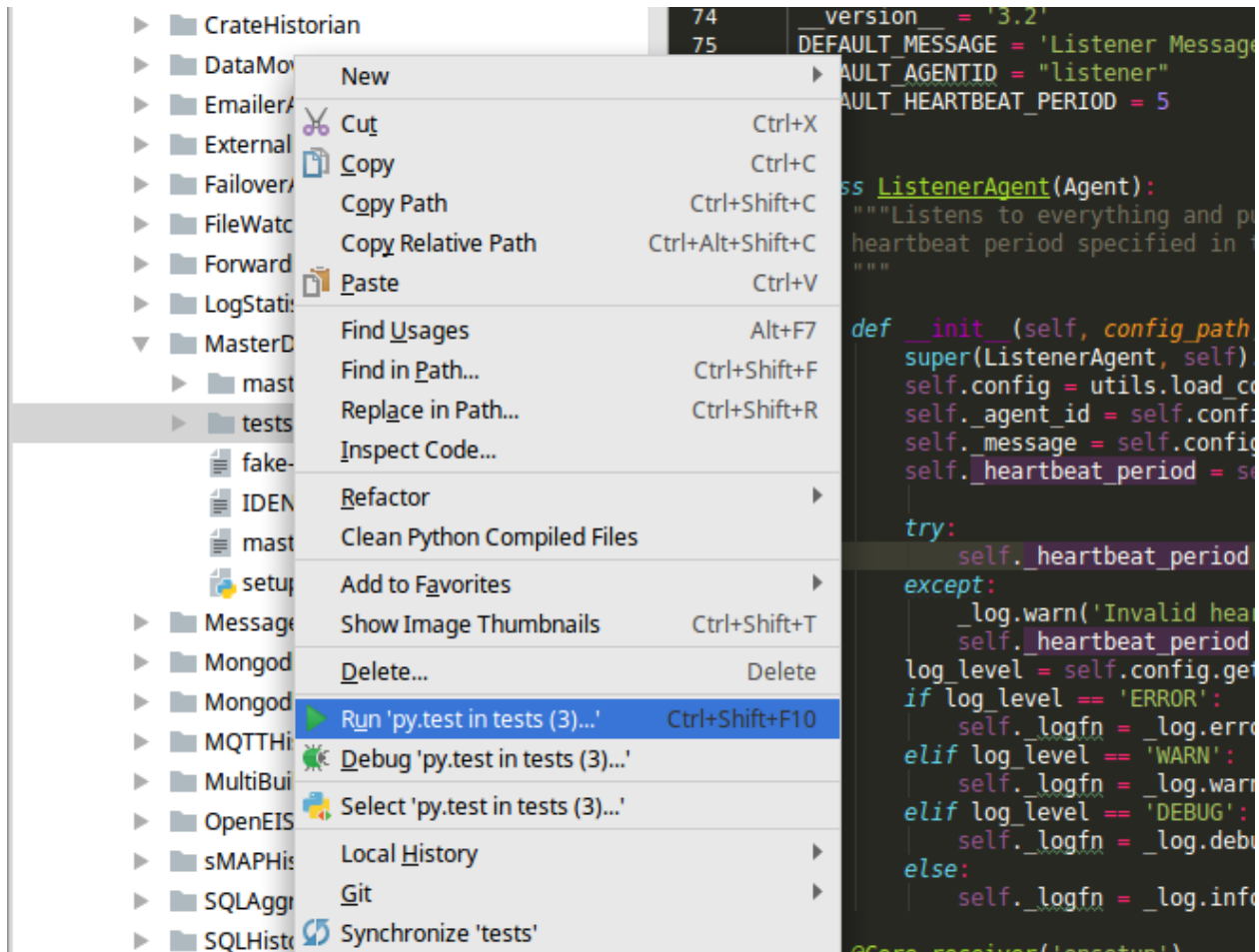
about this is to create a new folder and assign working directory to that folder as shown below.





Testing an Agent

Agent tests written in `pytest` can be run simply by right-clicking the tests directory and selecting *Run 'pytest in tests'*, so long as the root directory is set as the VOLTTRON source root.



1.8 Agent Development

The VOLTTTRON platform now has utilities to speed the creation and installation of new agents. To use these utilities the VOLTTTRON environment must be activated.

From the project directory, activate the VOLTTTRON environment with:

```
source env/bin/activate
```

1.8.1 Create Agent Code

Run the following command to start the Agent Creation Wizard:

```
vpkg init TestAgent tester
```

TestAgent is the directory that the agent code will be placed in. The directory must not exist when the command is run. *tester* is the name of the agent module created by wizard.

The Wizard will prompt for the following information:

```
Agent version number: [0.1]: 0.5
Agent author: []: VOLTTRON Team
Author's email address: []: volttron@pnnl.gov
Agent homepage: []: https://volttron.org/
Short description of the agent: []: Agent development tutorial.
```

Once the last question is answered the following will print to the console:

```
2018-08-02 12:20:56,604 () volttron.platform.packaging INFO: Creating TestAgent
2018-08-02 12:20:56,604 () volttron.platform.packaging INFO: Creating TestAgent/tester
2018-08-02 12:20:56,604 () volttron.platform.packaging INFO: Creating TestAgent/setup.
↪py
2018-08-02 12:20:56,604 () volttron.platform.packaging INFO: Creating TestAgent/config
2018-08-02 12:20:56,604 () volttron.platform.packaging INFO: Creating TestAgent/
↪tester/agent.py
2018-08-02 12:20:56,604 () volttron.platform.packaging INFO: Creating TestAgent/
↪tester/__init__.py
```

The TestAgent directory is created with the new Agent inside.

Agent Directory

At this point, the contents of the TestAgent directory should look like:

```
TestAgent/
├── setup.py
├── config
├── tester
│   ├── agent.py
│   └── __init__.py
```

Agent Skeleton

The *agent.py* file in the *tester* directory of the newly created agent module will contain skeleton code (below). Descriptions of the features of this code as well as additional development help are found in the rest of this document.

```
"""
Agent documentation goes here.
"""

__docformat__ = 'reStructuredText'

import logging
import sys
from volttron.platform.agent import utils
from volttron.platform.vip.agent import Agent, Core, RPC

_log = logging.getLogger(__name__)
utils.setup_logging()
__version__ = "0.1"

def tester(config_path, **kwargs):
    """Parses the Agent configuration and returns an instance of
```

(continues on next page)

(continued from previous page)

```

the agent created using that configuration.

:param config_path: Path to a configuration file.

:type config_path: str
:returns: Garbage
:rtype: Garbage
"""
try:
    config = utils.load_config(config_path)
except StandardError:
    config = {}

if not config:
    _log.info("Using Agent defaults for starting configuration.")

setting1 = int(config.get('setting1', 1))
setting2 = config.get('setting2', "some/random/topic")

return Tester(setting1,
               setting2,
               **kwargs)

class Tester(Agent):
    """
    Document agent constructor here.
    """

    def __init__(self, setting1=1, setting2="some/random/topic",
               **kwargs):
        super(Garbage, self).__init__(**kwargs)
        _log.debug("vip_identity: " + self.core.identity)

        self.setting1 = setting1
        self.setting2 = setting2

        self.default_config = {"setting1": setting1,
                               "setting2": setting2}

        #Set a default configuration to ensure that self.configure is called
        ↪ immediately to setup
        #the agent.
        self.vip.config.set_default("config", self.default_config)
        #Hook self.configure up to changes to the configuration file "config".
        self.vip.config.subscribe(self.configure, actions=["NEW", "UPDATE"], pattern=
        ↪ "config")

    def configure(self, config_name, action, contents):
        """
        Called after the Agent has connected to the message bus. If a configuration
        ↪ exists at startup
        this will be called before onstart.

        Is called every time the configuration in the store changes.
        """

```

(continues on next page)

(continued from previous page)

```

config = self.default_config.copy()
config.update(contents)

_log.debug("Configuring Agent")

try:
    setting1 = int(config["setting1"])
    setting2 = str(config["setting2"])
except ValueError as e:
    _log.error("ERROR PROCESSING CONFIGURATION: {}".format(e))
    return

self.setting1 = setting1
self.setting2 = setting2

self._create_subscriptions(self.setting2)

def _create_subscriptions(self, topic):
    #Unsubscribe from everything.
    self.vip.pubsub.unsubscribe("pubsub", None, None)

    self.vip.pubsub.subscribe(peer='pubsub',
                              prefix=topic,
                              callback=self._handle_publish)

def _handle_publish(self, peer, sender, bus, topic, headers,
                    message):
    pass

@Core.receiver("onstart")
def onstart(self, sender, **kwargs):
    """
    This is method is called once the Agent has successfully connected to the_
    ↪platform.
    This is a good place to setup subscriptions if they are not dynamic or
    do any other startup activities that require a connection to the message bus.
    Called after any configurations methods that are called at startup.

    Usually not needed if using the configuration store.
    """
    #Example publish to pubsub
    #self.vip.pubsub.publish('pubsub', "some/random/topic", message="HI!")

    #Exmaple RPC call
    #self.vip.rpc.call("some_agent", "some_method", arg1, arg2)

@Core.receiver("onstop")
def onstop(self, sender, **kwargs):
    """
    This method is called when the Agent is about to shutdown, but before it_
    ↪disconnects from
    the message bus.
    """
    pass

@RPC.export
def rpc_method(self, arg1, arg2, kwarg1=None, kwarg2=None):

```

(continues on next page)

(continued from previous page)

```

    """
    RPC method

    May be called from another agent via self.core.rpc.call """
    return self.setting1 + arg1 - arg2

def main():
    """Main method called to start the agent."""
    utils.vip_main(garbage,
                  version=__version__)

if __name__ == '__main__':
    # Entry point for script
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        pass

```

The resulting code is well documented with comments and documentation strings. It gives examples of how to do common tasks in VOLTTRON Agents. The main agent code is found in *tester/agent.py*.

1.8.2 Building an Agent

The following section includes guidance on several important components for building agents in VOLTTRON.

Parse Packaged Configuration and Create Agent Instance

The code to parse a configuration file packaged and installed with the agent is found in the *tester* function:

```

def tester(config_path, **kwargs):
    """Parses the Agent configuration and returns an instance of
    the agent created using that configuration.

    :param config_path: Path to a configuration file.

    :type config_path: str
    :returns: Tester
    :rtype: Tester
    """
    try:
        config = utils.load_config(config_path)
    except StandardError:
        config = {}

    if not config:
        _log.info("Using Agent defaults for starting configuration.")

    setting1 = int(config.get('setting1', 1))
    setting2 = config.get('setting2', "some/random/topic")

    return Tester(setting1,
                  setting2,
                  **kwargs)

```

The configuration is parsed with the `utils.load_config` function and the results are stored in the `config` variable. An instance of the Agent is created from the parsed values and is returned.

Initialization and Configuration Store Support

The *configuration store* is a powerful feature. The agent template provides a simple example of setting up default configuration store values and setting up a configuration handler.

```
class Tester(Agent):
    """
    Document agent constructor here.
    """

    def __init__(self, setting1=1, setting2="some/random/topic",
                 **kwargs):
        super(Tester, self).__init__(**kwargs)
        _log.debug("vip_identity: " + self.core.identity)

        self.setting1 = setting1
        self.setting2 = setting2

        self.default_config = {"setting1": setting1,
                               "setting2": setting2}

        #Set a default configuration to ensure that self.configure is called
        ↪ immediately to setup
        #the agent.
        self.vip.config.set_default("config", self.default_config)
        #Hook self.configure up to changes to the configuration file "config".
        self.vip.config.subscribe(self.configure, actions=["NEW", "UPDATE"], pattern=
        ↪ "config")

    def configure(self, config_name, action, contents):
        """
        Called after the Agent has connected to the message bus. If a configuration
        ↪ exists at startup
        this will be called before onstart.

        Is called every time the configuration in the store changes.
        """
        config = self.default_config.copy()
        config.update(contents)

        _log.debug("Configuring Agent")

        try:
            setting1 = int(config["setting1"])
            setting2 = str(config["setting2"])
        except ValueError as e:
            _log.error("ERROR PROCESSING CONFIGURATION: {}".format(e))
            return

        self.setting1 = setting1
        self.setting2 = setting2

        self._create_subscriptions(self.setting2)
```

Note: Support for the configuration store is instantiated by subscribing to configuration changes with `self.vip.config.subscribe`.

```
self.vip.config.subscribe(self.configure_main, actions=["NEW", "UPDATE"], pattern=
    ↪ "config")
```

Values in the default config can be built into the agent or come from the packaged configuration file. The subscribe method tells our agent which function to call whenever there is a new or updated config file. For more information on using the configuration store see [Agent Configuration Store](#).

`_create_subscriptions` (covered in a later section) will use the value in `self.setting2` to create a new subscription.

Agent Lifecycle Events

The agent lifecycle is controlled in the agents VIP *core*. The agent lifecycle manages *scheduling and periodic function calls*, the main agent loop, and trigger a number of signals for callbacks in the concrete agent code. These callbacks are listed and described in the skeleton code below:

Note: The lifecycle signals can trigger any method. To cause a method to be triggered by a lifecycle signal, use a decorator:

```
@Core.receiver("<lifecycle_method>")
def my_callback(self, sender, **kwargs):
    # do my lifecycle method callback
    pass
```

```
@Core.receiver("onsetup")
def onsetup(self, sender, **kwargs):
    """
    This method is called after the agent has successfully connected to the platform,
    ↪but before the scheduled
    methods loop has started. This method not often used, but is most commonly used
    ↪to define periodic
    functions or do some pre-configuration.
    """
    self.vip.core.periodic(60, send_request)

@Core.receiver("onstart")
def onstart(self, sender, **kwargs):
    """
    This method is called once the Agent has successfully connected to the platform.
    This is a good place to setup subscriptions if they are not dynamic or to
    do any other startup activities that require a connection to the message bus.
    Called after any configurations methods that are called at startup.

    Usually not needed if using the configuration store.
    """
    #Example publish to pubsub
    self.vip.pubsub.publish('pubsub', "some/random/topic", message="HI!")

    #Example RPC call
    self.vip.rpc.call("some_agent", "some_method", arg1, arg2)
```

(continues on next page)

(continued from previous page)

```

@Core.receiver("onstop")
def onstop(self, sender, **kwargs):
    """
    This method is called when the Agent is about to shutdown, but before it
    ↳disconnects from
    the message bus. Common use-cases for this method are to stop periodic
    ↳processing, closing connections and
    setting agent state prior to cleanup.
    """
    self.publishing = False
    self.cache.close()

@Core.receiver("onfinish")
def onfinish(self, sender, **kwargs):
    """
    This method is called after all scheduled threads have concluded. This method is
    ↳rarely used, but could be
    used to send shut down signals to other agents, etc.
    """
    self.vip.pubsub.publish('pubsub', 'some/topic', message=f'agent {self.core.
    ↳identity} shutdown')

```

Periodics and Scheduling

Periodic and Scheduled callback functions are callbacks made to functions in agent code from the thread scheduling in the agent core.

Scheduled Callbacks

Scheduled callback functions are often used similarly to cron jobs to perform tasks at specific times, or to schedule tasks ad-hoc as agent state is updated. There are 2 ways to schedule callbacks: using a decorator, or calling the core's scheduling function. Example usage follows.

```

# using the agent's core to schedule a task
self.core.schedule(periodic(5), self.sayhi)

def sayhi(self):
    print("Hello-World!")

```

```

# using the decorator to schedule a task
@Core.schedule(cron('0 1 * * *'))
def cron_function(self):
    print("this is a cron-scheduled function")

```

Note: Scheduled Callbacks can use CRON scheduling, a datetime object, a number of seconds (from current time), or a *periodic* which will make the schedule function as a periodic.

```

# inside some agent method
self.core.schedule(t, function)
self.core.schedule(periodic(t), periodic_function)
self.core.schedule(cron('0 1 * * *'), cron_function)

```


Periodic Callbacks

Periodic call back functions are functions which are repeatedly called at a regular interval until the periodic is cancelled in the agent code or the agent stops running. Like scheduled callbacks, periodics can be specified using either decorators or using core function calls.

```
self.core.periodic(10, self.saybye)

def saybye(self):
    print('Good-bye Cruel World!')
```

```
@Core.periodic(60)
def poll_api(self):
    return requests.get("https://lmgtyfy.com").json()
```

Note: Periodic intervals are specified in seconds.

Publishing Data to the Message Bus

The agent's VIP connection can be used to publish data to the message bus. The message published and topic to publish to are determined by the agent implementation. Classes of agents already *specified by VOLTRON* may have well-defined intended topic usage, see those agent specifications for further detail.

```
def publish_oscillating_update(self):
    self.publish_value = 1 if self.publish_value == 0 else 0
    self.vip.pubsub.publish('pubsub', 'some/topic/', message=f'{"oscillating_value":'
    ↪ f'{self.publish_value}"}')
```

Setting up a Subscription

The Agent creates a subscription to a topic on the message bus using the value of *self.setting2* in the method *_create_subscription*. The messages for this subscription are handled with the *_handle_publish* method:

```
def _create_subscriptions(self, topic):
    #Unsubscribe from everything.
    self.vip.pubsub.unsubscribe("pubsub", None, None)

    self.vip.pubsub.subscribe(peer='pubsub',
                              prefix=topic,
                              callback=self._handle_publish)

def _handle_publish(self, peer, sender, bus, topic, headers,
                    message):
    #By default no action is taken.
    pass
```

Alternatively, a decorator can be used to specify the function as a callback:

```
@PubSub.subscribe('pubsub', "topic_prefix")
def _handle_publish(self, peer, sender, bus, topic, headers,
                    message):
    #By default no action is taken.
    pass
```

To unsubscribe from a topic, the `self.vip.pubsub.unsubscribe` can be used:

```
self.vip.pubsub.unsubscribe(peer='pubsub',
                             prefix=topic,
                             callback=self._handle_publish)
```

Giving `None` as values for the prefix and callback argument will unsubscribe from everything on that bus. This is handy for subscriptions that must be updated base on a configuration setting.

Heartbeat

The heartbeat subsystem provides access to a periodic publish so that others can observe the agent's status. Other agents can subscribe to the *heartbeat* topic to see who is actively publishing to it. It is turned off by default.

Enabling the *heartbeat* publish:

Subscribing to the heartbeat topic:

Health

The health subsystem adds extra status information to the an agent's heartbeat. Setting the status will start the heartbeat if it wasn't already. Health is used to represent the internal state of the agent at runtime. *GOOD* health indicates that all is fine with the agent and it is operating normally. *BAD* health indicates some kind of problem, such as if an agent is unable to reach a remote web API.

Example of setting health:

Remote Procedure Calls

An agent may receive commands from other agents via a Remote Procedure Call (RPC). This is done with the `@RPC.export` decorator:

```
@RPC.export
def rpc_method(self, arg1, arg2, kwarg1=None, kwarg2=None):
    """
    RPC method

    May be called from another agent via self.core.rpc.call """
    return self.setting1 + arg1 - arg2
```

To send an RPC call to another agent running on the platform, the agent must invoke the `rpc.call` method of its VIP connection.

```
# in agent code
def send_remote_procedure_call(self):
    peer = "<agent identity>"
    peer_method = "<method in peer agent API>"
```

(continues on next page)

(continued from previous page)

```
args = ["list", "of", "peer", "method", "arguments", "..."]
self.vip.rpc.call(peer, peer_method, *args)
```

1.8.3 Packaging Configuration

The wizard will automatically create a *setup.py* file. This file sets up the name, version, required packages, method to execute, etc. for the agent based on your answers to the wizard. The packaging process will also use this information to name the resulting file.

```
from setuptools import setup, find_packages

MAIN_MODULE = 'agent'

# Find the agent package that contains the main module
packages = find_packages('.')
agent_package = 'tester'

# Find the version number from the main module
agent_module = agent_package + '.' + MAIN_MODULE
__temp = __import__(agent_module, globals(), locals(), ['__version__'], -1)
__version__ = _temp.__version__

# Setup
setup(
    name=agent_package + 'agent',
    version=__version__,
    author_email="voltttron@pnnl.gov",
    url="https://voltttron.org/",
    description="Agent development tutorial.",
    author="VOLTTTRON Team",
    install_requires=['voltttron'],
    packages=packages,
    entry_points={
        'setuptools.installation': [
            'eggsecutable = ' + agent_module + ':main',
        ]
    }
)
```

1.8.4 Launch Configuration

In TestAgent, the wizard will automatically create a JSON file called “config”. It contains configuration information for the agent. This file contains examples of every data type supported by the configuration system:

```
{
    # VOLTTTRON config files are JSON with support for python style comments.
    "setting1": 2, #Integers
    "setting2": "some/random/topic2", #Strings
    "setting3": true, #Booleans: remember that in JSON true and false are not_
    ↪capitalized.
    "setting4": false,
    "setting5": 5.1, #Floating point numbers.
    "setting6": [1,2,3,4], #Lists
```

(continues on next page)

(continued from previous page)

```
"setting7": {"setting7a": "a", "setting7b": "b"} #Objects
}
```

1.8.5 Packaging and Installation

To install the agent the platform must be running. Start the platform with the command:

```
./start-volttron
```

Note: If you are not in an activated environment, this script will start the platform running in the background in the correct environment. However the environment will not be activated for you; you must activate it yourself.

Now we must install it into the platform. Use the following command to install it and add a tag for easily referring to the agent. From the project directory, run the following command:

```
python scripts/install-agent.py -s TestAgent/ -c TestAgent/config -t testagent
```

To verify it has been installed, use the following command:

```
vctl list
```

This will result in output similar to the following:

	AGENT	IDENTITY	TAG	Status	Health	PRI
df	testeragent-0.5	testeragent-0.5_1	testagent			

- The first string is a unique portion of the full UUID for the agent
- AGENT is the “name” of the agent based on the contents of its class name and the version in its setup.py.
- IDENTITY is the agent’s identity in the platform. This is automatically assigned based on class name and instance number. This agent’s ID is `_1` because it is the first instance.
- TAG is the name we assigned in the command above
- Status indicates the running status of an agent - running agents are *running*, agents which are not running will have no listed status
- Health is an indication of the internal state of the agent. ‘Healthy’ agents will have GOOD health. If an agent enters an error state, it will continue to run, but its health will be BAD.
- PRI is the priority for agents which have been “enabled” using the `vctl enable` command.

When using lifecycle commands on agents, they can be referred to by the UUID (default) or AGENT (name) or TAG.

1.8.6 Running and Testing the Agent

Now that the first pass of the agent code is complete, we can see if the agent works. It is highly-suggested to build a set of automated tests for the agent code prior to writing the agent, and running those tests after the agent is code-complete. Another quick way to determine if the agent is going the right direction is to run the agent on the platform using the VOLTTRON command line interface.

From the Command Line

To test the agent, we will start the platform (if not already running), launch the agent, and check the log file. With the VOLTTTRON environment activated, start the platform by running (if needed):

```
./start-volttron
```

You can launch the agent in three ways, all of which you can find by using the `vctl list` command:

- By using the <uuid>:

```
vctl start <uuid>
```

- By name:

```
vctl start --name testeragent-0.1
```

- By tag:

```
vctl start --tag testagent
```

Check that it is running:

```
vctl status
```

- Start the ListenerAgent as in the *platform installation guide*.
- Check the log file for messages indicating the TestAgent is receiving the ListenerAgents messages:

```
TODO
```

Automated Test Cases and Documentation

Before contributing a new agent to the VOLTTTRON source code repository, please consider adding two other essential elements.

1. Integration and unit test cases
2. README file that includes details of pre-requisite software, agent setup details (such as setting up databases, permissions, etc.) and sample configuration

VOLTTTRON uses *pytest* as a framework for executing tests. All unit tests should be based on the *pytest* framework. For instructions on writing unit and integration tests with *pytest*, refer to the *Writing Agent Tests* documentation.

pytest is not installed with the distribution by default. To install *py.test* and its dependencies execute the following:

```
python bootstrap.py --testing
```

Note: There are other options for different agent requirements. To see all of the options use:

```
python bootstrap.py --help
```

in the Extra Package Options section.

To run a single test module, use the command

```
pytest <testmodule.py>
```

To run all of the tests in the volttron repository execute the following in the root directory using an activated command prompt:

```
./ci-integration/run-tests.sh
```

1.8.7 Scripts

In order to make repetitive tasks less repetitive the VOLTTRON team has create several scripts in order to help. These tasks are available in the *scripts* directory.

Note: In addition to the *scripts* directory, the VOLTTRON team has added the config directory to the .gitignore file. By convention this is where we store customized scripts and configuration that will not be made public. Please feel free to use this convention in your own processes.

The *scripts/core* directory is laid out in such a way that we can build scripts on top of a base core. For example the scripts in sub-folders such as the *historian-scripts* and *demo-comms* use the scripts that are present in the core directory.

The most widely used script is *scripts/install-agent.py*. The *install_agent.py* script will remove an agent if the tag is already present, create a new agent package, and install the agent to *VOLTTRON_HOME*. This script has three required arguments and has the following signature:

Note: Agent to Package must have a setup.py in the root of the directory. Additionally, the user must be in an activated Python Virtual Environment for VOLTTRON

```
cd $VOLTTRON_ROOT
source env/bin/activate
```

```
python scripts/install_agent.py -s <agent path> -c <agent config file> -i <agent VIP_
↪identity> --tag <Tag>
```

Note: The `--help` optional argument can be used with *scripts/install-agent.py* to view all available options for the script

The *install_agent.py* script will respect the *VOLTTRON_HOME* specified on the command line or set in the global environment. An example of setting *VOLTTRON_HOME* to */tmp/vlhome* is as follows.

```
VOLTTRON_HOME=/tmp/vlhome python scripts/install-agent.py -s <Agent to Package> -c
↪<Config file> --tag <Tag>
```

Agent Configuration Store Interface

The Agent Configuration Store Subsystem provides an interface for facilitating dynamic configuration via the platform configuration store. It is intended to work alongside the original configuration file to create a backwards compatible system for configuring agents with the bundled configuration file acting as default settings for the agent.

If an Agent Author does not want to take advantage of the platform configuration store they need to make no changes. To completely disable the Agent Configuration Store Subsystem an Agent may pass `enable_store=False` to the `Agent.__init__` method.

The Agent Configuration Store Subsystem caches configurations as the platform sends updates to the agent. Updates from the platform will usually trigger callbacks on the agent.

Agent access to the Configuration Store is managed through the `self.vip.config` object in the Agent class.

The “config” Configuration

The configuration name `config` is considered the canonical name of an Agents main configuration. As such the Agent will always run callbacks for that configuration first at startup and when a change to another configuration triggers any callbacks for `config`.

Configuration Callbacks

Agents may setup callbacks for different configuration events. The callback method must have the following signature:

```
my_callback(self, config_name, action, contents)
```

Note: The example above is for a class member method, however the method does not need to be a member of the agent class.

- **config_name** - The method to call when a configuration event occurs.
- **action** - The specific configuration event type that triggered the callback. Possible values are “NEW”, “UPDATE”, “DELETE”. See [Configuration Events](#)
- **contents** - The actual contents of the configuration. Will be a string, list, or dictionary for the actions “NEW” and “UPDATE”. None if the action is “DELETE”.

Note: All callbacks which are connected to the “NEW” event for a configuration will called during agent startup with the initial state of the configuration.

Configuration Events

- **NEW** - This event happens for every existing configuration at Agent startup and whenever a new configuration is added to the Configuration Store.
- **UPDATE** - This event happens every time a configuration is changed.
- **DELETE** - The event happens every time a configuration is removed from the store.

Setting Up a Callback

A callback is setup with the `self.vip.config.subscribe` method.

Note: Subscriptions may be setup at any point in the life cycle of an Agent. Ideally they are setup in `__init__`.

```
subscribe(callback, actions=["NEW", "UPDATE", "DELETE"], pattern="*")
```

- **callback** - The method to call when a configuration event occurs.
- **actions** - The specific configuration event that will trigger the callback. May be a string with the name of a single action or a list of actions.
- **pattern** - The pattern used to match configuration names to trigger the callback.

Configuration Name Pattern Matching

Configuration name matching uses Unix file name matching semantics. Specifically the python module `fnmatch` is used.

Name matching is not case sensitive regardless of the platform VOLTTRON is running on.

For example, the pattern `devices/*` will trigger the supplied callback for any configuration name that starts with `devices/`.

The default pattern matches all configurations.

Getting a Configuration

Once RPC methods are available to an agent (once onstart methods have been called or from any configuration callback) the contents of any configuration may be acquired with the `self.vip.config.get` method.

```
get(config_name="config")
```

If the Configuration Subsystem has not been initialized with the starting values of the agent configuration that will happen in order to satisfy the request.

If initialization occurs to satisfy the request callbacks will *not* be called before returning the results.

Typically an Agent will only obtain the contents of a configuration via a callback. This method is included for agents that want to save state in the store and only need to retrieve the contents of a configuration at startup and ignore any changes to the configuration going forward.

Setting a Configuration

Once RPC methods are available to an agent (once onstart methods have been called) the contents of any configuration may be set with the `self.vip.config.set` method.

```
set(config_name, contents, trigger_callback=False, send_update=False)
```

The contents of the configuration may be a string, list, or dictionary.

This method is intended for agents that wish to maintain a copy of their state in the store for retrieval at startup with the `self.vip.config.get` method.

Warning: This method may **not** be called from a configuration callback. The Configuration Subsystem will detect this and raise a `RuntimeError`, even if `trigger_callback` or `send_update` is `False`.

The platform has a locking mechanism to prevent concurrent configuration updates to the Agent. Calling `self.vip.config.set` would cause the Agent and the Platform configuration store for that Agent to deadlock until a timeout occurs.

Optionally an agent may trigger any callbacks by setting `trigger_callback` to True. If `trigger_callback` is set to False the platform will still send the updated configuration back to the agent. This ensures that a subsequent call to `self.vip.config.get` will still return the correct value. This way the agent's configuration subsystem is kept in sync with the platform's copy of the agent's configuration store at all times.

Optionally the agent may prevent the platform from sending the updated file to the agent by setting `send_update` to False. This setting is available strictly for performance tuning.

Warning: This setting will allow the agent's view of the configuration to fall out of sync with the platform. Subsequent calls to `self.vip.config.get` will return an old version of the file if it exists in the agent's view of the configuration store.

This will also affect any configurations that reference the configuration changed with this setting.

Care should be taken to ensure that the configuration is only retrieved at agent startup when using this option.

Setting a Default Configuration

In order to more easily allow agents to use both the Configuration Store while still supporting configuration via the tradition method of a bundled configuration file the `self.vip.config.set_default` method was created.

```
set_default(config_name, contents)
```

Warning: This method may **not** be called once the Agent Configuration Store Subsystem has been initialized. This method should only be called from `__init__` or an `onsetup` method.

The `set_default` method adds a temporary configuration to the Agents Configuration Subsystem. Nothing is sent to the platform. If a configuration with the same name exists in the platform store it will be presented to a callback method in place of the default configuration.

The normal way to use this is to set the contents of the packaged Agent configuration as the default contents for the configuration named `config`. This way the same callback used to process `config` configuration in the Agent will be called when the Configuration Subsystem can be used to process the configuration file packaged with the Agent.

Note: No attempt is made to merge a default configuration with a configuration from the store.

If a configuration is deleted from the store and a default configuration exists with the same name the Agent Configuration Subsystem will call the `UPDATE` callback for that configuration with the contents of the default configuration.

Other Methods

In a well thought out configuration scheme these methods should not be needed but are included for completeness.

List Configurations

A current list of all configurations for the Agent may be called with the `self.vip.config.list` method.

Unsubscribe

All subscriptions can be removed with a call to the `self.vip.config.unsubscribe_all` method.

Delete

A configuration can be deleted with a call to the `self.vip.config.delete` method.

```
delete(config_name, trigger_callback=False)
```

Note: This method may **not** be called from a callback for the same reason as the `self.vip.config.set` method.

Delete Default

A default configuration can be deleted with a call to the `self.vip.config.delete_default` method.

```
delete_default(config_name)
```

Warning: This method may **not** be called once the Agent Configuration Store Subsystem has been initialized. This method should only be called from `__init__` or an `onsetup` method.

Example Agent

The following example shows how to use `set_default` with a basic configuration and how to setup callbacks.

```
def my_agent(config_path, **kwargs):

    config = utils.load_config(config_path) #Now returns {} if config_path does not_
    ↪exist.

    setting1 = config.get("setting1", 42)
    setting2 = config.get("setting2", 2.5)

    return MyAgent(setting1, setting2, **kwargs)

class MyAgent(Agent):
    def __init__(self, setting1=0, setting2=0.0, **kwargs):
        super(MyAgent, self).__init__(**kwargs)

        self.default_config = {"setting1": setting1,
                               "setting2": setting2}

        self.vip.config.set_default("config", self.default_config)
```

(continues on next page)

(continued from previous page)

```

        #Because we have a default config we don't have to worry about "DELETE"
        self.vip.config.subscribe(self.configure_main, actions=["NEW", "UPDATE"],
↪pattern="config")
        self.vip.config.subscribe(self.configure_other, actions=["NEW", "UPDATE"],
↪pattern="other_config/*")
        self.vip.config.subscribe(self.configure_delete, actions="DELETE", pattern=
↪"other_config/*")

    def configure_main(self, config_name, action, contents):
        #Ensure that we use default values from anything missing in the configuration.
        config = self.default_config.copy()
        config.update(contents)

        _log.debug("Configuring MyAgent")

        #Sanity check the types.
        try:
            setting1 = int(config["setting1"])
            setting2 = float(config["setting2"])
        except ValueError as e:
            _log.error("ERROR PROCESSING CONFIGURATION: {}".format(e))
            #TODO: set a health status for the agent
            return

        _log.debug("Using setting1 {}, setting2 {}".format(setting1, setting2))
        #Do something with setting1 and setting2.

    def configure_other(self, config_name, action, contents):
        _log.debug("Configuring From {}".format(config_name))
        #Do something with contents of configuration.

    def configure_delete(self, config_name, action, contents):
        _log.debug("Removing {}".format(config_name))
        #Do something in response to the removed configuration.

```

Writing Agent Tests

The VOLTTTRON team strongly encourages developing agents with a set of unit and integration tests. Test-driven development can save developers significant time and effort by clearly defining behavioral expectations for agent code. We recommend developing agent tests using Pytest. Agent code contributed to VOLTTTRON is expected to include a set of tests using Pytest in the agent module directory. Following are instructions for setting up Pytest, structuring your tests, how to write unit and integration tests (including some helpful tools using Pytest and Mock) and how to run your tests.

Installation

To get started with Pytest, install it in an activated environment:

```
pip install pytest
```

Or when running VOLTTTRON's bootstrap process, specify the `--testing` optional argument.

```
python bootstrap.py --testing
```

Pytest on PyPI

Module Structure

We suggest the following structure for your agent module:



The test suite should be in a *tests* directory in the root agent directory, and should contain one or more test code files (with the *test_<name of test>* convention). *contest.py* can be used to give all agent tests access to some portion of the VOLTTRON code. In many cases, agents use *contest.py* to import VOLTTRON testing fixtures for integration tests.

Naming Conventions

Pytest tests are discovered and run using some conventions:

- Tests will be found recursively in either the directory specified when running Pytest, or the current working directory if no argument was supplied
- Pytest will search in those directories for files called *test_<name of test>.py* or *<name of test>_test.py*
- **In those files, Pytest will test:**
 - functions and methods prefixed by “test” outside of any class
 - functions and methods prefixed by “test” inside of any class prefixed by “test”



```
# test1.py

def helper_method():
    return 1

def test_success():
    assert helper_method()

# test2.py

def test_success():
```

(continues on next page)

(continued from previous page)

```

    assert True

def test_fail():
    assert False

# file.py

def test_success():
    assert True

def test_fail():
    assert False

```

In the above example, Pytest will run the tests *test_success* from the file *test1.py* and *test_success* and *test_fail* from *test2.py*. No tests will be run from *file.txt*, even though it contains test code, nor will it try to run *helper_method* from *test1.py* as a test.

Writing Unit Tests

These tests should test the various methods of the code base, checking for success and fail conditions. These tests should capture how the components of the system should function; and describe all the possible output conditions given the possible range of inputs including how they should fail if given improper input.

[Pytest guide to Unit Testing](#)

Mocking Dependencies

VOLTTTRON agents include code for many platform features; these features can be mocked to allow unit tests to test only the features of the agent without having to account for the behaviors of the core platform. While there are many tools that can mock dependencies of an agent, we recommend Volttron's AgentMock or Python's Mock testing library.

AgentMock

AgentMock was specifically created to run unit tests on agents. AgentMock takes an Agent class and mocks the attributes and methods of that Agent's dependencies. AgentMock also allows you to customize the behavior of dependencies within each individual test. Below is an example:

```

# Import the Pytest, Mock, base Agent, and Agent mock utility from VOLTTTRON's_
↪ repository
import pytest
import mock
from volttron.platform.vip.agent import Agent
from volttron.testing.utils.utils import AgentMock
# Import your agent code
from UserAgent import UserAgentClass

UserAgentClass.__bases__ = (AgentMock.imitate(Agent, Agent()),)
agent = UserAgentClass()

def test_success_case():
    result = agent.do_function("valid input")
    assert isinstance(result, dict)

```

(continues on next page)

(continued from previous page)

```

for key in ['test1', 'test2']:
    assert key in result
assert result.get("test1") == 10
assert isinstance(result.get("test2"), str)
# ...

def test_success_case_custom_mocks():
    agent.some_dependency.some_method.return_value = "foobar"
    agent.some_attribute = "custom, dummy value"
    result = agent.do_function_that_relies_on_custom_mocks("valid input")
    # ...

def test_failure_case():
    # pytest.raises can be useful for testing exceptions, more information about_
    ↪usage below
    with pytest.raises(ValueError, match=r'Invalid input string for do_function'):
        result = agent.do_function("invalid input")

```

Mock

Similar to AgentMock, Python's Mock testing library allows a user to replace the behavior of dependencies with a user-specified behavior. This is useful for replacing VOLTRON platform behavior, remote API behavior, modules, etc. where using them in unit or integration tests is impractical or impossible. Below is an example that uses the patch decorator to mock an Agent's web request.

Mock documentation

```

class UserAgent():

    def __init__():
        # Code here

    def get_remote_data():
        response = self._get_data_from_remote()
        return "Remote response: {}".format(response)

    # it can be useful to create private functions for use with mock for things like_
    ↪making web requests
    def _get_data_from_remote():
        url = "test.com/test1"
        headers = {}
        return requests.get(url, headers)

# ~~~~~

import pytest
import mock

def get_mock_response():
    return "test response"

# here we're mocking the UserAgent's _get_data_from_remote method and replacing it_
↪with our get_mock_response method
# to feed our test some fake remote data
@mock.patch.object(UserAgent, '_get_data_from_remote', get_mock_response)

```

(continues on next page)

(continued from previous page)

```
def test_get_remote_data():
    assert UserAgent.get_remote_Data() == "Remote response: test response"
```

Pytest Tools

Pytest includes many helpful tools for developing your tests. We'll highlight a few that have been useful for VOLTTRON core tests, but checkout [the Pytest documentation](#) for additional information on each tool as well as tools not covered in this guide.

Pytest Fixtures

Pytest fixtures can be used to create reusable code for tests that can be accessed by every test in a module based on scope. There are several kinds of scopes, but commonly used are “module” (the fixture is run once per module for all the tests of that module) or “function” (the fixture is run once per test). For fixtures to be used by tests, they should be passed as parameters.

Pytest Fixture documentation

Here is an example of a fixture, along with using it in a test:

```
# Fixtures with scope function will be run once per test if the test accepts the
↳ fixture as a parameter
@pytest.fixture(scope="function")
def cleanup_database():
    # This fixture cleans up a sqlite database in between each test run
    sqlite_conn = sqlite.connect("test.sqlite")
    cursor = sqlite_conn.cursor()
    cursor.execute("DROP TABLE 'TEST'")
    cursor.commit()

    cursor.execute("CREATE TABLE TEST (ID INTEGER, FirstName TEXT, LastName TEXT,
↳ Occupation Text)")
    cursor.commit()
    sqlite_conn.close()

# when we pass the cleanup function, we expect that the table will be dropped and
↳ rebuilt before the test runs
def test_store_data(cleanup_database):
    sqlite_conn = sqlite.connect("test.sqlite")
    cursor = sqlite_conn.cursor()
    # after this insert, we'd expect to only have 1 value in the table
    cursor.execute("INSERT INTO TEST VALUES(1, 'Test', 'User', 'Developer')")
    cursor.commit()

    # validate the row count
    cursor.execute("SELECT COUNT(*) FROM TEST")
    count = cursor.fetchone()
    assert count == 1
```

Pytest.mark

Pytest marks are used to set metadata for test functions. Defining your own custom marks can allow you to run subsections of your tests. Parametrize can be used to pass a series of parameters to a test, so that it can be run many

times to cover the space of potential inputs. Marks also exist to specify expected behavior for tests.

[Mark documentation](#)

Custom Marks

To add a custom mark, add the name of the mark followed by a colon then a description string to the ‘markers’ section of `Pytest.ini` (an example of this exists in the core VOLTTRON repository). Then add the appropriate decorator:

```
@pytest.mark.UserAgent
def test_success_case():
    # TODO unit test here
    pass
```

The VOLTTRON team also has a *dev* mark for running individual (or a few) one-off tests.

```
@pytest.mark.dev
@pytest.mark.UserAgent
def test_success_case():
    # TODO unit test here
    pass
```

Parametrize

Parametrize will allow tests to be run with a variety of parameters. Add the parametrize decorator, and for parameters include a list of parameter names matching the test parameter names as a comma-delimited string followed by a list of tuples containing parameters for each test.

[Parametrize docs](#)

```
@pytest.mark.parametrize("test_input1, test_input2, expected", [(1, 2, 3), (-1, 0, "
→")])
def test_user_agent(param1, param2, param3):
    # TODO unit test here
    pass
```

Skip, skipif, and xfail

The *skip* mark can be used to skip a test for any reason every time the test suite is run:

```
# This test will be skipped!
@pytest.mark.skip
def test_user_agent():
    # TODO unit test here
    pass
```

The *skipif* mark can be used to skip a test based on some condition:

```
# This test will be skipped if RabbitMQ hasn't been set up yet!
@pytest.mark.skipif(not isRabbitMQInstalled)
def test_user_agent():
    # TODO unit test here
    pass
```


The *xfail* mark can be used to run a test, but to show that the test is currently expected to fail

```
# This test will fail, but will not cause the module tests to be considered failing!
@pytest.mark.xfail
def test_user_agent():
    # TODO unit test here
    assert False
```

Skip, skipif, and xfail docs

Writing Integration Tests

Integration tests are useful for testing the faults that occur between integrated units. In the context of VOLTTRON agents, integration tests should test the interactions between the agent, the platform, and other agents installed on the platform that would interface with the agent. It is typical for integration tests to test configuration, behavior and content of RPC calls and agent Pub/Sub, the agent subsystems, etc.

Pytest best practices for Integration Testing

Volttrontesting Directory

The *Volttrontesting* directory includes several helpful fixtures for your tests. Including the following line at the top of your tests, or in *conf/test.py*, will allow you to utilize the platform wrapper fixtures, and more.

```
from volttrontesting.fixtures.volttron_platform_fixtures import *
```

Here is an example success case integration test:

```
import pytest
import mock
from volttrontesting.fixtures.volttron_platform_fixtures import *

# If the test requires user specified values, setting environment variables or having
↳ settings files is recommended
API_KEY = os.environ.get('API_KEY')

# request object is a pytest object for managing the context of the test
@pytest.fixture(scope="module")
def Weather(request, volttron_instance):
    config = {
        "API_KEY": API_KEY
    }
    # using the volttron_instance fixture (passed in by volttrontesting fixtures), we
↳ can install an agent
    # on the platform to test against
    agent = volttron_instance.install_agent(
        vip_identity=identity,
        agent_dir=source,
        start=False,
        config_file=config)

    volttron_instance.start_agent(agent)
    gevent.sleep(3)

    def stop_agent():
```

(continues on next page)

(continued from previous page)

```

        print("stopping weather service")
        if volttron_instance.is_running():
            volttron_instance.stop_agent(agent)
        # here we used the passed request object to add something to happen when the test_
        ↪is finished
        request.addfinalizer(stop_agent)
        return agent, identity

# Here we create a really simple agent which has only the core functionality, which_
        ↪we can use for Pub/Sub
# or JSON/RPC
@pytest.fixture(scope="module")
def query_agent(request, volttron_instance):
    # Create the simple agent
    agent = volttron_instance.build_agent()

    def stop_agent():
        print("In teardown method of query_agent")
        agent.core.stop()

    request.addfinalizer(stop_agent)
    return agent

# pass the 2 fixtures to our test, then we can run the test
def test_weather_success(Weather, query_agent):
    query_data = query_agent.vip.rpc.call(identity, 'get_current_weather', locations).
    ↪get(timeout=30)
    assert query_data.get("weather_results") = "Its sunny today!"

```

For more integration test examples, it is recommended to take a look at some of the VOLTTRON core agents, such as historian agents and weather service agents.

Using Docker for Limited-Integration Testing

If you want to run limited-integration tests which do not require the setup of a volttron system, you can use Docker containers to mimic dependencies of an agent. The `volttrontesting/fixtures/docker_wrapper.py` module provides a convenient function to create docker containers for use in limited-integration tests. For example, suppose that you had an agent with a dependency on a MySQL database. If you want to test the connection between the Agent and the MySQL dependency, you can create a Docker container to act as a real MySQL database. Below is an example:

```

from volttrontesting.fixtures.docker_wrapper import create_container
from UserAgent import UserAgentClass

def test_docker_wrapper_example():
    ports_config = {'3306/tcp': 3306}
    with create_container("mysql:5.7", ports=ports_config) as container:
        init_database(container)
        agent = UserAgent(ports_config)

        results = agent.some_method_that_talks_to_container()

```

Running your Tests and Debugging

Pytest can be run from the command line to run a test module.

```
pytest <path to module to be tested>
```

If using marks, you can add `-m <mark>` to specify your testing subset, and `-s` can be used to suppress standard output. For more information about optional arguments you can type `pytest --help` into your command line interface to see the full list of options.

Testing output should look something like this:

```
(voltttron) <user>@<host>:~/voltttron$ pytest services/core/SQLHistorian/
===== test session starts _
<-----
platform linux -- Python 3.6.9, pytest-5.4.1, py-1.8.1, pluggy-0.13.1 -- /home/<user>/
<----- voltttron/env/bin/python
cachedir: .pytest_cache
rootdir: /home/<user>/voltttron, inifile: pytest.ini
plugins: timeout-1.3.4
timeout: 240.0s
timeout method: signal
timeout func_only: False
collected 2 items

services/core/SQLHistorian/tests/test_sqlitehistorian.py::test_sqlite_
<----- timeout[voltttron_3-voltttron_instance0] ERROR [ 50%]
services/core/SQLHistorian/tests/test_sqlitehistorian.py::test_sqlite_
<----- timeout[voltttron_3-voltttron_instance1] PASSED [100%]

===== ERRORS _
<-----
_____ ERROR at setup of test_sqlite_timeout[voltttron_3-
<----- voltttron_instance0] _____

request = <SubRequest 'voltttron_instance' for <Function test_sqlite_timeout[voltttron_
<----- 3-voltttron_instance0]>>, kwargs = {}
address = 'tcp://127.0.0.113:5846'

    @pytest.fixture(scope="module",
                    params=[
                        dict(messagebus='zmq', ssl_auth=False),
                        pytest.param(dict(messagebus='rmq', ssl_auth=True), marks=rmq_
<----- skipif),
                    ])
    def voltttron_instance(request, **kwargs):
        """Fixture that returns a single instance of voltttron platform for testing

        @param request: pytest request object
        @return: voltttron platform instance
        """
        address = kwargs.pop("vip_address", get_rand_vip())
        wrapper = build_wrapper(address,
                                messagebus=request.param['messagebus'],
                                ssl_auth=request.param['ssl_auth'],
>                                **kwargs)

address      = 'tcp://127.0.0.113:5846'
kwargs       = {}
request      = <SubRequest 'voltttron_instance' for <Function test_sqlite_
<----- timeout[voltttron_3-voltttron_instance0]>>
```

(continues on next page)

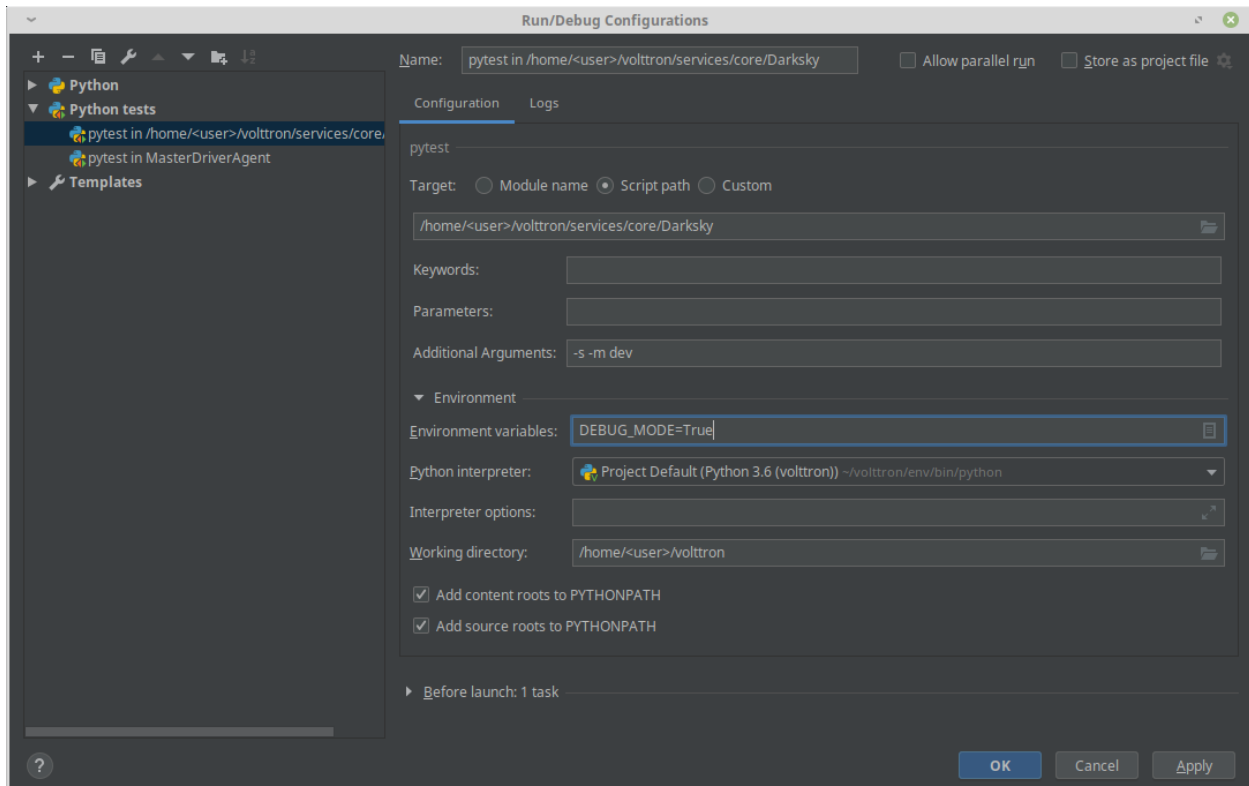
(continued from previous page)

```
volttrontesting/fixtures/volttron_platform_fixtures.py:106:
```

Running Tests Via PyCharm

To run our Pytests using PyCharm, we'll need to create a run configuration. To do so, select “edit configurations” from the “Run” menu (or if using the toolbar UI element you can click on the run configurations dropdown to select “edit configurations”). Use the plus symbol at the top right of the pop-up menu, scroll to “Python Tests” and expand this menu and select “pytest”. This will create a run configuration, which will then need to be filled out. We recommend the following in general:

- Set the “Script Path” radio and fill the form with the path to your module. Pytest will run any tests in that module using the discovery process described above (and any marks if specified)
- In the interpreter dropdown, select the VOLTTTRON virtual environment - this will likely be your project default
- Set the working directory to the VOLTTTRON root directory
- Add any environment variables - For debugging, add variable “DEBUG_MODE” = True or “DEBUG” 1
- Add any optional arguments (-s will prevent standard output from being displayed in the console window, -m is used to specify a mark)



[PyCharm testing instructions](#)

[More information on testing in Python](#)

Developing Historian Agents

VOLTTTRON provides a convenient base class for developing new historian agents. The base class automatically performs a number of important functions:

- subscribes to all pertinent topics
- caches published data to disk until it is successfully recorded to a historian
- creates the public facing interface for querying results
- spells out a simple interface for concrete implementation to meet to make a working Historian Agent
- breaks data to publish into reasonably sized chunks before handing it off to the concrete implementation for publication. The size of the chunk is configurable
- sets up a separate thread for publication. If publication code needs to block for a long period of time (up to 10s of seconds) this will no disrupt the collection of data from the bus or the functioning of the agent itself

The VOLTTTRON repository provides several historians which can be deployed without modification.

BaseHistorian

All Historians must inherit from the BaseHistorian class in `volttron.platform.agent.base_historian` and implement the following methods:

`publish_to_historian(self, to_publish_list)`

This method is called by the BaseHistorian class when it has received data from the message bus to be published. `to_publish_list` is a list of records to publish in the form:

```
[
  {
    '_id': 1,
    'timestamp': timestamp,
    'source': 'scrape',
    'topic': 'campus/building/unit/point',
    'value': 90,
    'meta': {'units': 'F'}
  }
  ...
]
```

- **_id** - ID of the record used for internal record tracking. All IDs in the list are unique
- **timestamp** - Python datetime object of the time data was published at timezone UTC
- **source** - Source of the data: can be scrape, analysis, log, or actuator
- **topic** - Topic data was published on, topic prefix's such as "device" are dropped
- **value** - Value of the data, can be any type.
- **meta** - Metadata for the value, some sources will omit this entirely.

For each item in the list the concrete implementation should attempt to publish (or discard if non-publishable) every item in the list. Publication should be batched if possible. For every successfully published record and every record that is to be discarded because it is non-publishable the agent must call *report_handled* on those records. Records

that should be published but were not for whatever reason require no action. Future calls to *publish_to* '_historian' will include these unpublished records. *publish_to_historian* is always called with the oldest unhandled records. This allows the historian to no lose data due to lost connections or other problems.

As a convenience *report_all_handled* can be called if all of the items in *published_list* were successfully handled.

query_topic_list(self)

Must return a list of all unique topics published.

query_historian(self, topic, start=None, end=None, skip=0, count=None, order=None)

This function must return the results of a query in the form:

```
{ "values": [(timestamp1: value1), (timestamp2: value2), ...],  
  "metadata": { "key1": value1, "key2": value2, ...} }
```

metadata is not required (The caller will normalize this to {} for you if you leave it out)

- **topic** - the topic the user is querying for
- **start** - datetime of the start of the query, *None* for the beginning of time
- **end** - datetime of the end of of the query, *None* for the end of time
- **skip** - skip this number of results (for pagination)
- **count** - return at maximum this number of results (for pagination)
- **order** - *FIRST_TO_LAST* for ascending time stamps, *LAST_TO_FIRST* for descending time stamps

historian_setup(self)

Implementing this is optional. This function is run on the same thread as the rest of the concrete implementation at startup. It is meant for connection setup.

Example Historian

An example historian can be found in the *examples/CSVHistorian* directory in the VOLTTRON repository. This example historian uses a CSV file as the persistent data store. It is recommended to use this agent as a reference for developing new historian agents.

Developing Market Agents

VOLTTRON provides a convenient base class for developing new market agents. The base class automatically subscribes to all pertinent topics, and spells out a simple interface for concrete implementation to make a working Market Agent.

Markets are implemented by the Market Service Agent which is a core service agent. The Market Service Agent publishes information on several topics to which the base agent automatically subscribes. The base agent also provides all the methods you will need to interact with the Market Service Agent to implement your market transactions.

MarketAgent

All Market Agents must inherit from the MarketAgent class in *volttron.platform.agent.base_market_agent* and call the following method:

```
self.join_market(market_name, buyer_seller, reservation_callback, offer_callback,
    ↳aggregate_callback, price_callback, error_callback)
```

This method causes the market agent to join a single market. If the agent wishes to participate in several markets it may be called once for each market. The first argument is the name of the market to join and this name must be unique across the entire volttron instance because all markets are implemented by a single market service agent for each volttron instance. The second argument describes the role that this agent wished to play in this market. The value is imported as:

```
from volttron.platform.agent.base_market_agent.buy_sell import BUYER, SELLER
```

Arguments 3-7 are callback methods that the agent may implement as needed for the agent's participation in the market.

The Reservation Callback

```
reservation_callback(self, timestamp, market_name, buyer_seller)
```

This method is called when it is time to reserve a slot in the market for the current market cycle. If this callback is not registered a slot is reserved for every market cycle. If this callback is registered it is called for each market cycle and returns *True* if a reservation is wanted and *False* if a reservation is not wanted.

The name of the market and the roll being played are provided so that a single callback can handle several markets. If the agent joins three markets with the same reservation callback routine it will be called three times with the appropriate market name and buyer/seller role for each call. The MeterAgent example illustrates the use of this of this method and how to determine whether to make an offer when the reservation is refused.

A market will only exist if there are reservations for at least one buyer or one seller. If the market fails to achieve the minimum participation the error callback will be called. If only buyers or only sellers make reservations any offers will be rejected with the reason that the market has not formed.

The Offer Callback

```
offer_callback(self, timestamp, market_name, buyer_seller)
```

If the agent has made a reservation for the market and a callback has been registered this callback is called. If the agent wishes to make an offer at this time the market agent computes either a supply or a demand curve as appropriate and offers the curve to the market service by calling the *make_offer* method.

The name of the market and the roll being played are provided so that a single callback can handle several markets.

For each market joined either an offer callback, an aggregate callback, or a cleared price callback is required.

The Aggregate Callback

```
aggregate_callback(self, timestamp, market_name, buyer_seller, aggregate_curve)
```

When a market has received all its buy offers it calculates an aggregate demand curve. When the market receives all of its sell offers it calculates an aggregate supply curve. This callback delivers the aggregate curve to the market agent whenever the appropriate curve becomes available.

If the market agent wants to use this opportunity to make an offer on this or another market it would do that using the `make_offer` method.

- If the aggregate demand curve is received, only a supply offer may be submitted for this market
- If the aggregate supply curve is received, only make a demand offer will be accepted by this market.

You may use this information to make an offer on another market; The example `AHUAgent` does this. The name of the market and the roll being played are provided so that a single callback can handle several markets.

For each market joined, either an offer callback, an aggregate callback, or a cleared price callback is required.

The Price Callback

```
price_callback(self, timestamp, market_name, buyer_seller, price, quantity)
```

This callback is called when the market clears. If the market agent wants to use this opportunity to make an offer on this or another market it would do that using the `make_offer` method.

Once the market has cleared you can not make an offer on that market. Again, you may use this information to make an offer on another market as in the example `AHUAgent`. The name of the market and the roll being played are provided so that a single callback can handle several markets.

For each market joined either an offer callback, an aggregate callback, or a cleared price callback is required.

The Error Callback

```
error_callback(self, timestamp, market_name, buyer_seller, error_code, error_message, aux)
```

This callback is called when an error occurs isn't in response to an RPC call. The error codes are documented in:

```
from volttron.platform.agent.base_market_agent.error_codes import NOT_FORMED, SHORT_
OFFERS, BAD_STATE, NO_INTERSECT
```

- `NOT_FORMED` - If a market fails to form this will be called at the offer time.
- `SHORT_OFFERS` - If the market doesn't receive all its offers this will be called while clearing the market.
- `BAD_STATE` - This indicates a bad state transition while clearing the market and should never happen, but may be called while clearing the market.
- `NO_INTERSECT` - If the market fails to clear this would be called while clearing the market and an auxillary array will be included. The auxillary array contains comparisons between the supply max, supply min, demand max and demand min. They allow the market client to make determinations about why the curves did not intersect that may be useful.

The error callback is optional, but highly recommended.

Example Agents

Some example agents are included with the platform to help explore its features. These agents represent concrete implementations of important agent sub-types such as Historians or Weather Agents, or demonstrate a development pattern for accomplishing common tasks.

More complex agents contributed by other researchers can also be found in the examples directory. It is recommended that developers new to VOLTTTRON understand the example agents first before diving into the other agents.

C Agent

The C Agent uses the *ctypes* module to load a shared object into memory so its functions can be called from Python.

There are two versions of the C Agent:

- A standard agent that can be installed with the agent installation process
- A driver which can be controlled using the Master Driver Agent

Building the Shared Object

The shared object library must be built before installing C Agent examples. Running `make` in the C Agent source directory will compile the provided C code using the position independent flag, a requirement for creating shared objects.

Files created by `make` can be removed by running

```
make clean
```

Agent Installation

After building the shared object library the standard agent can be installed with the `scripts/install-agent.py` script:

```
python scripts/install-agent.py -s examples/CAgent
```

The other is a driver interface for the Master Driver. To use the C driver, the driver code file must be moved into the Master Driver's *interfaces* directory:

```
examples/CAgent/c_agent/driver/cdriver -> services/core/MasterDriverAgent/  
↪ master_driver/interfaces
```

The C Driver configuration tells the interface where to find the shared object. An example is available in the C Agent's *driver* directory.

Config Actuation Example

The Config Actuation example attempts to set points on a device when files are added or updated in its configuration store.

Configuration

The name of a configuration file must match the name of the device to be actuated. The configuration file is a JSON dictionary of point name and value pairs. Any number of points on the device can be listed in the config.

```
{
    "point0": value,
    "point1": value
}
```

CSV Historian

The CSV Historian Agent is an example historian agent that writes device data to the CSV file specified in the configuration file.

Explanation of CSV Historian

The Utils module of the VOLTTRON platform includes functions for setting up global logging for the platform:

```
utils.setup_logging()
_log = logging.getLogger(__name__)
```

The historian method is called by `utils.vip_main` when the agents is started (see below). `utils.vip_main` expects a callable object that returns an instance of an Agent. This method of dealing with a configuration file and instantiating an Agent is common practice.

```
def historian(config_path, **kwargs):
    if isinstance(config_path, dict):
        config_dict = config_path
    else:
        config_dict = utils.load_config(config_path)

    output_path = config_dict.get("output", "~/historian_output.csv")

    return CSVHistorian(output_path = output_path, **kwargs)
```

All historians must inherit from *BaseHistorian*. The *BaseHistorian* class handles the capturing and caching of all device, logging, analysis, and record data published to the message bus.

```
class CSVHistorian(BaseHistorian):
```

The Base Historian creates a separate thread to handle publishing data to the data store. In this thread the Base Historian calls two methods on the created historian, `historian_setup` and `publish_to_historian`.

The Base Historian created the new thread in it's `__init__` method. This means that any instance variables must assigned in `__init__` before calling the Base Historian's `__init__` method.

```
def __init__(self, output_path="", **kwargs):
    self.output_path = output_path
    self.csv_dict = None
    super(CSVHistorian, self).__init__(**kwargs)
```

Historian setup is called shortly after the new thread starts. This is where a Historian sets up a connect the first time. In our example we create the *Dictwriter* object that we will use to create and add lines to the CSV file.

We keep a reference to the file object so that we may flush its contents to disk after writing the header and after we have written new data to the file.

The CSV file we create will have 4 columns: *timestamp*, *source*, *topic*, and *value*.

```
def historian_setup(self):
    self.f = open(self.output_path, "wb")
    self.csv_dict = csv.DictWriter(self.f, ["timestamp", "source", "topic", "value"])
    self.csv_dict.writeheader()
    self.f.flush()
```

`publish_to_historian` is called when data is ready to be published. It is passed a list of dictionaries. Each dictionary contains a record of a single value that was published to the message bus.

The dictionary takes the form:

```
{
    '_id': 1,
    'timestamp': timestamp1.replace(tzinfo=pytz.UTC), #Timestamp in UTC
    'source': 'scrape', #Source of the data point.
    'topic': "pnnl/isbl/hvac1/thermostat", #Topic that published to without prefix.
    'value': 73.0, #Value that was published
    'meta': {"units": "F", "tz": "UTC", "type": "float"} #Meta data published with_
    ↪the topic
}
```

Once the data is written to the historian we call `self.report_all_handled()` to inform the *BaseHistorian* that all data we received was successfully published and can be removed from the cache. Then we can flush the file to ensure that the data is written to disk.

```
def publish_to_historian(self, to_publish_list):
    for record in to_publish_list:
        row = {}
        row["timestamp"] = record["timestamp"]

        row["source"] = record["source"]
        row["topic"] = record["topic"]
        row["value"] = record["value"]

        self.csv_dict.writerow(row)

    self.report_all_handled()
    self.f.flush()
```

This agent does not support the Historian Query interface.

Agent Testing

The CSV Historian can be tested by running the included `launch_my_historian.sh` script.

Agent Installation

This Agent may be installed on the platform using the standard method.

Data Publisher

This is a simple agent that plays back data either from the config store or a CSV to the configured topic. It can also provide basic emulation of the Actuator Agent for testing agents that expect to be able to set points on a device in response to device publishes.

Installation notes

In order to simulate the actuator you must install the agent with the VIP identity of *platform.actuator*. If an actuator is already installed on the platform, this will cause VIP identity conflicts. To install the agent, the agent install script can be used:

```
python scripts/install-agent.py -s examples/DataPublisher -c <config file>
```

Configuration

```
{
    # basetopic can be devices, analysis, or custom base topic
    "basepath": "devices/PNNL/ISB1",

    # use_timestamp uses the included in the input_data if present.
    # Currently the column must be named `Timestamp`.
    "use_timestamp": true,

    # Only publish data at most once every max_data_frequency seconds.
    # Extra data is skipped.
    # The time windows are normalized from midnight.
    # ie 900 will publish one value for every 15 minute window starting from
    # midnight of when the agent was started.
    # Only used if timestamp in input file is used.
    "max_data_frequency": 900,

    # The meta data published with the device data is generated
    # by matching point names to the unittypes_map.
    "unittypes_map": {
        ".*Temperature": "Fahrenheit",
        ".*SetPoint": "Fahrenheit",
        "OutdoorDamperSignal": "On/Off",
        "SupplyFanStatus": "On/Off",
        "CoolingCall": "On/Off",
        "SupplyFanSpeed": "RPM",
        "Damper*": "On/Off",
        "Heating*": "On/Off",
        "DuctStatic*": "On/Off"
    },
    # Path to input CSV file.
    # May also be a list of records or reference to a CSV file in the config store.
    # Large CSV files should be referenced by file name and not
    # stored in the config store.
    "input_data": "econ_test2.csv",
    # Publish interval in seconds
    "publish_interval": 1,
```

(continues on next page)

(continued from previous page)

```

# Tell the playback to maintain the location a the file in the config store.
# Playback will be resumed from this point
# at agent startup even if this setting is changed to false before restarting.
# Saves the current line in line_marker in the DataPublishers's config store
# as plain text.
# default false
"remember_playback": true,

# Start playback from 0 even if the line_marker configuration is set a non 0_
↪value.
# default false
"reset_playback": false,

# Repeat data from the start if this flag is true.
# Useful for data that does not include a timestamp and is played back in real_
↪time.
"replay_data": false
}

```

CSV File Format

The CSV file must have a single header line. The column names are appended to the *basepath* setting in the configuration file and the resulting topic is normalized to remove extra ' / ' characters. The values are all treated as floating point values and converted accordingly.

The corresponding device for each point is determined and the values are combined together to create an *all* topic publish for each device.

If a *Timestamp* column is in the input it may be used to set the timestamp in the header of the published data.

Timestamp	centrifugal_chiller/OutsideAirTemperature	centrifugal_chiller/DischargeAirTemperatureSetPoint
2012/05/19 05:07:00	0	56
2012/05/19 05:08:00	0	56
2012/05/19 05:09:00	0	56
2012/05/19 05:10:00	0	56
2012/05/19 05:11:00	0	56
2012/05/19 05:12:00	0	56
2012/05/19 05:13:00	0	56
2012/05/19 05:14:00	0	56
2012/05/19 05:15:00	0	56
2012/05/19 05:16:00	0	56
2012/05/19 05:17:00	0	56
2012/05/19 05:18:00	0	56
2012/05/19 05:19:00	0	56
2012/05/19 05:20:00	0	56
2012/05/19 05:21:00	0	56
2012/05/19 05:22:00	0	56
2012/05/19 05:23:00	0	56
2012/05/19 05:24:00	0	56
2012/05/19 05:25:00	48.78	56
2012/05/19 05:26:00	48.88	56

Timestamp	centrifugal_chiller/OutsideAirTemperature	centrifugal_chiller/DischargeAirTemperatureSetPoint
2012/05/19 05:27:00	48.93	56
2012/05/19 05:28:00	48.95	56
2012/05/19 05:29:00	48.92	56
2012/05/19 05:30:00	48.88	56
2012/05/19 05:31:00	48.88	56
2012/05/19 05:32:00	48.99	56
2012/05/19 05:33:00	49.09	56
2012/05/19 05:34:00	49.11	56
2012/05/19 05:35:00	49.07	56
2012/05/19 05:36:00	49.05	56
2012/05/19 05:37:00	49.09	56
2012/05/19 05:38:00	49.13	56
2012/05/19 05:39:00	49.09	56
2012/05/19 05:40:00	49.01	56
2012/05/19 05:41:00	48.92	56
2012/05/19 05:42:00	48.86	56
2012/05/19 05:43:00	48.92	56
2012/05/19 05:44:00	48.95	56
2012/05/19 05:45:00	48.92	56
2012/05/19 05:46:00	48.86	56
2012/05/19 05:47:00	48.78	56
2012/05/19 05:48:00	48.69	56
2012/05/19 05:49:00	48.65	56
2012/05/19 05:50:00	48.65	56
2012/05/19 05:51:00	48.65	56
2012/05/19 05:52:00	48.61	56
2012/05/19 05:53:00	48.59	56
2012/05/19 05:54:00	48.55	56
2012/05/19 05:55:00	48.63	56
2012/05/19 05:56:00	48.76	56
2012/05/19 05:57:00	48.95	56
2012/05/19 05:58:00	49.24	56
2012/05/19 05:59:00	49.54	56
2012/05/19 06:00:00	49.71	56
2012/05/19 06:01:00	49.79	56
2012/05/19 06:02:00	49.94	56
2012/05/19 06:03:00	50.13	56
2012/05/19 06:04:00	50.18	56
2012/05/19 06:05:00	50.15	56

DDS Agent

The DDS example agent demonstrates VOLTTRON's capacity to be extended with tools and libraries not used in the core codebase. DDS is a messaging platform that implements a publish-subscribe system for well defined data types.

This agent example is meant to be run the command line, as opposed to installing it like other agents. From the *examples/DDSAgent* directory, the command to start it is:

```
$ AGENT_CONFIG=config python -m ddsagent.agent
```

The *rticonnextdds-connector* library needs to be installed for this example to function properly. We'll retrieve it from GitHub since it is not available through Pip. Download the source with:

```
$ wget https://github.com/rticommunity/rticonnextdds-connector/archive/master.zip
```

and unpack it in *examples/DDSAgent/ddsagent* with:

```
$ unzip master.zip
```

The `demo_publish()` output can be viewed with the *rtishapesdemo* available from RTI.

Configuration

Each data type that this agent will have access to needs to have an XML document defining its structure. The XML will include a participant name, publisher name, and a subscriber name. These are recorded in the configuration with the location on disk of the XML file.

```
{
  "square": {
    "participant_name": "MyParticipantLibrary::Zero",
    "xml_config_path": "./ddsagent/rticonnextdds-connector-master/examples/python/
↪ShapeExample.xml",
    "publisher_name": "MyPublisher::MySquareWriter",
    "subscriber_name": "MySubscriber::MySquareReader"
  }
}
```

Listener Agent

The `ListenerAgent` subscribes to all topics and is useful for testing that agents being developed are publishing correctly. It also provides a template for building other agents as it expresses the requirements of a platform agent.

Explanation of Listener Agent Code

Use `utils` to setup logging, which we'll use later.

```
utils.setup_logging()
_log = logging.getLogger(__name__)
```

The `Listener` agent extends (inherits from) the `Agent` class for its default functionality such as responding to platform commands:

```
class ListenerAgent (Agent) :
    '''
    Listens to everything and publishes a heartbeat according to the
    heartbeat period specified in the settings module.
    '''
```

After the class definition, the `Listener` agent reads the configuration file, extracts the configuration parameters, and initializes any `Listener` agent instance variable. This is done through the agent's `__init__` method:

```
def __init__(self, config_path, **kwargs):
    super(ListenerAgent, self).__init__(**kwargs)
    self.config = utils.load_config(config_path)
    self._agent_id = self.config.get('agentid', DEFAULT_AGENTID)
    log_level = self.config.get('log-level', 'INFO')
    if log_level == 'ERROR':
        self._logfn = _log.error
    elif log_level == 'WARN':
        self._logfn = _log.warn
    elif log_level == 'DEBUG':
        self._logfn = _log.debug
    else:
        self._logfn = _log.info
```

Next, the Listener agent will run its setup method. This method is tagged to run after the agent is initialized by the decorator `@Core.receiver('onsetup')`. This method accesses the configuration parameters, logs a message to the platform log, and sets the agent ID.

```
@Core.receiver('onsetup')
def onsetup(self, sender, **kwargs):
    # Demonstrate accessing a value from the config file
    _log.info(self.config.get('message', DEFAULT_MESSAGE))
    self._agent_id = self.config.get('agentid')
```

The Listener agent subscribes to all topics published on the message bus. Publish and subscribe interactions with the message bus are handled by the *PubSub* module located at `~/volttron/volttron/platform/vip/agent/subsystems/pubsub.py`.

The Listener agent uses an empty string to subscribe to all messages published. This is done in a [decorator](#) for simplifying subscriptions.

```
@PubSub.subscribe('pubsub', '')
def on_match(self, peer, sender, bus, topic, headers, message):
    '''Use match_all to receive all messages and print them out.'''
    if sender == 'pubsub.compat':
        message = compat.unpack_legacy_message(headers, message)
    self._logfn(
        "Peer: %r, Sender: %r, Bus: %r, Topic: %r, Headers: %r, "
        "Message: %r", peer, sender, bus, topic, headers, message)
```

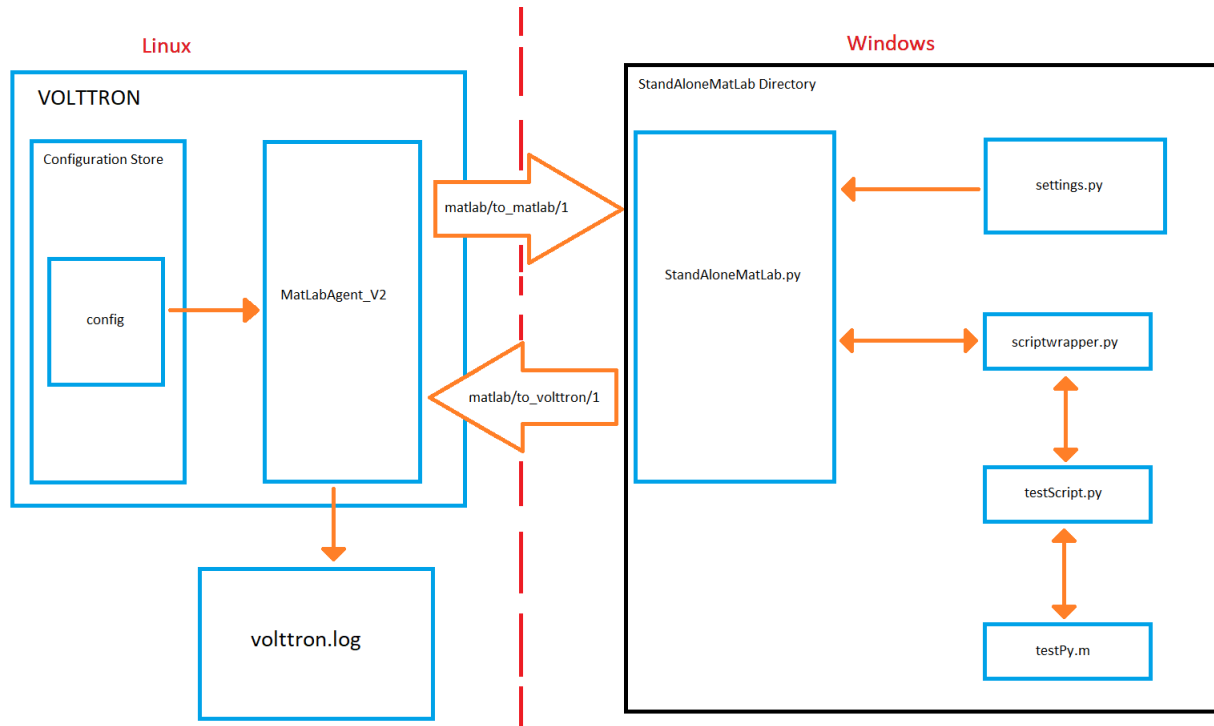
MatLab Agent

The MatLab agent and Matlab Standalone Agent together are example agents that allow for MatLab scripts to be run in a Windows environment and interact with the VOLTTRON platform running in a Linux environment.

The MatLab agent takes advantage of the config store to dynamically send scripts and commandline arguments across the message bus to one or more Standalone Agents in Windows. The Standalone Agent then executes the requested script and arguments, and sends back the results to the MatLab agent.

Overview of Matlab Agents

There are multiple components that are used for the MatLab agent. This diagram is to represent the components that are connected to the MatLab Agents. In this example, the scripts involved are based on the default settings in the MatLab Agent.



MatLabAgentV2

MatLabAgentV2 publishes the name of a python script along with any command line arguments that are needed for the script to the appropriate topic. The agent then listens on another topic, and whenever anything is published on this topic, it stores the message in the log file chosen when the VOLTTRON instance is started. If there are multiple standalone agents, the agent can send a script to each of them, along with their own set of command line arguments. In this case, each script name and set of command line arguments should be sent to separate subtopics. This is done so that no matter how many standalone agents are in use, MatLabAgentV2 will record all of their responses.

```

class MatlabAgentV2 (Agent) :

    def __init__(self, script_names=[], script_args=[], topics_to_matlab=[],
                  topics_to_volttron=None, **kwargs):

        super(MatlabAgentV2, self).__init__(**kwargs)
        _log.debug("vip_identity: " + self.core.identity)

        self.script_names = script_names
        self.script_args = script_args
        self.topics_to_matlab = topics_to_matlab
        self.topics_to_volttron = topics_to_volttron
        self.default_config = {"script_names": script_names,
                               "script_args": script_args,
                               "topics_to_matlab": topics_to_matlab,
                               "topics_to_volttron": topics_to_volttron}

        #Set a default configuration to ensure that self.configure is called_
        #immediately to setup
        #the agent.

```

(continues on next page)

(continued from previous page)

```

self.vip.config.set_default("config", self.default_config)
#Hook self.configure up to changes to the configuration file "config".
self.vip.config.subscribe(self.configure, actions=["NEW", "UPDATE"], pattern=
↪"config")

def configure(self, config_name, action, contents):
    """
    Called after the Agent has connected to the message bus.
    If a configuration exists at startup this will be
    called before onstart.
    Is called every time the configuration in the store changes.
    """
    config = self.default_config.copy()
    config.update(contents)

    _log.debug("Configuring Agent")

    try:
        script_names = config["script_names"]
        script_args = config["script_args"]
        topics_to_matlab = config["topics_to_matlab"]
        topics_to_volttron = config["topics_to_volttron"]

    except ValueError as e:
        _log.error("ERROR PROCESSING CONFIGURATION: {}".format(e))
        return

    self.script_names = script_names
    self.script_args = script_args
    self.topics_to_matlab = topics_to_matlab
    self.topics_to_volttron = topics_to_volttron
    self._create_subscriptions(self.topics_to_volttron)

    for script in range(len(self.script_names)):
        cmd_args = ""
        for x in range(len(self.script_args[script])):
            cmd_args += ", {}".format(self.script_args[script][x])
        _log.debug("Publishing on: {}".format(self.topics_to_matlab[script]))
        self.vip.pubsub.publish('pubsub', topic=self.topics_to_matlab[script],
            message="{}".format(self.script_names[script],cmd_args))
        _log.debug("Sending message: {}".format(self.script_names[script],cmd_
↪args))

    _log.debug("Agent Configured!")

```

For this example, the agent is publishing to the *matlab/to_matlab/1* topic, and is listening to the *matlab/to_volttron* topic. It is sending the script name *testScript.py* with the argument 20. These are the default values found in the agent, if no configuration is loaded.

```

script_names = config.get('script_names', ["testScript.py"])
script_args = config.get('script_args', [["20"]])
topics_to_matlab = config.get('topics_to_matlab', ["matlab/to_matlab/1"])
topics_to_volttron = config.get('topics_to_volttron', "matlab/to_volttron/")

```

StandAloneMatLab.py

The *StandAloneMatLab.py* script is a standalone agent designed to be able to run in a Windows environment. Its purpose is to listen to a topic, and when something is published to this topic, it takes the message, and sends it to the `script_runner` function in *scriptwrapper.py*. This function processes the inputs, and then the output is published to another topic.

```
class StandAloneMatLab(Agent):
    '''The standalone version of the MatLab Agent'''

    @PubSub.subscribe('pubsub', _topics['volttron_to_matlab'])
    def print_message(self, peer, sender, bus, topic, headers, message):
        print('The Message is: ' + str(message))
        messageOut = script_runner(message)
        self.vip.pubsub.publish('pubsub', _topics['matlab_to_volttron'],
        ↪message=messageOut)
```

settings.py

The topic to listen to and the topic to publish to are defined in *settings.py*, along with the information needed to connect the Standalone Agent to the primary VOLTTTRON instance. These should be the same topics that the MatLabAgentV2 is publishing and listening to, so that the communication can be successful. To connect the Standalone Agent to the primary VOLTTTRON instance, the IP address and port of the instance are needed, along with the server key.

```
_topics = {
    'volttron_to_matlab': 'matlab/to_matlab/1',
    'matlab_to_volttron': 'matlab/to_volttron/1'
}

# The parameters dictionary is used to populate the agent's
# remote vip address.
_params = {
    # The root of the address.
    # Note:
    # 1. volttron instance should be configured to use tcp. use command vcfg
    # to configure
    'vip_address': 'tcp://192.168.56.101',
    'port': 22916,

    # public and secret key for the standalone_matlab agent.
    # These can be created using the command: volttron-ctl auth keypair
    # public key should also be added to the volttron instance auth
    # configuration to enable standalone agent access to volttron instance. Use
    # command 'vctl auth add' Provide this agent's public key when prompted
    # for credential.

    'agent_public': 'dpu13XKPvGB3XJNVUusCNn2U0kIWcuyDIP5J8mAgBQ0',
    'agent_secret': 'Hlya-6BvfUot5USdeDHZ8eksDkWgEEHABs1SElmQhMs',

    # Public server key from the remote platform. This can be
    # obtained using the command:
    # volttron-ctl auth serverkey
    'server_key': 'QTIZrRGQ0-b-37AbEYDuMA0l2ETrythM2Vlac0v9CTA'
}
```

(continues on next page)

(continued from previous page)

```
def remote_url():
    return "{vip_address}:{port}?serverkey={server_key}" \
           "&publickey={agent_public}&" \
           "secretkey={agent_secret}".format(**_params)
```

The primary VOLTTRON instance will then need to add the public key from the Standalone Agent. In this example, the topic that the Standalone Agent is listening to is *matlab/to_matlab/1*, and the topic it is publishing to is *matlab/to_volttron/1*.

scriptwrapper.py

Scriptwrapper.py contains the *script_runner* function. The purpose of this function is to take in a string that contains a Python script and command line arguments separated by commas. This string is parsed and passed to the system arguments, which allows the script sent to the function to use the command line arguments. The function then redirects standard output to a *StringIO* file object, and then attempts to execute the script. If there are any errors with the script, the error that is generated is returned to the standalone agent. Otherwise, the file object stores the output from the script, is converted to a string, and is sent to the standalone agent. In this example, the script that is to be run is *testScript.py*.

```
#Script to take in a string, run the program,
#and output the results of the command as a string.

import time
import sys
from io import StringIO

def script_runner(message):
    original = sys.stdout
    # print(message)
    # print(sys.argv)
    sys.argv = message.split(',')
    # print(sys.argv)

    try:
        out = StringIO()
        sys.stdout = out
        exec(open(sys.argv[0]).read())
        sys.stdout = original
        return out.getvalue()
    except Exception as ex:
        out = str(ex)
        sys.stdout = original
        return out
```

Note: The script that is to be run needs to be in the same folder as the agent and the *scriptwrapper.py* script. The *script_runner* function needs to be edited if it is going to call a script at a different location.

testScript.py

This is a very simple test script designed to demonstrate the calling of a MatLab function from within Python. First it initializes the MatLab engine for Python. It then takes in a single command line argument, and passes it to the MatLab function *testPy.m*. If no arguments are sent, it will send 0 to the *testPy.m* function. It then prints the result of the *testPy.m* function. In this case, since standard output is being redirected to a file object, this is how the result is passed from this function to the Standalone Agent.

```
import matlab.engine
import sys

eng = matlab.engine.start_matlab()

if len(sys.argv) == 2:
    result = eng.testPy(float(sys.argv[1]))
else:
    result = eng.testPy(0.0)

print(result)
```

testPy.m

This MatLab function is a very simple example, designed to show a function that takes an argument, and produces an array as the output. The input argument is added to each element in the array, and the entire array is then returned.

```
function out = testPy(z)
x = 1:100
out = x + z
end
```

Setup on Linux

1. Setup and run VOLTTRON from develop branch using instructions [here](#).
2. Configure volttron instance using the `vcfg` command. When prompted for the vip address use `tcp://<ip address of the linux machine>`. This is necessary to enable volttron communication with external processes.

Note: If you are running VOLTTRON from within VirtualBox, it would be good to set one of your adapters as a *Host-only* adapter. This can be done within the VM's settings, under the *Network* section. Once this is done, use this IP for the VIP address.

3. Update the configuration for MatLabAgent_v2 at `<volttron source dir>/example/MatLabAgent_v2/config`.

The configuration file for the MatLab agent has four variables.

1. `script_names`
2. `script_args`
3. `topics_to_matlab`
4. `topics_to_volttron`

An example config file included with the folder.

```
{
  # VOLTTRON config files are JSON with support for python style comments.
  "script_names": ["testScript.py"],
  "script_args": [["20"]],
  "topics_to_matlab": ["matlab/to_matlab/1"],
  "topics_to_volttron": "matlab/to_volttron/"
}
```

To edit the configuration, the format should be as follows:

```
{
  "script_names": ["script1.py", "script2.py", "..."],
  "script_args": [["arg1", "arg2"], ["arg1"], [...]],
  "topics_to_matlab": ["matlab/to_matlab/1", "matlab/to_matlab/2", "..."],
  "topics_to_volttron": "matlab/to_volttron/"
}
```

The config requires that each script name lines up with a set of commandline arguments and a topic. A commandline argument must be included, even if it is not used. The placement of brackets are important, even when only communicating with one standalone agent.

For example, if only one standalone agent is used, and no command line arguments are in place, the config file may look like this.

```
{
  "script_names": ["testScript.py"],
  "script_args": [["0"]],
  "topics_to_matlab": ["matlab/to_matlab/1"],
  "topics_to_volttron": "matlab/to_volttron/"
}
```

4. Install MatLabAgent_v2 and start agent (from volttron root directory)

```
python ./scripts/install-agent.py -s examples/MatLabAgent_v2 --start
```

Note: The MatLabAgent_v2 publishes the command to be run to the message bus only on start or on a configuration update. Once we configure the *standalone_matlab* agent on the Windows machine, we will send a configuration update to the running MatLabAgent_v2. The configuration would contain the topics to which the Standalone Agent is listening to and will be publishing result to.

See also:

The MatLab agent uses the configuration store to dynamically change inputs. More information on the config store and how it used can be found [here](#).

- VOLTTRON Configuration Store
- Agent Configuration Store
- *Agent Configuration Store Interface*

5. Run the below command and make a note of the server key. This is required for configuring the stand alone agent on Windows. (This is run on the linux machine)

```
vctl auth serverkey
```

Setup on Windows

Install pre-requisites

1. Install Python3.6 64-bit from the [Python website](#).
2. Install the MatLab engine from [MathWorks](#).

Warning: The MatLab engine for Python only supports certain version of Python depending on the version of MatLab used. Please check [here](#) to see if the current version of MatLab supports your version of Python.

Note: At this time, you may want to verify that you are able to communicate with your Linux machine across your network. The simplest method would be to open up the command terminal and use `ping <ip of Linux machine>`, and `telnet <ip of Linux machine> <port of volttron instance, default port is 22916>`. Please make sure that the port is opened for outside access.

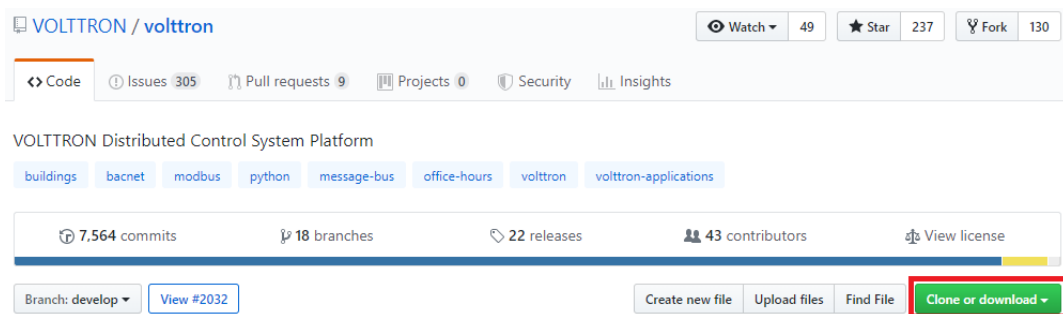
Install Standalone MatLab Agent

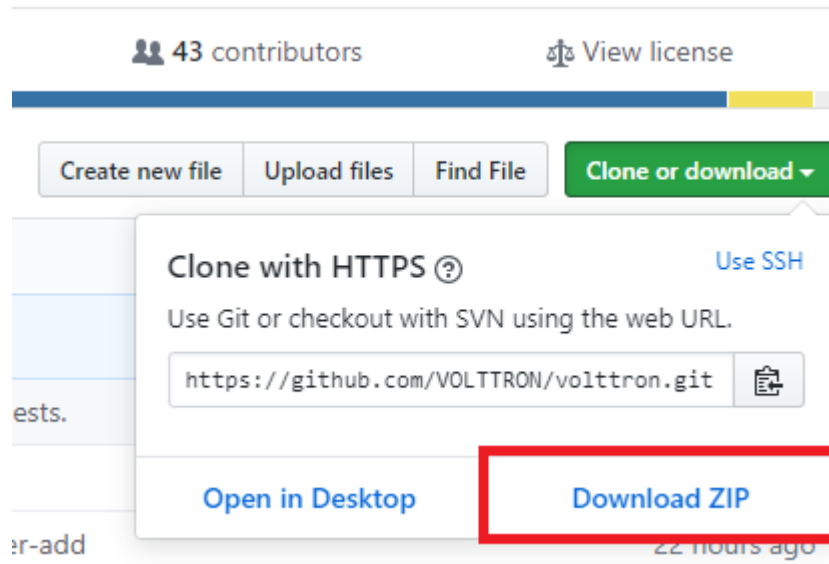
The standalone MatLab agent is designed to be usable in a Windows environment.

Warning: VOLTTTRON is not designed to run in a Windows environment. Outside of cases where it is stated to be usable in a Windows environment, it should be assumed that it will **NOT** function as expected.

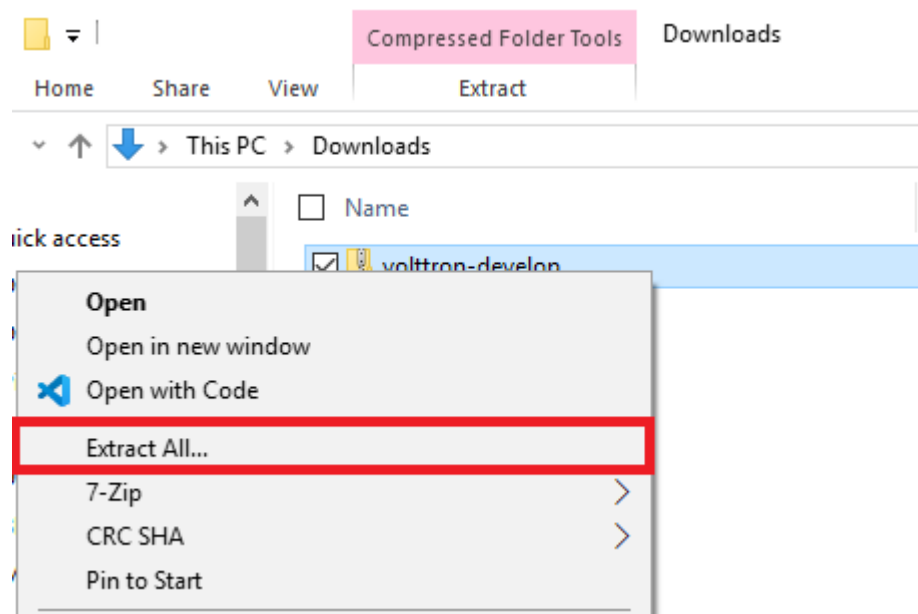
1. Download VOLTTTRON

Download the VOLTTTRON develop repository from Github. Download the zip from [GitHub](#).

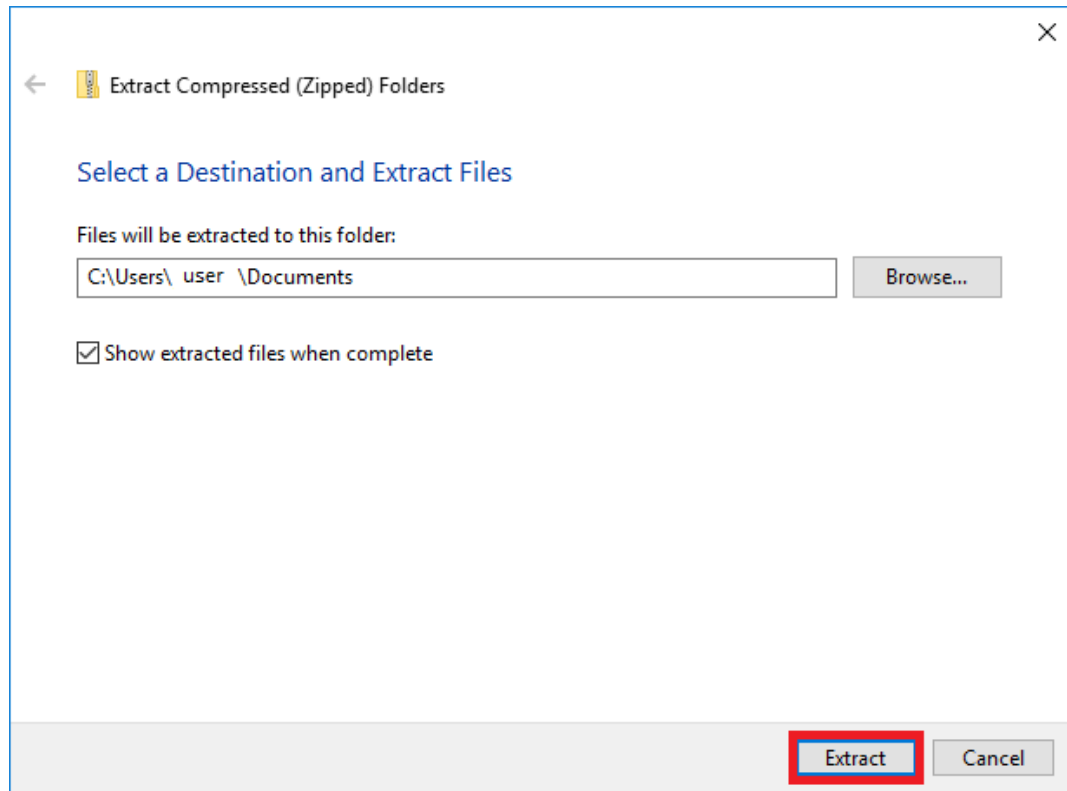




Once the zipped file has been downloaded, go to your *Downloads* folder, right-click on the file, and select *Extract All...*

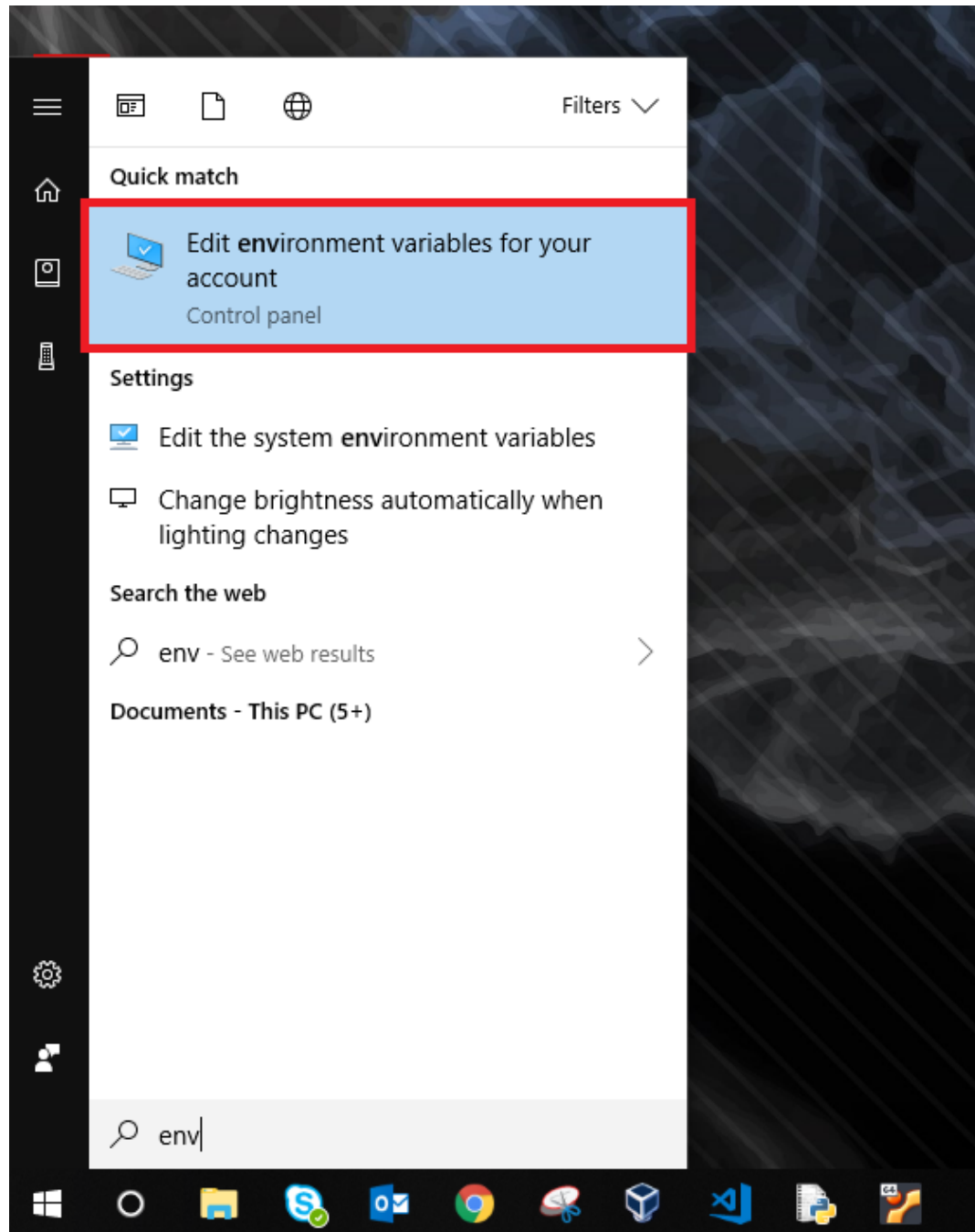


Choose a location for the extracted folder, and select “Extract”

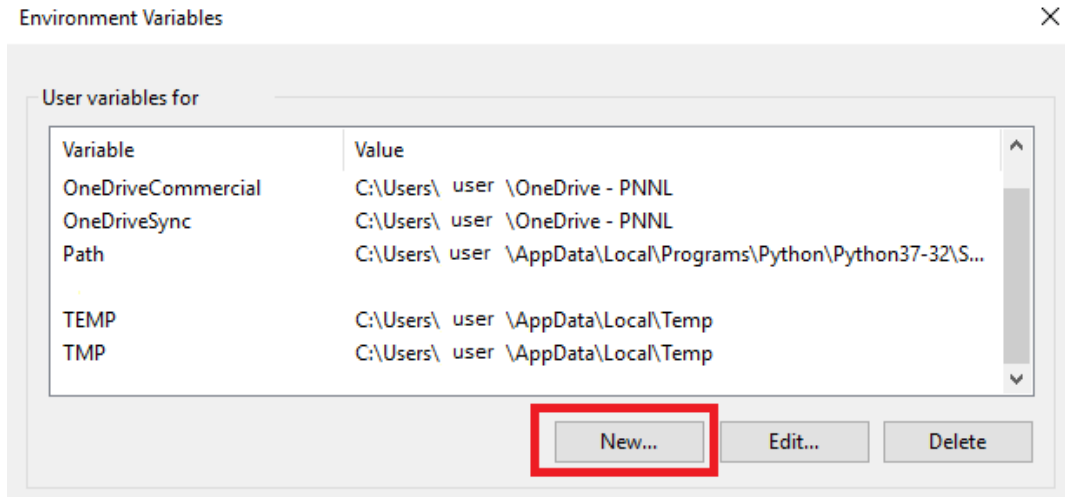


2. Setup the `PYTHONPATH`

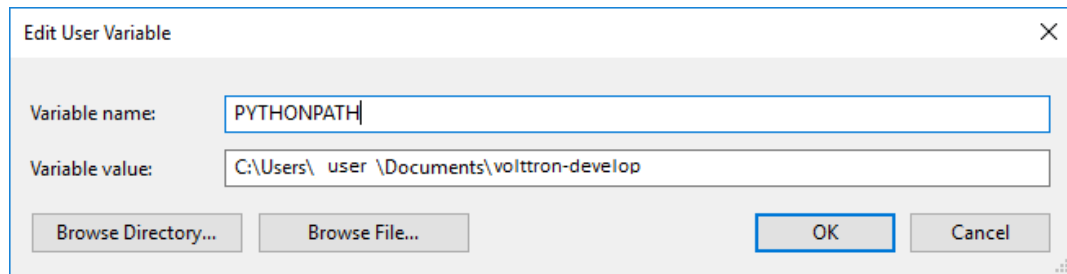
Open the Windows explorer, and navigate to *Edit environment variables for your account*.



Select “New”



For “Variable name” enter: PYTHONPATH For “Variable value” either browse to your VOLTTRON installation, or enter in the path to your VOLTTRON installation.



Select *OK* twice.

3. Set Python version in MatLab

Open your MatLab application. Run the command:

```
pyversion
```

This should print the path to Python2.7. If you have multiple versions of python on your machine and *pyversion* points to a different version of Python, use:

```
pyversion /path/to/python.exe
```

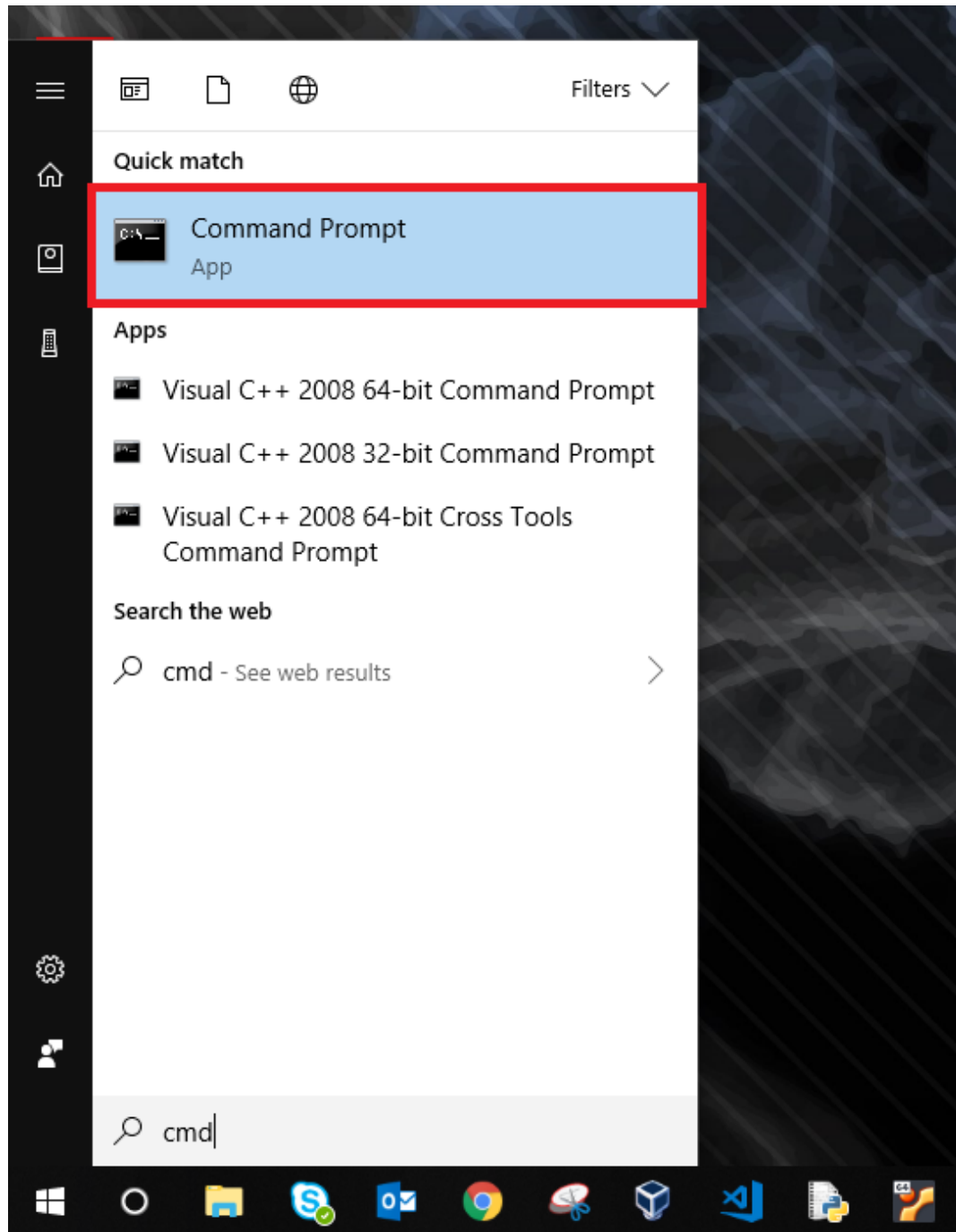
to set the appropriate version of python for your system.

For example, to use python 3.6 with MatLab:

```
pyversion C:\Python36\python.exe
```

4. Set up the environment.

Open up the command prompt



Navigate to your VOLTTRON installation

```
cd \Your\directory\path\to\volttron-develop
```

Use pip to install and setup dependencies.

```
pip install -r examples\StandAloneMatLab\requirements.txt
```

```
pip install -e .
```

Note: If you get the error doing the second step because of an already installed volttron from a different directory, manually delete the *volttron-egg*. link file from your *<python path>\Lib\site-*

packages directory (for example:

```
del C:\\Python27\\lib\\site-packages\\volttron-egg.link
```

and re-run the second command

5. Configure the agent

The configuration settings for the standalone agent are in `setting.py` (located in `volttron-develop\\examples\\StandAloneMatLab\\`)

settings.py

- `volttron_to_matlab` needs to be set to the topic that will send your script and command line arguments to your stand alone agent. This was defined in the [config](#).
- `matlab_to_volttron` needs to be set to the topic that will send your script's output back to your volttron platform. This was defined in [config](#).
- `vip_address` needs to be set to the address of your volttron instance
- `port` needs to be set to the port of your volttron instance
- `server_key` needs to be set to the public server key of your primary volttron platform. This can be obtained from the primary volttron platform using `vctl auth serverkey` (VOLTTRON must be running to use this command.)

It is possible to have multiple standalone agents running. In this case, copy the `StandAloneMatLab` folder, and make the necessary changes to the new `settings.py` file. Unless it is connecting to a separate VOLTTRON instance, you should only need to change the `volttron_to_matlab` setting.

Note: It is recommended that you generate a new “agent_public” and “agent_private” key for your standalone agent. This can be done using the `vctl auth keypair` command on your primary VOLTTRON platform on Linux. If you plan to use multiple standalone agents, they will each need their own keypair.

6. Add standalone agent key to VOLTTRON platform

- Copy the public key from `settings.py` in the `StandAloneMatLab` folder.
- While the primary VOLTTRON platform is running on the linux machine, add the agent public key using the `vctl auth` command on the Linux machine. This will make VOLTTRON platform allow connections from the standalone agent

```
vctl auth add --credentials <standalone agent public key>
```

7. Run standalone agent

At this point, the agent is ready to run. To use the agent, navigate to the example folder and use `python` to start the agent. The agent will then wait for a message to be published to the selected topic by the MatLab agent.

```
cd examples\\StandAloneMatLab\\  
  
python standalone_matlab.py
```

The output should be similar to this:

```
2019-08-01 10:42:47,592 volttron.platform.vip.agent.core DEBUG: identity:↵
↵standalone_matlab
2019-08-01 10:42:47,592 volttron.platform.vip.agent.core DEBUG: agent_
↵uuid: None
2019-08-01 10:42:47,594 volttron.platform.vip.agent.core DEBUG:↵
↵serverkey: None
2019-08-01 10:42:47,596 volttron.platform.vip.agent.core DEBUG: AGENT↵
↵RUNNING on ZMQ Core standalone_matlab
2019-08-01 10:42:47,598 volttron.platform.vip.zmq_connection DEBUG: ZMQ↵
↵connection standalone_matlab
2019-08-01 10:42:47,634 volttron.platform.vip.agent.core INFO: Connected↵
↵to platform: router: ebae9efa-5e8f-49e3-95a0-2020ddff9e8a version: 1.0↵
↵identity: standalone_matlab
2019-08-01 10:42:47,634 volttron.platform.vip.agent.core DEBUG: Running↵
↵onstart methods.
```

Note: If you have Python3 as your default Python run the command `python -2 standalone_matlab.py`

8. On the Linux machine configure the Matlab Agent to publish commands to the topic standalone agent is listening to. To load a new configuration or to change the current configuration enter

```
vctl config store <agent vip identity> config <path\to\configfile>
```

Whenever there is a change in the configuration in the config store, or whenever the agent starts, the MatLab Agent sends the configured command to the topic configured. As long as the standalone agent has been started and is listening to the appropriate topic, the output in the log should look similar to this:

```
2019-08-01 10:43:18,925 (matlab_agentV2agent-0.3 3539) matlab_agentV2.agent↵
↵DEBUG: Configuring Agent
2019-08-01 10:43:18,926 (matlab_agentV2agent-0.3 3539) matlab_agentV2.agent↵
↵DEBUG: Publishing on: matlab/to_matlab/1
2019-08-01 10:43:18,926 (matlab_agentV2agent-0.3 3539) matlab_agentV2.agent↵
↵DEBUG: Sending message: testScript2.py,20
2019-08-01 10:43:18,926 (matlab_agentV2agent-0.3 3539) matlab_agentV2.agent↵
↵DEBUG: Agent Configured!
2019-08-01 10:43:18,979 (matlab_agentV2agent-0.3 3539) matlab_agentV2.agent↵
↵INFO: Agent: matlab/to_volttron/1
Message:
'20'
```

Once the matlab agent publishes the message (in the above case, “testScript2.py,20”) on the windows command prompt running the standalone agent, you should see the message that was received by the standalone agent.

```
2019-08-01 10:42:47,671 volttron.platform.vip.agent.subsystems.configstore↵
↵DEBUG: Processing callbacks for affected files: {}
The Message is: testScript2.py,20
```

Note: If MatLabAgent_v2 has been installed and started, and you have not started the *standalone_matlab agent*, you will need to either restart the `matlab_agentV2`, or make a change to the configuration in the config store to send command to the topic standalone agent is actively listening to.

Node Red Example

Node Red is a visual programming language wherein users connect small units of functionality “nodes” to create “flows”.

There are two example nodes that allow communication between Node-Red and VOLTTRON. One node reads subscribes to messages on the VOLTTRON message bus and the other publishes to it.

Dependencies

The example nodes depend on *python-shell* to be installed and available to the Node Red environment.

Installation

Copy all files from *volttron/examples/NodeRed* to your *~/.node-red/nodes* directory. *~/.node-red* is the default directory for Node Red files. If you have set a different directory use that instead.

Set the variables at the beginning of the *volttron.js* file to be a valid VOLTTRON environment, VOLTTRON home, and Python PATH.

Valid CURVE keys need to be added to the *settings.py* file. If they are generated with the *vctl auth keypair* command then the public key should be added to VOLTTRON’s authorization file with the following:

```
$ vctl auth add
```

The serverkey can be found with:

```
$ vctl auth serverkey
```

Usage

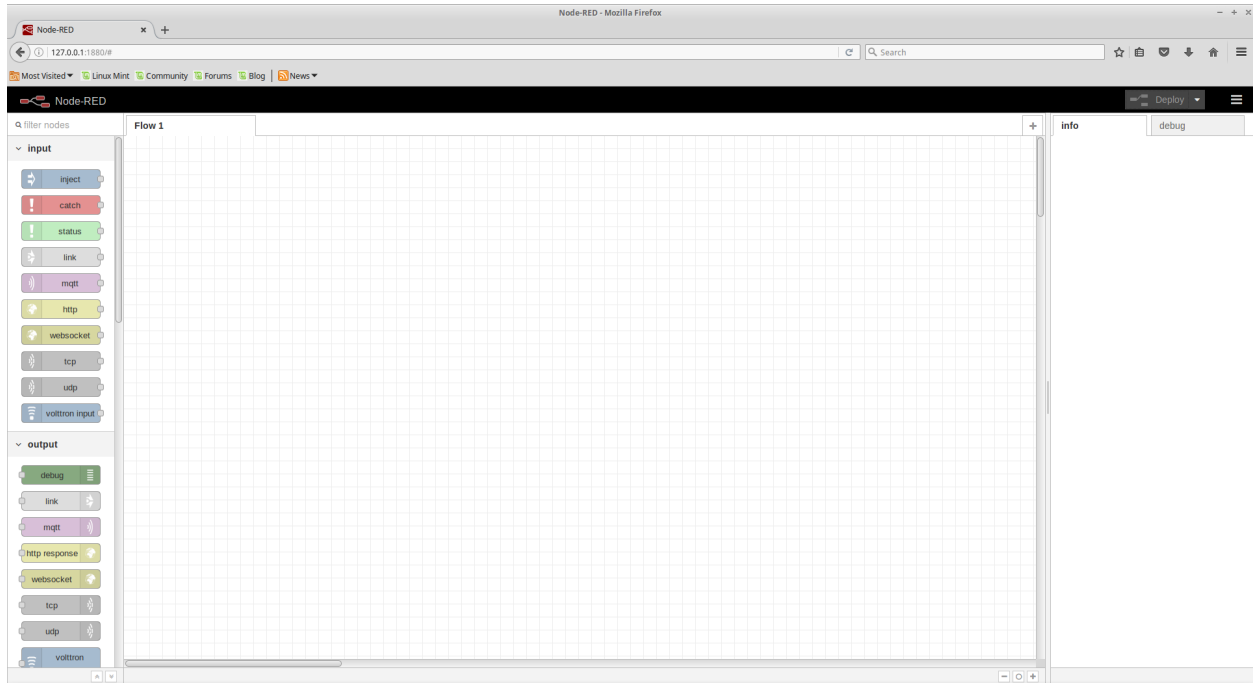
Start VOLTTRON and Node Red.

```
$ node-red

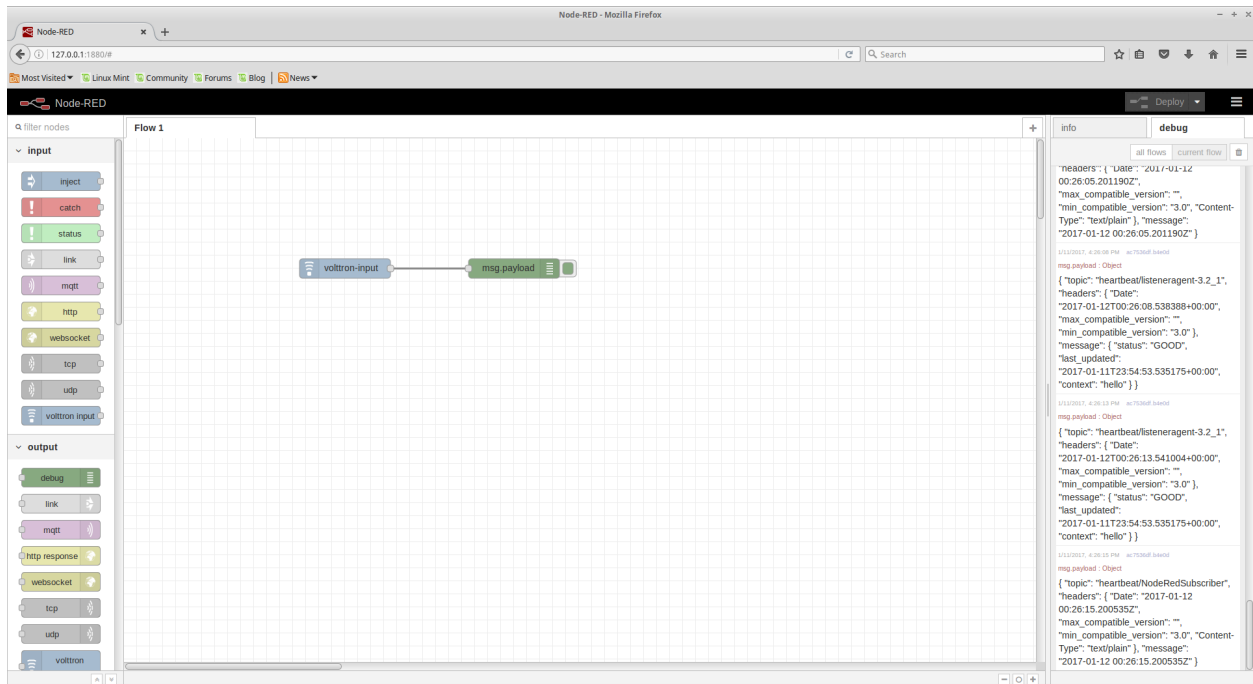
Welcome to Node-RED
=====

11 Jan 15:26:49 - [info] Node-RED version: v0.14.4
11 Jan 15:26:49 - [info] Node.js version: v0.10.25
11 Jan 15:26:49 - [info] Linux 3.16.0-38-generic x64 LE
11 Jan 15:26:49 - [info] Loading palette nodes
11 Jan 15:26:49 - [warn] -----
11 Jan 15:26:49 - [warn] [rpi-gpio] Info : Ignoring Raspberry Pi specific node
11 Jan 15:26:49 - [warn] -----
11 Jan 15:26:49 - [info] Settings file   : /home/volttron/.node-red/settings.js
11 Jan 15:26:49 - [info] User directory  : /home/volttron/.node-red
11 Jan 15:26:49 - [info] Flows file      : /home/volttron/.node-red/flows_volttron.json
11 Jan 15:26:49 - [info] Server now running at http://127.0.0.1:1880/
11 Jan 15:26:49 - [info] Starting flows
11 Jan 15:26:49 - [info] Started flows
```

The output from the Node Red command indicates the address of its web interface. Nodes available for use are in the left sidebar.



We can now use the VOLTTRON nodes to read from and write to VOLTTRON.



Scheduler Example Agent

The Scheduler Example Agent demonstrates how to use the scheduling feature of the :ref`Actuator Agent <Actuator-Agent>` as well as how to send a command. This agent publishes a request for a reservation on a (fake) device then takes an action when it's scheduled time appears. The ActuatorAgent must be running to exercise this example.

Note: Since there is no actual device, an error is produced when the agent attempts to take its action.

```
def publish_schedule(self):
    '''Periodically publish a schedule request'''
    headers = {
        'AgentID': agent_id,
        'type': 'NEW_SCHEDULE',
        'requesterID': agent_id, #The name of the requesting agent.
        'taskID': agent_id + "-ExampleTask", #The desired task ID for this task. It_
        ↪must be unique among all other scheduled tasks.
        'priority': 'LOW', #The desired task priority, must be 'HIGH', 'LOW', or 'LOW_
        ↪PREEMPT'
    }

    start = str(datetime.datetime.now())
    end = str(datetime.datetime.now() + datetime.timedelta(minutes=1))

    msg = [
        ['campus/building/unit', start, end]
    ]
    self.vip.pubsub.publish(
        'pubsub', topics.ACTUATOR_SCHEDULE_REQUEST, headers, msg)
```

The agent listens to schedule announcements from the actuator and then issues a command:

```
@PubSub.subscribe('pubsub', topics.ACTUATOR_SCHEDULE_ANNOUNCE(campus='campus',
                                                                building='building', unit='unit'))
def actuate(self, peer, sender, bus, topic, headers, message):
    print ("response:", topic, headers, message)
    if headers[headers_mod.REQUESTER_ID] != agent_id:
        return
    '''Match the announce for our fake device with our ID
    Then take an action. Note, this command will fail since there is no
    actual device'''
    headers = {
        'requesterID': agent_id,
    }
    self.vip.pubsub.publish(
        'pubsub', topics.ACTUATOR_SET(campus='campus',
                                     building='building', unit='unit',
                                     point='point'),
        headers, 0.0)
```

Simple Web Agent Walk-through

A simple web enabled agent that will hook up with a VOLTTRON message bus and allow interaction between it via HTTP. This example agent shows a simple file serving agent, a JSON-RPC based call, and a websocket based connection mechanism.

Starting VOLTTRON Platform

Note: Activate the environment first *active the environment*

In order to start the simple web agent, we need to bind the VOLTTRON instance to the a web server. We need to specify the address and the port for the web server. For example, if we want to bind the *localhost:8080* as the web server we start the VOLTTRON platform as follows:

```
./start-volttron --bind-web-address http://127.0.0.1:8080
```

Once the platform is started, we are ready to run the Simple Web Agent.

Running Simple Web Agent

Note: The following assumes the shell is located at the VOLTTRON_ROOT.

Copy the following into your shell (save it to a file for executing it again later):

```
python scripts/install-agent.py \  
    --agent-source examples/SimpleWebAgent \  
    --tag simpleWebAgent \  
    --vip-identity webagent \  
    --force \  
    --start
```

This will create a web server on `http://localhost:8080`. The *index.html* file under *simpleweb/webroot/simpleweb/* can be any HTML page which binds to the VOLTTRON message bus .This provides a simple example of providing a web endpoint in VOLTTRON.

Path based registration examples

- Files will need to be in *webroot/simpleweb* in order for them to be browsed from `http://localhost:8080/simpleweb/index.html`
- Filename is required as we don't currently auto-redirect to any default pages as shown in `self.vip.web.register_path("/simpleweb", os.path.join(WEBROOT))`

The following two examples show the way to call either a JSON-RPC (default) endpoint and one that returns a different content-type. With the JSON-RPC example from volttron central we only allow post requests, however this is not required.

- Endpoint will be available at `http://localhost:8080/simple/text` `self.vip.web.register_endpoint("/simple/text", self.text)`
- Endpoint will be available at `http://localhost:8080/simple/jsonrpc` `self.vip.web.register_endpoint("/simpleweb/jsonrpc", self.rpcendpoint)`
- `text/html` content type specified so the browser can act appropriately like `[("Content-Type", "text/html")]`
- The default response is `application/json` so our endpoint returns appropriately with a JSON based response.

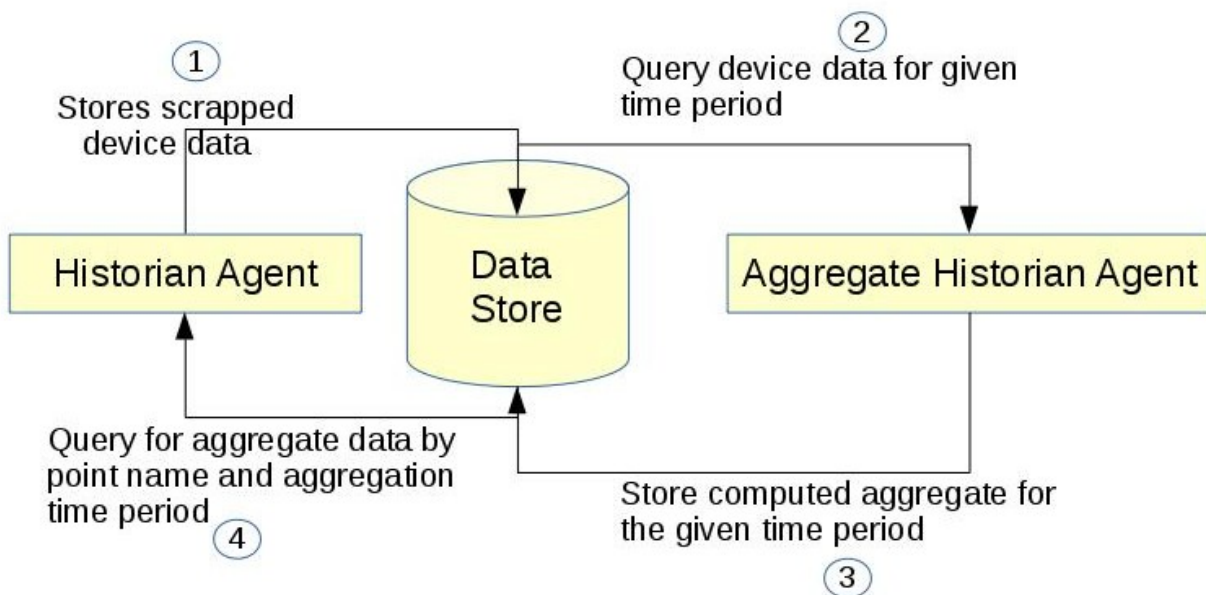
Agent Specifications

Documents included below are intended to provide a specification to classes of agents which include a base class in the VOLTRON repository and have a well defined set of functions and services.

Aggregate Historian

Description

An aggregate historian computes aggregates of data stored in a given volttron historian's data store. It runs periodically to compute aggregate data and store it in new tables/collections in the historian's data store. Each regular historian ([BaseHistorian](#)) needs a corresponding aggregate historian to compute and store aggregates of the data collected by the regular historian.



Software Interfaces

Data Collection - Data store that the aggregate historian uses as input source needs to be up. Access to it should be provided using an account that has create, read, and write privileges. For example, a `MongoAggregateHistorian` needs to be able to connect to the mongodb used by `MongoHistorian` using an account that has read and write access to the db used by the `MongoHistorian`.

Data retrieval Aggregate Historian Agent does not provide api for retrieving the aggregate data collected. Use Historian agent's query interface. Historian's query api will be modified as below

1. `topic_name` can now be a list of topic names or a single topic
2. Two near optional parameters have been added to the query api - `agg_type` (aggregation type), `agg_period` (aggregation time period). Both these parameters are mandatory for query aggregate data.
3. New api to get the list of aggregate topics available for querying

User Interfaces

Aggregation agent requires user to configure the following details as part of the agent configuration file

1. Connection details for historian's data store (same as historian agent configuration)
2. **List of aggregation groups where each group contains:**
 1. Aggregation period - integer followed by m/h/d/w/M (minutes, hours, days, weeks or months)
 2. Boolean parameter to indicate if aggregation periods should align to calendar times
 3. Optional collection start time in utc. If not provided, aggregation collection will start from current time
 4. List of aggregation points with topic name, type of aggregation (sum, avg, etc.), and minimum number of records that should be available for the aggregate to be computed
 5. Topic name can be specified either as a list of specific topic names (topic_names=[topic1, topic2]) or a regular expression pattern (topic_name_pattern="Building1/device_*/Zone*temperature")
 6. When aggregation is done for a single topic then name of topic will be used for the computed aggregation as well. You could optionally provide a unique aggregation_topic_name
 7. When topic_name_pattern or multiple topics are specified a unique aggregate topic name should be specified for the collected aggregate. Users can query for the collected aggregate data using this aggregate topic name.
 8. User should be able to configure multiple aggregations done with the same time period/time interval and these should be time synchronized.

Functional Capabilities

1. Should run periodically to compute aggregate data.
2. Same instance of the agent should be able to collect data at more than one time interval
3. For each configured time period/interval agent should be able to collect different type of aggregation for different topics/points
4. Support aggregation over multiple topics/points
5. Agent should be able to handle and normalize different time units such as minutes, hours, days, weeks and months
6. Agent should be able to compute aggregate both based on wall clock based time intervals and calendar based time interval. For example, agent should be able to calculate daily average based on 12.00AM to 11.59PM of a calendar day or between current time and the same time the previous day.
7. Data should be stored in such a way that users can easily retrieve multiple aggregate topics data within a given time interval

Data Structure

Collected aggregate data should be stored in the historian data store into new collection or tables and should be accessible by historian agent's query interface. Users should easily be able to query aggregate data of multiple points for which data is time synchronized.

Use Cases

Collect monthly average of multiple topic using data from MongoDBHistorian

1. Create a configuration file with connection details from Mongo Historian configuration file and add additional aggregation specific configuration

```
{
  # configuration from mongo historian - START
  "connection": {
    "type": "mongodb",
    "params": {
      "host": "localhost",
      "port": 27017,
      "database": "mongo_test",
      "user": "test",
      "passwd": "test"
    }
  },
  # configuration from mongo historian - START
  "aggregations": [
    # list of aggregation groups each with unique aggregation_period and
    # list of points that needs to be collected
    {
      "aggregation_period": "1M",
      "use_calendar_time_periods": true,
      "utc_collection_start_time": "2016-03-01T01:15:01.000000",
      "points": [
        {
          "topic_names": ["Building/device/point1", "Building/device/point2"],
          "aggregation_topic_name": "building/device/point1_2/month_sum",
          "aggregation_type": "avg",
          "min_count": 2
        }
      ]
    }
  ]
}
```

In the above example configuration, here is what each field under “aggregations” represent

- **aggregation_period**: can be minutes(m), hours(h), weeks(w), or months(M)
- **use_calendar_time_periods**: true or false - Should aggregation period align to calendar time periods. Default False. Exam
 - if “aggregation_period”:”1h” and “use_calendar_time_periods”: false, example periods: 10.15-11.15, 11.15-12.15, 12.15-13.15 etc.
 - if “aggregation_period”:”1h” and “use_calendar_time_periods”: true, example periods: 10.00-11.00, 11.00-12.00, 12.00-13.00 etc.
 - if “aggregation_period”:”1M” and “use_calendar_time_periods”: true, aggregation would be computed from the first day of the month to last day of the month
 - if “aggregation_period”:”1M” and “use_calendar_time_periods”: false, aggregation would be computed with a 30 day interval based on aggregation collection start time
- **utc_collection_start_time**: The time from which aggregation computation should start. If not provided this would default to current time.

- **points:** List of points, its aggregation type and min_count **topic_names:** List of topic_names across which aggregation should be computed. **aggregation_topic_name:** Unique name given for this aggregate. Optional if aggregation is for a single topic. **aggregation_type:** Type of aggregation to be done. Please see *Constraints and Limitations*

min_count: Optional. Minimum number of records that should exist within the configured time period for a aggregation to be computed.

2. install and starts the aggregate historian using the above configuration
3. Query aggregate data: Query using historian's query api by passing two additional parameters - agg_type and agg_period

```
result1 = query_agent.vip.rpc.call('platform.historian',
                                   'query',
                                   topic='building/device/point1_2/month_sum',
                                   agg_type='avg',
                                   agg_period='1M',
                                   count=20,
                                   order="FIRST_TO_LAST").get(10)
```

Collect weekly average(sunday to saturday) of single topic using data from MongoDBHistorian

1. Create a configuration file with connection details from Mongo Historian configuration file and add additional aggregation

- **aggregation_period:** "1w",
- **topic_names:** ["Building/device/point1"], #topic for which you want to compute aggregation
- **aggregation_topic_name** need not be provided

2. install and starts the aggregate historian using the above configuration
3. Query aggregate data: Query using historian's query api by passing two additional parameters - agg_type and agg_period. topic_name will be the same as the point name for which aggregation is collected

```
result1 = query_agent.vip.rpc.call('platform.historian',
                                   'query',
                                   topic='Building/device/point1',
                                   agg_type='avg',
                                   agg_period='1w',
                                   count=20,
                                   order="FIRST_TO_LAST").get(10)
```

Collect hourly average for multiple topics based on topic_name pattern

1. Create a configuration file with connection details from Mongo Historian configuration file and add additional aggregation

- **aggregation_period:** "1h",
- Insetead of topic_names provide **topic_name_pattern**. For example, **"topic_name_pattern": "Building1/device_a*/point1"**
- **aggregation_topic_name** provide a unique aggregation topic name

2. install and starts the aggregate historian using the above configuration

3. Query aggregate data: Query using historian's query api by passing two additional parameters - `agg_type` and `agg_period`. `topic_name` will be the same as the point name for which aggregation is collected

```
result1 = query_agent.vip.rpc.call('platform.historian',
                                   'query',
                                   topic="unique aggregation_topic_name provided in_
↪configuration",
                                   agg_type='avg',
                                   agg_period='1h',
                                   count=20,
                                   order="FIRST_TO_LAST").get(10)
```

Collect 7 day average of two topics and time synchronize them for easy comparison

1. Create a configuration file with connection details from Mongo Historian configuration file and add additional aggregation specific configuration. The configuration file should be similar to the below example

```
{
  # configuration from mongo historian - START
  "connection": {
    "type": "mongodb",
    "params": {
      "host": "localhost",
      "port": 27017,
      "database": "mongo_test",
      "user": "test",
      "passwd": "test"
    }
  },
  # configuration from mongo historian - START
  "aggregations": [
    # list of aggregation groups each with unique aggregation_period and
    # list of points that needs to be collected
    {
      "aggregation_period": "1w",
      "use_calendar_time_periods": false, #compute for last 7 days, then the next_
↪and so on..
      "points": [
        {
          "topic_names": ["Building/device/point1"],
          "aggregation_type": "avg",
          "min_count": 2
        },
        {
          "topic_names": ["Building/device/point2"],
          "aggregation_type": "avg",
          "min_count": 2
        }
      ]
    }
  ]
}
```

2. install and starts the aggregate historian using the above configuration
3. Query aggregate data: Query using historian's query api by passing two additional parameters - `agg_type` and `agg_period`. provide the list of topic names for which aggregate was configured above. Since both the points were

configured within a single “aggregations” array element, their aggregations will be time synchronized

```
result1 = query_agent.vip.rpc.call('platform.historian',
                                   'query',
                                   topic=['Building/device/point1' 'Building/device/
↪point2'],
                                   agg_type='avg',
                                   agg_period='1w',
                                   count=20,
                                   order="FIRST_TO_LAST").get(10)
```

Results will be of the format

```
{'values': [
  ['Building/device/point1', '2016-09-06T23:31:27.679910+00:00', 2],
  ['Building/device/point1', '2016-09-15T23:31:27.679910+00:00', 3],
  ['Building/device/point2', '2016-09-06T23:31:27.679910+00:00', 2],
  ['Building/device/point2', '2016-09-15T23:31:27.679910+00:00', 3]],
'metadata': {}}
```

Query list of aggregate data collected

```
result = query_agent.vip.rpc.call('platform.historian',
                                   'get_aggregate_topics').get(10)
```

The result will be of the format:

```
[(aggregate topic name, aggregation type, aggregation time period, configured list of
↪topics or topic name pattern), ...]
```

This shows the list of aggregation currently being computed periodically

Query list of supported aggregation types

```
result = query_agent.vip.rpc.call(
    AGG_AGENT_VIP,
    'get_supported_aggregations').get(timeout=10)
```

Constraints and Limitations

1. Initial implementation of this agent will not support any data filtering for raw data before computing data aggregation
2. Initial implementation should support all aggregation types directly supported by underlying data store. End user input is needed to figure out what additional aggregation methods are to be supported

MySQL

Name	Description
AVG()	Return the average value of the argument
BIT_AND()	Return bitwise AND
BIT_OR()	Return bitwise OR
BIT_XOR()	Return bitwise XOR
COUNT()	Return a count of the number of rows returned
GROUP_CONCAT()	Return a concatenated string
MAX()	Return the maximum value
MIN()	Return the minimum value
STD()	Return the population standard deviation
STDDEV()	Return the population standard deviation
STDDEV_POP()	Return the population standard deviation
STDDEV_SAMP()	Return the sample standard deviation
SUM()	Return the sum
VAR_POP()	Return the population standard variance
VAR_SAMP()	Return the sample variance
VARIANCE()	Return the population standard variance

SQLite

Name	Description
AVG()	Return the average value of the argument
COUNT()	Return a count of the number of rows returned
GROUP_CONCAT()	Return a concatenated string
MAX()	Return the maximum value
MIN()	Return the minimum value
SUM()	Return sum of all non-NULL values in the group. If there are no non-NULL input rows then returns NULL .
TOTAL()	Return sum of all non-NULL values in the group.If there are no non-NULL input rows returns 0.0

MongoDB

Name	Description
SUM	Returns a sum of numerical values. Ignores non-numeric values
AVG	Returns a average of numerical values. Ignores non-numeric values
MAX	Returns the highest expression value for each group.
MIN	Returns the lowest expression value for each group.
FIRST	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
LAST	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
PUSH	Returns an array of expression values for each group
AD-DTOSET	Returns an array of unique expression values for each group. Order of the array elements is undefined.
STDDEV-VPOP	Returns the population standard deviation of the input values
STDDEV-VSAMP	Returns the sample standard deviation of the input values

Tagging Service

Description

Tagging service provides VOLTTRON users the ability to add semantic tags to different topics so that topic can be queried by tags instead of specific topic name or topic name pattern.

Taxonomy

VOLLTTRON will use tags from [Project Haystack](#). Tags defined in haystack will be imported into VOLTTRON and grouped by categories to tag topics and topic name prefix.

Dependency

Once data in VOLTTRON has been tagged, users will be able to query topics based on tags and use the resultant topics to query the historian

Features

1. User should be able to tag individual components of a topic such as campus, building, device, point etc.
2. Using the tagging service users should only be able to add tags already defined in the volttron tagging schema. New tags should be explicitly added to the tagging schema before it can be used to tag topics or topic prefix
3. Users should be able batch process and tag multiple topic names or topic prefix using a template. At the end of this, users should be notified about the list of topics that did not confirm to the template. This will help users to individually add or edit tags for those specific topics
4. When users query for topics based on a tag, the results would correspond to the current metadata values. It is up to the calling agent/application to periodically query for latest updates if needed.
5. Users should be able query based on tags on a specific topic or its topic prefix/parents
6. Allow for count and skip parameters in queries to restrict count and allow pagination

API

1. Get the list of tag categories available

rpc call to tagging service method **'get_categories'** with optional parameters:

1. **include_description** - set to True to return available description for each category. Default = False
2. **skip** - number of categories to skip. this parameter along with count can be used for paginating results
3. **count** - limit the total number of tag categories returned to given count
4. **order** - ASCENDING or DESCENDING. By default, it will be sorted in ascending order

2. Get the list of tags for a specific category

rpc call to tagging service method **'get_tags_by_category'** with parameter:

1. **category** - <category name>

and optional parameters:

2. **include_kind** - indicate if result should include the kind/data type for tags returned. Defaults to False
3. **include_description** - indicate if result should include available description for tags returned. Defaults to False
4. **skip** - number of tags to skip. this parameter along with count can be used for paginating results
5. **count** - limit the total number of tags returned to given count
6. **order** - ASCENDING or DESCENDING. By default, it will be sorted in ascending order

3. Get the list of tags for a topic_name or topic_name_prefix

rpc call to tagging service method **get_tags_by_topic**

with parameter

1. **topic_prefix** - topic name or topic name prefix

and optional parameters:

2. **include_kind** - indicate if result should include the kind/data type for tags returned. Defaults to False
3. **include_description** - indicate if result should include available description for tags returned. Defaults to False
4. **skip** - number of tags to skip. this parameter along with count can be used for paginating results
5. **count** - limit the total number of tags returned to given count
6. **order** - ASCENDING or DESCENDING. By default, it will be sorted in ascending order

4. Find topic names by tags

rpc call to tagging service method **'get_topics_by_tags'** with the one or more of the following parameters

1. **and_condition** - dictionary of tag and its corresponding values that should be matched using equality operator or a list of tags that should exist/be true. Tag conditions are combined with AND condition. Only topics that match all the tags in the list would be returned
2. **or_condition** - dictionary of tag and its corresponding values that should be matched using equality operator or a list tags that should exist/be true. Tag conditions are combined with OR condition. Topics that match any of the tags in the list would be returned. If both **and_condition** and **or_condition** are provided then they are combined using AND operator.
3. **condition** - conditional statement to be used for matching tags. If this parameter is provided the above two parameters are ignored. The value for this parameter should be an expression that contains one or more query conditions combined together with an "AND" or "OR". Query conditions can be grouped together using parenthesis. Each condition in the expression should conform to one of the following format:

1. <tag name/ parent.tag_name> <binary_operator> <value>

2. <tag name/ parent.tag_name>
3. <tag name/ parent.tag_name> LIKE <regular expression within single quotes
4. the word NOT can be prefixed before any of the above three to negate the condition.
5. expressions can be grouped with parenthesis.

For example

```
condition="tag1 = 1 and not (tag2 < ' ' and tag2 > ' ') and tag3_
↪and NOT tag4 LIKE '^a.*b$'"
condition="NOT (tag5='US' OR tag5='UK') AND NOT tag3 AND NOT_
↪(tag4 LIKE 'a.*') "
condition="campusRef.geoPostalCode='20500' and equip and boiler"
```

6. **skip** - number of topics to skip. this parameter along with count can be used for paginating results
7. **count** - limit the total number of tag topics returned to given count
8. **order** - ASCENDING or DESCENDING. By default, it will be sorted in ascending order

5. Query data based on tags

Use above api to get topics by tags and then use the result to query historian's query api.

6. Add tags to specific topic name or topic name prefix

rpc call to tagging service method **'add_topic_tags'** with parameters:

1. **topic_prefix** - topic name or topic name prefix
2. **tags** - {<valid tag>:value, <valid_tag>: value,... }
3. **update_version** - True/False. Default to False. If set to True and if any of the tags update an existing tag value the older value would be preserved as part of tag version history. **NOTE:** This is a placeholder. Current version does not support versioning.

7. Add tags to multiple topics

rpc call to tagging service method **'add_tags'** with parameters:

1. **tags** - dictionary object containing the topic and the tag details. format:

```
<topic_name or prefix or topic_name pattern>: {<valid tag>:<value>, ... }, ... }
```

2. **update_version** - True/False. Default to False. If set to True and if any of the tags update an existing tag value the older value would be preserved as part of tag version history

Use case examples

1. Loading news tags for an existing VOLTTRON instance

Current topic names:

```

/campus1/building1/deviceA1/point1
/campus1/building1/deviceA1/point2
/campus1/building1/deviceA1/point3
/campus1/building1/deviceA2/point1
/campus1/building1/deviceA2/point2
/campus1/building1/deviceA2/point3
/campus1/building1/deviceB1/point1
/campus1/building1/deviceB1/point2
/campus1/building1/deviceB2/point1
/campus1/building1/deviceB1/point2

```

Step 1:

Create a python dictionary object contains topic name pattern and its corresponding tag/value pair. Use topic pattern names to fill out tags that can be applied to more than one topic or topic prefix. Use specific topic name and topic prefix for tags that apply only to a single entity. For example:

```

{
# tags specific to building1
'/campus1/building1':
    {
        'site': true,
        'dis': ": 'some building description'",
        'yearBuilt': 2015,
        'area': '24000sqft'
    },
# tags that apply to all device of a specific type
'/campus1/building1/deviceA*':
    {
        'dis': "building1 chilled water system - CHW",
        'equip': true,
        'campusRef': 'campus1',
        'siteRef': 'campus1/building1',
        'chilled': true,
        'water' : true,
        'secondaryLoop': true
    }
# tags that apply to point1 of all device of a specific type
'/campus1/building1/deviceA*/point1':
    {
        'dis': "building1 chilled water system - point1",
        'point': true,
        'kind': 'Bool',
        'campusRef': 'campus1',
        'siteRef': 'campus1/building1'
    }
# tags that apply to point2 of all device of a specific type
'/campus1/building1/deviceA*/point2':
    {
        'dis': "building1 chilled water system - point2",
        'point': true,
        'kind': 'Number',
        'campusRef': 'campus1',
        'siteRef': 'campus1/building1'
    }
}

```

(continues on next page)

(continued from previous page)

```
    }
    # tags that apply to point3 of all device of a specific type
    '/campus1/building1/deviceA*/point3':
    {
        'dis': "building1 chilled water system - point3",
        'point': true,
        'kind': 'Number',
        'campusRef': 'campus1',
        'siteRef': 'campus1/building1'
    }
    # tags that apply to all device of a specific type
    '/campus1/building1/deviceB*':
    {
        'dis': "building1 device of type B",
        'equip': true,
        'chilled': true,
        'water' : true,
        'secondaryLoop': true,
        'campusRef': 'campus1',
        'siteRef': 'campus1/building1'
    }
    # tags that apply to point1 of all device of a specific type
    '/campus1/building1/deviceB*/point1':
    {
        'dis': "building1 device B - point1",
        'point': true,
        'kind': 'Bool',
        'campusRef': 'campus1',
        'siteRef': 'campus1/building1',
        'command': true
    }
    # tags that apply to point1 of all device of a specific type
    '/campus1/building1/deviceB*/point2':
    {
        'dis': "building1 device B - point2",
        'point': true,
        'kind': 'Number',
        'campusRef': 'campus1',
        'siteRef': 'campus1/building1'
    }
}
```

Step 2: Create tags using template above

Make an RPC call to the `add_tags` method and pass the python dictionary object

Step 3: Create tags specific to a point or device

Any tags that were not included in step one and needs to be added later can be added using the rpc call to tagging service either the method `'add_topic_tags'` `'add_tags'`

For example:

```
agent.vip.rpc.call(
    'platform.tagging',
    'add_topic_tags',
    topic_prefix='/campus1/building1/deviceA1',
    tags={'tag1':'value'})
```

```
agent.vip.rpc.call(
    'platform.tagging',
    'add_topic_tags',
    tags={
        '/campus1/building1/deviceA2':
            {'tag1':'value'},
        '/campus1/building1/deviceA2/point1':
            {'equipRef':'campus1/building1/deviceA2'}
    }
)
```

2. Querying based on a topic's tag and it parent's tags

Query - Find all points that has the tag 'command' and belong to a device/unit that has a tag 'chilled'

```
agent.vip.rpc.call(
    'platform.tagging',
    'get_topics_by_tags',
    condition='temperature and equip.chilled')
```

In the above code block 'command' and 'chilled' are the tag names that would be searched, but since the tag 'chilled' is prefixed with 'equip.' the tag in a parent topic

The above query would match the topic '/campus1/building1/deviceB1/point1' if tags in the system are as follows

'/campus1/building1/deviceB1/point1' tags:

```
{
  'dis': "building1 device B - point1",
  'point': true,
  'kind': 'Bool',
  'campusRef': 'campus1',
  'siteRef': 'campus1/building1',
  'equipRef': 'campus1/building1/deviceB1',
  'command': true
}
```

'/campus1/building1/deviceB1' tags

```
{
  'dis': "building1 device of type B",
  'equip': true,
  'chilled': true,
  'water': true,
  'secondaryLoop': true,
  'campusRef': 'campus1',
  'siteRef': 'campus1/building1'
}
```

Possible future improvements

1. Versioning - When a value of a tag is changed, users should be prompted to verify if this change denotes a new version or a value correction. If this value denotes a new version, then older value of the tag should be preserved in a history/audit store
2. Validation of tag values based on data type
3. Support for units validation and conversions
4. Processing and saving geologic coordinates that can enable users to do geo-spatial queries in databases that support it.

Weather Service

Description

The weather service agent provides API to access current weather data, historical data and weather forecast data. There are several weather data providers, some paid and some free. Weather data providers differ from one another

1. In the kind of features provided - current data, historical data, forecast data
2. The data points returned
3. The naming schema used to represent the data returned
4. Units of data returned
5. Frequency of data updates

The weather service agent has a design similar to historians. There is a single base weather service that defines the api signatures and the ontology of the weather data points. There is one concrete weather service agent for each weather provider. Users can install one or more provider specific agent to access weather data.

The initial implementation is for [NOAA](#) and would support current and forecast data requests. NOAA does not support accessing historical weather data through their api. This agent implements request data caching.

The second implementation is for [darksky.net](#).

Features

Base weather agent features:

1. Caching

The weather service provides basic caching capability so that repeated request for same data can be returned from cache instead of network round trip to the weather data provider. This is also useful to limit the number of request made to the provider as most weather data provider have restrictions on number of requests for developer/free api keys. The size of the cache can be restricted by setting an optional configuration parameter 'max_size_gb'

2. Name mapping

Data points returned by concrete weather agents is mapped to standard names based on [CF standard names table](#) Name mapping is done using a CSV file. See [Configuration](#) section for an example configuration

3. Unit conversion

If data returned from the provider is of the format {"data_point_name":value}, base weather agent can do unit conversions on the value. Both name mapping and unit conversions can be specified as a csv file and

packaged with the concrete implementing agent. This feature is not mandatory. See [Configuration](#) section for an example configuration

Core weather data retrieval features :

1. Retrieve current weather data.
2. Retrieve hourly weather forecast data.
3. Retrieve historical weather data.
4. Periodic polling of current weather data for one or more locations. Users can configure one or more locations in a config file and weather agent will periodically poll for current weather data for the configured locations and publish the results to message bus.

The set of points returned from the above queries depends on the specific weather data provider, however the point names returned are from the standard schema.

Note:

1. Since individual weather data provider can support slightly different sets of features, users are able to query for the list of available features. For example a provider could provide daily weather forecast in addition to the hourly forecast data.

API

1. Get available features

rpc call to weather service method `'get_api_features'`

Parameters - None

Returns - dictionary of api features that can be called for this weather agent.

2. Get current weather data

rpc call to weather service method `'get_current_weather'`

Parameters:

1. **locations** - dictionary containing location details. The format of location accepted differs between different weather providers and even different APIs supported by the same provider For example the location input could be either `{"zipcode":value}` or `{"region":value, "country": value}`.

Returns: List of dictionary objects containing current weather data. The actual data points returned depends on the weather service provider.

3. Get hourly forecast data

rpc call to weather service method `'get_hourly_forecast'`

Parameters:

1. **locations** - dictionary containing location details. The format of location accepted differs between different weather providers and even different APIs supported by the same provider For example the location input could be either `{"zipcode":value}` or `{"region":value, "country": value}`.

optional parameters:

2. **hours** - The number of hours for which forecast data are returned. By default, it is 24 hours.

Returns: List of dictionary objects containing forecast data. If weather data provider returns less than requested number of hours result returned would contain a warning message in addition to the result returned by the provider

4. Get historical weather data

rpc call to weather service method **'get_hourly_historical'**

Parameters:

1. **locations** - dictionary containing location details. For example the location input could be either {"zip-code":value} or {"region":value, "country": value}.
2. **start_date** - start date of requested data
3. **end_date** - end date of requested data

Returns: List of dictionary objects containing historical data.

Note: Based on the weather data provider this api could do multiple calls to the data provider to get the requested data. For example, darksky.net allows history data query by a single date and not a date range.

5. Periodic polling of current weather data

This can be achieved by configuring the locations for which data is requested in the agent's configuration file along with polling interval. Results for each location configured, is published to its corresponding result topic. is no result topic prefix is configured, then results for all locations are posted to the topic weather/poll/current/all. poll_topic_suffixes when provided should be a list of string with the same length as the number of poll_locations. When topic prefix is specified, each location's result is published to weather/poll/current/<poll_topic_suffix for that location> topic_prefix.

Configuration

Example configuration:

```
{
  poll_locations: [
    {"zip": "22212"},
    {"zip": "99353"}
  ],
  poll_topic_suffixes: ["result_22212", "result_99353"],
  poll_interval: 20 #seconds,

  #optional cache arguments
  max_cache_size: ...
}
```

Example configuration for mapping point names returned by weather provider to a standard name and units:

```
Service_Point_Name, Standard_Point_Name, Service_Units, Standard_Units
temperature, air_temperature, fahrenheit, celsius
```

Caching

Weather agent will cache data until the configured size limit is reached (if provided).

1. Current and forecast data:

If current/forecast weather data exists in cache and if the request time is within the update time period of the api (specified by a concrete implementation) then by default cached data would be returned otherwise a new request is made for it. If hours is provided and the amount of cached data records is less than hours, this will also result in a new request.

2. Historical data cache:

Weather api will query the cache for available data for the given time period and fill and missing time period with data from the remote provider.

3. Clearing of cache:

Users can configure the maximum size limit for cache. For each api call, before data is inserted in cache, weather agent will check for this size limit and purge records in this order. - Current data older than update time period - Forecast data older than update time period - History data starting with the oldest cached data

Assumptions

1. User has api key for accessing weather api for a specific weather data provider, if a key is required.
2. Different weather agent might have different requirement for how input locations are specified. For example NOAA expects a station id for querying current weather and requires either a lat/long or gridpoints to query for forecast. weatherbit.io accepts zip code.
3. Not all features might be implemented by a specific weather agent. For example NOAA doesn't make history data available using their weather api.
4. Concrete agents could expose additional api features
5. Optionally, data returned will be based on standard names provided by the CF standard names table (see Ontology). Any points with a name not mapped to a standard name would be returned as is.

1.9 Driver Development

In order for VOLTTTRON agents to gather data from a device or to set device values, agents send requests to the Master Driver Agent to read or set points. The Master Driver Agent then sends these requests on to the appropriate driver for interfacing with that device based on the topic specified in the request and the configuration of the Master Driver. Drivers provide an interface between the device and the master driver by implementing portions of the devices' protocols needed to serve the functions of setting and reading points.

As a demonstration of developing a driver a driver can be made to read and set points in a CSV file. This driver will only differ from a real device driver in terms of the specifics of the protocol.

1.9.1 Create a Driver and Register class

When a new driver configuration is added to the Master Driver, the Master Driver will look for a file or directory in its interfaces directory (services/core/MasterDriverAgent/master_driver/interfaces) that shares the name of the value specified by "driver_type" in the configuration file. For the CSV Driver, create a file named csvdriver.py in that directory.

```

├── master_driver
│   ├── agent.py
│   ├── driver.py
│   ├── __init__.py
│   └── interfaces
│       ├── __init__.py
│       ├── bacnet.py
│       ├── csvdriver.py
│       └── modbus.py
├── socket_lock.py
├── master-driver.agent
└── setup.py

```

Following is an example using the directory type structure:

```

├── master_driver
│   ├── agent.py
│   ├── driver.py
│   ├── __init__.py
│   └── interfaces
│       ├── __init__.py
│       ├── bacnet.py
│       ├── csvdriver.py
│       ├── modbus.py
│       ├── modbus_tk.py
│       ├── __init__.py
│       ├── tests
│       ├── requirements.txt
│       └── README.rst

```

Note: Using this format, the directory must be the name specified by “driver_type” in the configuration file and the *Interface* class must be in the `__init__.py` file in that directory.

This format is ideal for including additional code files as well as requirements files, tests and documentation.

Interface Basics

A complete interface consists of two parts: the interface class and one or more register classes.

Interface Class Skeleton

When the Master Driver processes a driver configuration file it creates an instance of the interface class found in the interface file (such as the one we’ve just created). The interface class is responsible for managing the communication between the Volttron Platform, and the device. Each device has many registers which hold the values Volttron agents are interested in so generally the interface manages reading and writing to and from a device’s registers. At a minimum, the interface class should be configurable, be able to read and write registers, as well as read all registers with a single request. First create the csv interface class boilerplate.

```

class Interface(BasicRevert, BaseInterface):
    def __init__(self, **kwargs):
        super(Interface, self).__init__(**kwargs)

```

(continues on next page)

(continued from previous page)

```

def configure(self, config_dict, registry_config_str):
    pass

def get_point(self, point_name):
    pass

def _set_point(self, point_name, value):
    pass

def _scrape_all(self):
    pass

```

This class should inherit from the BaseInterface and at a minimum implement the configure, get_point, set_point, and scrape_all methods.

Note: In some sense, drivers are sub-agents running under the same process as the Master Driver. They should be instantiated following the agent pattern, so a function to handle configuration and create the Driver object has been included.

Register Class Skeleton

The interface needs some information specifying the communication for each register on the device. For each different type of register a register class should be defined which will help identify individual registers and determine how to communicate with them. Our CSV driver will be fairly basic, with one kind of “register”, which will be a column in a CSV file. Other drivers may require many kinds of registers; for instance, the Modbus protocol driver has registers which store data in byte sized chunks and registers which store individual bits, therefore the Modbus driver has bit and byte registers.

For the CSV driver, create the register class boilerplate:

```

class CsvRegister(BaseRegister):
    def __init__(self, csv_path, read_only, pointName, units, reg_type,
                 default_value=None, description=''):
        super(CsvRegister, self).__init__("byte", read_only, pointName, units,
        ↳description=description)

```

This class should inherit from the BaseRegister. The class should keep register metadata, and depending upon the requirements of the protocol/device, may perform the communication.

The BACNet and Modbus drivers may be used as examples of more specific implementations. For the purpose of this demonstration writing and reading points will be done in the register, however, this may not always be the case (as in the case of the BACNet driver).

Filling out the Interface class

The CSV interface will be writing to and reading from a CSV file, so the device configuration should include a path specifying a CSV file to use as the “device”. The CSV “device: path value is set at the beginning of the agent loop which runs the configure method when the Master Driver starts. Since this Driver is for demonstration, we’ll create the CSV with some default values if the configured path doesn’t exist. The CSV device will consist of 2 columns: “Point Name” specifying the name of the register, and “Point Value”, the current value of the register.

```

_log = logging.getLogger(__name__)

CSV_FIELDNAMES = ["Point Name", "Point Value"]
CSV_DEFAULT = [
    {
        "Point Name": "test1",
        "Point Value": 0
    },
    {
        "Point Name": "test2",
        "Point Value": 1
    },
    {
        "Point Name": "test3",
        "Point Value": "testpoint"
    }
]

type_mapping = {"string": str,
                "int": int,
                "integer": int,
                "float": float,
                "bool": bool,
                "boolean": bool}

class Interface(BasicRevert, BaseInterface):
    def __init__(self, **kwargs):
        super(Interface, self).__init__(**kwargs)
        self.csv_path = None

    def configure(self, config_dict, registry_config_str):
        self.csv_path = config_dict.get("csv_path", "csv_device.csv")
        if not os.path.isfile(self.csv_path):
            _log.info("Creating csv 'device'")
            with open(self.csv_path, "w+") as csv_device:
                writer = DictWriter(csv_device, fieldnames=CSV_FIELDNAMES)
                writer.writeheader()
                writer.writerows(CSV_DEFAULT)
            self.parse_config(registry_config_str)

```

At the end of the configuration method, the Driver parses the registry configuration. The registry configuration is a csv which is used to tell the Driver which register the user wishes to communicate with and includes a few meta-data values about each register, such as whether the register can be written to, if the register value uses a specific measurement unit, etc. After each register entry is parsed from the registry config a register is added to the driver's list of active registers.

```

def parse_config(self, config_dict):
    if config_dict is None:
        return

    for index, regDef in enumerate(config_dict):
        # Skip lines that have no point name yet
        if not regDef.get('Point Name'):
            continue

        read_only = regDef.get('Writable', "").lower() != 'true'
        point_name = regDef.get('Volttron Point Name')
        if not point_name:

```

(continues on next page)

(continued from previous page)

```

        point_name = regDef.get("Point Name")
        if not point_name:
            raise ValueError("Registry config entry {} did not have a point name or_
↪volttron point name".format(
                index))
        description = regDef.get('Notes', '')
        units = regDef.get('Units', None)
        default_value = regDef.get("Default Value", "").strip()
        if not default_value:
            default_value = None
        type_name = regDef.get("Type", 'string')
        reg_type = type_mapping.get(type_name, str)

        register = CsvRegister(
            self.csv_path,
            read_only,
            point_name,
            units,
            reg_type,
            default_value=default_value,
            description=description)

        if default_value is not None:
            self.set_default(point_name, register.value)

        self.insert_register(register)

```

Since the driver's registers will be doing the work of parsing the registers the interface only needs to select the correct register to read from or write to and instruct the register to perform the corresponding unit of work.

```

def get_point(self, point_name):
    register = self.get_register_by_name(point_name)
    return register.get_state()

def _set_point(self, point_name, value):
    register = self.get_register_by_name(point_name)
    if register.read_only:
        raise IOError("Trying to write to a point configured read only: " + point_
↪name)
    register.set_state(value)
    return register.get_state()

def _scrape_all(self):
    result = {}
    read_registers = self.get_registers_by_type("byte", True)
    write_registers = self.get_registers_by_type("byte", False)
    for register in read_registers + write_registers:
        result[register.point_name] = register.get_state()
    return result

```

Writing the Register class

The CSV driver's register class is responsible for parsing the CSV, reading the corresponding rows to return the register's current value and writing updated values into the CSV for the register. On a device which communicates via a protocol such as Modbus the same units of work would be done, but using pymodbus to perform the reads and writes. Here, Python's CSV library will be used as our "protocol implementation".

The Register class determines which file to read based on values passed from the Interface class.

```
class CsvRegister(BaseRegister):
    def __init__(self, csv_path, read_only, pointName, units, reg_type,
                  default_value=None, description=''):
        super(CsvRegister, self).__init__("byte", read_only, pointName, units,
                                          description=description)

        self.csv_path = csv_path
```

To find its value the register will read the CSV file, iterate over each row until a row with the point name the same as the register name at which point it extracts the point value, and returns it. The register should be written to handle problems which may occur, such as no correspondingly named row being present in the CSV file.

```
def get_state(self):
    if os.path.isfile(self.csv_path):
        with open(self.csv_path, "r") as csv_device:
            reader = DictReader(csv_device)
            for point in reader:
                if point.get("Point Name") == self.point_name:
                    point_value = point.get("Point Value")
                    if not point_value:
                        raise RuntimeError("Point {} not set on CSV Device".
→format(self.point_name))
                    else:
                        return point_value
            raise RuntimeError("Point {} not found on CSV Device".format(self.point_name))
    else:
        raise RuntimeError("CSV device at {} does not exist".format(self.csv_path))
```

Likewise to overwrite an existing value, the register will iterate over each row until the point name matches the register name, saving the output as it goes. When it finds the correct row it instead saves the output updated with the new value then continues on. Finally it writes the output back to the csv.

```
def set_state(self, value):
    _log.info("Setting state for {} on CSV Device".format(self.point_name))
    field_names = []
    points = []
    found = False
    with open(self.csv_path, "r") as csv_device:
        reader = DictReader(csv_device)
        field_names = reader.fieldnames
        for point in reader:
            if point["Point Name"] == self.point_name:
                found = True
                point_copy = point
                point_copy["Point Value"] = value
                points.append(point_copy)
            else:
                points.append(point)

    if not found:
        raise RuntimeError("Point {} not found on CSV Device".format(self.point_name))
    else:
        with open(self.csv_path, "w") as csv_device:
            writer = DictWriter(csv_device, fieldnames=field_names)
            writer.writeheader()
            writer.writerows([dict(row) for row in points])
    return self.get_state()
```


At this point we should be able to scrape the CSV device using the Master Driver and set points using the actuator.

Creating Driver Configurations

The configuration files for the CSV driver are very simple, but in general, the device configuration should specify the parameters which the interface requires to communicate with the device and the registry configuration contains rows which correspond to registers and specifies their usage.

Here's the driver configuration for the CSV driver:

```
{
  "driver_config": {"csv_path": "csv_driver.csv"},
  "driver_type": "csvdriver",
  "registry_config": "config://csv_registers.csv",
  "interval": 30,
  "timezone": "UTC"
}
```

Note: The “driver_type” value must match the name of the driver’s python file as this is what the Master Driver will look for when searching for the correct interface.

And here's the registry configuration:

Volttron Point Name	Point Name	Writable
test1	test1	true
test2	test2	true
test3	test3	true

The BACNet and Modbus driver docs and example configurations can be used to compare these configurations to more complex configurations.

1.9.2 Testing your driver

To test the driver’s scrape all functionality, one can install a ListenerAgent and Master Driver with the driver’s configurations, and run them. To do so for the CSV driver using the configurations above: activate the Volttron environment start the platform, tail the platform’s log file, then try the following:

```
python scripts/install-agent.py -s examples/ListenerAgent
python scripts/install-agent.py -s services/core/MasterDriverAgent -c services/core/
↪MasterDriverAgent/master-driver.agent
vctl config store platform.driver devices/<campus>/<building>/csv_driver <path to_
↪driver configuration>
vctl config store platform.driver <registry config path from driver configuration>
↪<path to registry configuration>
```

Note: *vctl config list platform.driver* will list device and registry configurations stored for the master driver and *vctl config delete platform.driver <config in configs list>* can be used to remove a configuration entry - these commands are very useful for debugging

After the Master Driver starts the driver’s output should appear in the logs at regular intervals based on the Master Driver’s configuration.

Here is some sample CSV driver output:

```
2019-11-15 10:32:00,010 (listeneragent-3.3 22996) listener.agent INFO: Peer: pubsub,
↳ Sender: platform.driver:, Bus:
, Topic: devices/pnnl/isbl/csv_driver/all, Headers: {'Date': '2019-11-15T18:32:00.
↳ 001360+00:00', 'TimeStamp':
'2019-11-15T18:32:00.001360+00:00', 'SynchronizedTimeStamp': '2019-11-15T18:32:00.
↳ 000000+00:00',
'min_compatible_version': '3.0', 'max_compatible_version': ''}, Message:
[{'test1': '0', 'test2': '1', 'test3': 'testpoint'},
 {'test1': {'type': 'integer', 'tz': 'UTC', 'units': None},
  'test2': {'type': 'integer', 'tz': 'UTC', 'units': None},
  'test3': {'type': 'integer', 'tz': 'UTC', 'units': None}}]
```

This output is an indication of the basic scrape all functionality working in the Interface class - in our implementation this is also an indication of the basic functionality of the Interface class “get_point” method and Register class “get_state” methods working (although edge cases should still be tested!).

To test the Interface’s “set_point” method and Register’s “set_state” method we’ll need to use the Actuator agent. The following agent code can be used to alternate a point’s value on a schedule using the actuator, as well as perform an action based on a pubsub subscription to a single point:

```
def CsvDriverAgent(config_path, **kwargs):
    """Parses the Agent configuration and returns an instance of
    the agent created using that configuration.

    :param config_path: Path to a configuration file.

    :type config_path: str
    :returns: Csvdriveragent
    :rtype: Csvdriveragent
    """
    _log.debug("Config path: {}".format(config_path))
    try:
        config = utils.load_config(config_path)
    except Exception:
        config = {}

    if not config:
        _log.info("Using Agent defaults for starting configuration.")
    _log.debug("config_dict before init: {}".format(config))
    utils.update_kwargs_with_config(kwargs, config)
    return Csvdriveragent(**kwargs)

class Csvdriveragent (Agent):
    """
    Document agent constructor here.
    """

    def __init__(self, csv_topic="", **kwargs):
        super(Csvdriveragent, self).__init__(**kwargs)
        _log.debug("vip_identity: " + self.core.identity)

        self.agent_id = "csv_actuation_agent"
        self.csv_topic = csv_topic

        self.value = 0
```

(continues on next page)

(continued from previous page)

```

self.default_config = {
    "csv_topic": self.csv_topic
}

# Set a default configuration to ensure that self.configure is called_
↳ immediately to setup
# the agent.
self.vip.config.set_default("config", self.default_config)

# Hook self.configure up to changes to the configuration file "config".
self.vip.config.subscribe(self.configure, actions=["NEW", "UPDATE"], pattern=
↳ "config")

def configure(self, config_name, action, contents):
    """
    Called after the Agent has connected to the message bus. If a configuration_
    ↳ exists at startup
    this will be called before onstart.

    Is called every time the configuration in the store changes.
    """
    config = self.default_config.copy()
    config.update(contents)

    _log.debug("Configuring Agent")
    _log.debug(config)

    self.csv_topic = config.get("csv_topic", "")

    # Unsubscribe from everything.
    self.vip.pubsub.unsubscribe("pubsub", None, None)

    self.vip.pubsub.subscribe(peer='pubsub',
                              prefix="devices/" + self.csv_topic + "/all",
                              callback=self._handle_publish)

def _handle_publish(self, peer, sender, bus, topic, headers, message):
    _log.info("Device {} Publish: {}".format(self.csv_topic, message))

@Core.receiver("onstart")
def onstart(self, sender, **kwargs):
    """
    This is method is called once the Agent has successfully connected to the_
    ↳ platform.
    This is a good place to setup subscriptions if they are not dynamic or
    do any other startup activities that require a connection to the message bus.
    Called after any configurations methods that are called at startup.

    Usually not needed if using the configuration store.
    """
    self.core.periodic(30, self.actuate_point)

def actuate_point(self):
    _now = get_aware_utc_now()
    str_now = format_timestamp(_now)
    _end = _now + td(seconds=10)
    str_end = format_timestamp(_end)

```

(continues on next page)

(continued from previous page)

```

        schedule_request = [[self.csv_topic, str_now, str_end]]
        result = self.vip.rpc.call(
            'platform.actuator', 'request_new_schedule', self.agent_id, 'my_test',
↪ 'HIGH', schedule_request).get(
            timeout=4)
        point_topic = self.csv_topic + "/" + "test1"
        result = self.vip.rpc.call(
            'platform.actuator', 'set_point', self.agent_id, point_topic, self.value).
↪ get(
            timeout=4)
        self.value = 0 if self.value is 1 else 1

    @Core.receiver("onstop")
    def onstop(self, sender, **kwargs):
        """
        This method is called when the Agent is about to shutdown, but before it_
↪ disconnects from
        the message bus.
        """
        pass

def main():
    """Main method called to start the agent."""
    utils.vip_main(CsvDriverAgent,
                   version=__version__)

if __name__ == '__main__':
    # Entry point for script
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        pass

```

While this code runs, since the Actuator is instructing the Interface to set points on the device, the pubsub all publish can be used to check that the values are changing as expected.

1.10 Contributing Code

As an open source project VOLTTRON requires input from the community to keep development focused on new and useful features. To that end we are revising our commit process to hopefully allow more contributors to be apart of the community. The following document outlines the process for source code and documentation to be submitted. There are GUI tools that may make this process easier, however this document will focus on what is required from the command line.

The only requirements for contributing are Git (Linux version control software) and your favorite web browser.

Note: The following guide assumes the user has already created a fork of the core VOLTTRON repository. Please review the [docs](#) if you have not yet created a fork.

The only technical requirements for contributing are Git (version control software) and your favorite web browser.

As a part of VOLTTRON joining the Eclipse community, Eclipse requires that all contributors sign the [Eclipse Contributor agreement](#) before making a pull request.

1.10.1 Reviewing Changes

Okay, we've written a cool new *foo.py* script to service *bar* in our deployment. Let's make sure our code is up-to-snuff.

Code

First, go through the code.

Note: We on the VOLTTRON team would recommend an internal code review - it can be really hard to catch small mistakes, typos, etc. for code you just finished writing.

- Does the code follow best-practices for Python, object-oriented programming, unit and integration testing, etc.?
- Does the code contain any typos and does it follow [Pep8 guidelines](#)?
- Does the code follow the guidelines laid out in the VOLTTRON documentation?

Docs

Next, Check out the documentation.

- Is it complete?
 - Has an introduction describing purpose
 - Describes configuration including all parameters
 - Includes installation instructions
 - Describes behavior at runtime
 - Describes all available endpoints (JSON-RPC, pub/sub messages, Web-API endpoints, etc.)
- Does it follow the [VOLTTRON documentation guidelines](#)?

Tests

You've included tests, right? Unit and integration tests show users that *foo.py* is better than their wildest dreams - all of the features work, and include components they hadn't even considered themselves!

- Are the unit tests thorough?
 - Success and failure cases
 - Tests for each independent component of the code
- Do the integration tests capture behavior with a running VOLTTRON platform?
 - Success and Failure cases
 - Tests for each endpoint
 - Tests for interacting with other agents if necessary
 - Are status, health, etc. updating as expected when things go wrong or the code recovers?

- Can the tests be read to describe the behavior of the code?

Structure

For agents and drivers, the VOLTTRON team has some really simple structure recommendations. These make your project structure nice and tidy, and integrate nicely with the core repository.

For agents:

```
TestAgent/
├── setup.py
├── config
├── README.rst
├── tester
├── |
├── |   ├── agent.py
├── |   └── __init__.py
├── tests
├── |
├── |   └── test_agent.py
```

For drivers, the interface should be a file named after the driver in the Master Driver's interfaces directory:

```
├── master_driver
├── |
├── |   ├── agent.py
├── |   ├── driver.py
├── |   ├── __init__.py
├── |   ├── interfaces
├── |   ├── |
├── |   ├── |   ├── __init__.py
├── |   ├── |   ├── bacnet.py
├── |   ├── |   ├── csvdriver.py
├── |   ├── |   └── new_driver.py
```

Or in the `__init__.py` file in a directory named after the driver in the Master Driver's interfaces directory:

```
├── master_driver
├── |
├── |   ├── agent.py
├── |   ├── driver.py
├── |   ├── __init__.py
├── |   ├── interfaces
├── |   ├── |
├── |   ├── |   ├── __init__.py
├── |   ├── |   ├── bacnet.py
├── |   ├── |   ├── new_driver
├── |   ├── |   └── |
├── |   ├── |   └── |   └── __init__.py
```

This option is ideal for adding additional code files, and including documentation and tests.

1.10.2 Creating a Pull Request to the main VOLTTRON repository

After reviewing changes to our fork of the VOLTTRON repository, we want our changes to be added into the main VOLTTRON repository. After all, our *foo.py* can cure a lot of the world's problems and of course it is always good to have a copyright with the correct year. Open your browser to https://github.com/VOLTTRON/volttron/compare/develop...YOUR_USERNAME:develop.

On that page the base fork should always be VOLTTRON/volttron with the base develop, the head fork should be <YOUR USERNAME>/volttron and the compare should be the branch in your repository to pull from. Once you have verified that you have got the right changes made then, click on create pull request, enter a title and description that represent your changes and submit the pull request.

The VOLTTRON repository has a description template to use to format your PR:

```
# Description

Please include a summary of the change and which issue is fixed. Please also include
↳relevant motivation and context. List any dependencies that are required for this
↳change.

Fixes # (issue)

## Type of change

Please delete options that are not relevant.

- [ ] Bug fix (non-breaking change which fixes an issue)
- [ ] New feature (non-breaking change which adds functionality)
- [ ] Breaking change (fix or feature that would cause existing functionality to not
↳work as expected)
- [ ] This change requires a documentation update

# How Has This Been Tested?

Please describe the tests that you ran to verify your changes. Provide instructions
↳so we can reproduce. Please also list any relevant details for your test
↳configuration

- [ ] Test A
- [ ] Test B

**Test Configuration**:
* Firmware version:
* Hardware:
* Toolchain:
* SDK:

# Checklist:

- [ ] My code follows the style guidelines of this project
- [ ] I have performed a self-review of my own code
- [ ] I have commented my code, particularly in hard-to-understand areas
- [ ] I have made corresponding changes to the documentation
- [ ] My changes generate no new warnings
- [ ] I have added tests that prove my fix is effective or that my feature works
- [ ] New and existing unit tests pass locally with my changes
- [ ] Any dependent changes have been merged and published in downstream modules
```

Note: The VOLTTRON repository includes a stub for completing your pull request. Please follow the stub to facilitate the reviewing and merging processes.

1.10.3 What happens next?

Once you create a pull request, one or more VOLTTRON team members will review your changes and either accept them as is ask for modifications in order to have your commits accepted. Typical response time is approximately two weeks; please be patient, your pull request will be reviewed. You will be automatically emailed through the GitHub notification system when this occurs (assuming you haven't changed your GitHub preferences).

Merging changes from the main VOLTTRON repository

As time goes on the VOLTTRON code base will continually be modified so the next time you want to work on a change to your files the odds are your local and remote repository will be out of date. In order to get your remote VOLTTRON repository up to date with the main VOLTTRON repository you could simply do a pull request to your remote repository from the main repository. To do so, navigate your browser to [https://github.com/YOUR_USERNAME/volttron/compare/develop... VOLTTRON:develop](https://github.com/YOUR_USERNAME/volttron/compare/develop...VOLTTRON:develop).

Click the 'Create Pull Request' button. On the following page click the 'Create Pull Request' button. On the next page click 'Merge Pull Request' button.

Once your remote is updated you can now pull from your remote repository into your local repository through the following command:

```
git pull
```

The other way to get the changes into your remote repository is to first update your local repository with the changes from the main VOLTTRON repository and then pushing those changes up to your remote repository. To do that you need to first create a second remote entry to go along with the origin. A remote is simply a pointer to the url of a different repository than the current one. Type the following command to create a new remote called 'upstream':

```
git remote add upstream https://github.com/VOLTTRON/volttron
```

To update your local repository from the main VOLTTRON repository then execute the following command where upstream is the remote and develop is the branch to pull from:

```
git pull upstream develop
```

Finally to get the changes into your remote repository you can execute:

```
git push origin
```

Other commands to know

At this point in time you should have enough information to be able to update both your local and remote repository and create pull requests in order to get your changes into the main VOLTTRON repository. The following commands are other commands to give you more information that the preceding tutorial went through

Viewing what the remotes are in our local repository

```
git remote -v
```

Stashing changed files so that you can do a merge/pull from a remote

```
git stash save 'A comment to be listed'
```

Applying the last stashed files to the current repository

```
git stash pop
```


Finding help about any git command

```
git help
git help branch
git help stash
git help push
git help merge
```

Creating a branch from the branch and checking it out

```
git checkout -b newbranchname
```

Checking out a branch (if not local already will look to the remote to checkout)

```
git checkout branchname
```

Removing a local branch (cannot be current branch)

```
git branch -D branchname
```

Determine the current and show all local branches

```
git branch
```

Using Travis Continuous Integration Tools

The main VOLTTRON repository is hooked into an automated build tool called travis-ci. Your remote repository can be automatically built with the same tool by hooking your account into travis-ci's environment. To do this go to <https://travis-ci.org> and create an account. You can use your GitHub login directly to this service. Then you will need to enable the syncing of your repository through the travis-ci service. Finally you need to push a new change to the repository. If the build fails you will receive an email notifying you of that fact and allowing you to modify the source code and then push new changes out.

1.11 Contributing Documentation

The Community is encouraged to contribute documentation back to the project as they work through use cases the developers may not have considered or documented. By contributing documentation back, the community can learn from each other and build up a more extensive knowledge base.

VOLTTRON™ documentation utilizes ReadTheDocs: <http://volttron.readthedocs.io/en/develop/> and is built using the Sphinx Python library with static content in Restructured Text.

1.11.1 Building the Documentation

Static documentation can be found in the *docs/source* directory. Edit or create new .rst files to add new content using the [Restructured Text](#) format. To see the results of your changes the documentation can be built locally through the command line using the following instructions:

If you've already *bootstrapped* VOLTTRON™, do the following while activated. If not, this will also pull down the necessary VOLTTRON™ libraries.

```
python bootstrap.py --documentation
cd docs
make html
```

Then, open your browser to the created local files:

```
file:///home/<USER>/git/volttron/docs/build/html/overview/index.html
```

When complete, changes can be contributed back using the same process as code [contributions](#) by creating a pull request. When the changes are accepted and merged, they will be reflected in the ReadTheDocs site.

1.11.2 Documentation Styleguide

Naming Conventions

- File names and directories should be all lower-case and use only dashes/minus signs (-) as word separators

```
index.rst
├── first-document.rst
├── more-documents
│   └── second-document.rst
```

- Reference Labels should be Capitalized and dash/minus separated:

```
.. _Reference-Label:
```

- Headings and Sub-headings should be written like book titles:

```
=====
The Page Title
=====
```

Headings

Each page should have a main title:

```
=====
This is the Main Title of the Page
=====
```

It can be useful to include reference labels throughout the document to use to refer back to that section of documentation. Include reference labels above titles and important headings:

```
.. _Main-Title:
```

```
=====
This is the main title of the page
=====
```

Heading Levels

- Page titles and documentation parts should use over-line and underline hashes:

```
=====
Title
=====
```

- Chapter headings should be over-lined and underlined with asterisks

```
*****
Chapter
*****
```

- For sections, subsections, sub-subsections, etc. underline the heading with the following:
 - =, for sections
 - -, for subsections
 - ^, for sub-subsections
 - “, for paragraphs

In addition to following guidelines for styling, please separate headers from previous content by two newlines.

```
=====
Title
=====

    Content

Subheading
=====
```

Example Code Blocks

Use bash for commands or user actions:

```
ls -al
```

Use this for the results of a command:

```
total 5277200
drwxr-xr-x 22 volttron volttron    4096 Oct 20 09:44 .
drwxr-xr-x 23 volttron volttron    4096 Oct 19 18:39 ..
-rwxr-xr-x  1 volttron volttron     164 Sep 29 17:08 agent-setup.sh
drwxr-xr-x  3 volttron volttron    4096 Sep 29 17:13 applications
```

Use this when Python source code is displayed

```
@RPC.export
def status_agents(self):
    return self._aip.status_agents()
```

Directives

Danger: Something very bad!

Tip: This is something good to know

Some other directives

“attention”, “caution”, “danger”, “error”, “hint”, “important”, “note”, “tip”, “warning”, “admonition”

Links

Linking to external sites is simple:

```
Link to `Google <www.google.com>`_
```

References

You can reference other sections of documentation using the *ref* directive:

```
This will reference the :ref:`platform installation <Platform-Installation>`
```

Other resources

- <http://pygments.org/docs/lexers/>
- <http://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html>
- <http://www.sphinx-doc.org/en/stable/markup/code.html>

1.12 Jupyter Notebooks

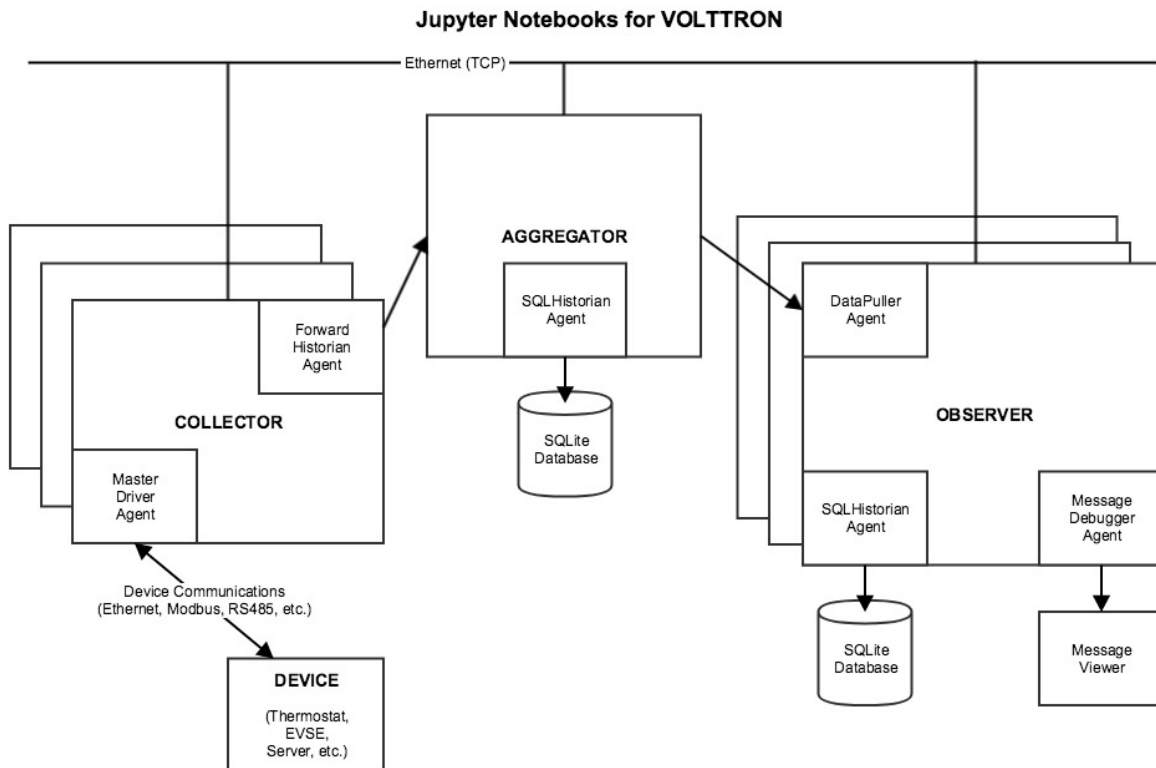
Jupyter is an open-source web application that lets you create and share “notebook” documents. A notebook displays formatted text along with live code that can be executed from the browser, displaying the execution output and preserving it in the document. Notebooks that execute Python code used to be called *iPython Notebooks*. The iPython Notebook project has now merged into Project Jupyter.

1.12.1 Using Jupyter to Manage a Set of VOLTRON Servers

The following Jupyter notebooks for VOLTRON have been provided as examples:

- **Collector notebooks.** Each Collector notebook sets up a particular type of device driver and forwards device data to another VOLTRON instance, the Aggregator.
 - **SimulationCollector notebook.** This notebook sets up a group of Simulation device drivers and forwards device data to another VOLTRON instance, the Aggregator.
 - **BacnetCollector notebook.** This notebook sets up a Bacnet (or Bacnet gateway) device driver and forwards device data to another VOLTRON instance, the Aggregator.
 - **ChargePointCollector notebook.** This notebook sets up a ChargePoint device driver and forwards device data to another VOLTRON instance, the Aggregator.
 - **SEP2Collector notebook.** This notebook sets up a SEP2.0 (IEEE 2030.5) device driver and forwards device data to another VOLTRON instance, the Aggregator. The Smart Energy Profile 2.0 (“SEP2”) protocol implements IEEE 2030.5, and is capable of connecting a wide array of smart energy devices to the Smart Grid. The standard is designed to run over TCP/IP and is physical layer agnostic.
- **Aggregator notebook.** This notebook sets up and executes aggregation of forwarded data from other VOLTRON instances, using a historian to record the data.
- **Observer notebook.** This notebook sets up and executes a DataPuller that captures data from another VOLTRON instance, using a Historian to record the data. It also uses the Message Debugger agent to monitor messages flowing across the VOLTRON bus.

Each notebook configures and runs a set of VOLTRON Agents. When used as a set they implement a multiple-VOLTRON-instance architecture that captures remote device data, aggregates it, and reports on it, routing the data as follows:



1.12.2 Install VOLTTRON and Jupyter on a Server

The remainder of this guide describes how to set up a host for VOLTTRON and Jupyter. Use this setup process on a server in order to prepare it to run Jupyter notebook for VOLTTRON.

Set Up the Server and Install VOLTTRON

The following is a complete, but terse, description of the steps for installing and running VOLTTRON on a server. For more detailed, general instructions, see *Installing Volttron*.

The VOLTTRON server should run on the same host as the Jupyter server.

- Load third-party software:

```
$ sudo apt-get update
$ sudo apt-get install build-essential python-dev openssl libssl-dev libevent-dev git
$ sudo apt-get install sqlite3
```

- Clone the VOLTTRON repository from github:

```
$ cd ~
$ mkdir repos
$ cd repos
$ git clone https://github.com/VOLTTRON/volttron/
```

- Check out the develop (or master) branch and bootstrap the development environment:

```
$ cd volttron
$ git checkout develop
$ python bootstrap.py
```

- Activate and initialize the VOLTTRON virtual environment:

Run the following each time you open a new command-line shell on the server:

```
$ export VOLTTRON_ROOT=~/.repos/volttron
$ export VOLTTRON_HOME=~/.volttron
$ cd $VOLTTRON_ROOT
$ source env/bin/activate
```

Install Extra Libraries

- Add Python libraries to the VOLTTRON virtual environment:

These notebooks use third-party software that's not included in VOLTTRON's standard distribution that was loaded by *bootstrap.py*. The following additional packages are required:

- Jupyter
- SQLAlchemy (for the Message Debugger)
- Suds (for the ChargePoint driver, if applicable)
- Numpy and Matplotlib (for plotted output)

Note: A Jupyter installation also installs and/or upgrades many dependent libraries. Doing so could disrupt other work on the OS, so it's safest to load Jupyter (and any other library code) in a virtual environment. VOLTTRON

runs in a virtual environment during normal operation, so if you're using Jupyter in conjunction with VOLTTRON, it should be installed in your VOLTTRON virtual environment (In other words, be sure to use `cd $VOLTRON_ROOT` and `source env/bin/activate` to activate the virtual environment before running `pip install`.)

- Install the third-party software:

```
$ pip install SQLAlchemy==1.1.4
$ pip install suds-jurko==0.6
$ pip install numpy
$ pip install matplotlib
$ pip install jupyter
```

Note: If `pip install` fails due to an untrusted cert, try using this command instead:

```
$ pip install --trusted-host pypi.python.org <libraryname>
```

An `InsecurePlatformWarning` may be displayed, but it typically won't stop the installation from proceeding.

1.12.3 Configure VOLTTRON

Use the `vcfg` wizard to configure the VOLTTRON instance. By default, the wizard configures a VOLTTRON instance that communicates with agents only on the local host (ip 127.0.0.1). This set of notebooks manages communications among multiple VOLTTRON instances on different hosts. To enable this cross-host communication on VOLTTRON's web server, replace 127.0.0.1 with the host's IP address, as follows:

```
$ vcfg
```

Accept all defaults, except as follows:

- If a prompt defaults to 127.0.0.1 as an IP address, substitute the *host's IP address* (this may happen multiple times).
- When asked whether this is a volttron central, answer *Y*.
- When prompted for a username and password, use *admin* and *admin*.

1.12.4 Start VOLTTRON

Start the main VOLTTRON process, logging to `$VOLTRON_ROOT/volttron.log`:

```
$ volttron -vv -l volttron.log --msgdebug
```

This runs VOLTTRON as a foreground process. To run it in the background, use:

This also enables the Message Debugger, a non-production VOLTTRON debugging aid that's used by some notebooks. To run with the Message Debugger disabled (VOLTTRON's normal state), omit the `--msgdebug` flag.

Now that VOLTTRON is running, it's ready for agent configuration and execution. Each Jupyter notebook contains detailed instructions and executable code for doing that.

1.12.5 Configure Jupyter

More detailed information about installing, configuring and using Jupyter Notebooks is available on the Project Jupyter site, <http://jupyter.org/>.

- Create a Jupyter configuration file:

```
$ jupyter notebook --generate-config
```

- Revise the Jupyter configuration:

Open `~/jupyter/jupyter_notebook_config.py` in your favorite text editor. Change the configuration to accept connections from any IP address (not just from localhost) and use a specific, non-default port number:

- Un-comment `c.NotebookApp.ip` and set it to: `*` instead of `localhost`
- Un-comment `c.NotebookApp.port` and set it to: `8891` instead of `8888`

Save the config file.

- Open ports for TCP connections:

Make sure that your Jupyter server host's security rules allow inbound TCP connections on port `8891`.

If the VOLTTRON instance needs to receive TCP requests, for example ForwardHistorian or DataPuller messages from other VOLTTRON instances, make sure that the host's security rules also allow inbound TCP communications on VOLTTRON's port, which is usually `22916`.

1.12.6 Launch Jupyter

- Start the Jupyter server:

In a separate command-line shell, set up VOLTTRON's environment variables and virtual environment, and then launch the Jupyter server:

```
$ export VOLTTRON_HOME=(your volttron home directory, e.g. ~/.volttron)
$ export VOLTTRON_ROOT=(where volttron was installed; e.g. ~/repos/volttron)
$ cd $VOLTTRON_ROOT
$ source env/bin/activate
$ cd examples/JupyterNotebooks
$ jupyter notebook --no-browser
```

- Open a Jupyter client in a web browser:

Look up the host's IP address (e.g., using `ifconfig`). Open a web browser and navigate to the URL that was displayed when you started jupyter, replacing `localhost` with that IP address. A Jupyter web page should display, listing your notebooks.

1.13 Python for Matlab Users

Matlab is a popular proprietary programming language and tool suite with built in support for matrix operations and graphically plotting computation results. The purpose of this document is to introduce Python to those already familiar Matlab so it will be easier for them to develop tools and agents in VOLTTRON.

1.13.1 A Simple Function

Python and Matlab are similar in many respects, syntactically and semantically. With the addition of the NumPy library in Python, almost all numerical operations in Matlab can be emulated or directly translated. Here are functions in each language that perform the same operation:

```
% Matlab
function [result] = times_two(number)
    result = number * 2;
end
```

```
# Python
def times_two(number):
    result = number * 2
    return result
```

Some notes about the previous functions:

1. Values are explicitly returned with the *return* statement. It is possible to return multiple values, as in Matlab, but doing this without a good reason can lead to overcomplicated functions.
2. Semicolons are not used to end statements in python, and white space is significant. After a block is started (if, for, while, functions, classes) subsequent lines should be indented with four spaces. The block ends when the programmer stops adding the extra level of indentation.

1.13.2 Translating

The following may be helpful if you already have a Matlab file or function that will be translated into Python. Many of the syntax differences between Matlab and Python can be rectified with your text editor's find and replace feature.

Start by copying all of your Matlab code into a new file with a *.py* extension. It is recommended to start by commenting everything out and uncommenting the Matlab code in chunks. This way it is possible to write valid Python and verify it as you translate, instead of waiting till the whole file is "translated". Editors designed to work with Python should be able to highlight syntax errors as well.

1. Comments are created with a *%*. Find and replace these with *#*.

```
def test_function():
    # single line Python comment
    """
    Multi-line Python comment
    """
    pass # inline Python comment
```

1. Change *elseif* blocks to *elif* blocks.

```
if thing == 0:
    do_thing1()
elif thing == 1:
    do_thing2()
else:
    do_the_last_thing()
```

1. Python indexes start at zero instead of one. Array slices and range operations don't include the upper bound, so only the lower bound should decrease by one. The following examples are of Python code in the console:

```
>>> test_array = [0, 1, 2, 3, 4]
>>> test_array[0]
0
>>> test_array[1]
1
>>> test_array[0:2]
[0, 1]
>>>>> test_array[:2]
[0, 1]
>>> test_array[2:]
[2, 3, 4]
>>>
```

1. Semicolons in Matlab are used to suppress output at the end of lines and for organizing array literals. After arranging the arrays into nested lists, all semicolons can be removed.
2. The *end* keyword in Matlab is used both to access the last element in an array and to close blocks. The array use case can be replaced with *-1* and the others can be removed entirely.

```
>>> test_array = [0, 1, 2, 3, 4]
>>> test_array[-1]
4
>>>
```

A More Concrete Example

In the [Building Economic Dispatch](#) project, a sibling project to VOLTTRON, a number of components written in Matlab would create a matrix out of some collection of columns and perform least squares regression using the *matrix division* operator. This is straightforward and very similar in both languages assuming that all of the columns are defined and are the same length.

```
% Matlab
XX = [U, xbp, xbp2, xbp3, xbp4, xbp5];
AA = XX \ ybp;
```

```
# Python
import numpy as np

XX = np.column_stack((U, xbp, xbp2, xbp3, xbp4, xbp5))
AA, resid, rank, s = np.linalg.lstsq(XX, ybp)
```

This pattern also included the creation of the *U* column, a column of ones used as the bias term in the linear equation. In order to make the Python version more readable and more robust, the pattern was removed from each component and replaced with a single function call to *least_squares_regression*.

This function does some validation on the input parameters, automatically creates the bias column, and returns the least squares solution to the system. Now if we want to change how the solution is calculated we only have to change the one function, instead of each instance where the pattern was written originally.

```
def least_squares_regression(inputs=None, output=None):
    if inputs is None:
        raise ValueError("At least one input column is required")
    if output is None:
        raise ValueError("Output column is required")
```

(continues on next page)

(continued from previous page)

```

if type(inputs) != tuple:
    inputs = (inputs,)

ones = np.ones(len(inputs[0]))
x_columns = np.column_stack((ones,) + inputs)

solution, resid, rank, s = np.linalg.lstsq(x_columns, output)
return solution

```

1.13.3 Lessons Learned (sometimes the hard way)

Variable Names

Use descriptive function and variable names whenever possible. The most important things to consider here are reader comprehension and searching. Consider a variable called *hdr*. Is it *header* without any vowels, or is it short for *high-dynamic-range*? Spelling out full words in variable names can save someone else a lot of guesswork.

Searching comes in when we're looking for instances of a string or variable. Single letter variable names are impossible to search for. Variables names describing the value being stored in a concise but descriptive manner are preferred.

Matlab load/save

Matlab has built-in functions to automatically save and load variables from your programs to disk. Using these functions can lead to poor program design and should be avoided if possible. It would be best to refactor as you translate if they are being used. Few operations are so expensive that that cannot be redone every time the program is run. For part of the program that saves variables, consider making a function that simply returns them instead.

If your Matlab program is loading csv files then use the Pandas library when working in python. Pandas works well with NumPy and is the go-to library when using csv files that contain numeric data.

1.13.4 More Resources

[NumPy for Matlab Users](#) Has a nice list of common operations in Matlab and NumPy.

[NumPy Homepage](#)

[Pandas Homepage](#)

1.14 Bootstrap Process

The *bootstrap.py* Python script in the root directory of the VOLTTRON repository may be used to create VOLTTRON's Python virtual environment and install or update service agent dependencies.

The first running of *bootstrap.py* will be against the systems *python3* executable. During this initial step a virtual environment is created using the *venv* module. Additionally, all requirements for running a base volttron instance are installed. A user can specify additional arguments to the *bootstrap.py* script allowing a way to quickly install dependencies for service agents (e.g. *bootstrap.py --mysql*).

```

# bootstrap with additional dependency requirements for web enabled agents.
user@machine$ python3 bootstrap.py --web

```

After activating an environment (source env/bin/activate) one can use the *bootstrap.py* script to install more service agent dependencies by executing the same bootstrap.py command.

Note: In the following example one can tell the environment is activated based upon the (volttron) prefix to the command prompt

```
# Adding additional database requirement for crate
(volttron) user@machine$ python3 bootstrap.py --crate
```

If a fresh install is necessary one can use the `--force` argument to rebuild the virtual environment from scratch.

```
# Rebuild the environment from the system's python3
user@machine$ python3 bootstrap.py --force
```

Note: Multiple options can be specified on the command line *python3 bootstrap.py --web --crate* installs dependencies for web enabled agents as well as the Crate database historian.

1.14.1 Bootstrap Options

The *bootstrap.py* script takes several options that allow customization of the environment, installing and update packages, and setting the package locations. The following sections can be reproduced by executing:

```
# Show the help output from bootstrap.py
user@machine$ python3 bootstrap --help
```

The options for customizing the location of the virtual environment are as follows.

```
--envdir VIRTUAL_ENV  alternate location for virtual environment
--force                force installing in non-empty directory
-o, --only-virtenv    create virtual environment and exit (skip install)
--prompt PROMPT       provide alternate prompt in activated environment
                      (default: volttron)
```

Additional options are available for customizing where an environment will retrieve packages and/or upgrade existing packages installed.

```
update options:
  --offline            install from cache without downloading
  -u, --upgrade        upgrade installed packages
  -w, --wheel          build wheels in the pip wheelhouse
```

To help bootstrap an environment in the shortest number of steps we have grouped dependency packages under named collections. For example, the `--web` argument will install six different packages from a single call to *bootstrap.py --web*. The following collections are available to use.

```
...

Extra packaging options:
  --all                All dependency groups.
  --crate              Crate database adapter
  --databases          All of the databases (crate, mysql, postgres, etc).
  --dnp3               Dependencies for the dnp3 agent.
```

(continues on next page)

(continued from previous page)

```

--documentation    All dependency groups to allow generation of documentation.
↳without error.
--drivers          All drivers known to the platform driver.
--influxdb         Influx database adapter
--market           Base market agent dependencies
--mongo            Mongo database adapter
--mysql            Mysql database adapter
--pandas           Pandas numerical analysis tool
--postgres         Postgres database adapter
--testing          A variety of testing tools for running unit/integration tests.
--web              Packages facilitating the building of web enabled agents.
--weather          Packages for the base weather agent

rabbitmq options:
  --rabbitmq [RABBITMQ]
                        install rabbitmq server and its dependencies. optional
                        argument: Install directory that exists and is
                        writeable. RabbitMQ server will be installed in a
                        subdirectory. Defaults to /home/osboxes/rabbitmq_server

...

```

1.15 Platform Configuration

Each instance of the VOLTTTRON platform includes a *config* file which is used to configure the platform instance on startup. This file is kept in *VOLTTTRON_HOME* and is created using the *volttron-cfg* (*vcfg*) command, or will be created with default values on start up of the platform otherwise.

Following is helpful information about the *config* file and the *vcfg* command.

1.15.1 VOLTTTRON_HOME

By default, the VOLTTTRON project bases its files out of *VOLTTTRON_HOME* which defaults to *~/voltttron*. This directory features directories and files used by the platform for important operation and management tasks as well as containing packaged agents and their individual runtime environments (including data directories, identity files, etc.)

- **\$VOLTTTRON_HOME/agents** - contains the agents installed on the platform
- **\$VOLTTTRON_HOME/auth.json** - file containing authentication and authorization rules for agents connecting to the VOLTTTRON instance.
- **\$VOLTTTRON_HOME/certificates** - contains the certificates for use with the Licensed VOLTTTRON code.
- **\$VOLTTTRON_HOME/configuration_store** - agent configuration store files are stored in this directory. Each agent may have a file here in which JSON representations of their stored configuration files are stored.
- **\$VOLTTTRON_HOME/run** - contains files create by the platform during execution. The main ones are the ZMQ files created for publish and subscribe functionality.
- **\$VOLTTTRON_HOME/ssh** - keys used by agent mobility in the Licensed VOLTTTRON code
- **\$VOLTTTRON_HOME/config** - Default location to place a config file to override any platform settings.
- **\$VOLTTTRON_HOME/packaged** - agent packages created with *volttron-pkg* are created in this directory

- **\$VOLTRON_HOME/VOLTRON_PID** - File containing the Unix process ID for the VOLTRON platform
- used for tracking platform status.

1.15.2 VOLTRON Config File

The *config* file in *VOLTRON_HOME* is the config file used by the platform. This configuration file specifies the behavior of the platform at runtime, including which message bus it uses, the name of the platform instance, the address bound to by VIP, and so-on. It is recommended to use the *VOLTRON Config* wizard (explained below) for configuring an instance for the first time as it will create a thorough template unique to your deployment. After using the wizard the file may be edited by the user as necessary for operations. The following is a simple example *config* for a multi-platform deployment:

```
[volttron]
message-bus = zmq
vip-address = tcp://127.0.0.1:22916
bind-web-address = <web service bind address>
web-ssl-cert = <VOLTRON_HOME>/certificates/certs/master_web-server.crt
web-ssl-key = <VOLTRON_HOME>/certificates/private/master_web-server.pem
instance-name = volttron1
volttron-central-address = <VC address>
```

The example consists of the following entries:

- **message-bus** - message bus being used for this instance (rmq/zmq)
- **vip-address** - address bound to by VIP for message bus communication
- **bind-web-address** - Optional, needed if platform has to support web feature. Represents address bound to by the platform web service for handling HTTP(s) requests. Typical address would be <https://<hostname>:8443>
- **web-ssl-cert** - Optional, needed if platform has to support web feature. Represents path to the certificate for the instance's web service
- **web-ssl-key** - Optional, needed if platform has to support web feature. Represents secret key or path to secret key file used by web service authenticate requests
- **instance-name** - name of this VOLTRON platform instance, should be unique for the deployment
- **volttron-central-address** - Optional, needed if instance is running Volttron Central. Represents web address of VOLTRON Central agent managing this platform instance. Typical address would be <https://<hostname>:8443>

1.15.3 VOLTRON Config

The *volttron-cfg* or *vcfg* command allows for an easy configuration of the VOLTRON environment. The command includes the ability to set up the platform configuration, an instance of the platform historian, VOLTRON Central UI, and VOLTRON Central Platform agent.

Running *vcfg* will create a *config* file in *VOLTRON_HOME* which will be populated according to the answers to prompts. This process should be repeated for each platform instance, and can be re-run to reconfigure a platform instance.

Note: To create a simple instance of VOLTRON, leave the default response, or select yes (y) if prompted for a yes or no response [Y/N]. You must choose a username and password for the VOLTRON Central admin account if selected.

A set of example responses are included here (*username* is *user*, *localhost* is *volttron-pc*):

```
(voltttron) user@voltttron-pc:~/voltttron$ vcfg

Your VOLTTTRON_HOME currently set to: /home/user/.voltttron

Is this the voltttron you are attempting to setup? [Y]:
What type of message bus (rmq/zmq)? [zmq]:
What is the vip address? [tcp://127.0.0.1]:
What is the port for the vip address? [22916]:
Is this instance web enabled? [N]: y
What is the protocol for this instance? [https]:
Web address set to: https://voltttron-pc
What is the port for this instance? [8443]:
Would you like to generate a new web certificate? [Y]:
WARNING! CA certificate does not exist.
Create new root CA? [Y]:

Please enter the following details for web server certificate:
  Country: [US]:
  State: WA
  Location: Richland
  Organization: PNNL
  Organization Unit: VOLTTTRON
Created CA cert
Creating new web server certificate.
Is this an instance of voltttron central? [N]: y
Configuring /home/user/voltttron/services/core/VoltttronCentral.
Installing voltttron central.
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]: y
VC admin and password are set up using the admin web interface.
After starting VOLTTTRON, please go to https://voltttron-pc:8443/admin/login.html to
→complete the setup.
Will this instance be controlled by voltttron central? [Y]:
Configuring /home/user/voltttron/services/core/VoltttronCentralPlatform.
What is the name of this instance? [voltttron1]:
Voltttron central address set to https://voltttron-pc:8443
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]: y
Would you like to install a platform historian? [N]: y
Configuring /home/user/voltttron/services/core/SQLHistorian.
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]: y
Would you like to install a master driver? [N]: y
Configuring /home/user/voltttron/services/core/MasterDriverAgent.
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Would you like to**install a fake device on the master driver? [N]: y
Should the agent autostart? [N]: y
Would you like to install a listener agent? [N]: y
Configuring examples/ListenerAgent.
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]: y
Finished configuration!

You can now start the voltttron instance.

If you need to change the instance configuration you can edit
the config file is at /home/user/.voltttron/config
```

Once this is finished, run VOLTTRON and test the new configuration.

Optional Arguments

- **-v, --verbose** - Enables verbose output in standard-output (PIP output, etc.)
- **--vhome VHOME** - Provide a path to set `VOLTTRON_HOME` for this instance
- **--instance-name INSTANCE_NAME** - Provide a name for this instance. Required for running secure agents mode
- **--list-agents** - Display a list of configurable agents (Listener, Master Driver, Platform Historian, VOLTTRON Central, VOLTTRON Central Platform)
- **--agent AGENT [AGENT ...]** - Configure listed agents
- **--rabbitmq RABBITMQ [RABBITMQ ...]** - Configure rabbitmq for single instance, federation, or shovel either based on configuration file in yml format or providing details when prompted.

Usage:

```
vcfg --rabbitmq single|federation|shovel [rabbitmq config file]``
```

- **--secure-agent-users** - Require that agents run as their own Unix users (this requires running `scripts/secure_user_permissions.sh` as `sudo`)

1.16 Planning a Deployment

The 3 major installation types for VOLTTRON are doing development, doing research using VOLTTRON, and collecting and managing physical devices.

Development and Research installation tend to be smaller footprint installations. For development, the data is usually synthetic or copied from another source. The existing documentation covers development installs in significant detail.

Other deployments will have a better installation experience if they consider certain kinds of questions while they plan their installation.

1.16.1 Questions

- Do you want to send commands to the machines ?
- Do you want to store the data centrally ?
- How many machines do you expect to collect data from on each “collector” ?
- How often will the machines collect data ?
- Are all the devices visible to the same network ?
- What types of VOLTTRON applications do you want to run ?

Commands

If you wish to send commands to the devices, you will want to install and configure the Volttron Central agent. If you are only using VOLTTRON to securely collect the data, you can turn off the extra agents to reduce the footprint.

Storing Data

VOLTTRON supports multiple historians. MySQL and MongoDB are the most commonly used. As you plan your installation, you should consider how quickly you need access to the data and where. If you are looking at the health and well-being of an entire suite of devices, its likely that you want to do that from a central location. Analytics can be performed at the edge by VOLTTRON applications or can be performed across the data usually from a central data repository. The latency that you can tolerate in your data being available will also determine choices in different agents (ForwardHistorian versus Data Mover)

How Many

The ratio of how many devices-to-collector machine is based on several factors. These include:

- how much memory and network bandwidth the collection machine has. More = More devices
- how fast the local storage is can affect how fast the data cache can be written. Very slow storage devices can fall behind

The second half of the “how many” question is how many collector platforms are writing to a single VOLTTRON platform to store data - and whether that storage is local, remote, big enough, etc.

If you are storing more than moderate amount of data, you will probably benefit from installing your database on a different machine than your concrete historian machine.

Note: This is contra-indicated if you have a slow network connection between you concrete historian and your database machine.

In synthetic testing up to 6 virtual machines hosting 500 devices each (18 points) were easily supported by a single centralized platform writing to a Mongo database - using a high speed network. That central platform experienced very little CPU or memory load when the VOLTTRON Central agent was disabled.

How Often

This question is closely related to the last. A higher sampling frequency will create more data. This will place more work in the storage phase.

Networks

In many cases, there are constraints on how networks can interact with each other. In many cases, these include security considerations. On some sites, the primary network will be protected from less secure networks and may require different installation considerations. For example, if a data collector machine and the database machine are on the same network with sufficient security, you may choose to have the data collector write directly to the database. If the collector is on an isolated building network then you will likely need to use the ForwardHistorian to bridge the two networks.

Other Considerations

Physical location and maintenance of collector machines must be considered in all live deployments. Although the number of data points may imply a heavy load on a data collection box, the physical constraints may limit the practicality of having more than a single box. The other side of that discussion is deploying many collector boxes may be simpler initially, but may create a maintenance challenge if you don’t plan ahead on how you apply patches, etc.

Naming conventions should also be considered. The ability to trace data through the system and identify the collector machine and device can be invaluable in debugging and analysis.

1.16.2 Deployment Options

There are several ways to deploy the VOLTTRON platform in a Linux environment. It is up to the user to determine which is right for them. The following assumes that the platform has already been bootstrapped and is ready to run.

Simple Command Line

With the VOLTTRON environment activated the platform can be started simply by running VOLTTRON on the command line.

```
$volttron -vv
```

This will start the platform in the current terminal with very verbose logging turned on. This is most appropriate for testing Agents or testing a deployment for problems before switching to a more long term solution. This will print all log messages to the console in real time.

This should not be used for long term deployment. As soon as an SSH session is terminated for whatever reason the processes attached to that session will be killed. This also will not capture log message to a file.

Running VOLTTRON as a Background Process

A simple, more long term solution, is to run volttron in the background and disown it from the current terminal.

Warning: If you plan on running VOLTTRON in the background and detaching it from the terminal with the `disown` command be sure to redirect `stderr` and `stdout` to `/dev/null`. Even if logging to a file is used some libraries which VOLTTRON relies on output directly to `stdout` and `stderr`. This will cause problems if those file descriptors are not redirected to `/dev/null`.

```
$volttron -vv -l volttron.log > /dev/null 2>&1&
```

Alternatively:

```
``./start-volttron``
```

Note: If you are not in an activated environment, this script will start the platform running in the background in the correct environment, however the environment will not be activated for you, you must activate it yourself.

If there are other jobs running in your terminal be sure to disown the correct one.

```
$jobs
[1]+  Running                  something else
[2]+  Running                  ./start-volttron

#Disown VOLTTRON
$disown %2
```

This will run the VOLTTTRON platform in the background and turn it into a daemon. The log output will be directed to a file called `voltttron.log` in the current directory.

To keep the size of the log under control for more longer term deployments us the rotating log configuration file `examples/rotatinglog.py`.

```
$voltttron -vv --log-config examples/rotatinglog.py > /dev/null 2>&1&
```

This will start a rotate the log file at midnight and limit the total log data to seven days worth.

The main downside to this approach is that the VOLTTTRON platform will not automatically resume if the system is restarted. It will need to be restarted manually after reboot.

Setting up VOLTTTRON as a System Service

Systemd

An example service file `scripts/admin/voltttron.service` for systemd cas be used as a starting point for setting up VOLTTTRON as a service. Note that as this will redirect all the output that would be going to stdout - to the syslog. This can be accessed using *journalctl*. For systems that run all the time or have a high level of debugging turned on, we recommend checking the system's logrotate settings.

```
[Unit]
Description=VOLTTTRON Platform Service
After=network.target

[Service]
Type=simple

#Change this to the user that VOLTTTRON will run as.
User=voltttron
Group=voltttron

#Uncomment and change this to specify a different VOLTTTRON_HOME
#Environment="VOLTTTRON_HOME=/home/voltttron/.voltttron"

#Change these to settings to reflect the install location of VOLTTTRON
WorkingDirectory=/var/lib/voltttron
ExecStart=/var/lib/voltttron/env/bin/voltttron -vv
ExecStop=/var/lib/voltttron/env/bin/voltttron-ctl shutdown --platform

[Install]
WantedBy=multi-user.target
```

After the file has been modified to reflect the setup of the platform you can install it with the following commands. These need to be run as root or with sudo as appropriate.

```
#Copy the service file into place
cp scripts/admin/voltttron.service /etc/systemd/system/

#Set the correct permissions if needed
chmod 644 /etc/systemd/system/voltttron.service

#Notify systemd that a new service file exists (this is crucial!)
systemctl daemon-reload
```

(continues on next page)

(continued from previous page)

```
#Start the service
systemctl start volttron.service
```

Init.d

An example init script `scripts/admin/volttron` can be used as a starting point for setting up VOLTTRON as a service on init.d based systems.

Minor changes may be needed for the file to work on the target system. Specifically the `USER`, `VLHOME`, and `VOLTTRON_HOME` variables may need to be changed.

```
...
#Change this to the user VOLTTRON will run as.
USER=volttron
#Change this to the install location of VOLTTRON
VLHOME=/var/lib/volttron

...

#Uncomment and change this to specify a different VOLTTRON_HOME
#export VOLTTRON_HOME=/home/volttron/.volttron
```

The script can be installed with the following commands. These need to be run as root or with `sudo` as appropriate.

```
#Copy the script into place
cp scripts/admin/volttron /etc/init.d/

#Make the file executable
chmod 755 /etc/init.d/volttron

#Change the owner to root
chown root:root /etc/init.d/volttron

#These will set it to startup automatically at boot
update-rc.d volttron defaults

#Start the service
/etc/init.d/volttron start
```

1.17 Single Machine

The purpose of this demonstration is to show the process of setting up a simple VOLTTRON instance for use on a single machine.

Note: The simple deployment example below considers only the ZeroMQ deployment scenario. For RabbitMQ deployments, read and perform the RabbitMQ installation steps from the [platform installation](#) instructions and configuration steps from [VOLTTRON Config](#).

1.17.1 Install and Build VOLTTTRON

First, *install* VOLTTTRON:

For a quick reference for Ubuntu machines:

```
sudo apt-get update
sudo apt-get install build-essential libffi-dev python3-dev python3-venv openssl_
↳ libssl-dev libevent-dev git
git clone https://github.com/VOLTTTRON/voltttron/
cd voltttron
python3 bootstrap.py --drivers --databases
```

Note: For additional detail and more information on installing in other environments, please see the *platform install* section. See the *bootstrap process* docs for more information on its operation and available options.

Activate the Environment

After the build is complete, activate the VOLTTTRON environment.

```
source env/bin/activate
```

Run VOLTTTRON Config

The *voltttron-cfg* or *vcfg* commands can be used to configure platform communication. For an example single machine deployment, most values can be left at their default values. The following is a simple case example of running *vcfg*:

```
(voltttron) user@voltttron-pc:~/voltttron$ vcfg

Your VOLTTTRON_HOME currently set to: /home/james/.voltttron

Is this the voltttron you are attempting to setup? [Y]:
What type of message bus (rmq/zmq)? [zmq]:
What is the vip address? [tcp://127.0.0.1]:
What is the port for the vip address? [22916]:
Is this instance web enabled? [N]:
Will this instance be controlled by voltttron central? [Y]: N
Would you like to install a platform historian? [N]:
Would you like to install a master driver? [N]:
Would you like to install a listener agent? [N]:
Finished configuration!

You can now start the voltttron instance.

If you need to change the instance configuration you can edit
the config file is at /home/james/.voltttron/config
```

To learn more, read the *voltttron-config* section of the Platform Features docs.

Note: Steps below highlight manually installing some example agents. To skip manual install, supply *y* or *Y* for the platform historian, master driver and listener agent installation options.

Start VOLTRON

The most convenient way to start the platform is with the `.start-voltron` command (from the voltron root directory).

```
./start-voltron
```

The output following the platform starting successfully will appear like this:

```
2020-10-27 11:34:33,593 () voltron.platform.agent.utils DEBUG: value from env None
2020-10-27 11:34:33,593 () voltron.platform.agent.utils DEBUG: value from config_
↳False
2020-10-27 11:34:35,656 () root DEBUG: Creating ZMQ Core config.store
2020-10-27 11:34:35,672 () voltron.platform.store INFO: Initializing configuration_
↳store service.
2020-10-27 11:34:35,717 () root DEBUG: Creating ZMQ Core platform.auth
2020-10-27 11:34:35,728 () voltron.platform.auth INFO: loading auth file /home/james/_
↳.voltron/auth.json
2020-10-27 11:34:35,731 () voltron.platform.auth INFO: auth file /home/james/_
↳voltron/auth.json loaded
2020-10-27 11:34:35,732 () voltron.platform.agent.utils INFO: Adding file watch for /
↳home/james/.voltron/auth.json dirname=/home/james/.voltron, filename=auth.json
2020-10-27 11:34:35,734 () voltron.platform.agent.utils INFO: Added file watch for /
↳home/james/.voltron/auth.json
2020-10-27 11:34:35,734 () voltron.platform.agent.utils INFO: Adding file watch for /
↳home/james/.voltron/protected_topics.json dirname=/home/james/.voltron,_
↳filename=protected_topics.json
2020-10-27 11:34:35,736 () voltron.platform.agent.utils INFO: Added file watch for /
↳home/james/.voltron/protected_topics.json
2020-10-27 11:34:35,737 () voltron.platform.vip.pubsubservice INFO: protected-topics_
↳loaded
2020-10-27 11:34:35,739 () voltron.platform.vip.agent.core INFO: Connected to_
↳platform: router: fc054c9f-aa37-4842-a618-6e70d53530f0 version: 1.0 identity:_
↳config.store
2020-10-27 11:34:35,743 () voltron.platform.vip.agent.core INFO: Connected to_
↳platform: router: fc054c9f-aa37-4842-a618-6e70d53530f0 version: 1.0 identity:_
↳platform.auth
2020-10-27 11:34:35,746 () voltron.platform.vip.pubsubservice INFO: protected-topics_
↳loaded
2020-10-27 11:34:35,750 () voltron.platform.vip.agent.subsystems.configstore DEBUG:_
↳Processing callbacks for affected files: {}
2020-10-27 11:34:35,879 () root DEBUG: Creating ZMQ Core control
2020-10-27 11:34:35,908 () root DEBUG: Creating ZMQ Core keydiscovery
2020-10-27 11:34:35,913 () root DEBUG: Creating ZMQ Core pubsub
2020-10-27 11:34:35,924 () voltron.platform.auth INFO: loading auth file /home/james/_
↳.voltron/auth.json
2020-10-27 11:34:38,010 () voltron.platform.vip.agent.core INFO: Connected to_
↳platform: router: fc054c9f-aa37-4842-a618-6e70d53530f0 version: 1.0 identity:_
↳control
2020-10-27 11:34:38,066 () voltron.platform.vip.agent.core INFO: Connected to_
↳platform: router: fc054c9f-aa37-4842-a618-6e70d53530f0 version: 1.0 identity: pubsub
2020-10-27 11:34:38,069 () voltron.platform.vip.agent.core INFO: Connected to_
↳platform: router: fc054c9f-aa37-4842-a618-6e70d53530f0 version: 1.0 identity:_
↳keydiscovery
2020-10-27 11:34:38,429 () voltron.platform.auth WARNING: Attempt 1 to get peerlist_
↳failed with exception 0.5 seconds
2020-10-27 11:34:38,430 () voltron.platform.auth WARNING: Get list of peers from_
↳subsystem directly
2020-10-27 11:34:38,433 () voltron.platform.auth INFO: auth file /home/james/_
↳voltron/auth.json loaded
```

(continues on next page)

(continued from previous page)

```

2020-10-27 11:34:38,434 () volttron.platform.auth INFO: loading auth file /home/james/
↳.volttron/auth.json
2020-10-27 11:34:40,961 () volttron.platform.auth WARNING: Attempt 1 to get peerlist_
↳failed with exception 0.5 seconds
2020-10-27 11:34:40,961 () volttron.platform.auth WARNING: Get list of peers from_
↳subsystem directly
2020-10-27 11:34:40,969 () volttron.platform.auth INFO: auth file /home/james/.
↳volttron/auth.json loaded

```

Note: While running the platform with verbose logging enabled, the *volttron.log* file is useful for confirming successful platform operations or debugging. It is commonly recommended to open a new terminal window and run the following command to view the VOLTTRON logs as they are created:

```
tail -f volttron.log
```

1.17.2 Install Agents and Historian

Out of the box, VOLTTRON includes a number of agents which may be useful for single machine deployments:

- **historians** - Historians automatically record a data from a number of topics published to the bus. For more information on the historian framework or one of the included concrete implementations, view the docs
- **Listener** - This example agent can be useful for debugging drivers or other agents publishing to the bus. [docs](#)
- **Master Driver** - The Master-Driver is responsible for managing device communication on a platform instance.
- **weather agents** - weather agents can be used to collect weather data from sources like Weather.gov

Note: The *services/core*, *services/ops*, and *examples* directories in the repository contain additional agents to use to fit individual use cases.

For a simple setup example, a Master Driver, SQLite Historian, and Listener are installed using the following steps:

1. Create a configuration file for the Master Driver and SQLite Historian (it is advised to create a *configs* directory in volttron root to keep configs for a deployment). For information on how to create configurations for these agents, view their docs:
 - Master Driver
 - SQLite Historian
 - [Listener](#)

For a simple example, the configurations can be copied as-is to the *configs* directory:

```

cp services/core/MasterDriverAgent/master-driver.agent configs
cp services/core/SQLiteHistorian/config.sqlite configs
cp examples/ListenerAgent/config configs/listener.config

```

2. Use the *install-agent.py* script to install the agent on the platform:

```
python scripts/install-agent.py -s services/core/SQLHistorian -c configs/config.
↳sqlite --tag listener
python scripts/install-agent.py -s services/core/MasterDriverAgent -c configs/master-
↳driver.agent --tag master_driver
python scripts/install-agent.py -s examples/ListenerAgent -c configs/listener.config -
↳-tag platform_historian

.. note::

    The `volttron.log` file will contain logging indicating that the agent has
    ↳installed successfully.

    .. code-block:: console

        2020-10-27 11:42:08,882 () volttron.platform.auth INFO: AUTH: After
        ↳authenticate user id: control.connection, b'c61dff8e-f362-4906-964f-63c32b99b6d5'
        2020-10-27 11:42:08,882 () volttron.platform.auth INFO: authentication success:
        ↳userid=b'c61dff8e-f362-4906-964f-63c32b99b6d5' domain='vip', address=
        ↳'localhost:1000:1000:3249', mechanism='CURVE', credentials=[
        ↳'ZrDvPG4JNLE26GoPUrTP22rV0PV8uGCnrXThrNFk_Ec'], user='control.connection'
        2020-10-27 11:42:08,898 () volttron.platform.aip DEBUG: Using name template
        ↳"listeneragent-3.3_{n}" to generate VIP ID
        2020-10-27 11:42:08,899 () volttron.platform.aip INFO: Agent b3e7053c-28e8-414f-
        ↳b685-8522eb230c7a setup to use VIP ID listeneragent-3.3_1
        2020-10-27 11:42:08,899 () volttron.platform.agent.utils DEBUG: missing file /
        ↳home/james/.volttron/agents/b3e7053c-28e8-414f-b685-8522eb230c7a/listeneragent-3.3/
        ↳listeneragent-3.3.dist-info/keystore.json
        2020-10-27 11:42:08,899 () volttron.platform.agent.utils INFO: creating file /
        ↳home/james/.volttron/agents/b3e7053c-28e8-414f-b685-8522eb230c7a/listeneragent-3.3/
        ↳listeneragent-3.3.dist-info/keystore.json
        2020-10-27 11:42:08,899 () volttron.platform.keystore DEBUG: calling generate
        ↳from keystore
        2020-10-27 11:42:08,909 () volttron.platform.auth INFO: loading auth file /home/
        ↳james/.volttron/auth.json
        2020-10-27 11:42:11,415 () volttron.platform.auth WARNING: Attempt 1 to get
        ↳peerlist failed with exception 0.5 seconds
        2020-10-27 11:42:11,415 () volttron.platform.auth WARNING: Get list of peers
        ↳from subsystem directly
        2020-10-27 11:42:11,419 () volttron.platform.auth INFO: auth file /home/james/.
        ↳volttron/auth.json loaded
```

1. Use the `vctl status` command to ensure that the agents have been successfully installed:

```
vctl status
```

```
(volttron)user@volttron-pc:~/volttron$ vctl status
```

AGENT	IDENTITY	TAG	STATUS	
↳HEALTH				
8 listeneragent-3.2	listeneragent-3.2_1	listener		
0 master_driveragent-3.2	platform.driver	master_driver		
3 sqlhistorianagent-3.7.0	platform.historian	platform_historian		

Note: After installation, the *STATUS* and *HEALTH* columns of the `vctl status` command will be vacant, indicating that the agent is not running. The `--start` option can be added to the `install-agent.py` script arguments to automatically start agents after they have been installed.

1.17.3 Install a Fake Driver

The following are the simplest steps for installing a fake driver for example use. For more information on installing concrete drivers such as the BACnet or Modbus drivers, view their respective documentation in the Driver framework section.

Note: This section will assume the user has created a *configs* directory in the volttron root directory, activated the Python virtual environment, and started the platform as noted above.

```
cp examples/configurations/drivers/fake.config <VOLTTRON root>/configs
cp examples/configurations/drivers/fake.csv <VOLTTRON root>/configs
vctl config store platform.driver devices/campus/building/fake configs/fake.config
vctl config store platform.driver fake.csv devices/fake.csv
```

Note: For more information on the fake driver, or the configurations used in the above example, view the docs

1.17.4 Testing the Deployment

To test that the configuration was successful, start an instance of VOLTTRON in the background:

```
./start-volttron
```

Note: This command must be run from the root VOLTTRON directory.

Having following the examples above, the platform should be ready for demonstrating the example deployment. Start the Listener, SQLite historian and Master Driver.

```
vctl start --tag listener platform_historian master_driver
```

The output should look similar to this:

```
(volttron)user@volttron-pc:~/volttron$ vctl status
```

	AGENT	IDENTITY	TAG	STATUS	
→	HEALTH				
8	listeneragent-3.2	listeneragent-3.2_1	listener	running [2810]	GOOD
0	master_driveragent-3.2	platform.driver	master_driver	running [2813]	GOOD
3	sqlhistorianagent-3.7.0	platform.historian	platform_historian	running [2811]	GOOD

Note: The *STATUS* column indicates whether the agent is running. The *HEALTH* column indicates whether the current state of the agent is within intended parameters (if the Master Driver is publishing, the platform historian has not been backlogged, etc.)

You can further verify that the agents are functioning correctly with `tail -f volttron.log`.

ListenerAgent:

```
2020-10-27 11:43:33,997 (listeneragent-3.3 3294) __main__ INFO: Peer: pubsub, Sender:
↳ listeneragent-3.3_1:, Bus: , Topic: heartbeat/listeneragent-3.3_1, Headers: {
↳ 'TimeStamp': '2020-10-27T18:43:33.988561+00:00', 'min_compatible_version': '3.0',
↳ 'max_compatible_version': ''}, Message:
'GOOD'
```

Master Driver with Fake Driver:

```
2020-10-27 11:47:50,037 (listeneragent-3.3 3294) __main__ INFO: Peer: pubsub, Sender:
↳ platform.driver:, Bus: , Topic: devices/campus/building/fake/all, Headers: {'Date':
↳ '2020-10-27T18:47:50.005349+00:00', 'TimeStamp': '2020-10-27T18:47:50.005349+00:00',
↳ 'SynchronizedTimeStamp': '2020-10-27T18:47:50.000000+00:00', 'min_compatible_
↳ version': '3.0', 'max_compatible_version': ''}, Message:
[{'EKG': -0.8660254037844386,
  'EKG_Cos': -0.8660254037844386,
  'EKG_Sin': -0.8660254037844386,
  'Heartbeat': True,
  'OutsideAirTemperature1': 50.0,
  'OutsideAirTemperature2': 50.0,
  'OutsideAirTemperature3': 50.0,
  'PowerState': 0,
  'SampleBool1': True,
  'SampleBool2': True,
  'SampleBool3': True,
  'SampleLong1': 50,
  ...
```

SQLite Historian:

```
2020-10-27 11:50:25,021 (master_driveragent-4.0 3535) master_driver.driver DEBUG:
↳ finish publishing: devices/campus/building/fake/all
2020-10-27 11:50:25,052 (sqlhistorianagent-3.7.0 3551) volttron.platform.dbutils.
↳ sqlitefuncts DEBUG: Managing store - timestamp limit: None GB size limit: None
```

1.18 Multi-Platform Connection

There are multiple ways to establish connection between external VOLTTRON platforms. Given that VOLTTRON now supports ZeroMq and RabbitMQ type of message bus with each using different type authentication mechanism, the number of different ways that agents can connect to external platforms has significantly increased. Various multi-platform deployment scenarios will be covered in this section.

1. Agents can directly connect to external platforms to send and receive messages. Forward historian, Data Mover agents fall under this category. The deployment steps for forward historian is described in [Forward Historian Deployment](#) and data mover historian in [DataMover Historian Deployment](#)
2. The platforms maintain the connection with other platforms and agents can send to and receive messages from external platforms without having to establish connection directly. The deployment steps is described in [Multi Platform Router Deployment](#)
3. RabbitMQ has ready made plugins such as shovel and federation to connect to external brokers. This feature is leveraged to make connections to external platforms. This is described in [Multi Platform RabbitMQ Deployment](#)
4. A web based admin interface to authenticate multiple instances (ZeroMq or RabbitMQ) wanting to connect to single central instance is now available. The deployment steps is described in [Multi Platform Multi-Bus Deployment](#)

5. VOLTTRON Central is a platform management web application that allows platforms to communicate and to be managed from a centralized server. The deployment steps is described in VOLTTRON Central Demo

1.18.1 Assumptions

- *Data Collector* is the deployment box that has the drivers and is collecting data from devices which will be forwarded to a *VOLTTRON Central*.
- *Volttron Central (VC)* is the deployment box that has the historian which will save data from all Data Collectors to the central database.
- *VOLTTRON_HOME* is assumed to the default on both boxes (*/home/<user>/voltron*).

Note: *VOLTTRON_HOME* is the directory used by the platform for managing state and configuration of the platform and agents installed locally on the platform. Auth keys, certificates, the configuration store, etc. are stored in this directory by the platform.

Forward Historian

This guide describes a simple setup where one VOLTTRON instance collects data from a fake devices and sends to another instance . Lets consider the following example.

We are going to create two VOLTTRON instances and send data from one VOLTTRON instance running a fake driver(subscribing values from a fake device) and sending the values to the second VOLTTRON instance.

VOLTTRON instance 1 forwards data to VOLTTRON instance 2

VOLTTRON instance 1

- `vctl shutdown -platform` (if the platform is already working)
- `vcfg` (this helps in configuring the volttron instance http://volttron.readthedocs.io/en/releases-4.1/core_services/control/VOLTTRON-Config.html)
 - Specify the IP of the machine: `tcp://130.20.*.*:22916`
 - Specify the port you want to use
 - Specify if you want to run VC(Volttron Central) here or this this instance would be controlled by a VC and the IP and port of the VC
 - * Then install agents like Master Driver Agent with a fake driver for the instance.
 - * Install a listener agent so see the topics that are coming from the diver agent
 - * Then run the volttron instance by using the following command: `./start-volttron`
- Volttron authentication: We need to add the IP of the instance 2 in the *auth.config* file of the VOLTTRON agent. This is done as follows:
 - `vctl auth-add`
 - We specify the IP of the instance 2 and the credentials of the agent (read Agent Authentication
 - For specifying authentication for all the agents , we specify `/*/*`
 - This should enable authentication for all the volttron-instance based on the IP you specify here

For this documentation, the topics from the driver agent will be send to the instance 2

- We use the existing agent called the Forward Historian for this purpose which is available in service/core in the VOLTTRON directory.
- In the config file under the Forward Historian directory, we modify the following fields:
 - Destination-vip: the IP of the volttron instance to which we have to forward the data to along with the port number. Example : `tcp://130.20.*.*:22916`
 - Destination-serverkey: The server key of the VOLTTRON instance to which we need to forward the data to. This can be obtained at the VOLTTRON instance by typing `vctl auth serverkey`
- Service_topic_list: specify the topics you want to forward specifically instead of all the values.
- Once the above values are set, your forwarder is all set .
- You can create a script file for the same and execute the agent.

VOLTTRON instance 2

- `vctl shutdown -platform` (if the platform is already working)
- `volttron-cfg` (this helps in configuring the volttron instance) http://volttron.readthedocs.io/en/releases-4.1/core_services/control/VOLTTRON-Config.html
 - Specify the IP of the machine : `tcp://130.20.*.*:22916`
 - Specify the port you want to use.
 - Install the listener agent (this will show the connection from instance 1 if its successful and then show all the topics from instance 1.
- Volttron authentication: We need to add the IP of the instance 1 in the auth.config file of the VOLTTRON agent . This is done as follows:
 - `vctl auth-add`
 - We specify the IP of the instance 1 and the credentials of the agent
 - For specifying authentication for all the agents , we specify `/.*/`
 - This should enable authentication for all the volttron-instance based on the IP you specify here

Listener Agent

Run the listener agent on this instance to see the values being forwarded from instance 1. Once the above setup is done, you should be able to see the values from instance 1 on the listener agent of instance 2.

DataMover Historian

This guide describes how a DataMover historian can be used to transfer data from one VOLTTRON instance to another. The DataMover historian is different from Forward historian in the way it sends the data to the remote instance. It first batches the data and makes a RPC call to a remote historian instead of publishing data on the remote message bus instance. The remote historian then stores the data into it's database.

The walk-through below demonstrates how to setup DataMover historian to send data from one VOLTTRON instance to another.

VOLTTRON instance 1 sends data to platform historian on VOLTTRON instance 2

As an example two VOLTTRON instances will be created and to send data from one VOLTTRON instance running a fake driver (subscribing to publishes from a fake device) and sending the values to a remote historian running on the second VOLTTRON instance.

VOLTTRON instance 1

- `vctl shutdown -platform` (if the platform is already working)
- `volttron-cfg` (this helps in configuring the volttron instance http://volttron.readthedocs.io/en/releases-4.1/core_services/control/VOLTTRON-Config.html)
 - Specify the VIP address of the instance: `tcp://127.0.0.1:22916`
 - Install Master Driver Agent with a fake driver for the instance.
 - Install a listener agent so see the topics that are coming from the diver agent
- Then run the volttron instance by using the following command: `./start-volttron`

VOLTTRON instance 2

- `vctl shutdown -platform` (if the platform is already working)
- `volttron-cfg` (this helps in configuring the volttron instance) http://volttron.readthedocs.io/en/releases-4.1/core_services/control/VOLTTRON-Config.html
 - Specify the VIP address of the instance: `tcp://127.0.0.2:22916`
 - Install a platform historian. `volttron-cfg` installs a default SQL historian.
- Start the VOLTTRON instance by using following command: `./start-volttron`

DataMover Configuration

An example config file is available in `services/core/DataMover/config`. We need to update the *destination-vip*, *destination-serverkey*, and *destination-historian-identity* entries as per our setup.

Note: Here the topics from the driver on VOLTTRON instance 1 will be sent to instance 2.

- **destination-vip:** The VIP address of the volttron instance to which we need to send data. Example : `tcp://127.0.0.2:22916`
- **destination-serverkey:** The server key of remote VOLTTRON instance - Get the server key of VOLTTRON instance 2 and set *destination-serverkey* property with the server key

```
vctl auth serverkey
```

- **destination-historian-identity:** Identity of remote platform historian. Default is “platform.historian”
-

Running DataMover Historian

- Install the DataMover historian on the VOLTTRON instance 1

```
python scripts/install-agent.py -s services/core/DataMover -c services/core/DataMover/  
↪config -i datamover --start
```

- Add the public key of the DataMover historian on VOLTTRON instance 2 to enable authentication of the DataMover on VOLTTRON instance 2.
 - Get the public key of the DataMover. Run the below command on instance 1 terminal.

```
vctl auth publickey --name datamoveragent-0.1
```

- Add the credentials of the DataMover historian in VOLTTRON instance 2

```
vctl auth add --credentials <public key of data mover>
```

Check data in SQLite database

To check if data is transferred and stored in the database of remote platform historian, we need to check the entries in the database. The default location of SQL database (if not explicitly specified in the config file) will be in the *data* directory inside the platform historian's installed directory within its *\$VOLTTRON_HOME*.

- Get the uuid of the platform historian. This can be found by running the `vctl status` on the terminal of instance 2. The first column of the data mover historian entry in the status table gives the first alphabet/number of the uuid.
- Go the *data* directory of platform historian's install directory. For example, */home/ubuntu/.platform2/agents/6292302c-32cf-4744-bd13-27e78e96184f/sqlhistorianagent-3.7.0/data*
- **Run the SQL command to see the data**

```
sqlite3 platform.historian.sqlite  
select * from data;
```

- You will see similar entries

```
2020-10-27T15:07:55.006549+00:00|14|true  
2020-10-27T15:07:55.006549+00:00|15|10.0  
2020-10-27T15:07:55.006549+00:00|16|20  
2020-10-27T15:07:55.006549+00:00|17|true  
2020-10-27T15:07:55.006549+00:00|18|10.0  
2020-10-27T15:07:55.006549+00:00|19|20  
2020-10-27T15:07:55.006549+00:00|20|true  
2020-10-27T15:07:55.006549+00:00|21|0  
2020-10-27T15:07:55.006549+00:00|22|0
```

Multi-Platform Between Routers

Multi-Platform between routers alleviates the need for an agent in one platform to connect to another platform directly in order for it to send/receive messages from the other platform. Instead with this new type of connection, connections to external platforms will be maintained by the platforms itself and agents do not have the burden to manage the connections directly. This guide will show how to connect three VOLTTRON instances with a fake driver running on

VOLTTTRON instance 1 publishing to topic with prefix="devices" and listener agents running on other 2 VOLTTTRON instances subscribed to topic "devices".

- *Getting Started*
- *Multi-Platform Configuration*
- *Configuration and Authentication in Setup Mode*
- *Setup Configuration and Authentication Manually*
- *Start Master driver on VOLTTTRON instance 1*
- *Start Listener agents on VOLTTTRON instance 2 and 3*
- *Stopping All the Platforms*

Getting Started

Modify the subscribe annotate method parameters in the listener agent (examples/ListenerAgent/listener/agent.py in the VOLTTTRON root directory) to include `all_platforms=True` parameter to receive messages from external platforms.

```
@PubSub.subscribe('pubsub', '')
```

to

```
@PubSub.subscribe('pubsub', 'devices', all_platforms=True)
```

or add below line in the *onstart* method

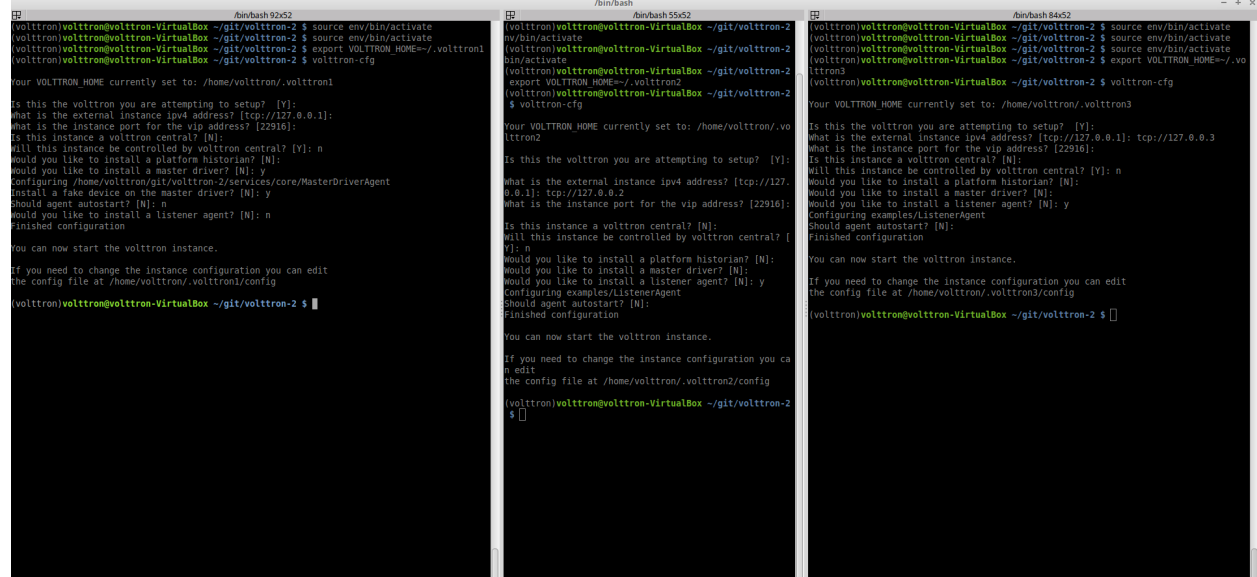
```
self.vip.pubsub.subscribe('pubsub', 'devices', self.on_match, all_platforms=True)
```

Note: If using the *onstart* method remove the `@PubSub.subscribe('pubsub', '')` from the top of the method.

After *installing VOLTTTRON*, open three shells with the current directory the root of the VOLTTTRON repository. Then activate the VOLTTTRON environment and export the `VOLTTTRON_HOME` variable. The home variable needs to be different for each instance.

```
$ source env/bin/activate
$ export VOLTTTRON_HOME=~/.volttron1
```

Run *vcfg* in all the three shells. This command will ask how the instance should be set up. Many of the options have defaults and that will be sufficient. Enter a different VIP address for each platform. Configure fake master driver in the first shell and listener agent in second and third shell.



Multi-Platform Configuration

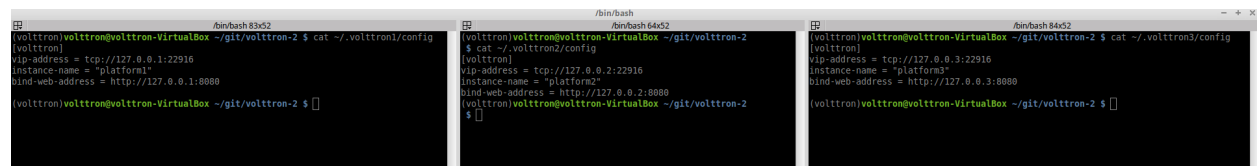
For each instance, specify the instance name in platform config file under it's `VOLTTRON_HOME` directory. If the platform supports web server, add the `bind-web-address` as well.

Here is an example,

Path of the config: `$VOLTRON_HOME/config`

```
[volttron]
vip-address = tcp://127.0.0.1:22916
instance-name = "platform1"
bind-web-address = http://127.0.0.1:8080
```

Instance name and bind web address entries added into each VOLTTRON platform's config file is shown below.



Next, each instance needs to know the VIP address, platform name and server keys of the remote platforms that it is connecting to. In addition, each platform has to authenticate or accept the connecting instances' public keys. We can do this step either by running VOLTTRON in setup mode or configure the information manually.

Configuration and Authentication in Setup Mode

Note: It is necessary for **each** platform to have a web server if running in setup mode

Add list of web addresses of remote platforms in `$VOLTRON_HOME/external_addresses.json`

Start VOLTTTRON instances in setup mode in the three terminal windows. The “-l” option in the following command tells VOLTTTRON to log to a file. The file name should be different for each instance.

```
$ ./start-volttron --setup-mode
```

A new auth entry is added for each new platform connection. This can be checked with below command in each terminal window.

```
$ vctl auth list
```

After all the connections are authenticated, we can start the instances in normal mode.

```
$ ./stop-volttron
$ ./start-volttron
```

Setup Configuration and Authentication Manually

If you do not need web servers in your setup, then you will need to build the platform discovery config file manually. The config file should contain an entry containing VIP address, instance name and serverkey of each remote platform connection.

Name of the file: *external_platform_discovery.json*

Directory path: Each platform’s VOLTTTRON_HOME directory.

For example, since VOLTTTRON instance 1 is connecting to VOLTTTRON instance 2 and 3, contents of *external_platform_discovery.json* will be

```
{
  "platform2": { "vip-address": "tcp://127.0.0.2:22916",
    "instance-name": "platform2",
    "serverkey": "YFyIqXy2H7gIKC1x6uPMdDOB_i9lzfAPB1IgbxfXLGc"},
  "platform3": { "vip-address": "tcp://127.0.0.3:22916",
    "instance-name": "platform3",
    "serverkey": "hzU2bnlacAhZSaI0rI8a6XK_bqLSpA0JRK4jq8ttZxw"}
}
```

We can obtain the serverkey of each platform using below command in each terminal window:

```
$ vctl auth serverkey
```

Contents of `external_platform_discovery.json` of VOLTTRON instance 1, 2, 3 is shown below.

```
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $ cat ~/.volttron1/external_platform_discovery.json
{"platform2":{"vip-address":"tcp://127.0.0.2:22916","instance-name":"platform2","serverkey":"Obj-BezSuDBuBh30cn2nIdMH8s9jnFyc0II3ctcyllw"},
"platform3":{"vip-address":"tcp://127.0.0.3:22916","instance-name":"platform3","serverkey":"1_fc30-fUGIveh_33Pg7nwlBHEoDgwDifP08QzJnhxY"}}
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $ cat ~/.volttron2/external_platform_discovery.json
{"platform3":{"vip-address":"tcp://127.0.0.3:22916","instance-name":"platform3","serverkey":"1_fc30-fUGIveh_33Pg7nwlBHEoDgwDifP08QzJnhxY"},
"platform1":{"vip-address":"tcp://127.0.0.1:22916","instance-name":"platform1","serverkey":"bbBFTKXTmXGzIezVC47scqhwdlyc3WSBTCUzLDqynA"}}
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $ cat ~/.volttron3/external_platform_discovery.json
{"platform2":{"vip-address":"tcp://127.0.0.2:22916","instance-name":"platform2","serverkey":"Obj-BezSuDBuBh30cn2nIdMH8s9jnFyc0II3ctcyllw"},
"platform1":{"vip-address":"tcp://127.0.0.1:22916","instance-name":"platform1","serverkey":"bbBFTKXTmXGzIezVC47scqhwdlyc3WSBTCUzLDqynA"}}
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $ nano ~/.volttron3/external_platform_discovery.json
(volttron)volttron@volttron-VirtualBox ~/git/volttron-2 $
```

After this, you will need to add the server keys of the connecting platforms using the `vctl` utility. Type `vctl auth add` command on the command prompt and simply hit Enter to select defaults on all fields except **credentials**. Here, we can either add serverkey of connecting platform or type `./.*` to allow ALL connections.

Warning: `./.*` allows ALL agent and platform connections without authentication.

```
$ vctl auth add
domain []:
address []:
user_id []:
capabilities (delimit multiple entries with comma) []:
roles (delimit multiple entries with comma) []:
groups (delimit multiple entries with comma) []:
mechanism [CURVE]:
credentials []: ./.*
comments []:
enabled [True]:
added entry domain=None, address=None, mechanism='CURVE', credentials=u'./.*', user_
↪id=None
```

For more information on authentication see authentication.

Once the initial configuration are setup, you can start all the VOLTTRON instances in normal mode.

```
$ ./start-volttron
```

Next step is to start agents in each platform to observe the multi-platform PubSub communication behavior.

Start Master driver on VOLTRON instance 1

If master driver is not configured to auto start when the instance starts up, we can start it explicitly with this command.

```
$ vctl start --tag master_driver
```

Start Listener agents on VOLTRON instance 2 and 3

If the listener agent is not configured to auto start when the instance starts up, we can start it explicitly with this command.

```
$ vctl start --tag listener
```

We should start seeing messages with prefix="devices" in the logs of VOLTRON instances 2 and 3.

```

# Terminal 1 (Instance 1)
$ vctl start --tag master_driver
[1] 17668
(volttron@volttron-VirtualBox ~/git/volttron-2 $ volttron-ctl status
on -v -l 12.log
AGENT IDENTITY TAG STATUS
9 listeneragent-3.2 listeneragent-3.2.1 listener
(volttron@volttron-VirtualBox ~/git/volttron-2 $ volttron-ctl start --tag
master_driver
Starting 94674d-68de-4c8c-8653-13cf9d388e25 master_driveragent-3.1.1
volttron@volttron-VirtualBox ~/git/volttron-2 $ []

# Terminal 2 (Instance 2)
$ vctl start --tag listener
[1] 17630
(volttron@volttron-VirtualBox ~/git/volttron-2 $ volttron-ctl status
AGENT IDENTITY TAG STATUS
9 listeneragent-3.2 listeneragent-3.2.1 listener
(volttron@volttron-VirtualBox ~/git/volttron-2 $ volttron-ctl start --tag
listener
Starting ba38be29-e0b7-4d6b-b07a-4a393bb42f01 listeneragent-3.2
volttron@volttron-VirtualBox ~/git/volttron-2 $ []

# Terminal 3 (Instance 3)
$ vctl start --tag listener
[1] 17630
(volttron@volttron-VirtualBox ~/git/volttron-2 $ volttron-ctl status
AGENT IDENTITY TAG STATUS
9 listeneragent-3.2 listeneragent-3.2.1 listener
(volttron@volttron-VirtualBox ~/git/volttron-2 $ volttron-ctl start --tag
listener
Starting 94674d-68de-4c8c-8653-13cf9d388e25 master_driveragent-3.1.1
volttron@volttron-VirtualBox ~/git/volttron-2 $ []

# Log Output (Instance 2)
2017-10-02 16:04:24,416 () volttron.platform.auth INFO: authentication success: don
sine'vip', address='localhost:1000:17002', mechanism='CURVE', credentials='bb
3FfXtXkG2ieZvC47scqubdyC3wSdCUzLbDyNA', user_id='platform'
2017-10-02 16:04:52,273 () volttron.platform.auth INFO: authentication success: don
sine'vip', address='localhost:1000:17740', mechanism='CURVE', credentials='bb
3FfXtXkG2ieZvC47scqubdyC3wSdCUzLbDyNA', user_id='platform'
2017-10-02 16:04:52,280 () volttron.platform.aio INFO: starting agent /home/volttr
/volttron/agents/9f4674d-68de-4c8c-8653-13cf9d388e25/master_driveragent-3.1.1
2017-10-02 16:04:52,351 () volttron.platform.aio INFO: agent /home/volttron/volttr
on/agents/9f4674d-68de-4c8c-8653-13cf9d388e25/master_driveragent-3.1.1 has PID 17
54
2017-10-02 16:04:52,514 () volttron.platform.auth INFO: authentication success: don
sine'vip', address='localhost:1000:17754', mechanism='CURVE', credentials='XJ
88AeE5T3Vz16YmKZJm6RmIAIYdZk1yAHKas1', user_id='platform.driver'
2017-10-02 16:04:52,526 (master_driveragent-3.1.1 17754) volttron.platform.vip agen
t.core INFO: Connected to platform: router: ae708021-e22a-4bd3-a996-6ce5ad41ba2f ve
rsion: 3.0 identity: platform.driver
2017-10-02 16:04:52,540 (master_driveragent-3.1.1 17754) master_driver.agent INFO:
maximum concurrently open sockets limited to 600000 (derived from system limits)
2017-10-02 16:04:52,541 (master_driveragent-3.1.1 17754) master_driver.agent INFO:
maximum concurrent driver publishes limited to 10000
2017-10-02 16:04:52,541 (master_driveragent-3.1.1 17754) master_driver.agent INFO:
starting driver: fake-campus/fake-building/fake-device
'PowerState': {'type': 'Integer', 'tz': 'US/Pacific', 'units':
'1/0'},
'ValveState': {'type': 'Integer', 'tz': 'US/Pacific', 'units':
'1/0'},
'temperature': {'type': 'Integer',
'tz': 'US/Pacific',
'units': 'Fahrenheit'}}]]
2017-10-02 16:05:45,012 (listeneragent-3.2 17780) listener.agent
INFO: Peer: 'pubsub', Sender: 'platform.driver', Bus: 'u', Top
ic: 'devices/fake-campus/fake-building/fake-device/all', Headers
: {'Date': '2017-10-02T23:05:45.002978+00:00', 'Timestamp': '201
7-10-02T23:05:45.002978+00:00', 'min compatible version': '3.0',
'Heartbeat': True, 'PowerState': 0, 'ValveState': 0, 'temperat
ure': 50.0},
[{'Heartbeat': {'type': 'Integer', 'tz': 'US/Pacific', 'units':
'On/Off'},
'PowerState': {'type': 'Integer', 'tz': 'US/Pacific', 'units':
'1/0'},
'ValveState': {'type': 'Integer', 'tz': 'US/Pacific', 'units':
'1/0'},
'temperature': {'type': 'Integer',
'tz': 'US/Pacific',
'units': 'Fahrenheit'}}]]
2017-10-02 16:05:45,008 (listeneragent-3.2 17819) listener.agent INFO: Peer: 'pubsub
', Sender: 'platform.driver', Bus: 'u', Topic: 'devices/fake-campus/fake-building/f
ake-device/all', Headers: {'Date': '2017-10-02T23:05:45.002978+00:00', 'Timestamp':
'2017-10-02T23:05:45.002978+00:00', 'min compatible version': '3.0', 'max compatible
version': 'u'}, Message:
[{'Heartbeat': True, 'PowerState': 0, 'ValveState': 0, 'temperature': 50.0},
[{'Heartbeat': {'type': 'Integer', 'tz': 'US/Pacific', 'units': 'On/Off'},
'PowerState': {'type': 'Integer', 'tz': 'US/Pacific', 'units': '1/0'},
'ValveState': {'type': 'Integer', 'tz': 'US/Pacific', 'units': '1/0'},
'temperature': {'type': 'Integer',
'tz': 'US/Pacific',
'units': 'Fahrenheit'}}]]

```

Stopping All the Platforms

We can stop all the VOLTRON instances by executing below command in each terminal window.

```
$ vctl shutdown --platform
```

Platform External Address Configuration

In the configuration file located in `$VOLTRON_HOME/config` add `vip-address=tcp://ip:port` for each address you want to listen on:

```
Example
vip-address=tcp://127.0.0.102:8182
vip-address=tcp://127.0.0.103:8083
vip-address=tcp://127.0.0.103:8183
```

Note: The config file is generated after running the `vcfg` command. The VIP-address is for the local platform, NOT the remote platform.

Multi-platform RabbitMQ Deployment

With ZeroMQ based VOLTTRON, multi-platform communication was accomplished in three different ways:

1. Direct connection to remote instance - Write an agent that would connect to a remote instance directly.
2. Special agents - Use special agents such as forward historian/data puller agents that would forward/receive messages to/from remote instances. In RabbitMQ-VOLTTRON, we make use of the *shovel* plugin to achieve this behavior. Please refer to Shovel Plugin to get an overview of shovels.
3. Multi-Platform RPC and PubSub - Configure VIP address of all remote instances that an instance has to connect to in its `$VOLTTRON_HOME/external_discovery.json` and let the router module in each instance manage the connection and take care of the message routing for us. In RabbitMQ-VOLTTRON, we make use of the *federation* plugin to achieve this behavior. Please refer to Federation Plugin get an overview of federation.

Using the Federation Plugin

We can connect multiple VOLTTRON instances using the federation plugin. Before setting up federation links, we need to first identify the upstream server and downstream server. The upstream server is the node that is publishing some message of interest and downstream server is the node that wants to receive messages from the upstream server. A federation link needs to be established from the downstream VOLTTRON instance to the upstream VOLTTRON instance. To setup a federation link, we will need to add upstream server information in a RabbitMQ federation configuration file:

Path: `$VOLTTRON_HOME/rabbitmq_federation_config.yml`

```
# Mandatory parameters for federation setup
federation-upstream:
  rabbit-4:
    port: '5671'
    virtual-host: volttron4
  rabbit-5:
    port: '5671'
    virtual-host: volttron5
```

To configure the VOLTTRON instance to setup federation, run the following command:

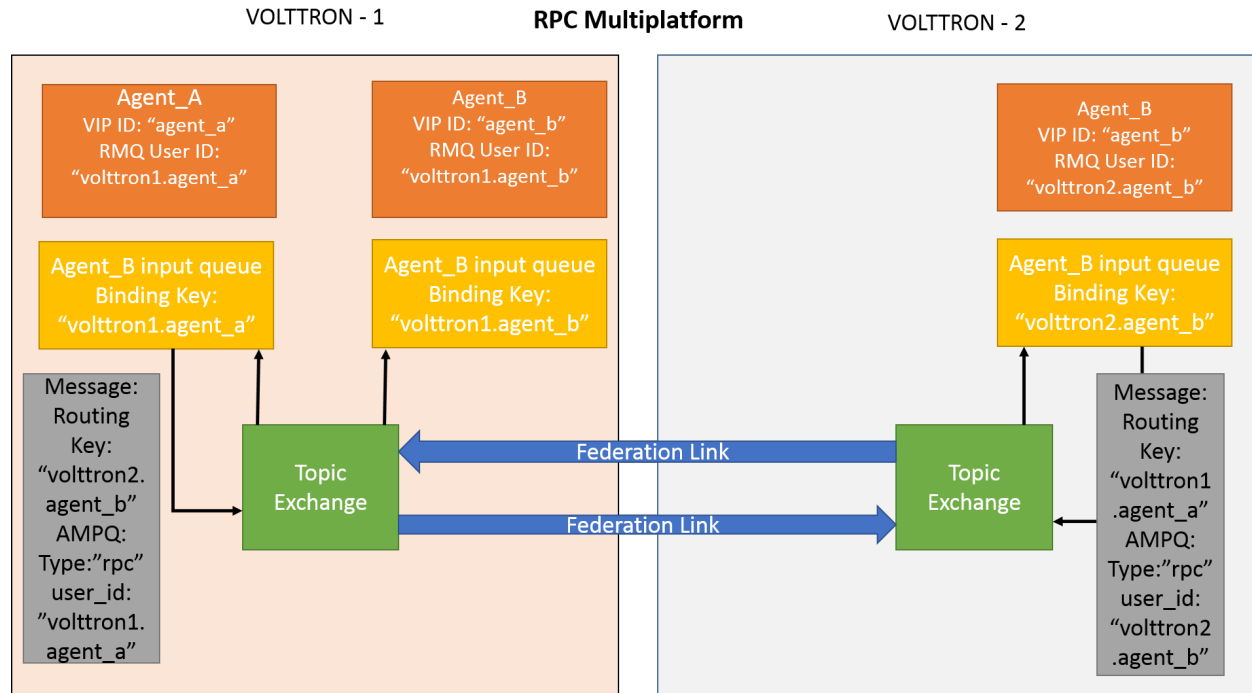
```
vcfg --rabbitmq federation [optional path to rabbitmq_federation_config.yml]
```

This will setup federation links to upstream servers and sets policy to make the VOLTTRON exchange *federated*. Once a federation link is established to remote instance, the messages published on the remote instance become available to local instance as if it were published on the local instance.

For detailed instructions to setup federation, please refer to the [platform installation docs](#).

Multi-Platform RPC With Federation

For multi-platform RPC communication, federation links need to be established on both the VOLTTTRON nodes. Once the federation links are established, RPC communication becomes fairly simple.



Consider Agent A on VOLTTTRON instance "volttron1" on host "host_A" wants to make RPC call to Agent B on VOLTTTRON instance "volttron2" on host "host_B".

1. Agent A makes RPC call.

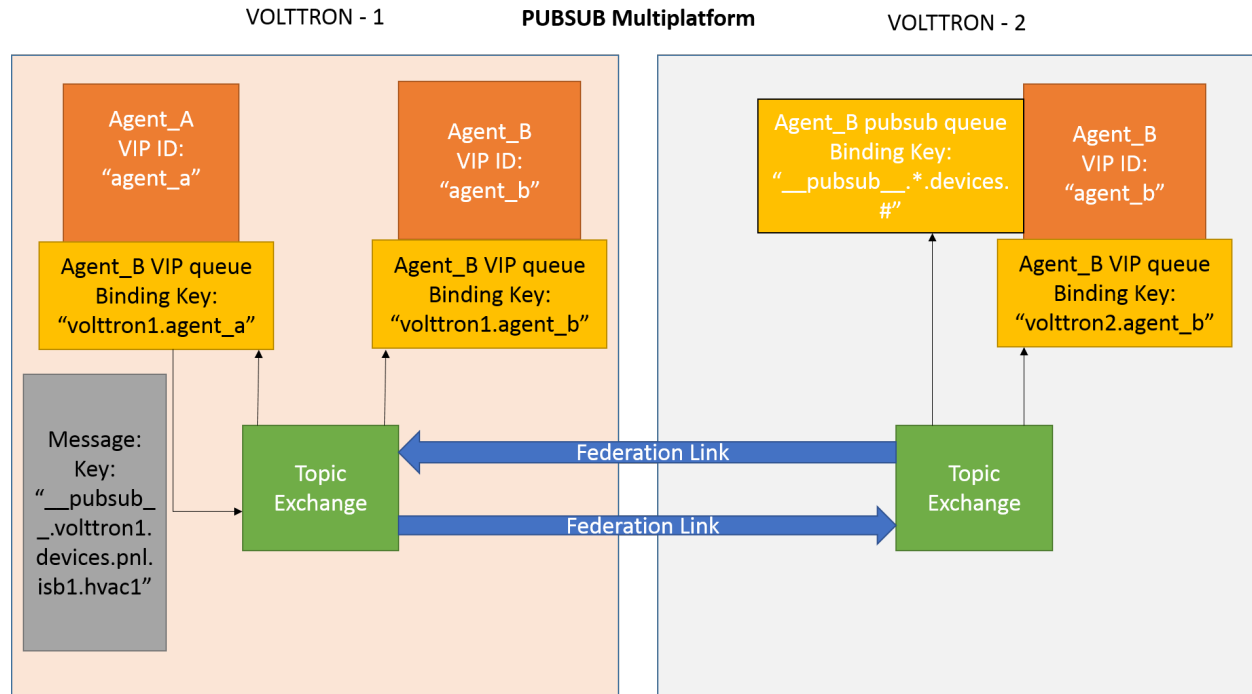
```

kwargs = {"external_platform": self.destination_instance_name}
agent_a.vip.rpc.call("agent_b", set_point, "point_name", 2.5, **kwargs)
  
```

2. The message is transferred over federation link to VOLTTTRON instance "volttron2" as both the exchanges are made *federated*.
3. The RPC subsystem of Agent B calls the actual RPC method and gets the result. It encapsulates the message result into a VIP message object and sends it back to Agent A on VOLTTTRON instance "volttron1".
4. The RPC subsystem on Agent A receives the message result and gives it to the Agent A application.

Multi-Platform PubSub With Federation

For multi-platform PubSub communication, it is sufficient to have federation link from the downstream server to the upstream server. In case of bi-directional data flow, links have to established in both the directions.



Consider Agent B on VOLTTRON instance “volttron2” on host “host_B” which wants to subscribe to messages from VOLTTRON instance “volttron2” on host “host_B”. First, a federation link needs to be established from “volttron2” to “volttron1”.

1. Agent B makes a subscribe call:

```
agent_b.vip.subscribe.call("pubsub", prefix="devices", all_platforms=True)
```

2. The PubSub subsystem converts the prefix to `__pubsub__.*.devices.#`. Here, “*” indicates that agent is subscribing to the “devices” topic from all VOLTTRON platforms.
3. A new queue is created and bound to VOLTTRON exchange with the above binding key. Since the VOLTTRON exchange is a *federated exchange*, any subscribed message on the upstream server becomes available on the federated exchange and Agent B will be able to receive it.
4. Agent A publishes message to topic `devices/pnl/isb1/hvac1`
5. The PubSub subsystem publishes this message on it’s VOLTTRON exchange.
6. Due to the federation link, message is received by the Pubsub subsystem of Agent A.

Using the Shovel Plugin

Shovels act as well written client applications which move messages from a source to a destination broker. The below configuration shows how to setup a shovel to forward PubSub messages or perform multi-platform RPC communication from local to a remote instance. It expects *hostname*, *port* and *virtual host* configuration values for the remote instance.

Path: `$VOLTTRON_HOME/rabbitmq_shovel_config.yml`

```
# Mandatory parameters for shovel setup
shovel:
  rabbit-2:
```

(continues on next page)

(continued from previous page)

```

port: '5671'
virtual-host: volttron
# Configuration to forward pubsub topics
pubsub:
    # Identity of agent that is publishing the topic
    platform.driver:
        - devices
# Configuration to make remote RPC calls
rpc:
    # Remote instance name
    volttron2:
        # List of pair of agent identities (local caller, remote callee)
        - [scheduler, platform.actuator]

```

To forward PubSub messages, the topic and agent identity of the publisher agent is needed. To perform RPC, the instance name of the remote instance and agent identities of the local agent and remote agent are needed.

To configure the VOLTTRON instance to setup shovel, run the following command.

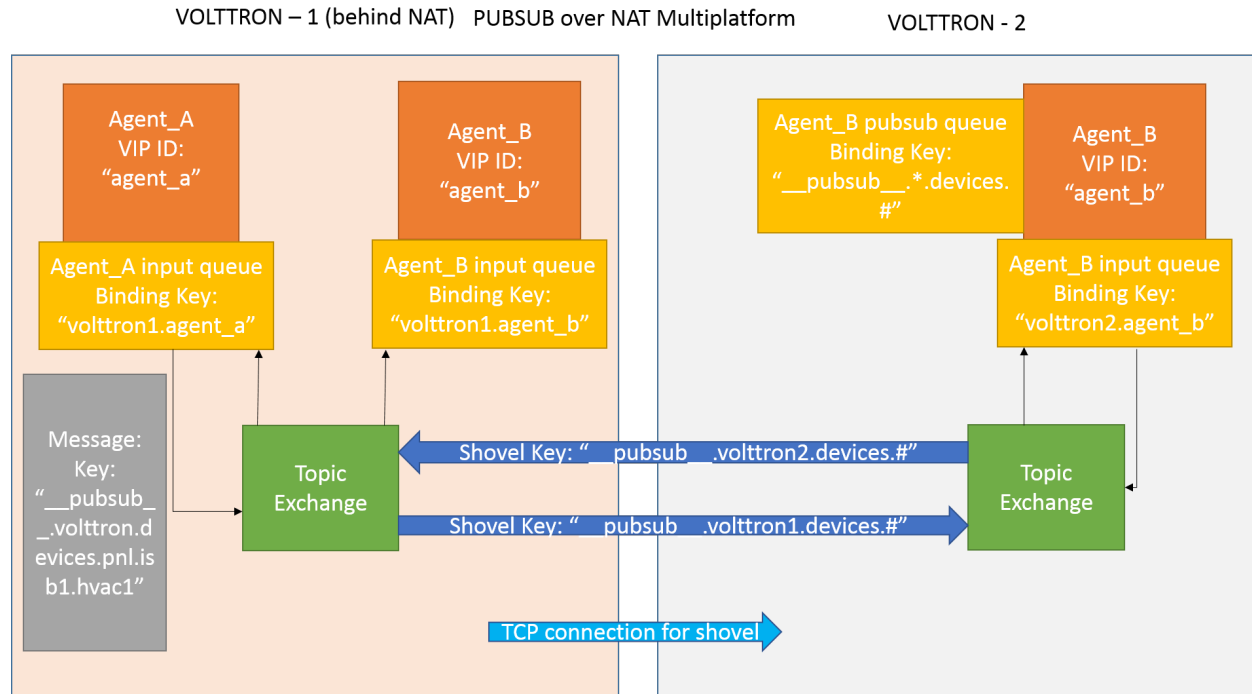
```
vcfg --rabbitmq shovel [optional path to rabbitmq_shovel_config.yml]
```

This setups up a shovel that forwards messages (either PubSub or RPC) from local exchange to remote exchange.

Multi-Platform PubSub With Shovel

After the shovel link is established for Pubsub, the below figure shows how the communication happens.

Note: For bi-directional pubsub communication, shovel links need to be created on both the nodes. The “blue” arrows show the shovel binding key. The pubsub topic configuration in `$VOLTTRON_HOME/rabbitmq_shovel_config.yml` gets internally converted to the shovel binding key: “`__pubsub__.<local instance name>.<actual topic>`”.



Now consider a case where shovels are setup in both the directions for forwarding “devices” topic.

1. Agent B makes a subscribe call to receive messages with topic “devices” from all connected platforms.

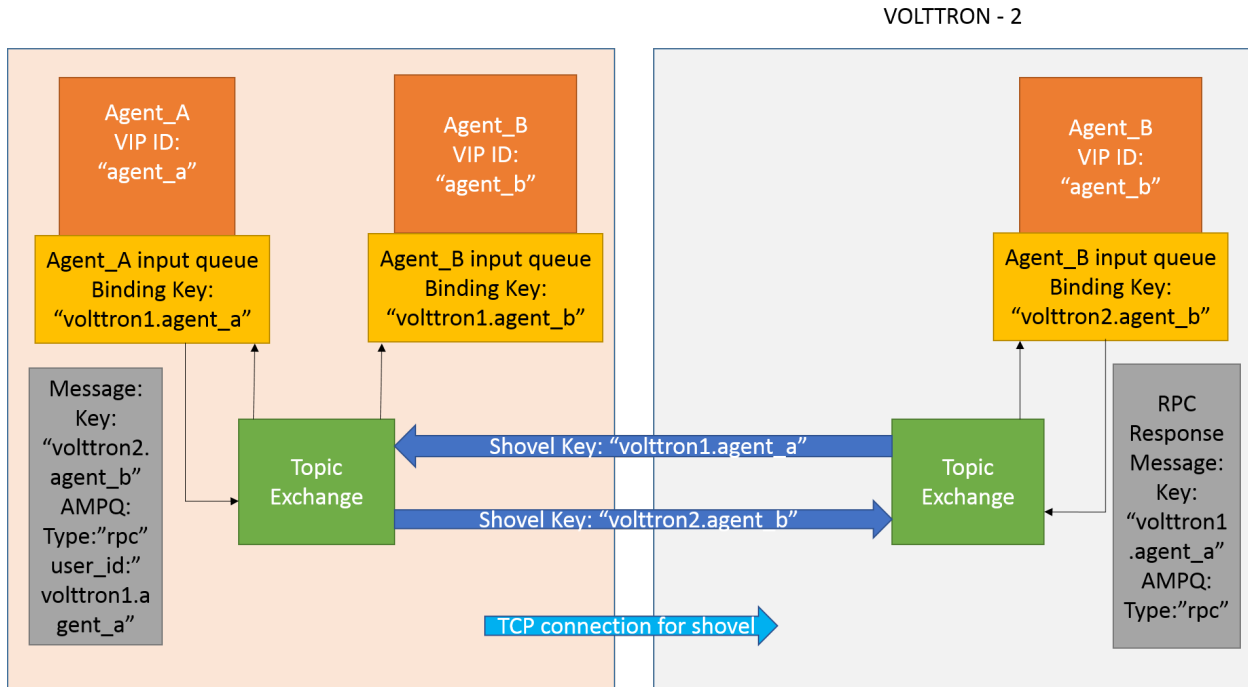
```
agent_b.vip.subscribe.call("pubsub", prefix="devices", all_platforms=True)
```

2. The PubSub subsystem converts the prefix to `__pubsub__.*.devices.#` “*” indicates that agent is subscribing to the “devices” topic from all the VOLTTRON platforms.
3. A new queue is created and bound to VOLTTRON exchange with above binding key.
4. Agent A publishes message to topic `devices/pnl/isb1/hvac1`
5. PubSub subsystem publishes this message on it’s VOLTTRON exchange.
6. Due to a shovel link from VOLTTRON instance “volttron1” to “volttron2”, the message is forwarded from VOLTTRON exchange “volttron1” to “volttron2” and is picked up by Agent A on “volttron2”.

Multi-Platform RPC With Shovel

After the shovel link is established for multi-platform RPC, the below figure shows how the RPC communication happens.

Note: It is mandatory to have shovel links on both directions as it is request-response type of communication. We will need to set the agent identities for caller and callee in the `$VOLTRON_HOME/rabbitmq_shovel_config.yml`. The “blue” arrows show the resulting the shovel binding key.



Consider Agent A on VOLTTRON instance "volttron1" on host "host_A" wants to make RPC call on Agent B on VOLTTRON instance "volttron2" on host "host_B".

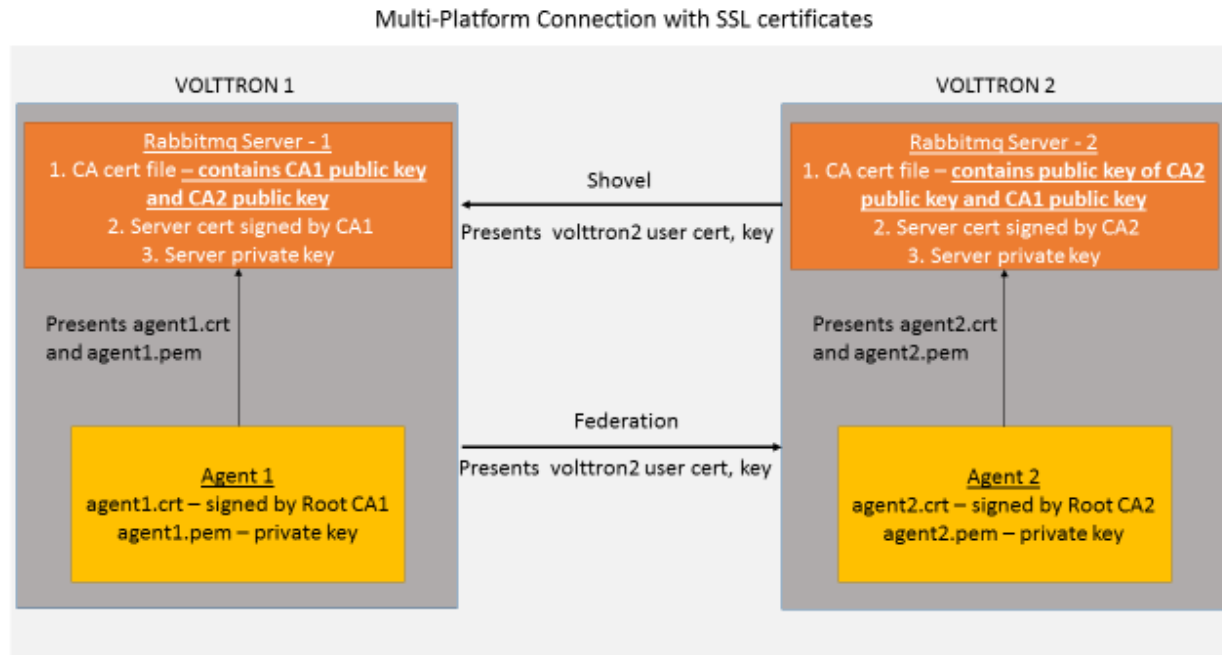
1. Agent A makes RPC call:

```
kwargs = {"external_platform": self.destination_instance_name}
agent_a.vip.rpc.call("agent_b", set_point, "point_name", 2.5, **kwargs)
```

2. The message is transferred over shovel link to VOLTTRON instance "volttron2".
3. The RPC subsystem of Agent B calls the actual RPC method and gets the result. It encapsulates the message result into a VIP message object and sends it back to Agent A on VOLTTRON instance "volttron1".
4. The RPC subsystem on Agent A receives the message result and gives it to Agent A's application.

Multi-Platform Communication With RabbitMQ SSL

For multi-platform communication over federation and shovel, we need the connecting instances to trust each other.



Suppose there are two VMs (VOLTTRON1 and VOLTTRON2) running single instances of RabbitMQ, and VOLTTRON1 and VOLTTRON2 want to talk to each other via either the federation or shovel plugins. In order for VOLTTRON1 to talk to VOLTTRON2, VOLTTRON1's root certificate must be appended to VOLTTRON2's trusted CA certificate, so that when VOLTTRON1 presents its root certificate during connection, VOLTTRON2's RabbitMQ server can trust the connection. VOLTTRON2's root CA must be appended to VOLTTRON1's root CA and it must in turn present its root certificate during connection, so that VOLTTRON1 will know it is safe to talk to VOLTTRON2.

Agents trying to connect to remote instance directly need to have a public certificate signed by the remote instance for authenticated SSL based connection. To facilitate this process, the VOLTTRON platform exposes a web based server API for requesting, listing, approving and denying certificate requests. For more detailed description, refer to Agent communication to Remote RabbitMQ instance

Multi-Platform Multi-Bus

This guide describes the setup process for a multi-platform connection that has a combination of ZeroMQ and RabbitMQ instances. For this example, we want to use the Forwarder to pass device data from two VOLTTRON instance to a single "central" instance for storage. It will also have a Volttron Central agent running on the "central" instance and Volttron Central Platform agents on all 3 instances and connected to "central" instance to provide operational status of its instance to the "central" instance. For this document "node" will be used interchangeably with VOLTTRON instance.

Node Setup

For this example we will have two types of nodes; a data collector and a central node. Each of the data collectors will have different message buses (VOLTTRON supports both RabbitMQ and ZeroMQ). The nodes will be configured as in the following table.

Table 2: Node Configuration

	Central	Node-ZMQ	Node-RMQ
Node Type	Central	Data Collector	Data Collector
Master Driver		yes	yes
Forwarder		yes	yes
SQL Historian	yes		
Volttron Central	yes		
Volttron Central Platform	yes	yes	yes
Exposes RMQ Port	yes		
Exposes ZMQ Port	yes		
Exposes HTTPS Port	yes		

The goal of this is to be able to see the data from Node-ZMQ and Node-RMQ in the Central SQL Historian and on the trending charts of Volttron Central.

Virtual Machine Setup

The first step in creating a VOLTTRON instance is to make sure the machine is ready for VOLTTRON. Each machine should have its hostname setup. For this walk-through, the hostnames “central”, “node-zmq” and “node-rmq” will be used.

For Central and Node-RMQ follow the instructions [platform installation steps for RMQ](#). For Node-ZMQ use [Platform Installation steps for ZeroMQ](#).

Instance Setup

The following conventions/assumptions are made for the rest of this document:

- Commands should be run from the VOLTTRON root
- Default values are used for VOLTTRON_HOME(\$HOME/.volttron), VIP port (22916), HTTPS port (8443), rabbitmq ports (5671 for AMQPs and 15671 for RabbitMQ management interface). If using different *VOLTTRON_HOME* or ports, please replace accordingly.
- Replace central, node-zmq and node-rmq with your own hostnames.
- user will represent your current user.

The following will use *vcfg* (volttron-cfg) to configure the individual platforms.

Central Instance Setup

Note: This instance must have been bootstrapped using `--rabbitmq` see [RabbitMQ installation instructions](#).

Next step would be to configure the instance to have a web interface to accept/deny incoming certificate signing requests from other instances. Additionally, we will need to install a Volttron Central agent, Volttron Central Platform agent, SQL historian agent and a Listener agent. The following shows an example command output for this setup.

```
(volttron)user@central:~/volttron$ vcfg

Your VOLTTRON_HOME currently set to: /home/user/.volttron

Is this the volttron you are attempting to setup? [Y]:
What type of message bus (rmq/zmq)? [zmq]: rmq
Name of this volttron instance: [volttron1]: central
RabbitMQ server home: [/home/user/rabbitmq_server/rabbitmq_server-3.7.7]:
Fully qualified domain name of the system: [central]:
Would you like to create a new self signed root CA certificate for this instance: [Y]:

Please enter the following details for root CA certificate
  Country: [US]:
  State: WA
  Location: Richland
  Organization: PNNL
  Organization Unit: volttron
Do you want to use default values for RabbitMQ home, ports, and virtual host: [Y]:
2020-04-13 13:29:36,347 rmq_setup.py INFO: Starting RabbitMQ server
2020-04-13 13:29:46,528 rmq_setup.py INFO: Rmq server at /home/user/rabbitmq_server/
↳rabbitmq_server-3.7.7 is running at
2020-04-13 13:29:46,554 volttron.utils.rmq_mgmt DEBUG: Creating new VIRTUAL HOST:↳
↳volttron
2020-04-13 13:29:46,582 volttron.utils.rmq_mgmt DEBUG: Create READ, WRITE and↳
↳CONFIGURE permissions for the user: central-admin
Create new exchange: volttron, {'durable': True, 'type': 'topic', 'arguments': {
↳'alternate-exchange': 'undeliverable'}}
Create new exchange: undeliverable, {'durable': True, 'type': 'fanout'}
2020-04-13 13:29:46,600 rmq_setup.py INFO:
Checking for CA certificate

2020-04-13 13:29:46,601 rmq_setup.py INFO:
  Creating root ca for volttron instance: /home/user/.volttron/certificates/certs/
↳central-root-ca.crt
2020-04-13 13:29:46,601 rmq_setup.py INFO: Creating root ca with the following info: {
↳'C': 'US', 'ST': 'WA', 'L': 'Richland', 'O': 'PNNL', 'OU': 'VOLTTRON', 'CN':
↳'central-root-ca'}
Created CA cert
2020-04-13 13:29:49,668 rmq_setup.py INFO: **Stopped rmq server
2020-04-13 13:30:00,556 rmq_setup.py INFO: Rmq server at /home/user/rabbitmq_server/
↳rabbitmq_server-3.7.7 is running at
2020-04-13 13:30:00,557 rmq_setup.py INFO:

#####

Setup complete for volttron home /home/user/.volttron with instance name=central
Notes:
  - On production environments, restrict write access to /home/user/.volttron/
↳certificates/certs/central-root-ca.crt to only admin user. For example: sudo chown↳
↳root /home/user/.volttron/certificates/certs/central-root-ca.crt and /home/user/.
↳volttron/certificates/certs/central-trusted-cas.crt
  - A new admin user was created with user name: central-admin and password=default_
↳passwd.
  You could change this user's password by logging into https://central:15671/↳
↳Please update /home/user/.volttron/rabbitmq_config.yml if you change password

#####
```

(continues on next page)

(continued from previous page)

```

The rmq message bus has a backward compatibility
layer with current zmq instances. What is the
zmq bus's vip address? [tcp://127.0.0.1]: tcp://192.168.56.101
What is the port for the vip address? [22916]:
Is this instance web enabled? [N]: y
Web address set to: https://central
What is the port for this instance? [8443]:
Is this an instance of voltttron central? [N]: y
Configuring /home/user/voltttron/services/core/VoltttronCentral.
Installing voltttron central.
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]: y
VC admin and password are set up using the admin web interface.
After starting VOLTTTRON, please go to https://central:8443/admin/login.html to
→complete the setup.
Will this instance be controlled by voltttron central? [Y]:
Configuring /home/user/voltttron/services/core/VoltttronCentralPlatform.
What is the name of this instance? [central]:
Voltttron central address set to https://central:8443
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]:
Would you like to install a platform historian? [N]: y
Configuring /home/user/voltttron/services/core/SQLHistorian.
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]: y
Would you like to install a master driver? [N]:
Would you like to install a listener agent? [N]: y
Configuring examples/ListenerAgent.
['voltttron', '-vv', '-l', '/home/user/.voltttron/voltttron.cfg.log']
Should the agent autostart? [N]: y
Finished configuration!

You can now start the voltttron instance.

If you need to change the instance configuration you can edit
the config file is at /home/user/.voltttron/config

```

Start VOLTTTRON instance and check if the agents are installed.

```

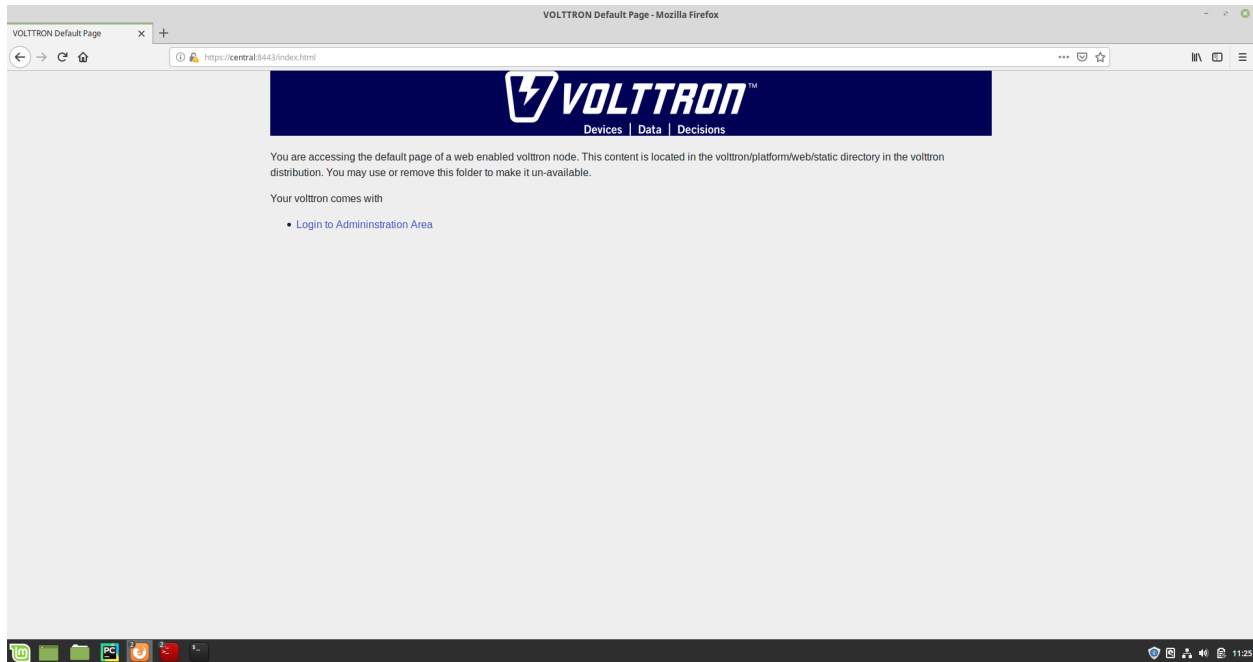
./start-voltttron
vctl status

```

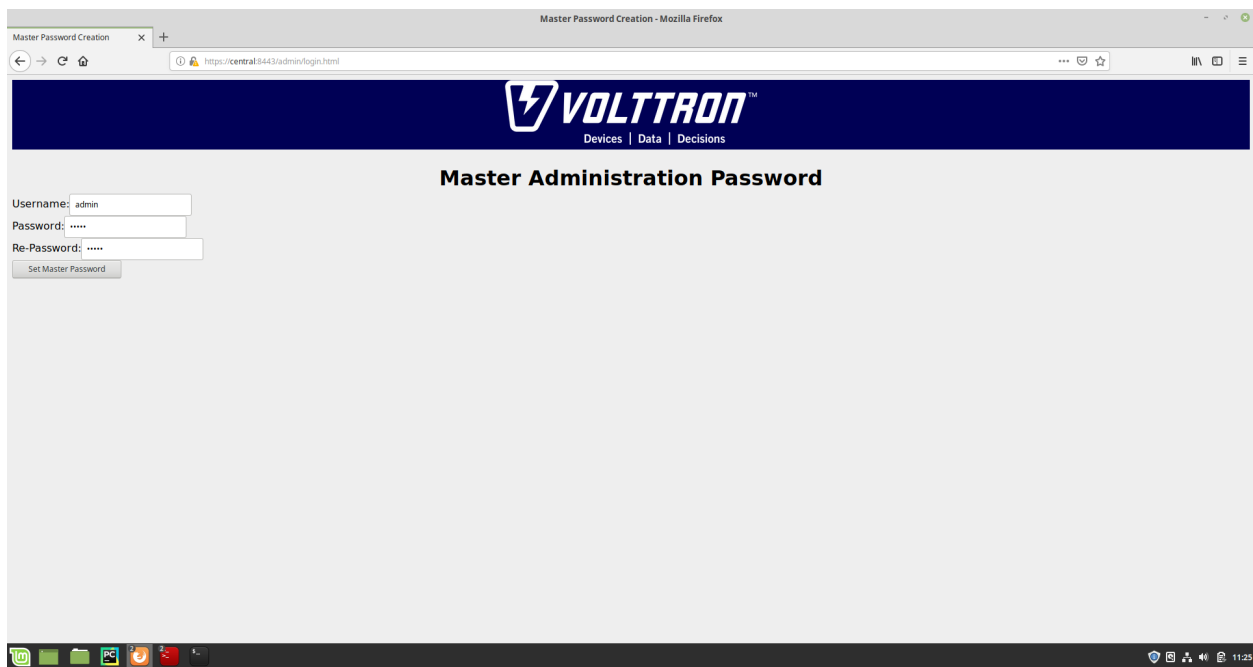
Open browser and go to master admin authentication page <https://central:8443/index.html> to accept/reject incoming certificate signing request (CSR) from other platforms.

Note: Replace “central” with the proper hostname of VC instance in the admin page URL. If opening the admin page from a different system, then please make that the hostname is resolvable in that machine.

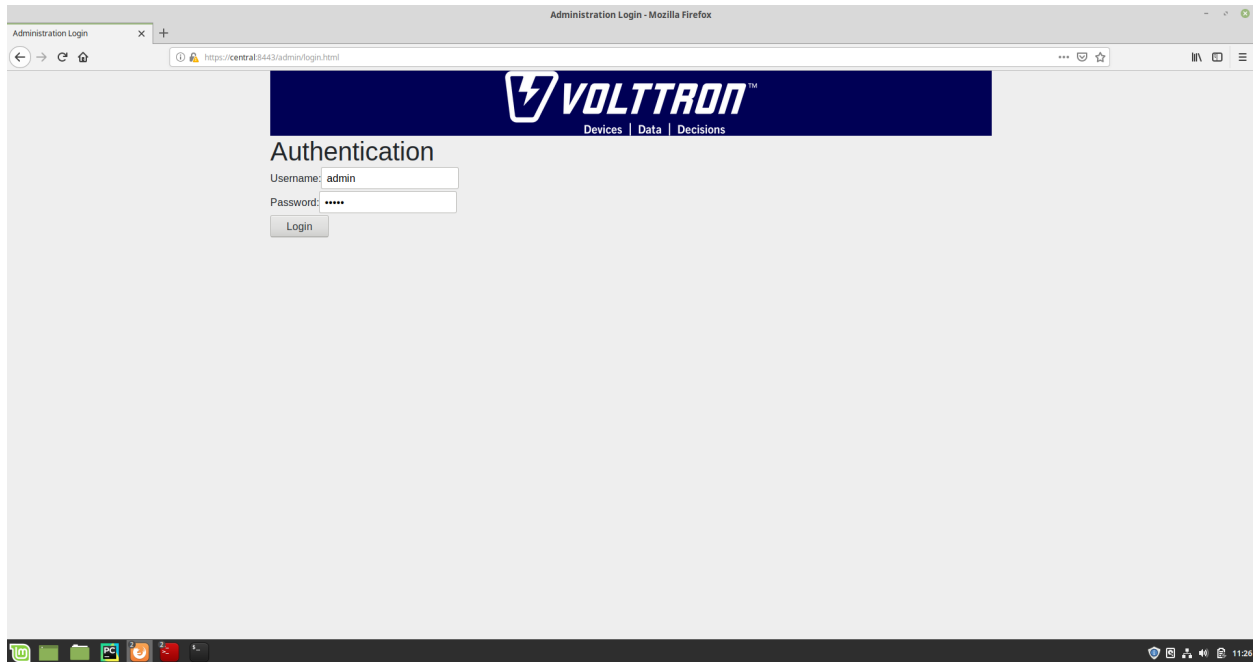
Click on “Login To Administration Area”.



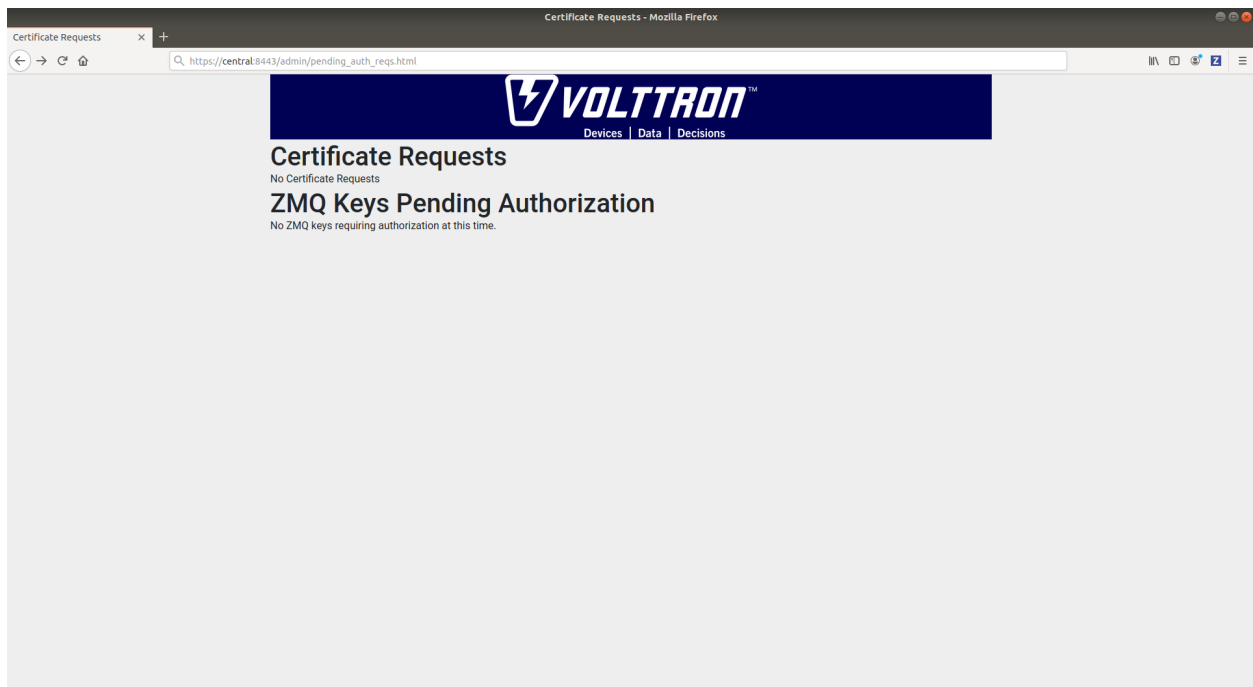
Set the master admin username and password. This can be later used to login into master admin authentication page. This username and password will also be used to log in to Volttron Central.



Login into the Master Admin page.



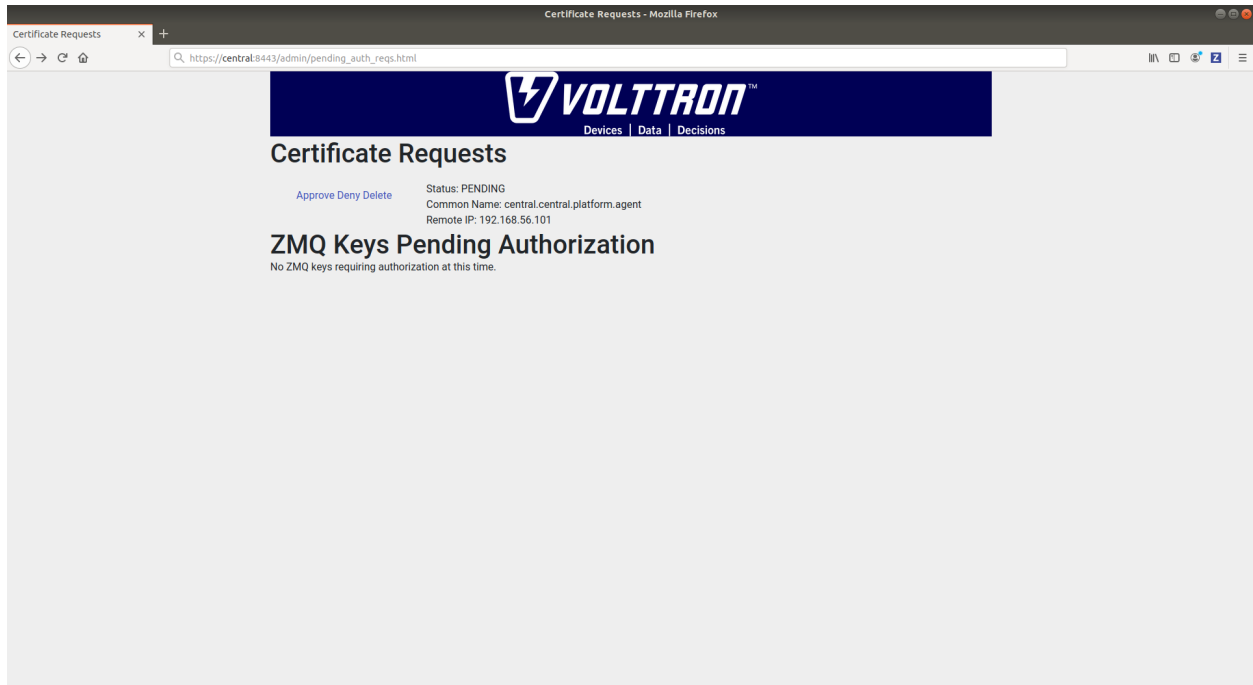
After logging in, you will see no CSR requests initially.



Go back to the terminal and start Volttron Central Platform agent on the “central” instance. The agent will send a CSR request to the web interface.

```
vctl start --tag vcp
```

Now go to master admin page to check if there is a new pending CSR request. You will see a “PENDING” request from “central.central.platform.agent”



Approve the CSR request to allow authenticated SSL based connection to the “central” instance.

Go back to the terminal and check the status of Volttron Central Platform agent. It should be set to “GOOD”.

Node-ZMQ Instance Setup

On the “node-zmq” VM, setup a ZeroMQ based VOLTTRON instance. Using “vcfg” command, install Volttron Central Platform agent, a master driver agent with a fake driver.

Note: This instance will use old ZeroMQ based authentication mechanism using CURVE keys.

```
(volttron)user@node-zmq:~/volttron$ vcfg

Your VOLTTRON_HOME currently set to: /home/user/.volttron

Is this the volttron you are attempting to setup? [Y]:
What type of message bus (rmq/zmq)? [zmq]:
What is the vip address? [tcp://127.0.0.1]:
What is the port for the vip address? [22916]:
Is this instance web enabled? [N]:
Will this instance be controlled by volttron central? [Y]:
Configuring /home/user/volttron/services/core/VolttronCentralPlatform.
What is the name of this instance? [volttron1]: collector1
What is the hostname for volttron central? [http://node-zmq]: https://central
What is the port for volttron central? [8080]: 8443
['volttron', '-vv', '-l', '/home/user/.volttron/volttron.cfg.log']
Should the agent autostart? [N]:
Would you like to install a platform historian? [N]:
Would you like to install a master driver? [N]: y
Configuring /home/user/volttron/services/core/MasterDriverAgent.
['volttron', '-vv', '-l', '/home/user/.volttron/volttron.cfg.log']
```

(continues on next page)

(continued from previous page)

```
Would you like to install a fake device on the master driver? [N]: y
Should the agent autostart? [N]: y
Would you like to install a listener agent? [N]:
Finished configuration!
```

You can now start the volttron instance.

If you need to change the instance configuration you can edit the config file is at /home/user/.volttron/config

Please note the Volttron Central web-address should point to that of the “central” instance.

Start VOLTRON instance and check if the agents are installed.

```
./start-volttron
vctl status
```

Start Volttron Central Platform on this platform manually.

```
vctl start --tag vcp
```

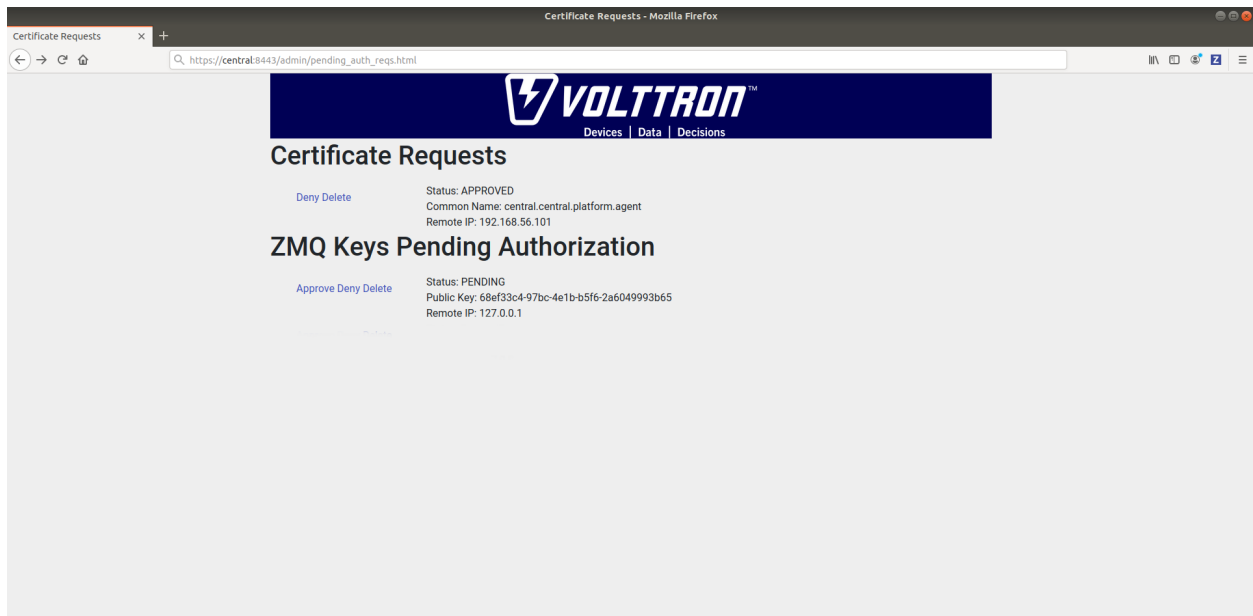
Check the VOLTRON log in the “central” instance, you will see “authentication failure” entry from the incoming connection. You will need to add the public key of VCP agent on the “central” instance.

```
2019-06-13 11:37:52,023 () volttron.platform.auth INFO: authentication failure: domain='vip', address='172.20.214.69', mechanism='CURVE', credentials=['f5XjmsCN n4MLV5uKpZmWn49ZzleRYinGrMpgeYmQwg']
```

At this point, you can either accept the connection through the admin page or the command line.

Using the admin page:

Navigate back to the master admin authentication page. You should see a pending request under the ZMQ Keys Pending Authorization header.



Accept the credential in the same method as a CSR.

Using the command line:

On the “node-zmq” box execute this command and grab the public key of the VCP agent.

```
vctl auth publickey
```

Add auth entry corresponding to VCP agent on “central” instance using the below command. Replace the user id value and credentials value appropriately before running

```
vctl auth add --user_id <any unique user id. for example zmq_node_vcp> --credentials  
↪ <public key of vcp on zmq node>
```

Complete similar steps to start a forwarder agent that connects to “central” instance. Modify the configuration in *services/core/ForwardHistorian/rmq_config.yml* to have a destination VIP address pointing to VIP address of the “central” instance and server key of the “central” instance.

```
---  
destination-vip: tcp://<ip>:22916  
destination-serverkey: <serverkey>
```

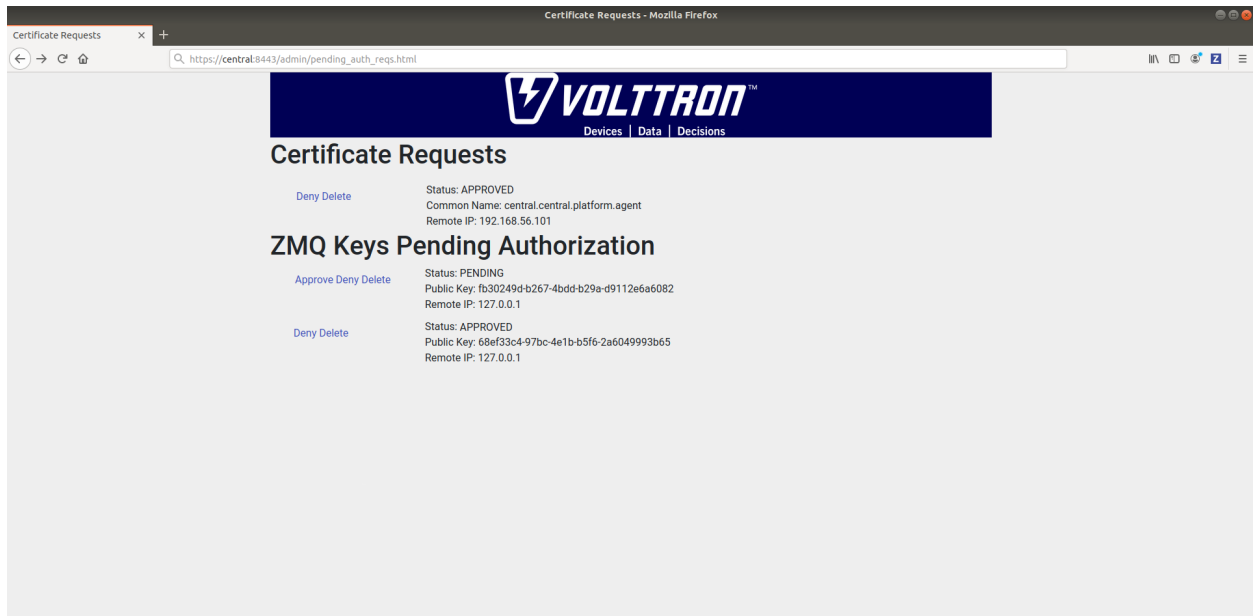
Note: Replace <ip> with public facing IP-address of “central” instance and <serverkey> with serverkey of “central” instance. Use the command **vctl auth serverkey** on the “central” instance to get the server key of the instance

Install and start forwarder agent.

```
python scripts/install-agent.py -s services/core/ForwardHistorian -c services/core/  
↪ ForwardHistorian/rmq_config.yml --start
```

To accept the credential using the admin page:

Navigate back to the master admin authentication page. You should see another pending request under the ZMQ Keys Pending Authorization header.



Accept this credential in the same method as before.

To accept the credential using the command line:

Grab the public key of the forwarder agent.

```
vctl auth publickey
```

Add auth entry corresponding to VCP agent on **central** instance.

```
vctl auth add --user_id <any unique user id. for example zmq_node_forwarder> --
↪credentials <public key of forwarder on zmq node>
```

In either case, you should start seeing messages from “collector1” instance on the “central” instance’s VOLTTTRON log now.

```
2019-06-13 12:02:45.099 (listeneragent-3.2 10403) listener.agent INFO: Peer: pubsub, Sender: proxy.router, Bus: , Topic: devices/fake-campus/fake-building/fake-device/
all, Headers: {'X-Forwarded': True, 'SynchronizedTimeStamp': '2019-06-13T19:02:45.000000+00:00', 'TimeStamp': '2019-06-13T19:02:45.001920+00:00', 'X-Forwarded-From': 'c
ollector1', 'Date': '2019-06-13T19:02:45.001920+00:00', 'min compatible version': '5.0', 'max compatible version': u''}, Message:
[{'Heartbeat': True, 'PowerState': 0, 'ValveState': 0, 'temperature': 50.0},
 {'Heartbeat': {'type': 'integer', 'tz': 'US/Pacific', 'units': 'On/Off'},
  'PowerState': {'type': 'integer', 'tz': 'US/Pacific', 'units': '1/0'},
  'ValveState': {'type': 'integer', 'tz': 'US/Pacific', 'units': '1/0'},
  'temperature': {'type': 'integer',
                  'tz': 'US/Pacific',
                  'units': 'Fahrenheit'}}]
2019-06-13 12:02:45.097 (listeneragent-3.2 10403) listener.agent INFO: Peer: pubsub, Sender: proxy.router, Bus: , Topic: devices/fake-campus/fake-building/fake-device/
all, Headers: {'X-Forwarded': True, 'SynchronizedTimeStamp': '2019-06-13T19:02:45.000000+00:00', 'TimeStamp': '2019-06-13T19:02:45.001920+00:00', 'X-Forwarded-From': 'c
ollector1', 'Date': '2019-06-13T19:02:45.001920+00:00', 'min compatible version': '5.0', 'max compatible version': u''}, Message:
[{'Heartbeat': True, 'PowerState': 0, 'ValveState': 0, 'temperature': 50.0},
 {'Heartbeat': {'type': 'integer', 'tz': 'US/Pacific', 'units': 'On/Off'},
  'PowerState': {'type': 'integer', 'tz': 'US/Pacific', 'units': '1/0'},
  'ValveState': {'type': 'integer', 'tz': 'US/Pacific', 'units': '1/0'},
  'temperature': {'type': 'integer',
                  'tz': 'US/Pacific',
                  'units': 'Fahrenheit'}}]
```

Node-RMQ Instance Setup

Note: This instance must have been bootstrapped using `–rabbitmq` see [RabbitMQ installation instructions](#).

Using “vcfg” command, install Volttron Central Platform agent, a master driver agent with fake driver. The instance name is set to “collector2”.

```
(volttron)user@node-rmq:~/volttron$ vcfg

Your VOLTTTRON_HOME currently set to: /home/user/.volttron

Is this the volttron you are attempting to setup? [Y]:
What type of message bus (rmq/zmq)? [zmq]: rmq
Name of this volttron instance: [volttron1]: collector2
RabbitMQ server home: [/home/user/rabbitmq_server/rabbitmq_server-3.7.7]:
Fully qualified domain name of the system: [node-rmq]:
Would you like to create a new self signed root CA certificate for this instance: [Y]:

Please enter the following details for root CA certificate
Country: [US]:
State: WA
Location: Richland
Organization: PNNL
Organization Unit: volttron
Do you want to use default values for RabbitMQ home, ports, and virtual host: [Y]:
2020-04-13 13:29:36,347 rmq_setup.py INFO: Starting RabbitMQ server
2020-04-13 13:29:46,528 rmq_setup.py INFO: Rmq server at /home/user/rabbitmq_server/
↪rabbitmq_server-3.7.7 is running at
2020-04-13 13:29:46,554 volttron.utils.rmq_mgmt DEBUG: Creating new VIRTUAL HOST:↪
↪volttron
2020-04-13 13:29:46,582 volttron.utils.rmq_mgmt DEBUG: Create READ, WRITE and↪
↪CONFIGURE permissions for the user: collector2-admin
Create new exchange: volttron, {'durable': True, 'type': 'topic', 'arguments': {
↪'alternate-exchange': 'undeliverable'}} (continues on next page)
```

(continued from previous page)

```

Create new exchange: undeliverable, {'durable': True, 'type': 'fanout'}
2020-04-13 13:29:46,600 rmq_setup.py INFO:
Checking for CA certificate

2020-04-13 13:29:46,601 rmq_setup.py INFO:
  Creating root ca for volttron instance: /home/user/.volttron/certificates/certs/
  ↳ collector2-root-ca.crt
2020-04-13 13:29:46,601 rmq_setup.py INFO: Creating root ca with the following info: {
  ↳ 'C': 'US', 'ST': 'WA', 'L': 'Richland', 'O': 'PNNL', 'OU': 'VOLTTRON', 'CN':
  ↳ 'collector2-root-ca'}
Created CA cert
2020-04-13 13:29:49,668 rmq_setup.py INFO: **Stopped rmq server
2020-04-13 13:30:00,556 rmq_setup.py INFO: Rmq server at /home/user/rabbitmq_server/
  ↳ rabbitmq_server-3.7.7 is running at
2020-04-13 13:30:00,557 rmq_setup.py INFO:

#####

Setup complete for volttron home /home/user/.volttron with instance name=collector2
Notes:
  - On production environments, restrict write access to /home/user/.volttron/
  ↳ certificates/certs/collector2-root-ca.crt to only admin user. For example: sudo_
  ↳ chown root /home/user/.volttron/certificates/certs/collector2-root-ca.crt and /home/
  ↳ user/.volttron/certificates/certs/collector2-trusted-cas.crt
  - A new admin user was created with user name: collector2-admin and password=default_
  ↳ passwd.
  You could change this user's password by logging into https://node-rmq:15671/_
  ↳ Please update /home/user/.volttron/rabbitmq_config.yml if you change password

#####

The rmq message bus has a backward compatibility
layer with current zmq instances. What is the
zmq bus's vip address? [tcp://127.0.0.1]:
What is the port for the vip address? [22916]:
Is this instance web enabled? [N]:
Will this instance be controlled by volttron central? [Y]:
Configuring /home/user/volttron/services/core/VolttronCentralPlatform.
What is the name of this instance? [collector2]:
What is the hostname for volttron central? [http://node-rmq]: https://central
What is the port for volttron central? [8443]:
['volttron', '-vv', '-l', '/home/user/.volttron/volttron.cfg.log']
Should the agent autostart? [N]:
Would you like to install a platform historian? [N]:
Would you like to install a master driver? [N]: y
Configuring /home/user/volttron/services/core/MasterDriverAgent.
['volttron', '-vv', '-l', '/home/user/.volttron/volttron.cfg.log']
Would you like to install a fake device on the master driver? [N]: y
Should the agent autostart? [N]: y
Would you like to install a listener agent? [N]:
Finished configuration!

You can now start the volttron instance.

If you need to change the instance configuration you can edit
the config file is at /home/user/.volttron/config

```

Note: The Volttron Central web-address should point to that of the “central” instance.

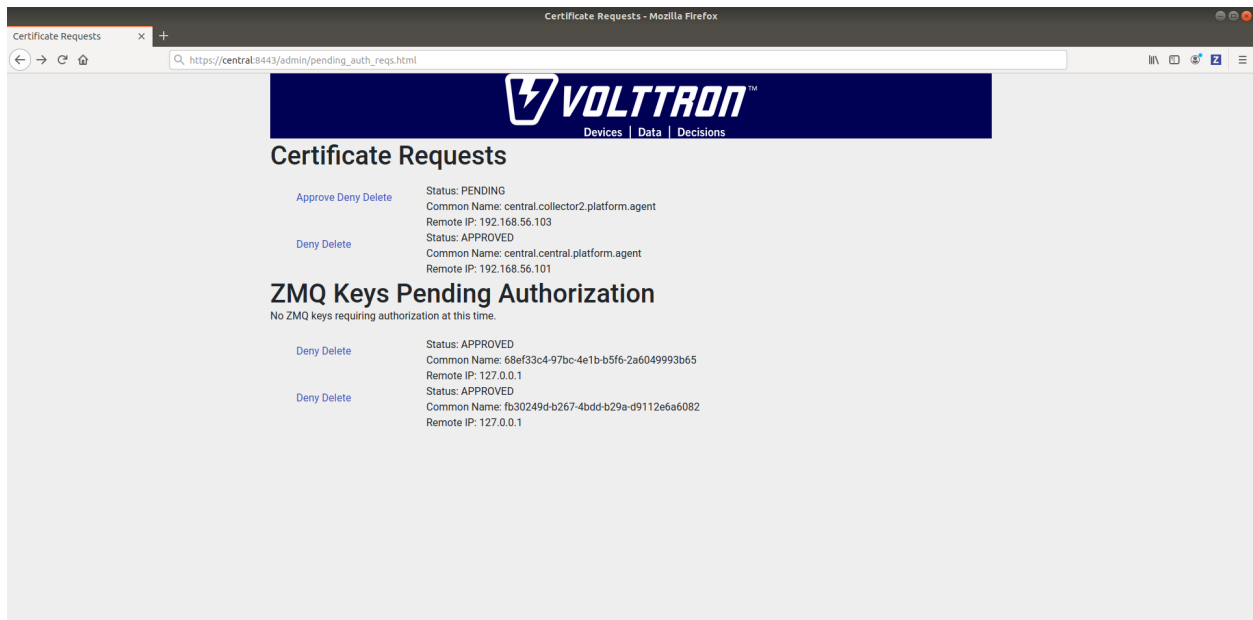
Start VOLTTTRON instance and check if the agents are installed.

```
./start-voltttron
vctl status
```

Start Volttron Central Platform on this platform manually.

```
vctl start --tag vcp
```

Go the master admin authentication page and check if there is a new pending CSR request from VCP agent of “collector2” instance.



Approve the CSR request to allow authenticated SSL based connection to the “central” instance.

Now go back to the terminal and check the status of Volttron Central Platform agent. It should be set to “GOOD”.

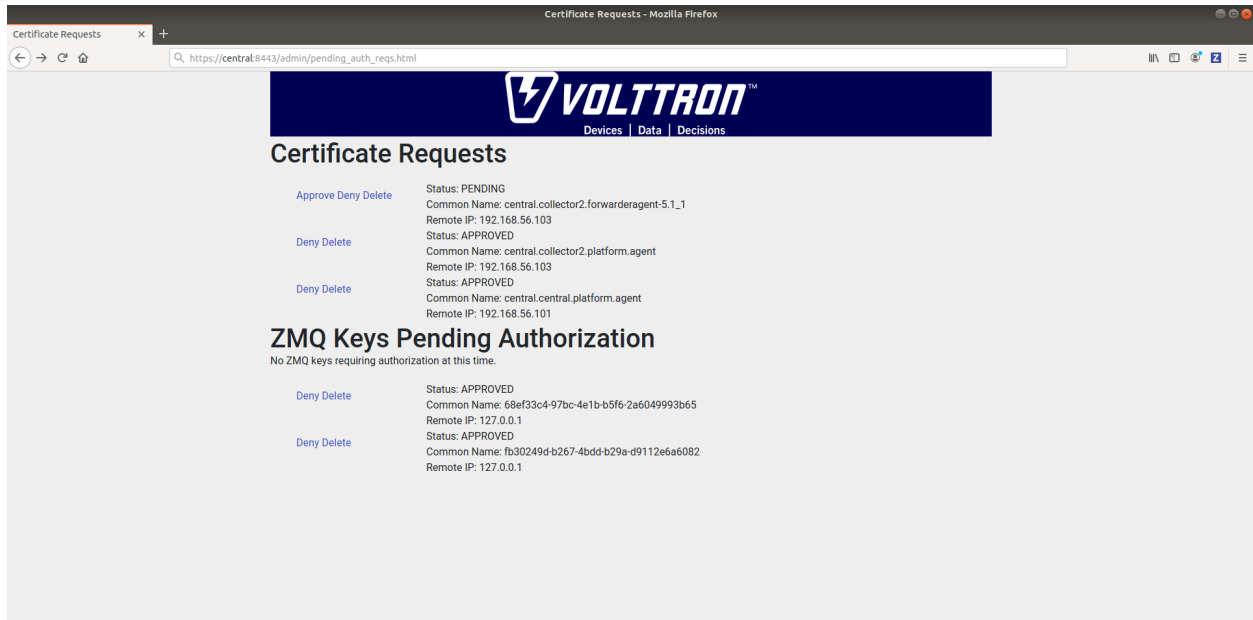
Let’s now install a forwarder agent on this instance to forward local messages matching “devices” topic to external “central” instance. Modify the configuration in `services/core/ForwardHistorian/rmq_config.yml` to have a destination address pointing to web address of the “central” instance.

```
---
destination-address: https://central:8443
```

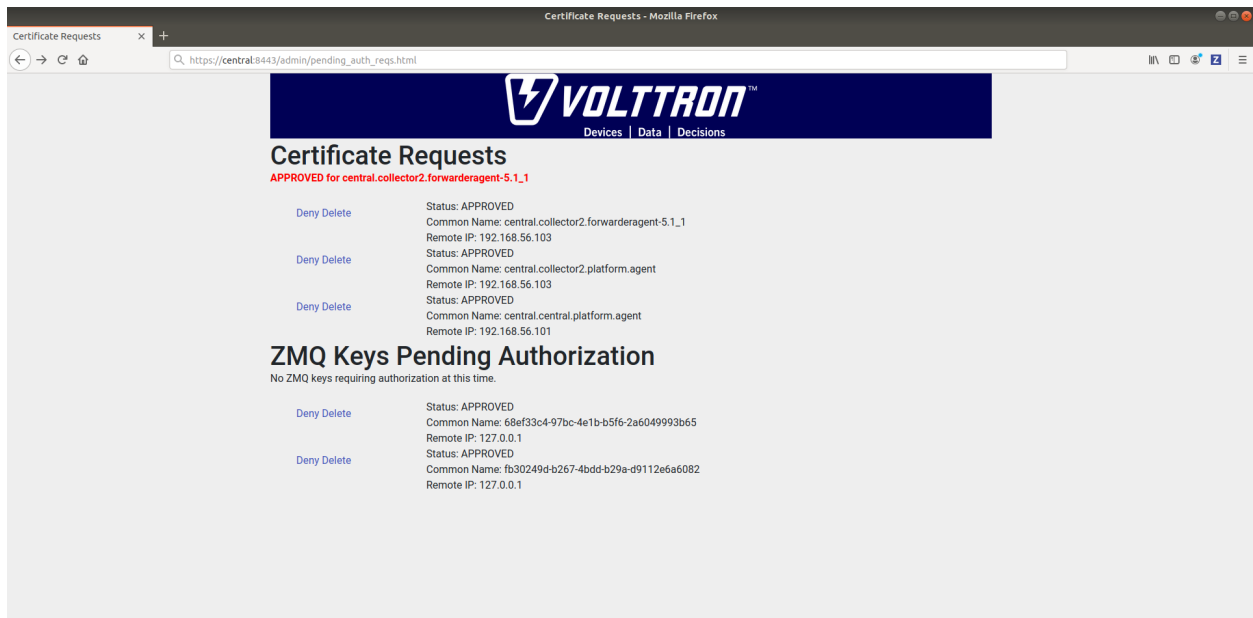
Start forwarder agent.

```
python scripts/install-agent.py -s services/core/ForwardHistorian -c services/core/
  ↳ ForwardHistorian/rmq_config.yml --start
```

Go the master admin authentication page and check if there is a new pending CSR request from forwarder agent of “collector2” instance.



Approve the CSR request to allow authenticated SSL based connection to the “central” instance.

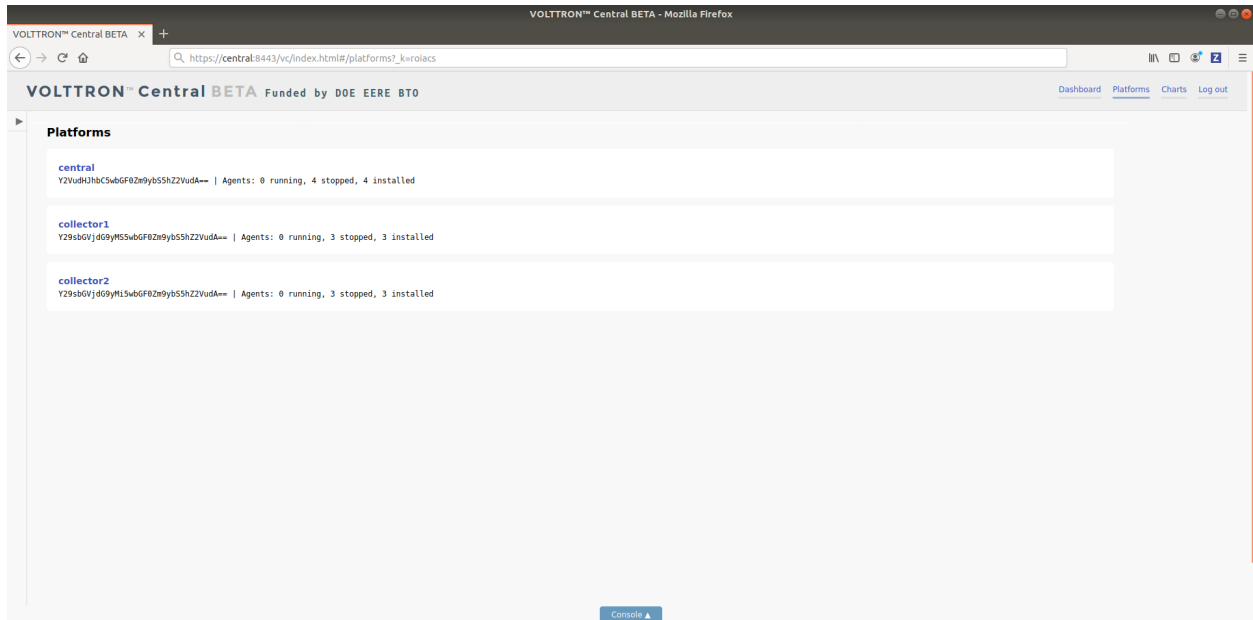


Now go back to the terminal and check the status of forwarder agent. It should be set to “GOOD”.

Check the VOLTTRON log of “central” instance. You should see messages with “devices” topic coming from “collector2” instance.

```
2019-06-13 12:00:00.025 [listeneragent-3.2 10403] listener.agent INFO: Peer: pubsub, Sender: collector2.forwarderagent-5.1.1, Bus: , Topic: devices/fake-campus/fake-building/fake-device/all, Headers: {'X-Forwarded': True, 'SynchronizedTimestamp': '2019-06-13T19:00:00.000000+00:00', 'Timestamp': '2019-06-13T19:00:00.004159+00:00', 'X-Forwarded-From': 'collector2', 'Date': '2019-06-13T19:00:00.004159+00:00', 'min-compatible-version': '5.0', 'max-compatible-version': 'u''}, Message:
[{'Heartbeat': True, 'PowerState': 0, 'ValveState': 0, 'temperature': 50.0},
 {'Heartbeat': {'type': 'integer', 'tz': 'US/Pacific', 'units': 'on/off'},
  'PowerState': {'type': 'integer', 'tz': 'US/Pacific', 'units': '1/0'},
  'ValveState': {'type': 'integer', 'tz': 'US/Pacific', 'units': '1/0'},
  'temperature': {'type': 'integer',
                  'tz': 'US/Pacific',
                  'units': 'Fahrenheit'}}]
```

To confirm that VolttronCentral is monitoring the status of all the 3 platforms, open a browser and type this URL <https://central:8443/vc/index.html>. Login using credentials (username and password) earlier set during the VC configuration step (using vcfg command in “central” instance). Click on “platforms” tab in the far right corner. You should see all three platforms listed in that page. Click on each of the platforms and check the status of the agents.



VOLTTTRON Central Deployment

VOLTTTRON Central is a platform management web application that allows platforms to communicate and to be managed from a centralized server. This agent alleviates the need to ssh into independent nodes in order to manage them. The demo will start up three different instances of VOLTTTRON with three historians and different agents on each host. The following entries will help to navigate around the VOLTTTRON Central interface.

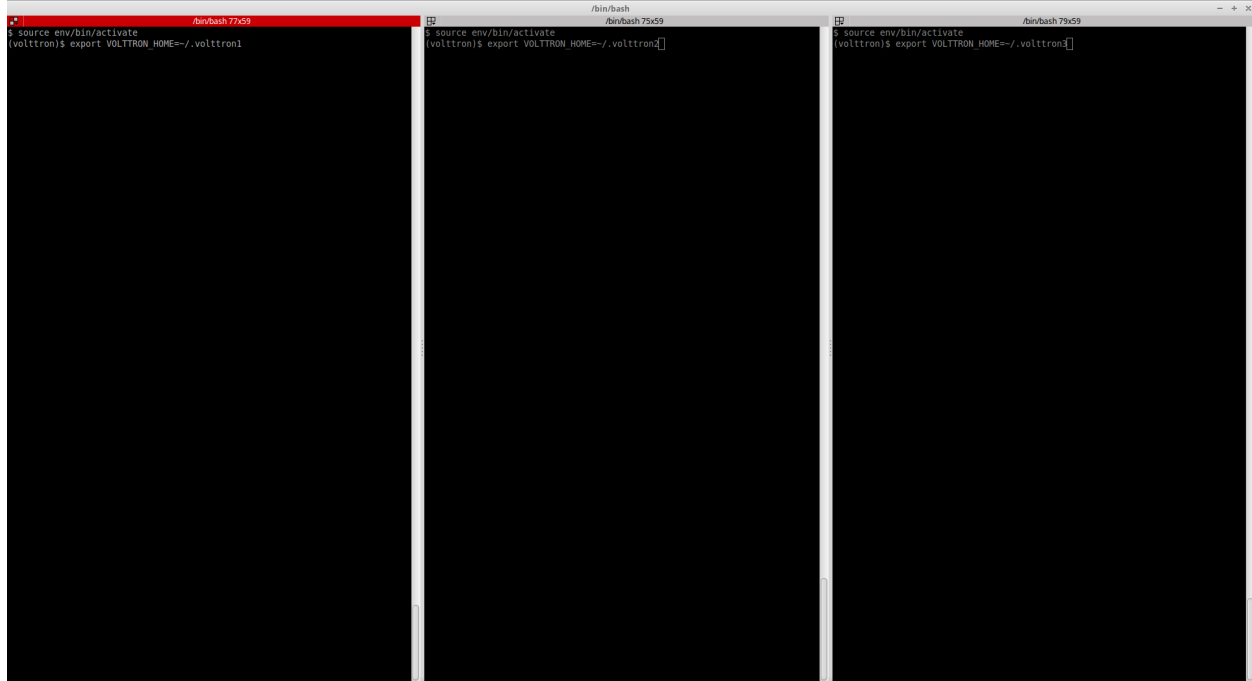
- *Getting Started*
- *Remote Platform Configuration*
- *Starting the Demo*
- *Stopping the Demo*
- *Log In*
- *Log Out*
- *Platforms Tree*
- *Loading the Tree*
- *Health Status*
- *Filter the Tree*
- *Platforms Screen*
- *Register New Platform*
- *Deregister Platform*
- *Platform View*
- *Add Charts*
- *Dashboard Charts*
- *Remove Charts*

Getting Started

After *installing VOLTTRON*, open three shells with the current directory the root of the VOLTTRON repository. Then activate the VOLTTRON environment and export the `VOLTTRON_HOME` variable. The home variable needs to be different for each instance.

If you are using Terminator you can right click and select “Split Vertically”. This helps us keep from losing terminal windows or duplicating work.

```
$ source env/bin/activate
$ export VOLTTRON_HOME=~/.volttron1
```



One of our instances will have a VOLTTRON Central agent. We will install a platform agent and a historian on all three platforms. Please note, for this demo all the instances run on the ZeroMQ message bus. For multi-platform, multi-bus deployment setup please follow the steps described in [Multi Platform Multi-Bus Deployment](#).

Run `vcfg` in the first shell. This command will ask how the instance should be set up. Many of the options have defaults that will be sufficient. When asked if this instance is a VOLTTRON Central enter `y`. Read through the options and use the enter key to accept default options. There are no default credentials for VOLTTRON Central. You can have it install the agents at this time. Below is an example configuration. In this case, username is `user` and localhost is `volttron-pc`.

```
(volttron)user@volttron-pc:~/volttron$ vcfg

Your VOLTTRON_HOME currently set to: /home/user/.volttron1

Is this the volttron you are attempting to setup? [Y]:
What type of message bus (rmq/zmq)? [zmq]:
What is the vip address? [tcp://127.0.0.1]:
What is the port for the vip address? [22916]:
Is this instance web enabled? [N]: y
What is the protocol for this instance? [https]:
Web address set to: https://volttron-pc
```

(continues on next page)

(continued from previous page)

```

What is the port for this instance? [8443]:
Would you like to generate a new web certificate? [Y]:
WARNING! CA certificate does not exist.
Create new root CA? [Y]:

Please enter the following details for web server certificate:
    Country: [US]:
    State: WA
    Location: Richland
    Organization: PNNL
    Organization Unit: VOLTTRON
Created CA cert
Creating new web server certificate.
Is this an instance of volttron central? [N]: y
Configuring /home/user/volttron/services/core/VolttronCentral.
Installing volttron central.
Should the agent autostart? [N]: y
VC admin and password are set up using the admin web interface.
After starting VOLTTRON, please go to https://volttron-pc:8443/admin/login.
→html to complete the setup.
Will this instance be controlled by volttron central? [Y]: y
Configuring /home/user/volttron/services/core/VolttronCentralPlatform.
What is the name of this instance? [volttron1]:
Volttron central address set to https://volttron-pc:8443
Should the agent autostart? [N]: y
Would you like to install a platform historian? [N]: y
Configuring /home/user/volttron/services/core/SQLHistorian.
Should the agent autostart? [N]: y
Would you like to install a master driver? [N]: y
Configuring /home/user/volttron/services/core/MasterDriverAgent.
Would you like to install a fake device on the master driver? [N]: y
Should the agent autostart? [N]: y
Would you like to install a listener agent? [N]: y
Configuring examples/ListenerAgent.
Should the agent autostart? [N]: y
Finished configuration!

You can now start the volttron instance.

If you need to change the instance configuration you can edit
the config file is at /home/user/.volttron1/config

(volttron)user@volttron-pc:~/volttron$

```

VOLTTRON Central needs to accept the connecting instances' public keys. For this example we'll allow any CURVE credentials to be accepted. After *starting*, the command **vctl auth add** will prompt the user for information about how the credentials should be used. We can simply hit Enter to select defaults on all fields except **credentials**, where we will type `./.*`

```

$ vctl auth add --credentials "/*.*/"
added entry domain=None, address=None, mechanism='CURVE', credentials=u'/*.*/', user_
→id='63b126a7-2941-4ebe-8588-711d1e6c70d1'

```

For more information on authorization see authentication.

Remote Platform Configuration

The next step is to configure the instances that will connect to VOLTTTRON Central. In the second and third terminal windows run `vcfg`. Like the `VOLTTTRON_HOME` variable, these instances need to have unique VIP addresses and unique instance names.

Install a platform agent and a historian as before. Since we used the default options when configuring VOLTTTRON Central, we can use the default options when configuring these platform agents as well. The configuration will be a little different. The example below is for the second volttron instance. Note the unique VIP address and instance name. Please ensure the web-address of the volttron central is configured correctly.

```
(voltttron)user@voltttron-pc:~/voltttron$ vcfg

Your VOLTTTRON_HOME currently set to: /home/user/.voltttron2

Is this the voltttron you are attempting to setup? [Y]:
What type of message bus (rmq/zmq)? [zmq]:
What is the vip address? [tcp://127.0.0.1]: tcp://127.0.0.2
What is the port for the vip address? [22916]:
Is this instance web enabled? [N]:
Will this instance be controlled by voltttron central? [Y]:
Configuring /home/user/voltttron/services/core/VoltttronCentralPlatform.
What is the name of this instance? [voltttron1]: voltttron2
What is the hostname for voltttron central? [https://voltttron-pc]:
What is the port for voltttron central? [8443]:
Should the agent autostart? [N]: y
Would you like to install a platform historian? [N]: y
Configuring /home/user/voltttron/services/core/SQLHistorian.
Should the agent autostart? [N]: y
Would you like to install a master driver? [N]:
Would you like to install a listener agent? [N]:
Finished configuration!

You can now start the voltttron instance.

If you need to change the instance configuration you can edit
the config file is at /home/user/.voltttron2/config

(voltttron)user@voltttron-pc:~/voltttron$
```

Starting the Demo

Start each Voltttron instance after configuration. You have two options.

Option 1: The following command starts the volttron process in the background. The “-l” option tells volttron to log to a file. The file name should be different for each instance.

```
$ volttron -vv -l volttron.log&
```

Option 2: Use the utility script `start-volttron`. This will override the default log file each time the script is ran unless the script is modified with a different filename for each instance.

```
$ ./start-volttron
```

Note: If you chose to not start your agents with their platforms they will need to be started by hand.

List the installed agents with

```
$ vctl status
```

A portion of each agent's uuid makes up the leftmost column of the status output. This is all that is needed to start or stop the agent. If any installed agents share a common prefix then more of the uuid will be needed to identify it.

```
$ vctl start uuid
```

or

```
$ vctl start --tag tag
```

Note: In each of the above examples one could use * suffix to match more than one agent.

VOLTTRON Admin

The admin page is used to set the master username and password for both admin page and VOLTTRON Central page. Admin page can then be used to manage RMQ and ZMQ certificates and credentials.

Open a web browser and navigate to <https://volttron-pc:8443/admin/login.html>

There may be a message warning about a potential security risk. Check to see if the certificate that was created in vcfg is being used. The process below is for firefox.



Warning: Potential Security Risk Ahead

Firefox detected a potential security threat and did not continue to `volttron-pc`. If you visit this site, attackers could try to steal information like your passwords, emails, or credit card details.

What can you do about it?

The issue is most likely with the website, and there is nothing you can do to resolve it.

If you are on a corporate network or using anti-virus software, you can reach out to the support teams for assistance. You can also notify the website's administrator about the problem.

[Learn more...](#)

Go Back (Recommended)

Advanced...

☐ Report errors like this to help Mozilla identify and block malicious sites



Someone could be trying to impersonate the site and you should not continue.

Websites prove their identity via certificates. Firefox does not trust `volttron-pc :8443` because its certificate issuer is unknown, the certificate is self-signed, or the server is not sending the correct intermediate certificates.

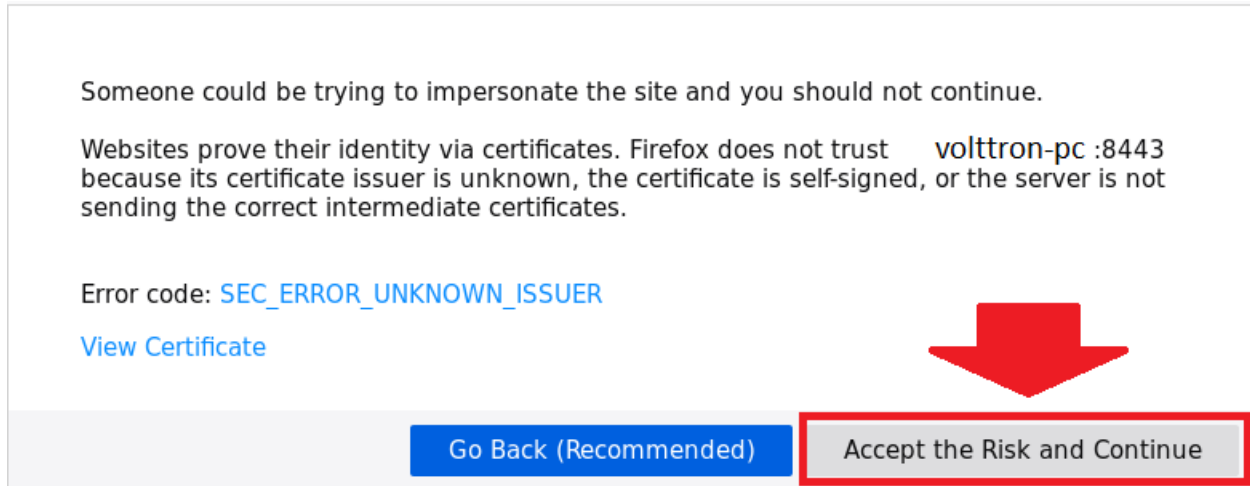
Error code: [SEC_ERROR_UNKNOWN_ISSUER](#)

[View Certificate](#)

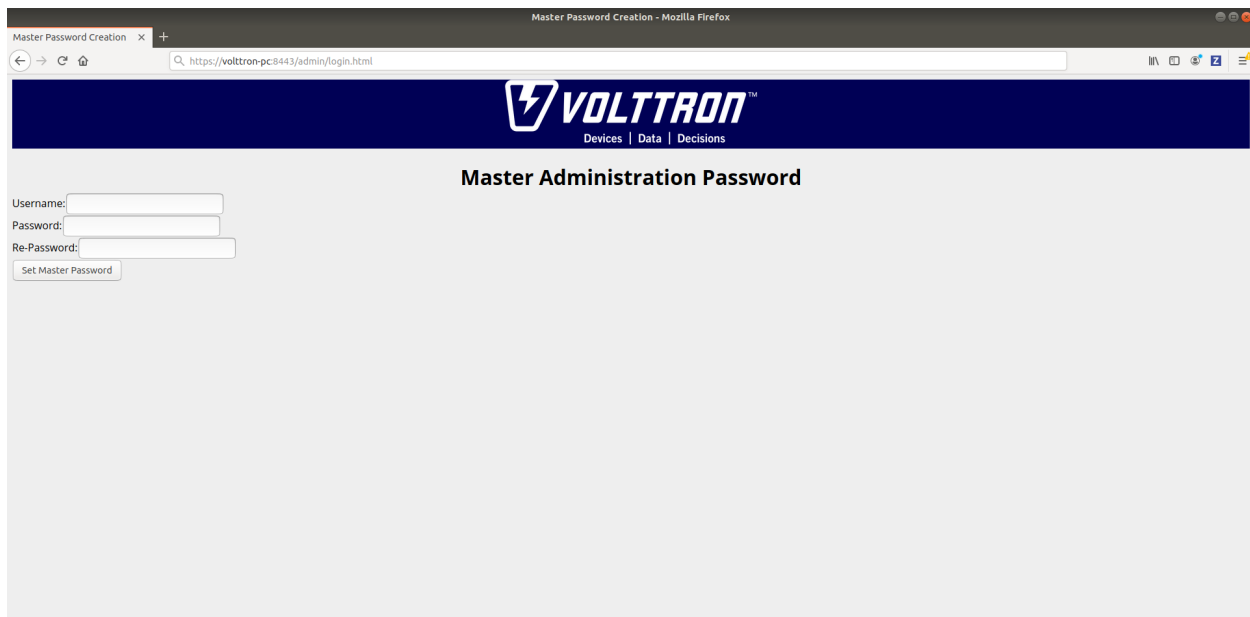
Go Back (Recommended)

Accept the Risk and Continue





When the admin page is accessed for the first time, the user will be prompted to set up a master username and password.



Open your browser to the web address that you specified for the VOLTTRON Central agent that you configured for the first instance. In the above examples, the configuration file would be located at `~/.volttron1/config` and the VOLTTRON Central address would be defined in the “volttron-central-address” field. The VOLTTRON Central address takes the pattern: `https://<localhost>:8443/vc/index.html`, where localhost is the hostname of your machine. In the above examples, our hostname is `volttron-pc`; thus our VC interface would be `https://volttron-pc:8443/vc/index.html`.

You will need to provide the username and password set earlier through admin web page.

Stopping the Demo

Once you have completed your walk through of the different elements of the VOLTTRON Central demo you can stop the demos by executing the following command in each terminal window.

```
$ ./stop-volttron
```

Once the demo is complete you may wish to see the VOLTTRON Central Management Agent page for more details on how to configure the agent for your specific use case.

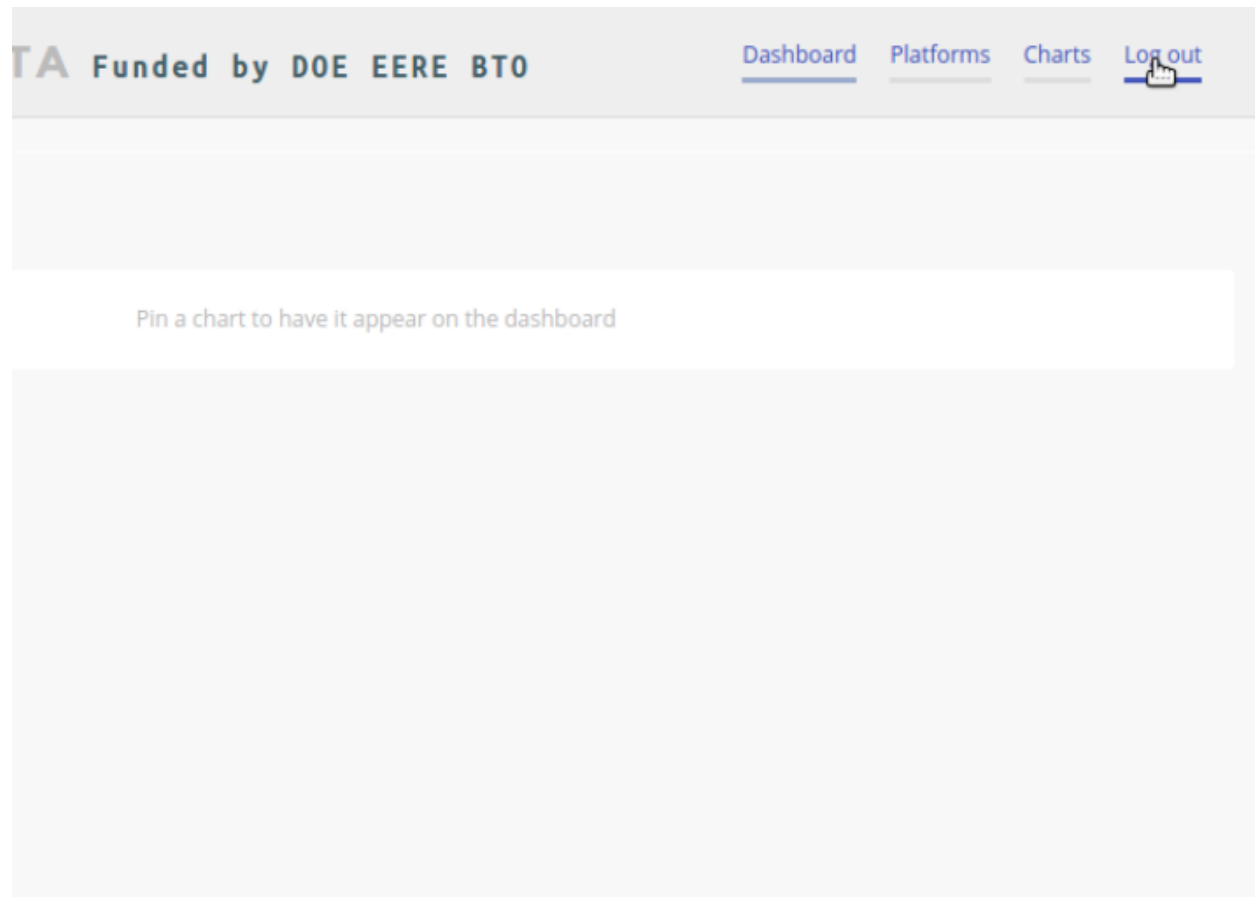
Log In

To log in to VOLTTRON Central, open a browser and login to the Volttron web interface, which takes the form `https://localhost:8443/vc/index.html` where localhost is the hostname of your machine. In the above example, we open the following URL in which our localhost is “volttron-pc”: `https://volttron-pc:8443/vc/index.html` and enter the user name and password on the login screen.



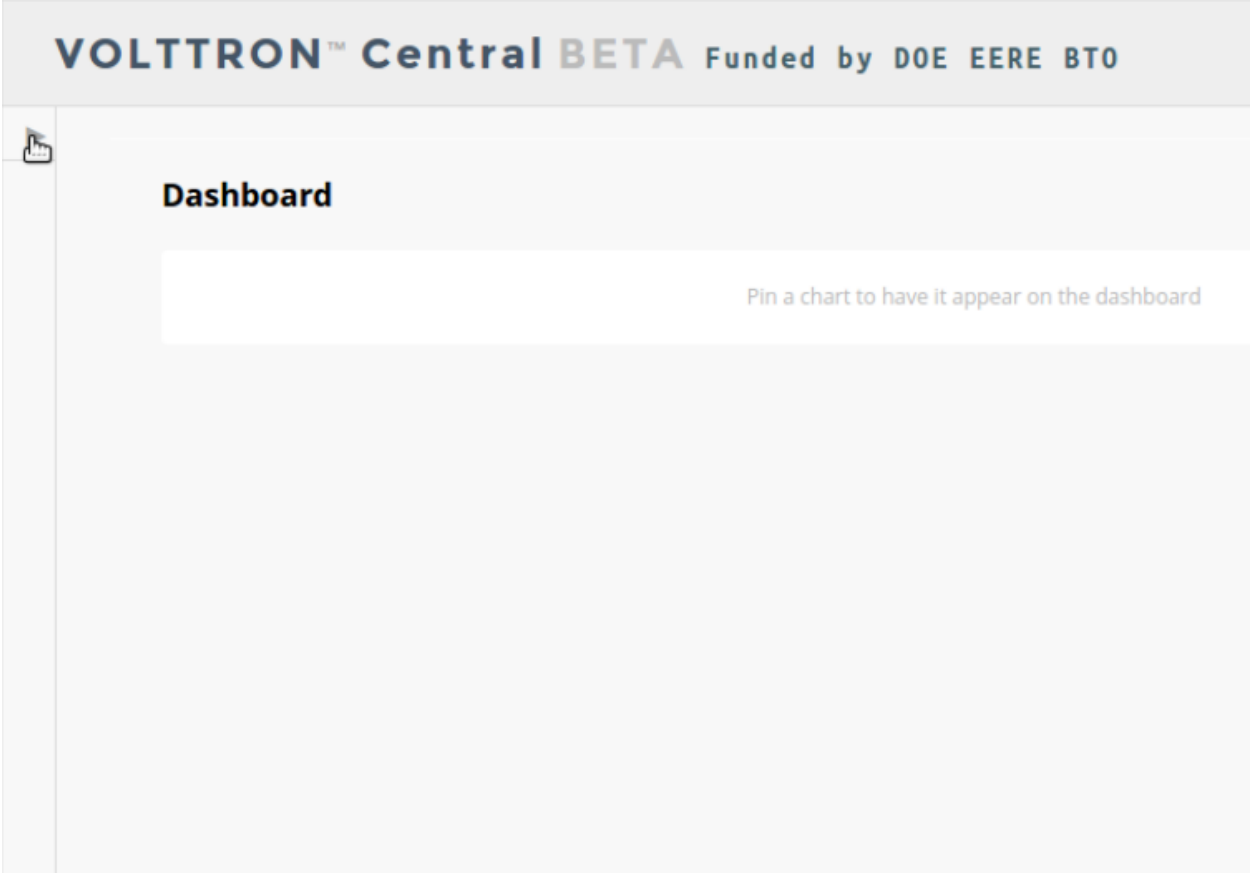
Log Out

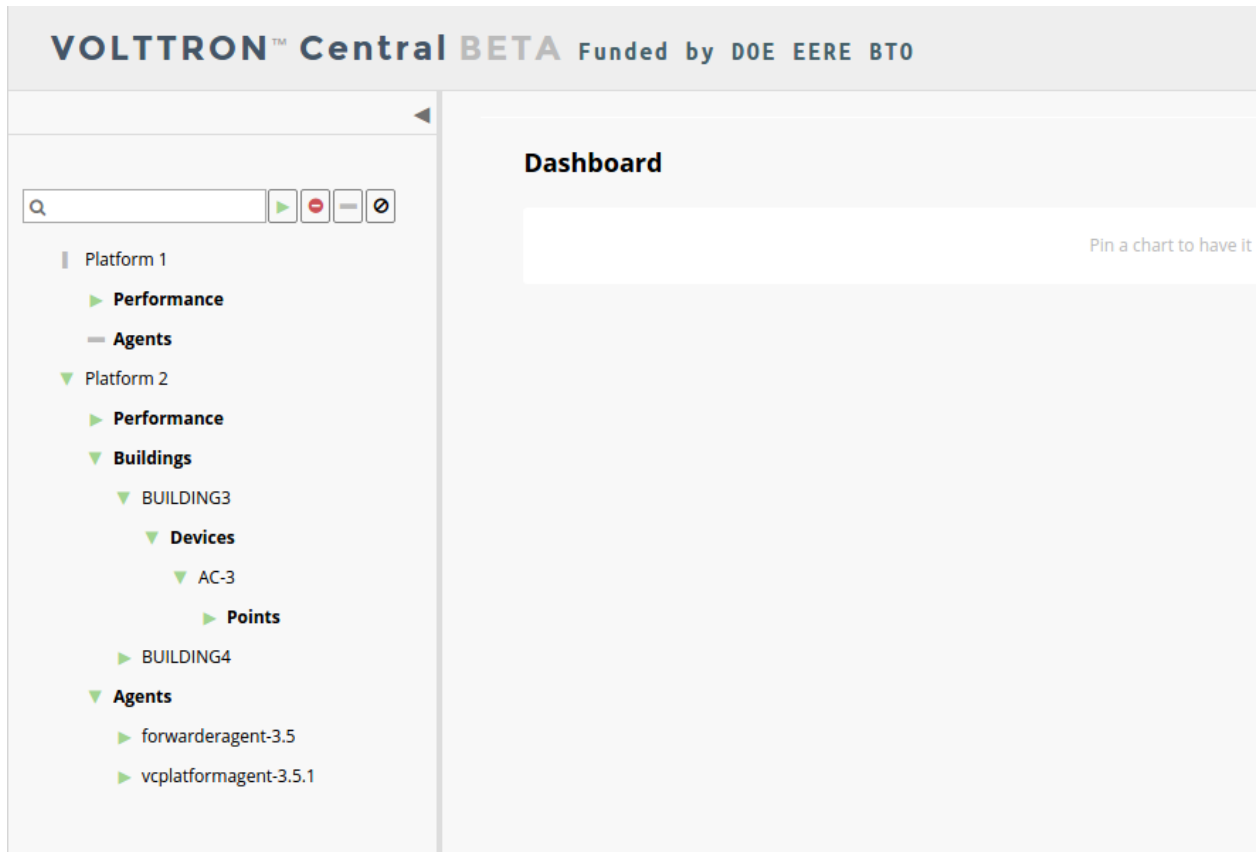
To log out of VOLTTRON Central, click the link at the top right of the screen.



Platforms Tree

The side panel on the left of the screen can be extended to reveal the tree view of registered platforms.

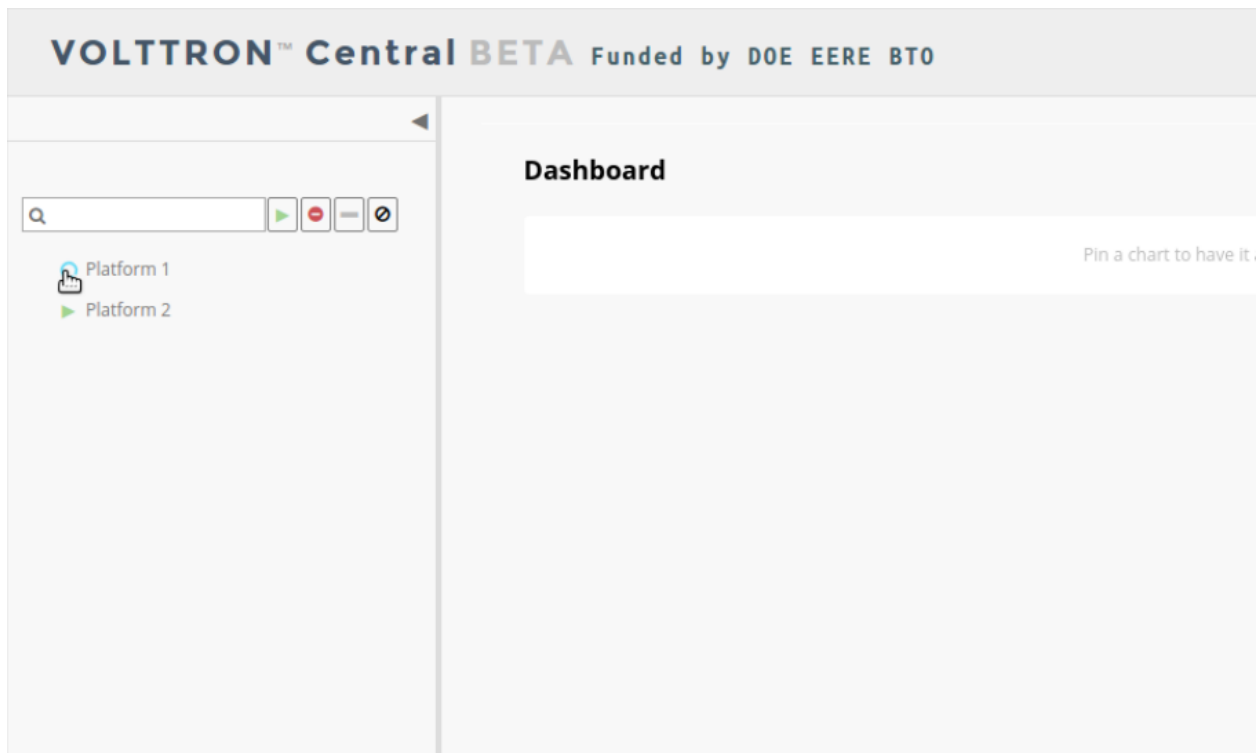




Top-level nodes in the tree are platforms. Platforms can be expanded in the tree to reveal installed agents, devices on buildings, and performance statistics about the platform instances.

Loading the Tree

The initial state of the tree is not loaded. The first time a top-level node is expanded is when the items for that platform are loaded.

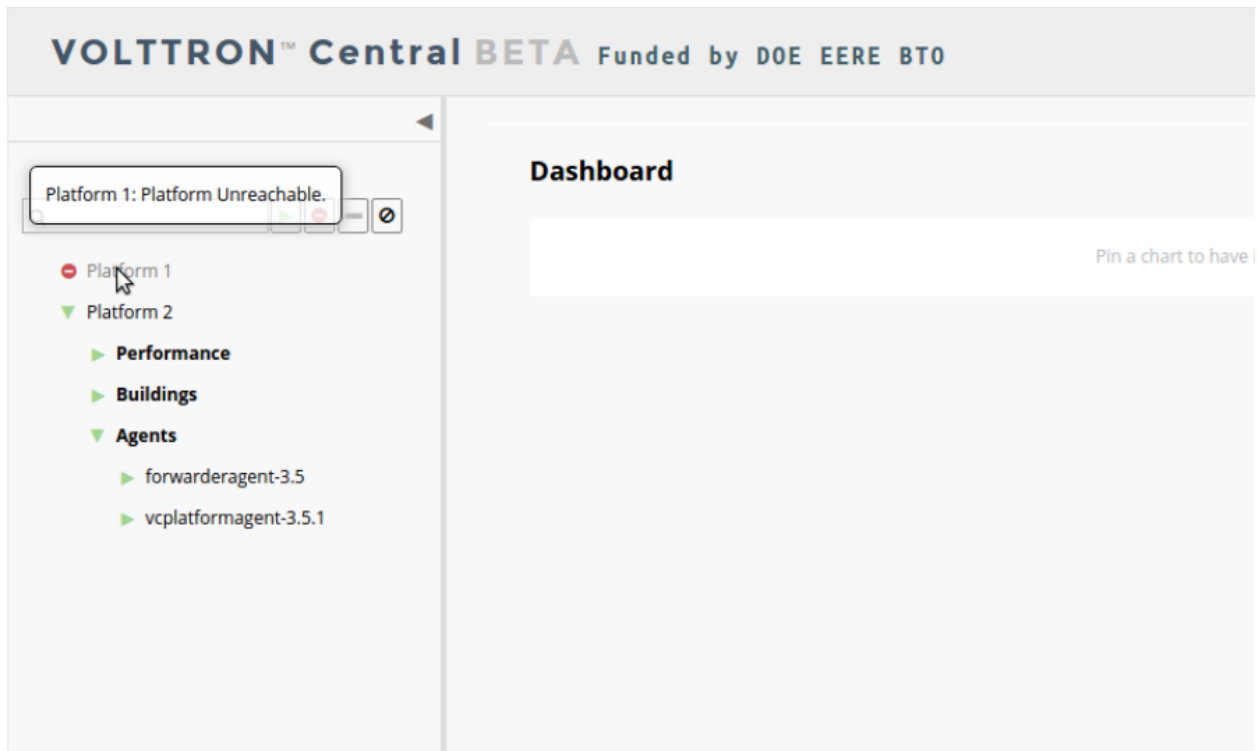


After a platform has been loaded in the tree, all the items under a node can be quickly expanded by double-clicking on the node.

Health Status

The health status of an item in the tree is indicated by the color and shape next to it. A green triangle means healthy, a red circle means there's a problem, and a gray rectangle means the status can't be determined.

Information about the health status also may be found by hovering the cursor over the item.



Filter the Tree

The tree can be filtered by typing in the search field at the top or clicking on a status button next to the search field.

VOLTRON™ Central BETA Funded by DOE EERE BTO

Search: [Apply] [Reset] [Clear] [Close]

- ▼ Platform 2
 - ▼ Buildings
 - ▼ BUILDING3
 - ▼ Devices
 - ▼ AC-3
 - ▼ Points
 - ▶ ☐ OutsideAirTemp
 - ▶ ☐ OutsideAirTemp
 - ▶ ☐ OutsideAirTemp
 - ▼ BUILDING4
 - ▼ Devices
 - ▼ HEATER4
 - ▼ Points
 - ▶ ☐ OutsideAirTemp
 - ▶ ☐ OutsideAirTemp
 - ▶ ☐ OutsideAirTemp

Dashboard

Pin a chart to have it

VOLTRON™ Central BETA Funded by DOE EERE BTO

Search: [Apply] [Reset] [Clear] [Close]

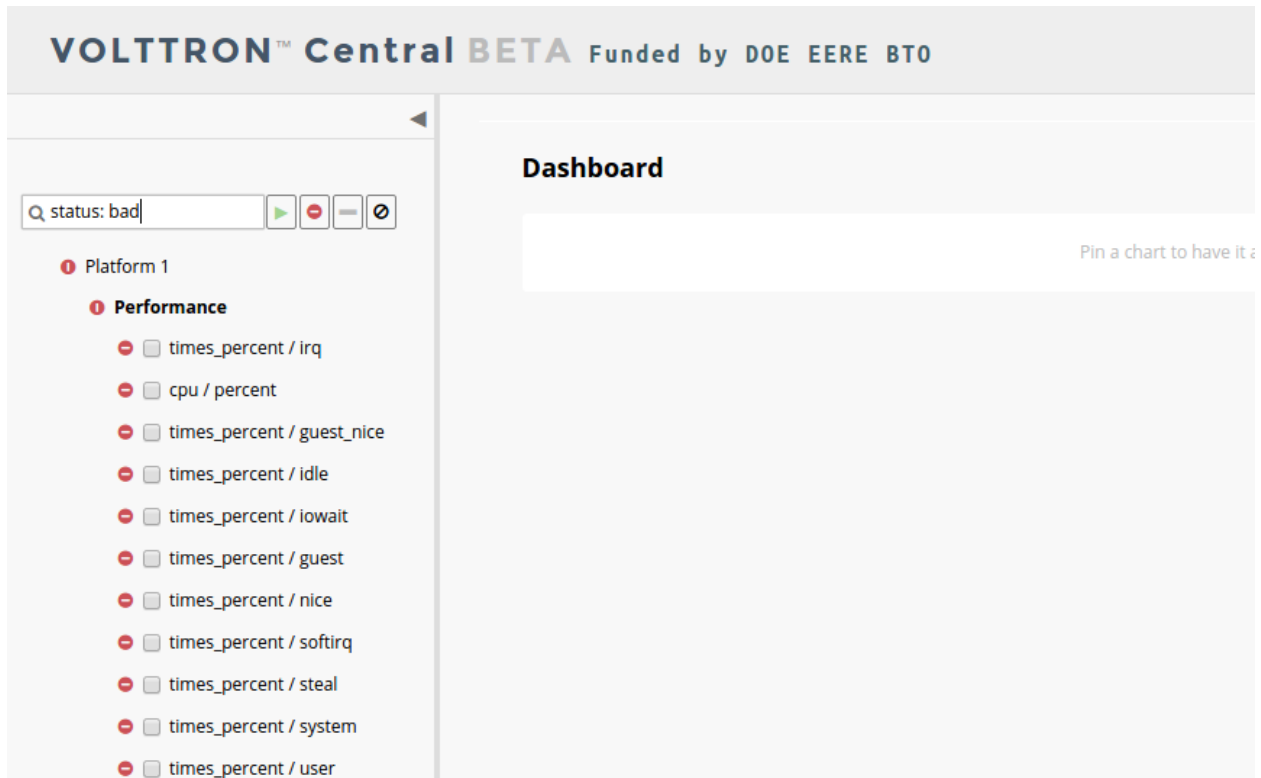
- Platform 1
 - Agents
 - bacnet_proxyagent-0.1
 - listeneragent-3.0

Dashboard

Pin a chart to have it

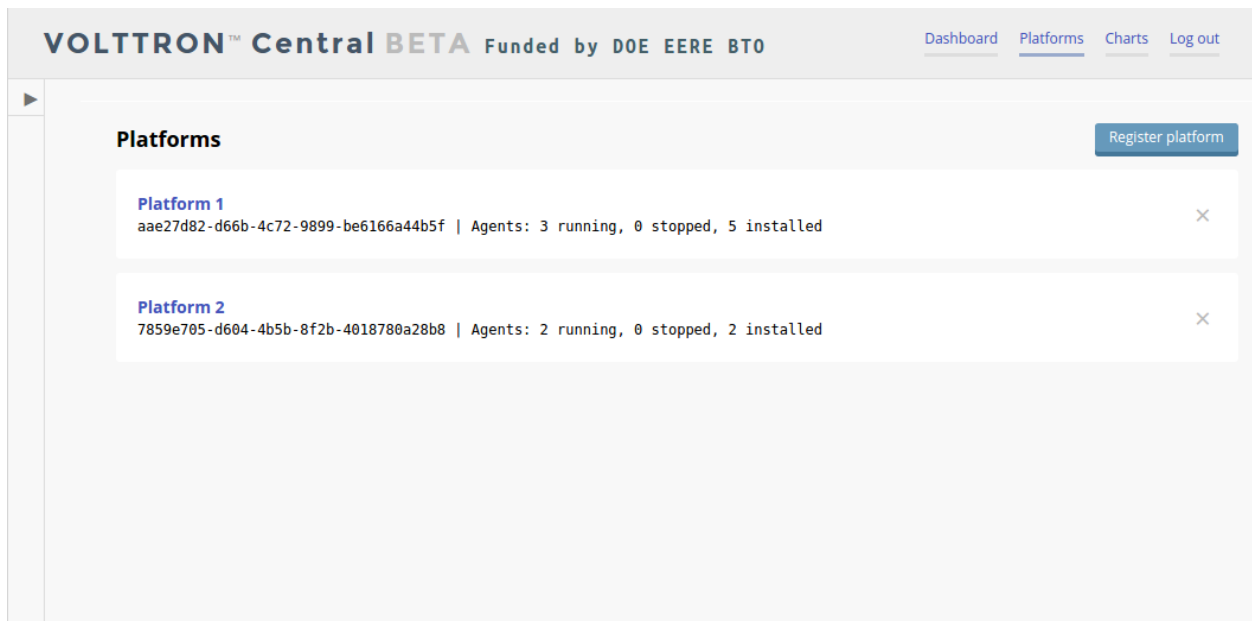
Meta terms such as “status” can also be used as filter keys. Type the keyword “status” followed by a colon, and then

the word “good,” “bad,” or “unknown.”



Platforms Screen

This screen lists the registered VOLTTRON platforms and allows new platforms to be registered by clicking the Register Platform button. Each platform is listed with its unique ID and the number and status of its agents. The platform’s name is a link that can be clicked on to go to the platform management view.



Platform View

From the platforms screen, click on the name link of a platform to manage it. Managing a platform includes installing, starting, stopping, and removing its agents.

VOLTRON™ Central BETA Funded by DOE EERE BTO

Dashboard Platforms Charts Log out

Platforms / Platform 1 (aae27d82-d66b-4c72-9899-be6166a44b5f)

Agents Name	UUID	Status	Action
bacnet_proxyagent-0.1	6a3be214-27a7-476d-a1d3-f3aa6c66db6e	Never started	Start Remove
listeneragent-3.0	0385b12e-2a5a-4d8a-abbf-6c3b6bab1acb	Never started	Start Remove
sqlhistorianagent-3.5.0	1fcb2c10-cb8e-4715-b248-69b4ad5d62b2	Running (PID 14296)	Stop Remove
vcplatformagent-3.5.1	dc720b2a-403a-4961-b316-74a964e5e038	Running (PID 14263)	Stop Remove
volttroncentralagent-3.5.3	f321f0b1-247a-476a-933b-d03cfc8d81500	Running (PID 12933)	Stop Remove

Install agents
[Choose Files](#) No file chosen

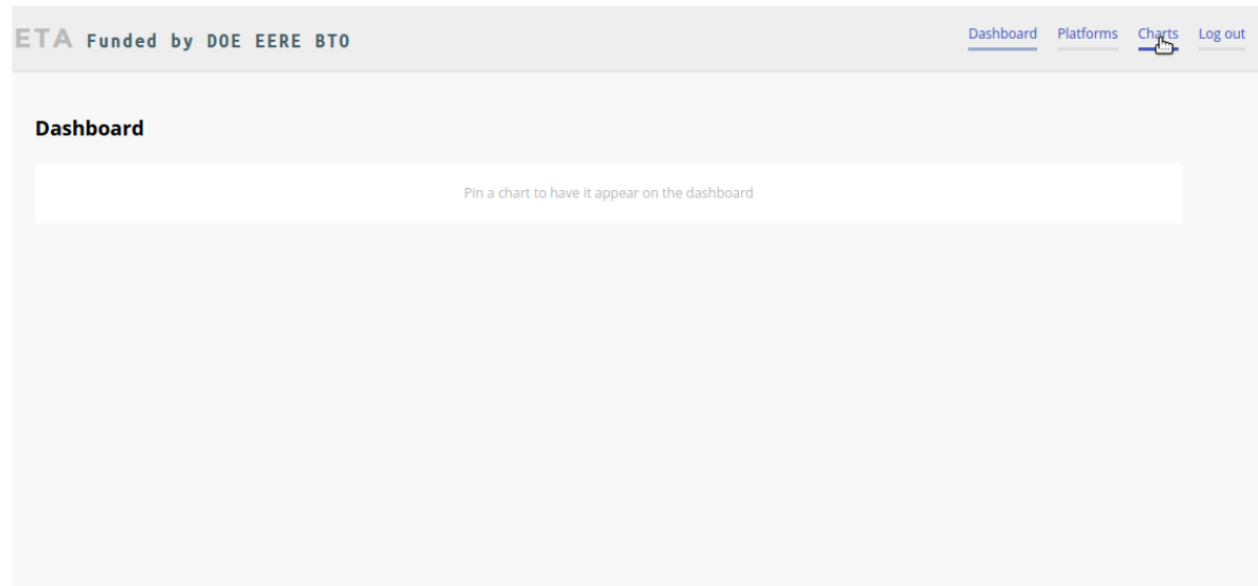
To install a new agent, all you need is the agent's wheel file. Click on the button and choose the file to upload it and install the agent.

To start, stop, or remove an agent, click on the button next to the agent in the list. Buttons may be disabled if the user lacks the correct permission to perform the action or if the action can't be performed on a specific type of agent. For instance, platform agents and VOLTRON Central agents can't be removed or stopped, but they can be restarted if they've been interrupted.

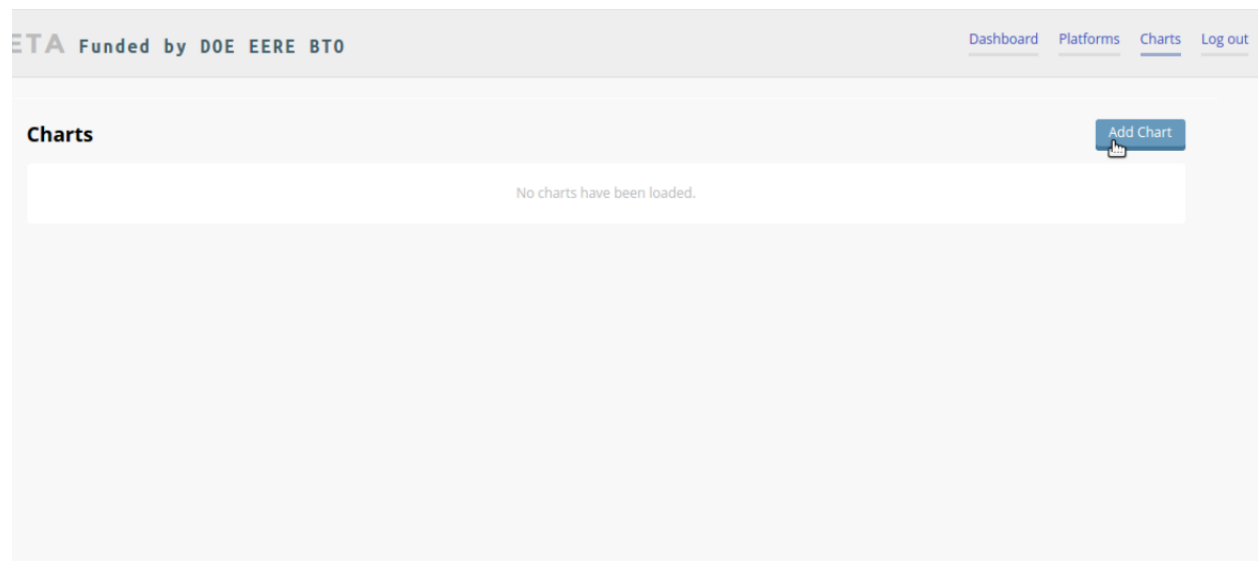
Add Charts

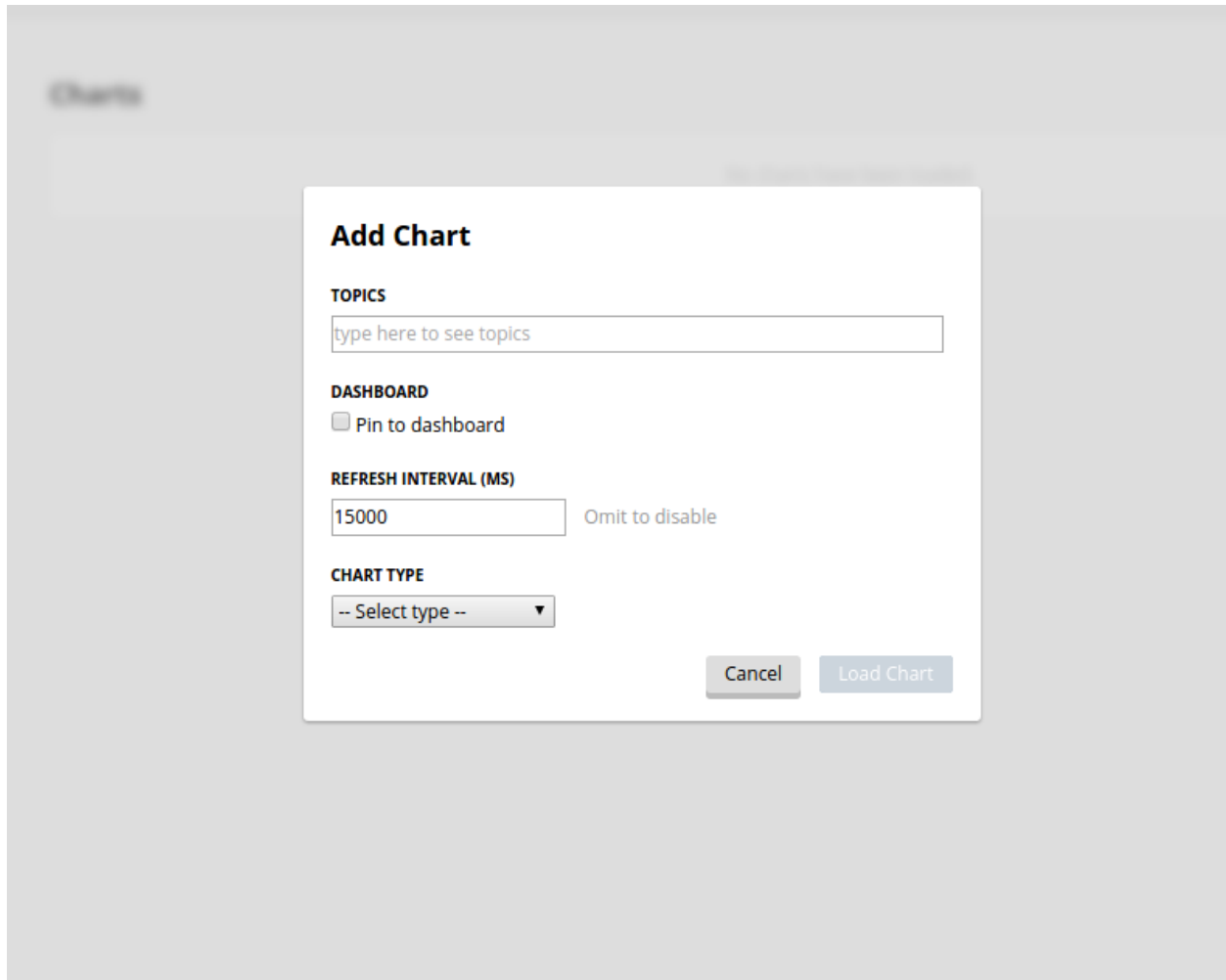
Performance statistics and device points can be added to charts either from the Charts page or from the platforms tree in the side panel.

Click the Charts link at the top-right corner of the screen to go to the Charts page.

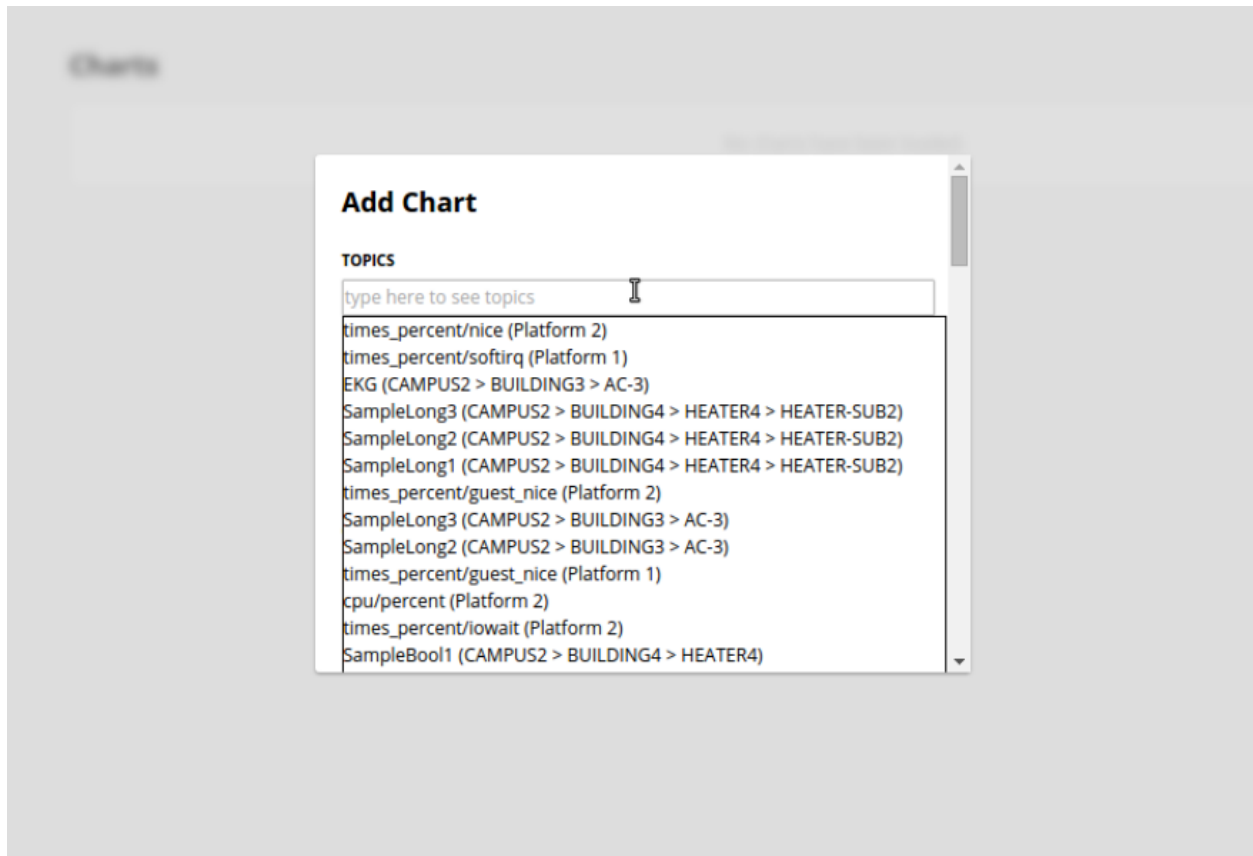


From the Charts page, click the Add Chart button to open the Add Chart window.

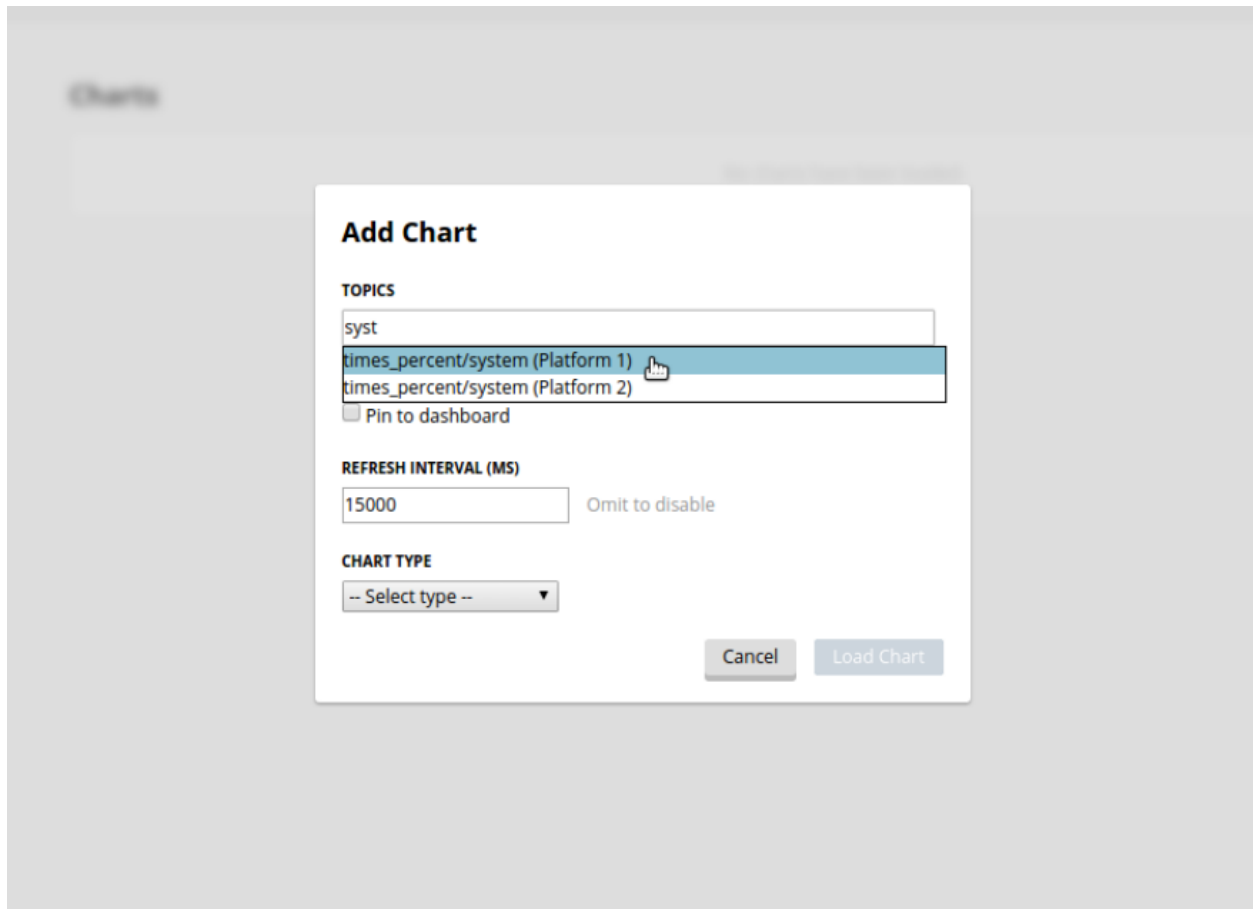




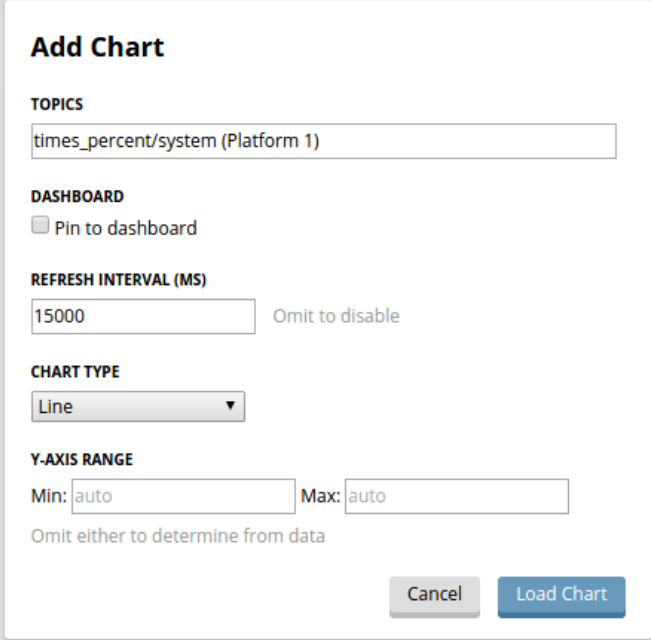
Click in the topics input field to make the list of available chart topics appear.



Scroll and select from the list, or type in the field to filter the list, and then select.



Select a chart type and click the Load Chart button to close the window and load the chart.



Add Chart

TOPICS

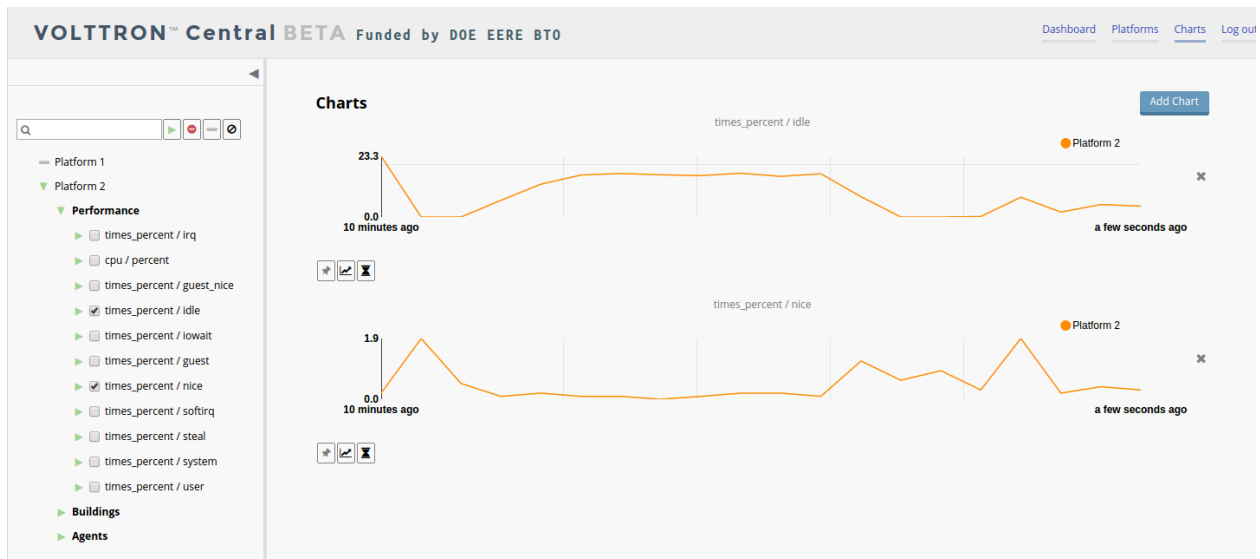
DASHBOARD
☐ Pin to dashboard

REFRESH INTERVAL (MS)
 Omit to disable

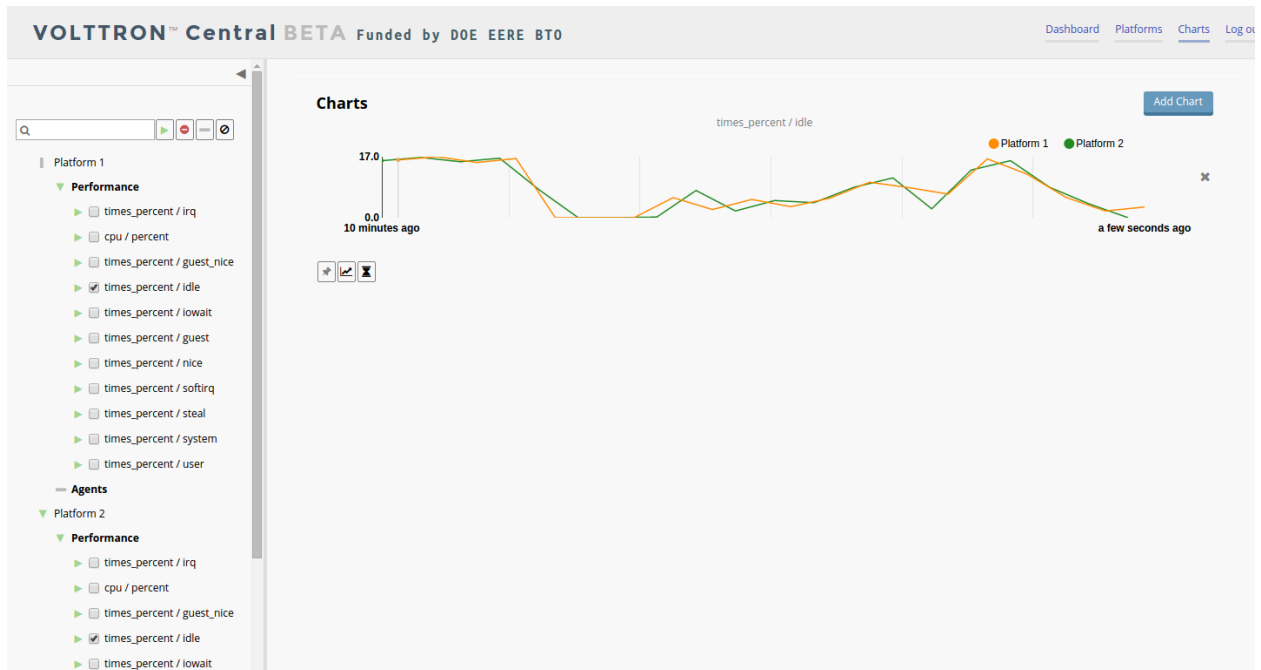
CHART TYPE

Y-AXIS RANGE
 Min: Max:
 Omit either to determine from data

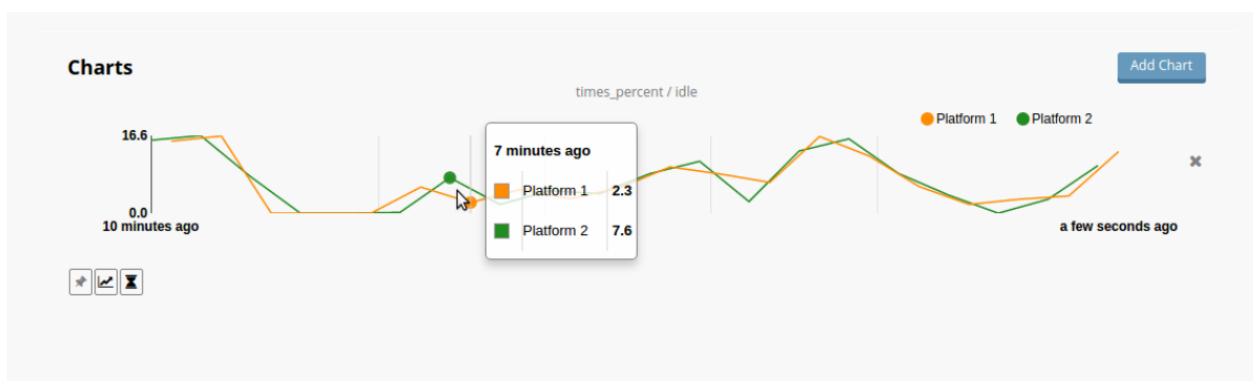
To add charts from the side panel, check boxes next to items in the tree.



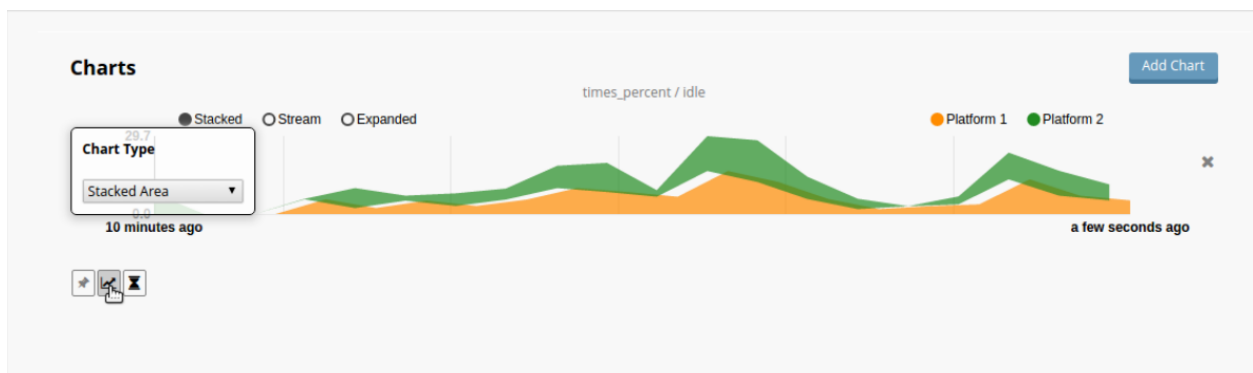
Choose points with the same name from multiple platforms or devices to plot more than one line in a chart.



Move the cursor arrow over the chart to inspect the graphs.

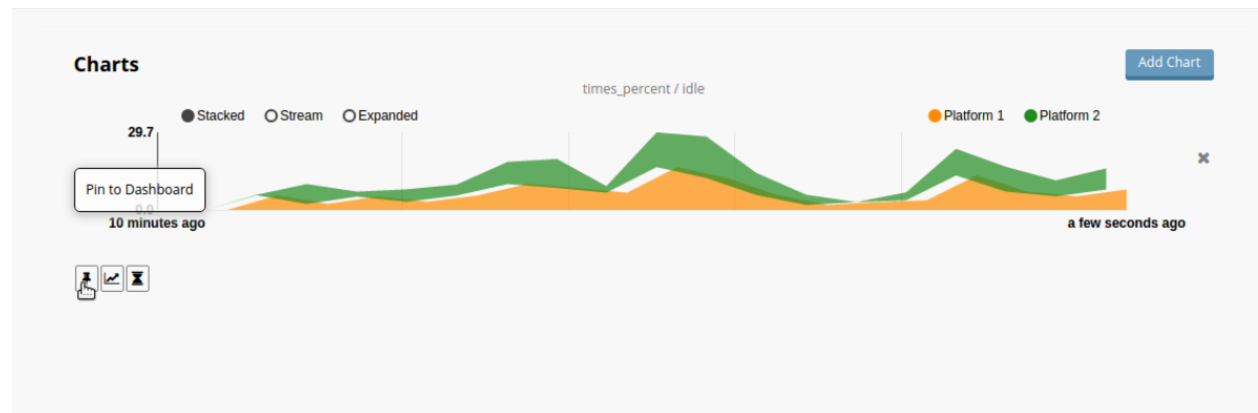


To change the chart's type, click on the Chart Type button and choose a different option.



Dashboard Charts

To pin a chart to the Dashboard, click the Pin Chart button to toggle it. When the pin image is black and upright, the chart is pinned; when the pin image is gray and diagonal, the chart is not pinned and won't appear on the Dashboard.



Charts that have been pinned to the Dashboard are saved to the database and will automatically load when the user logs in to VOLTTRON Central. Different users can save their own configurations of dashboard charts.

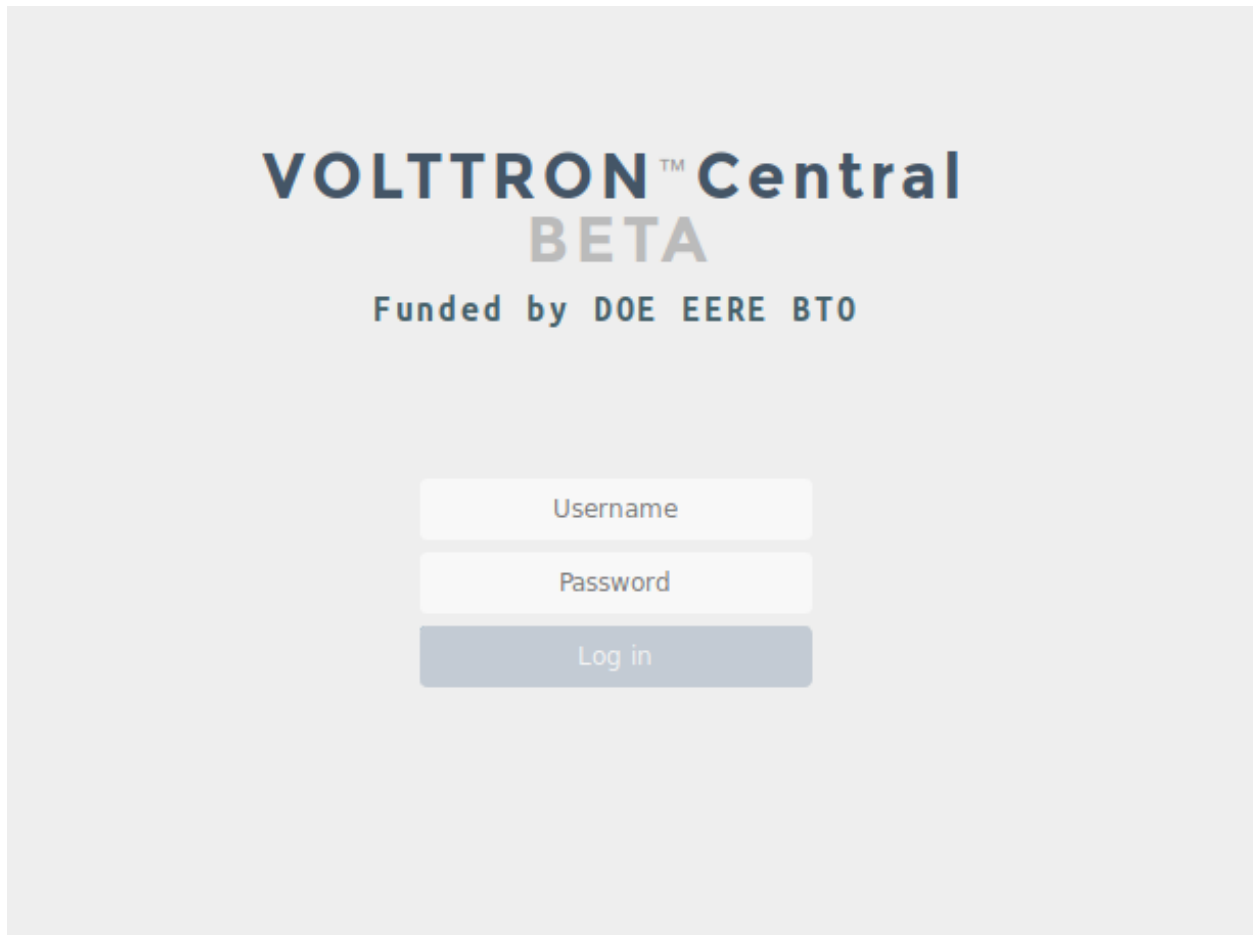
Remove Charts

To remove a chart, uncheck the box next to the item in the tree or click the X button next to the chart on the Charts page. Removing a chart removes it from the Charts page and the Dashboard.

VOLTTRON Central

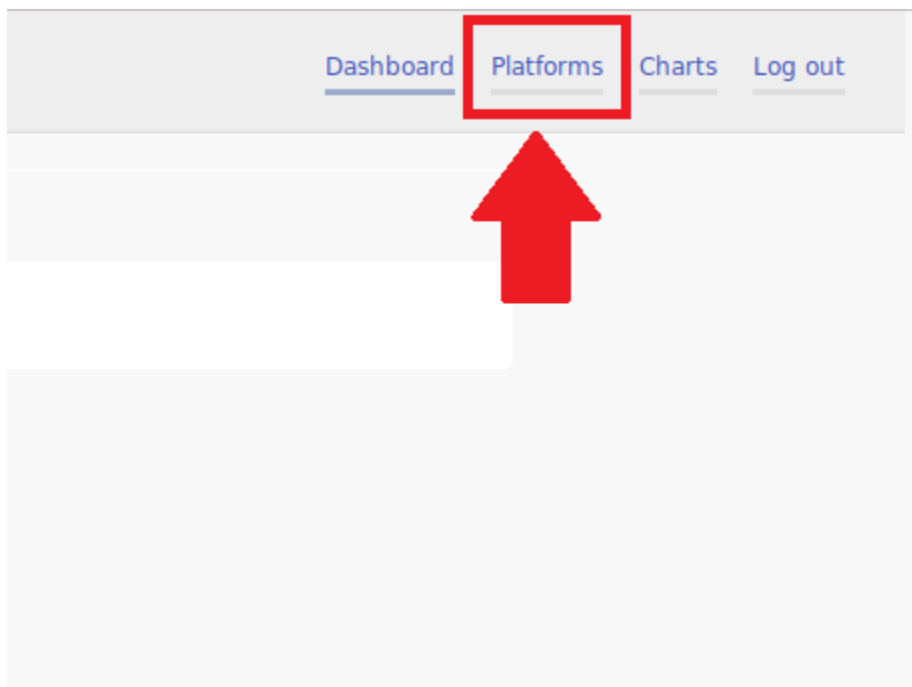
Navigate to <https://volttron-pc:8443/vc/index.html>

Log in using the username and password you set up on the admin web page.



The image shows the login page for VOLTRON Central BETA. The title "VOLTRON™ Central BETA" is centered at the top, with "BETA" in a lighter blue font. Below the title, it says "Funded by DOE EERE BTO". The login form consists of three stacked input fields: "Username", "Password", and a "Log in" button. The "Log in" button is a solid blue color, while the input fields are white with a light blue border.

Once you have logged in, click on the Platforms tab in the upper right corner of the window.



Once in the Platforms screen, click on the name of the platform.



You will now see a list of agents. They should all be running.

Platforms / volttron1 (dm9sdHRYb24xLnBsYXRmb3JtLmFnZW50)

Agents

Name	Identity	UUID	Status	Action
listeneragent-3.2	listeneragent-3.2_1	f8b93d0-b499-4d66-8304-94ac9fd5d9b4	Running (PID 3842)	<button>Stop</button> <button>Remove</button>
master_driveragent-3.2	platform.driver	30bf6384-3bde-4e28-8bf7-6695448f7b7d	Running (PID 3847)	<button>Stop</button> <button>Remove</button>
sqlhistorianagent-3.7.0	platform.historian	769d570d-e9b5-4263-8d07-31416cccec5c8	Running (PID 3845)	<button>Stop</button> <button>Remove</button>
vcplatformagent-4.8	platform.agent	f03b0ade-d161-4ddb-946b-30e129725584	Running (PID 3844)	<button>Stop</button> <button>Remove</button>
volttroncentralagent-5.0	volttron.central	63f67dd6-a27a-4064-949f-7c6649c43f73	Running (PID 3846)	<button>Stop</button> <button>Remove</button>

Install agents

No files selected.

For more information on VOLTTRON Central, please see:

- VOLTTRON Central Management
- VOLTTRON Central Demo

1.19 Linux System Hardening

1.19.1 Introduction

VOLTTRON is built with modern security principles in mind [security-wp] and implements many security features for hosted agents. However, VOLTTRON is built on top of Linux and the underlying Linux platform also needs to be secured in order to declare the resulting control system as “secure.”

Any system is only as secure as its weakest link. This document is dedicated to making recommendations for hardening of the underlying Linux platform that VOLTTRON is deployed to.

Warning: No system can be 100% secure and the cyber security strategy that is recommended in this document is based on risk management. For the following guidance, it is intended that the user consider the risk, impact of risks, and perform the appropriate corresponding mitigation techniques.

1.19.2 Recommendations

Here are the non-exhaustive recommendations for Linux hardening from the VOLTTRON team:

- **Physical Security:** Keep the system in locked cabinets or a locked room. Limit physical access to systems and to the networks to which they are attached. The goal should be to avoid physical access by untrusted personnel. This could be extended to blocking or locking USB ports, removable media drives, etc.

Drive encryption could be used to avoid access via alternate-media booting (off USB stick or DVD) if physical access can't be guaranteed. The downside of drive encryption would be needing to enter a passphrase to start system. Alternately, the *Trusted Platform Module* (TPM) may be used, but the drive might still be accessible to those with physical access. Enable chassis intrusion detection and reporting if supported. If available, use a physical tamper seal along with or in place of an interior switch.

- **Low level device Security:** Keep firmware of all devices (including BIOS) up-to-date. Password-protect the BIOS. Disable unneeded/unnecessary devices:
 - serial
 - parallel
 - USB (Leaving a USB port enabled may be helpful if a breach occurs to allow saving forensic data to an external drive.)
 - Firewire, etc.
 - ports
 - optical drives
 - wireless devices, such as Wi-Fi and Bluetooth
- **Boot security:**
 - Disable auto-mounting of external devices
 - Restrict the boot device:
 - * Disable PXE and other network boot options (unless that is the primary boot method)
 - * Disable booting from USB and other removable drives
 - Secure the boot loader:
 - * Require an administrator password to do anything but start the default kernel
 - * Do not allow editing of kernel parameters
 - * Disable, remove, or password-protect emergency/recovery boot entries
- **Security Updates:** First and foremost, configure the system to automatically download security updates. Most security updates can be installed without rebooting the system, but some updated (e.g. shared libraries, kernel, etc) require the system to be rebooted. If possible, configure the system to install the security updates automatically and reboot at a particular time. We also recommend reserving the reboot time (e.g. 1:30AM on a Saturday morning) using the Actuator Agent so that no control actions can happen during that time.
- **System Access only via Secured Protocols:**
 - Disallow all clear text access to VOLTTRON systems

- No telnet, no rsh, no ftp and no exceptions!
- Use ssh to gain console access, and scp/sftp to get files in and out of the system
- Disconnect excessively idle SSH Sessions
- Disable remote login for “root” users. Do not allow a user to directly access the system as the “root” user from a remote network location. Root access to privileged operations can be accomplished using `sudo`. This adds an extra level of security by restricting access to privileged operations and tracking those operations through the system log.
- Manage users and usernames, limit the number of user accounts, use complex usernames rather than first names.
- Authentication: If possible, use two factor authentication to allow access to the system. Informally, two factor authentication uses a combination of “something you know” and “something you have” to allow access to the system. RSA *SecurID* tokens are commonly used for two factor authentication but other tools are available. When not using two-factor authentication, use strong passwords and do not share accounts.
- Scan for weak passwords. Use password cracking tools such as [John the Ripper](#) or [Nmap](#) with password cracking modules to look for weak passwords.
- Utilize *Pluggable Authentication Modules* (PAM) to strengthen passwords and the login process. We recommend:
 - `pam_abl`: Automated blacklisting on repeated failed authentication attempts
 - `pam_captcha`: A visual text-based CAPTCHA challenge module for PAM
 - `pam_passwdqc`: A password strength checking module for PAM-aware password changing programs
 - `pam_cracklib`: PAM module to check the password against dictionary words
 - `pam_pwhistory`: PAM module to remember last passwords
- Disable unwanted services. Most desktop and server Linux distributions come with many unnecessary services enabled. Disable all unnecessary services. Refer to your distribution’s documentation to discover how to check and disable these services.
- Just as scanning for weak passwords is a step to more secure systems; regular network scans using [Nmap](#) to find what network services are being offered is another step towards a more secure system.

Warning: use *Nmap* or similar tools very carefully on BACnet and modbus environments. These scanning tools are known to crash/reset BACnet and modbus devices.

- Control incoming and outgoing network traffic. Use the built-in host-based firewall to control who/what can connect to this system. Many *iptables* frontends offer a set of predefined rules that provide a default deny policy for incoming connections and provide rules to prevent or limit other well known attacks (i.e. rules that limit certain responses that might amplify a DDoS attack). *ufw* (uncomplicated firewall) is a good example.

Examples:

- If the system administrators for the VOLTTRON device are all located in `10.10.10.0/24` subnetwork, then allow SSH and SCP logins from only that IP address range.
- If the VOLTTRON system exports data to a historian at `10.20.20.1` using TCP over port 443, allow outgoing traffic to that port on that server.

The idea here is to limit the attack surface of the system. The smaller the surface, the better we can analyze the communication patterns of the system and detect anomalies.

Note: While some system administrators disable network-based diagnostic tools such as ICMP ECHO responses, the VOLTTTRON team believes that this hampers usability. As an example, monitoring which incoming and outgoing firewall rules are triggering can be accomplished with this command:

```
watch --interval=5 'iptables -nvL | grep -v "0      0"'
```

- Rate limit incoming connections to discourage brute force hacking attempts. Use a tool such as [fail2ban](#) to dynamically manage firewall rules to rate limit incoming connections and discourage brute force hacking attempts. [sshguard](#) is similar to *fail2ban* but only used for ssh connections. Further rate limiting can be accomplished at the firewall level. As an example, you can restrict the number of connections used by a single IP address to your server using iptables. Only allow 4 ssh connections per client system:

```
iptables -A INPUT -p tcp --syn --dport 22 -m connlimit --connlimit-above 4 -j DROP
```

You can limit the number of connections per minute. The following example will drop incoming connections if an IP address makes more than 10 connection attempts to port 22 within 60 seconds:

```
iptables -A INPUT -p tcp -dport 22 -i eth0 -m state --state NEW -m recent --set
iptables -A INPUT -p tcp -dport 22 -i eth0 -m state --state NEW -m recent --
↪update --seconds 60 --hitcount 10 -j DROP
```

- Use a file system integrity tool to monitor for unexpected file changes. Tools such as [tripwire](#) monitor filesystems for changed files. Another file integrity checking tool to consider is [AIDE \(Advanced Intrusion Detect Environment\)](#).
- Use filesystem scanning tools periodically to check for exploits. Available tools such as [checkrootkit](#), [rkhunter](#) and others should be used to check for known exploits on a periodic basis and report their results.
- VOLTTTRON does not use Apache or require it. If Apache is being used, we recommend using the *mod_security* and *mod_evasive* modules.

Raspberry Pi

System hardening recommendations for Raspberry Pi closely match those for other Linux operating systems such as Ubuntu. VOLTTTRON has only been officially tested with Raspbian, and there is one important consideration, which is noted in the Raspbian documentation as well:

Warning: The Raspbian operating system includes only the default *pi* user on install, which uses a well-known default password. For any operational deployment, it is recommended to create a new user with a complex password (this user must have sudoers permissions).

Summarizing the process of creating a new user *alice* from the Raspberry Pi documentation:

```
sudo adduser alice
sudo usermod -a -G adm,dialout,cdrom,sudo,audio,video,plugdev,games,users,input,
↪netdev,gpio,i2c,spi alice
sudo su - alice
sudo visudo /etc/sudoers.d/010_pi-nopasswd
```

When the editor opens for the sudoer's file, add an entry for *alice*:

```
alice ALL=(ALL) PASSWD: ALL
```

Also, update the default *pi* user's default password:

```
pi@raspberrypi:~/volttron$ passwd
Changing password for pi.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

Note: The Raspberry Pi documentation states that ideally one would remove the *pi* user from the system, however this is not currently recommended as some aspects of the Raspberry Pi OS are tied to the *pi* user. This will be changed in the future.

For more information, please visit the [Raspberry Pi security site](#).

System Monitoring

- Monitor system state and resources. Use a monitoring tool such as [Xymon](#) or [Big Brother](#) to remotely monitor the system resources and state. Set the monitoring tools to alert the system administrators if anomalous use of resources (e.g. connections, memory, etc) are detected. An administrator can also use Unix commands such as *netstat* to look for open connections periodically.
- Watch system logs and get logs off the system. Use a utility such as [logwatch](#) or [logcheck](#) to get a daily summary of system activity via email. For Linux distributions that use *systemd* (such as Ubuntu), use [journalwatch](#) to accomplish the same task.

Additionally, use a remote syslog server to collect logs from all VOLTTRON systems in the field at a centralized location for analysis. A tool such as *Splunk* is ideal for this task and comes with many built-in analysis applications. Another benefit of sending logs remotely off the platform is the ability to inspect the logs even when the platform may be compromised.

- An active intrusion sensor such as [PSAD](#) can be used to look for intrusions as well.

Security Testing

Every security control discussed in the previous sections must be tested to determine correct operation and impact. For example, if we inserted a firewall rule to ban connections from an IP address such as 10.10.10.2, then we need to test that the connections actually fail.

In addition to functional correctness testing, common security testing tools such as [Nessus](#) and [Nmap](#) should be used to perform cyber security testing.

1.19.3 Conclusion

No system is 100% secure unless it is disconnected from the network and is in a physically secure location. The VOLTTRON team recommends a risk-based cyber security approach that considers each risk, and the impact of an exploit. Mitigating technologies can then be used to mitigate the most impactful risks first. VOLTTRON is built with security in mind from the ground up, but it is only as secure as the operating system that it runs on top of. This document is intended to help VOLTTRON users to secure the underlying Linux operating system to further improve the robustness of the VOLTTRON platform. Any security questions should be directed to volttron@pnnl.gov.

1.20 Deployment Recipes (Multi-Machine)

For more details about ansible recipes for scalable deployment strategies see [VOLTTRON Deployment Recipes](#)

CHAPTER 2

Indices and tables

- `genindex`
- `modindex`

A

Agent, [13](#)

B

BACNet, [13](#)

D

DNP3 (*Distributed Network Protocol 3*), [13](#)

I

IEEE 2030.5, [13](#)

J

JSON (*JavaScript Object Notation*), [13](#)

JSON-RPC (*JSON-Remote Procedure Call*), [13](#)

M

Modbus, [13](#)

P

PLC (*Programmable Logic Controller*), [13](#)

Publish/Subscribe, [13](#)

Python Virtual Environment, [13](#)