PNNL-22935

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

VOLTTRON Lite: Integration Platform for the Transactional Network

JN Haack
S Katipamula
BA Akyol
RG Lutes

October 2013

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

Printed in the United States of America

PNNL-22935

VOLTTRON Lite: Integration Platform for the Transactional Network

JN Haack
BA Akyol
S Katipamula
RG Lutes

October 2013

Prepared for
U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99352

Abstract

In FY13, Pacific Northwest National Laboratory (PNNL), with funding from the Department of Energy's (DOE's) Building Technologies Office (BTO), designed, prototyped and tested a transactional network platform. The platform consists of VOLTTRON Lite™ agent execution software, a number of agents that perform specific function (fault detection, demand response, weather service, logging service, etc.). The platform is intended to support energy, operational and financial transactions between networked entities (equipment, organizations, buildings, grid, etc.). Initially, in FY13, the concept demonstrated transactions between packaged rooftop air conditioners and heat pumps units (RTUs) and the electric grid using applications or "agents" that reside on the platform, on the equipment, on local building controller or in the Cloud.

This document describes the core of the transactional network platform, the VOLTTRON Lite software and associated services hosted on the platform. Future enhancements are also discussed. The appendix of the document provides examples of how to use the various services hosted on the platform.

Table of Contents

Figures

1.0 Introduction

In FY13, Pacific Northwest National Laboratory (PNNL), with funding from the Department of Energy's (DOE's) Building Technologies Office (BTO), designed, prototyped and tested a transactional network platform. The platform consists of VOLTTRON Lite™ agent execution software, a number of agents that perform specific function (fault detection, demand response, weather service, logging service, etc.). The platform is intended to support energy, operational and financial transactions between networked entities (equipment, organizations, buildings, grid, etc.). Initially, in FY13, the PNNL demonstrated transactions between packaged rooftop air conditioners and heat pumps units (RTUs) and the electric grid using applications or "agents" that reside on the platform, on the equipment, on local building controller or in the Cloud.

The transactional network project is a multi-laboratory effort with Oakridge National Laboratory (ORNL) and Lawrence Berkeley National Laboratory (LBNL) also contributing to the effort. PNNL coordinated the project and also was responsible for the development of the transactional network (TN) platform and three different applications associated with RTUs. This document describes the core of the TN platform, VOLTTRON Lite. The details of the RTU agents are described in another companion document.

2.0 VOLTTRON Lite Overview

The purpose of the Transactional Network project is to demonstrate and propagate an open-source, open-architecture platform that enables a variety of site/equipment specific applications to be applied in a cost-effective and scalable way. Such an open-source platform will lower the cost of entry for both existing and new service providers because the data transport or information exchange typically required for operational and energy related products and services will be ubiquitous and interoperable.

2.1 VOLTTRON Lite

VOLTTRON Lite serves as an integrating platform for the components of the Transactional Network project. It provides an environment for agent execution and serves as a single point of contact for interfacing with devices (RTU heating, ventilation and air conditioners (HVACs); power meters, etc.), external resources, and platform services such as data archival and retrieval. The VOLTTRON platform provides a collection of utility and helper classes, which simplifies agent development.
In the Transactional Network project, VOLTTRON Lite connects devices to applications implemented in the platform and in the Cloud, a data historian, and signals from the power grid. It also provides helper classes to ease development and deployment of agents into the environment.

This project was initiated to create an open source version of the VOLTTRON™ software, developed by Pacific Northwest National Laboratory (PNNL). This completely new codebase has replicated VOLTTRON functionality, added several new capabilities, and incorporated a number of open source projects to build a flexible and powerful platform:

- sMAP: VOLTTRON utilizes sMAP[1] for data storage and retrieval. The VOLTTRON MODBUS driver publishes data from devices to the platform and also stores the data in the sMAP historian. During development of this driver, the VOLTTRON team contributed error reports and resolved a bug in the sMAP software.

- Drivers for sMAP are written using another open source product called twistd[2]. Twistd is an event based networking engine.

- 0MQ: The VOLTTRON message bus, which allows agents and services to exchange data, is backed by Zero MQ[3]. This free software is used by National Aeronautics and Space Administration (NASA), Cisco, etc. to provide scalable, reliable, and fast communication. Security and bug fixes have been reported by the VOLTTRON Lite team as well as a fix being contributed back.

- PyModbus: The VOLTTRON MODBUS[4] driver builds on PyModbus[5], which enables Python code to easily interact with Modbus devices.

- Other open source Python modules being used are:

[1] http://www.cs.berkeley.edu/~stevedh/smap2/index.html
[2] http://twistedmatrix.com/trac/
[3] http://zeromq.org/
[4] http://www.modbus.org/
[5] http://code.google.com/p/pymodbus/

  - 'avro', 'configobj', 'gevent', 'flexible-jsonrpc', 'numpy', 'posix-clock', ‘pyopenssl', 'python-dateutil', 'requests', 'setuptools', 'simplejson', 'zope.interface'

2.2 VOLTTRON Lite Agents and Services

Agents deployed on VOLTTRON can perform one or more roles, which can be broadly classified into the following groups:

- Platform Agents: Agents that are part of the platform and provide a service to other agents. Examples are agents that interface with devices to publish readings and handle control signals from other agents.

- Cloud Agents: These agents are part of a remote application that needs access to the messages and data on the platform. A cloud agent would subscribe to topics of interest to the remote application and would also allow it to publish data to the platform.

- Control Agents: These agents control the devices of interest and interact with other resources to achieve a goal.

Platform Services:

- Message Bus: All agents and services publish and subscribe to topics on the message bus. This provides a single and uniform interface that abstracts the details of devices and agents from each other. At the most basic level, agents and components running in the platform produce and consume messages. The details of how agents produce events and how they process received events are left up to the agents.

- Weather Information: This agent periodically retrieves data from the Weather Underground site. It then reformats the data and publishes it out to the platform on a weather topic.

- Application Scheduling: This service allows the scheduling of agents’ access to devices to prevent conflicts

- Logging Service: Agents can publish integer or double data to arbitrary paths to a logging topic and this service will push them to the sMAP historian for later analysis. The primary use of the Logging Service is to allow agents to record actions or results resulting from the agent executing its services.

- MODBUS-based Device Interface: The MODBUS driver publishes device data onto the message bus. It also handles the locking of devices to prevent multiple conflicting directives.

3.0 Architectural Overview

Figure 1 shows the various components of the Transactional Network. The device interface communicates to the HVAC controller using MODBUS. It periodically scrapes data off the controller and both pushes data to the sMAP historian and publishes data to the message bus on a topic for each device. The device interface also responds to lock and control commands published on the requests topic. Agents must first request and receive a device lock. The lock gives the agent sole command access for the device. Scheduling of lock access for agents allows for greater control of when agents run and is user configurable. The actuator agent, which is detailed later in Section 5, controls execution of this schedule. A user configurable lock timeout will release an agent's lock on a device after a pre-determined time of inactivity.

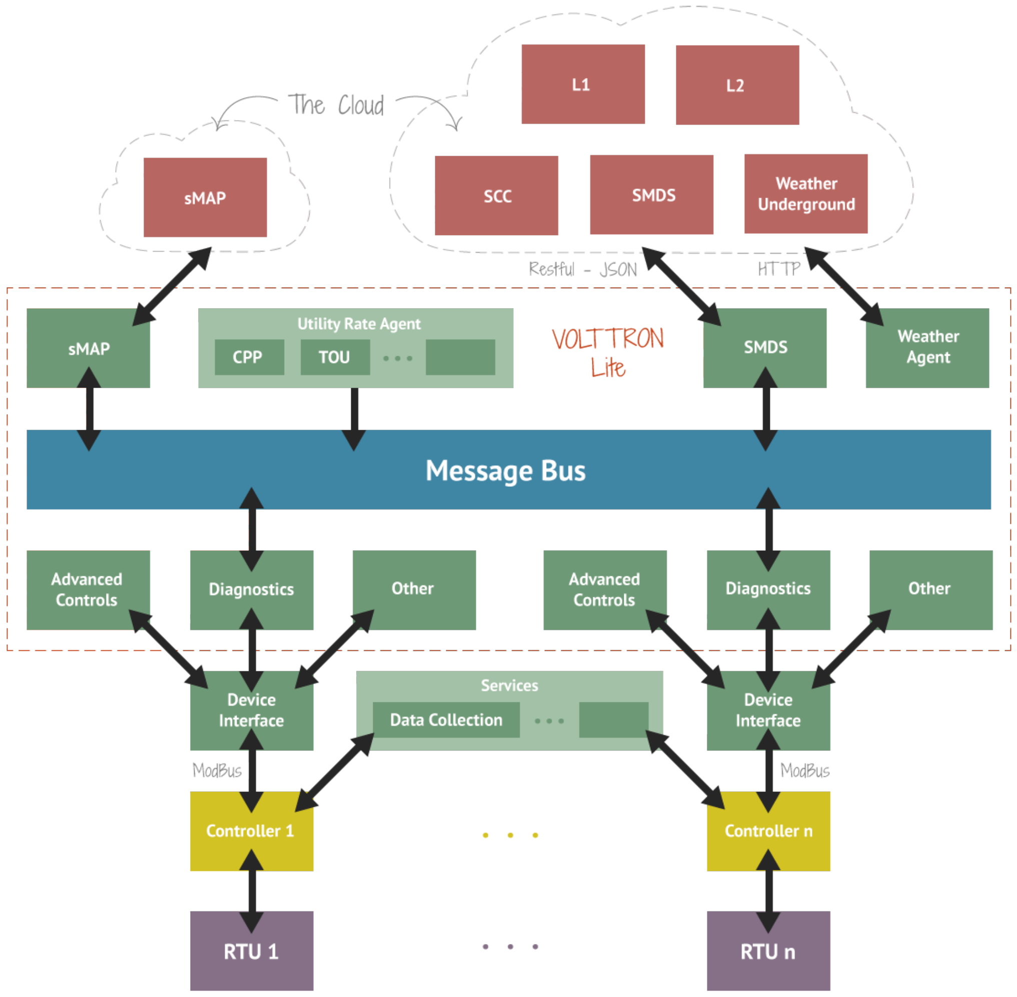

Figure 1: Illustration of the various components of the Transactional Network

The sMAP box in Figure represents the archiver agent, which allows agents to request data from sMAP over the message bus. This isolates agents from the historian and would allow the platform to use different or additional solutions. For example, since sMAP does not accept string data, a separate database could be used on the backend, and the interface to the agents would remain unchanged.

Agents and platform services shown in Figure 1 communicate with each other via the message bus using publish/subscribe over a variety of topics. For example, the weather agent would publish weather information to a “weather” topic that interested agents would subscribe to. The platform itself publishes platform related messages to the “platform” topic (such as “shutdown”). Topics are hierarchical following the format “topic/subtopic/subtopic”, allowing agents to get as general or as specific as they want with their subscriptions. Agents could subscribe to “weather/all” and get all data or “weather/temperature” for only temperature data.

3.1 Cloud Agent Use Case

The smart monitoring and diagnostic system (SMDS) agent provides a good example of a Cloud proxy agent (Figure 2). This agent does not use live data from the controller but pulls it in hourly batches from sMAP and sends it to an external application.

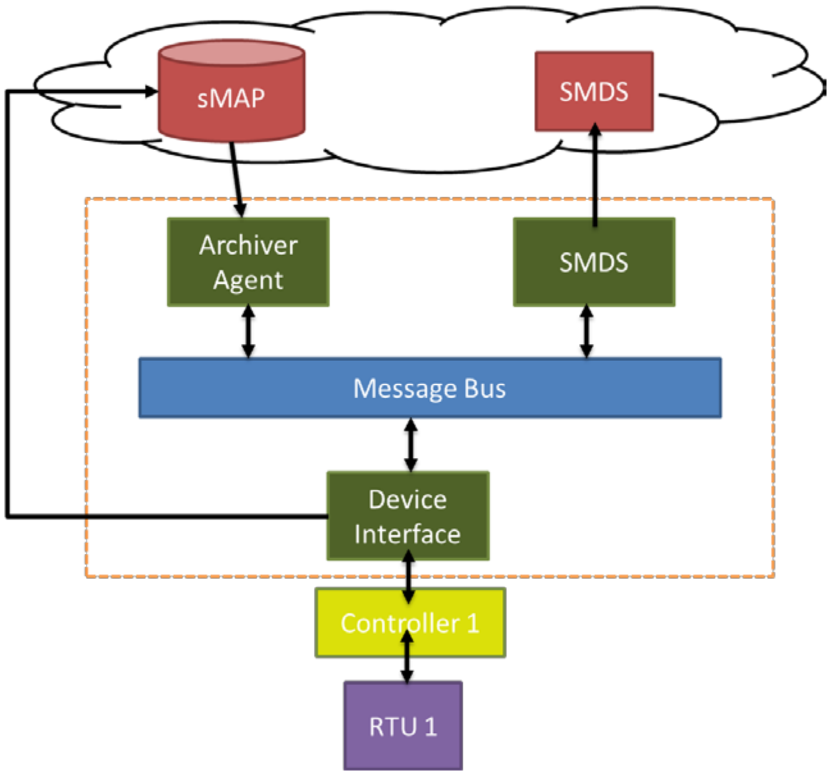

Figure 2: An example agent

- The data scraping device interface takes readings from the HVAC controller every minute and both pushes that data to sMAP and publishes out on the message bus
- Every hour, the SMDS proxy agent publishes a request to the archiver agent for the last hour of catalyst data for the points: unit power, supply fan speed, and outdoor air temperature
- The archiver agent queries sMAP and publishes the results on the message bus
- The SMDS agent receives its data, reformats it, and the pushes it to the SMDS application in the Cloud.

3.2 General Agent Use Case

This general use case is meant to illustrate features of the platform and does not describe actual control agents (Figure 3).

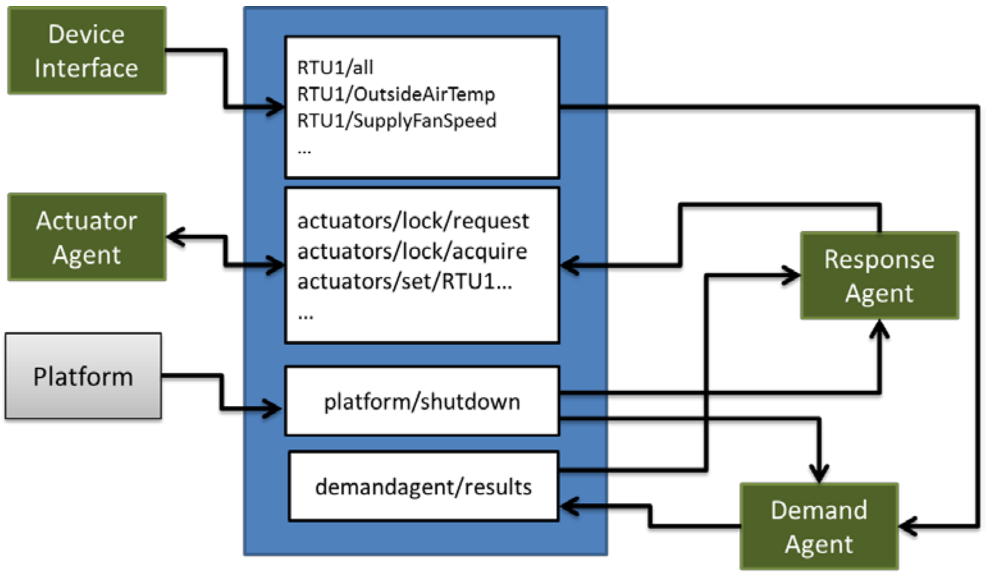

Figure 3: Illustration of a general use case

Agents in this example are each publishing and subscribing to various topics:

Device Interface:

- Subscriptions: none
- Publishes: RTU1 topics

Actuator Agent

- Subscriptions: actuators topics
- Publishes: actuator/lock/results

Platform

- Subscriptions: none
- Publishes: platform topics

Response Agent

- Subscriptions: platform topics, demand agent/results
- Publishes: actuator/lock/request, actuators/set

Demand Agent

- Subscriptions: platform topics, RTU1 data topics
- Publishes: demand agent/results

- The device interface publishes data from the RTU onto topics for each point as well as an “all” topic
- The demand agent subscribes to the RTU1 topics to gather data on the behavior of the RTU
- The demand agent calculates an expected demand forecast and publishes it on its results topic

- Response agent subscribes to events from the demand agent topic and combines it with other forecasts from other agents and compile a set of RTU commands to execute
- Response agent publishes a lock request
- Actuator agent assigns the lock and publishes a lock result response of success
- Response agent receives the lock success then publishes a series of set messages
- Actuator agent receives the set messages and issues the commands to the controller
- Response agent gets the results of the set commands
- At some point, the platform publishes a shutdown command
- Demand and response agents receive shutdown message and exit

4.0 Platform Base

The base platform has a set of capabilities that the platform services are built upon.

4.1 Command Structure

The VOLTTRON platform uses an rpc-based control interface that allows it to receive commands for management of the platform.

disable-agent	prevent agent from starting automatically
enable-agent	enable agent to start automatically
help	display help about commands
install-executable	install agent executable
list-agents	list agents
list-executables	list agent executables
load-agent	install agent launch file
remove-executable	remove agent executable
run-agent	run agent(s) defined in config file(s)
shutdown	stop all agents
start-agent	start installed agent
stop-agent	stop running agent
unload-agent	remove agent launch file.

Through the use of these commands, agents can be installed and set up to run automatically on the platform.

4.2 Autostart

Agents set up to autostart will start up with the platform. A naming convention has been established so that agent configuration files ending with .service (indicating a platform agent configuration file) will start before any .agent files. This allows the platform time to start up and begin providing these essential platform services.

4.3 Agent Execution

Agents executed on the platform must first be installed using the platform commands. In the case of Python-based agents, they should be built as an executable archive file (Python egg[1]). A JSON[2]-based launch configuration file is also installed, which tells the platform how to launch the agent and also contains configuration information for the agent.

Commands to set up an agent are as follows (a more detailed discussion is found in Section 5):

- Install the agent executable: volttron-ctrl install-executable <path to .egg file>
- Install agent launch file: volttron-ctrl load-agent <path to .json launch file> [<new agent name>]
- Enable automatic starting of agent: volttron-ctrl enable-agent <agent name>
- Test start the agent: volttron-ctrl start <agent name>

[1] http://mrtopf.de/blog/en/a-small-introduction-to-python-eggs/
[2] http://www.json.org/

5.0 Platform Services

The section describes various platform services supported by VOLTTRON Lite software.

5.1 Message Bus (ZMQ)

The messaging bus in VOLTTRON utilizes ZeroMQ[1] to provide a scalable multi-language solution for inter-agent communication. Agents post to topics and other agents can subscribe to their events if they are interested. Agents that need to work together can establish their own topics for communication. Utilities in the platform allow agents to have methods triggered by a message coming through on a subscribed topic that meets certain requirements. For example, an agent method for changing a set point based on outdoor air temperature would be triggered by events on an RTU's outdoor air temperature topic.

5.1.1 Topics

Agents in VOLTTRON Lite communicate with each other using a publish/subscribe mechanism built on the Zero MQ Python library. This allows for great flexibility because topics can be created dynamically, and the messages sent can be any format, as long as the sender and receiver understand it. An agent with data to share publishes to a topic, then any agents interested in that data subscribe to that topic.
While this flexibility is powerful, it could also lead to confusion if some standard is not followed. The current conventions for communicating in the VOLTTRON Lite are:

- Topics and subtopics follow the format: topic/subtopic/subtopic.
- Agents (subscribers) can subscribe to any and all levels. Subscriptions to "topic" will include messages for the base topic and all subtopics. Subscriptions to "topic/subtopic1" will only receive messages for that subtopic and any children subtopics. Subscriptions to empty string ("") will receive ALL messages. This is not recommended.
- All agents should subscribe to the "platform" topic. This is the topic the VOLTTRON Lite will use to send messages, such as "shutdown".

Agents should set the "From" header. This will allow agents to filter on the "To" message sent back. This is especially useful for requests to the archiver agent, so agents do not receive replies not meant for their request.

Actuator Agent Topics:

- actuators/get/<actuation point>
- actuators/set/<actuation point>
- actuators/lock/<acquire, result, release>/<device>
- actuators/schedule_announce

Archiver Agent Topics:

- archiver/request/<data point>
- archiver/response/<data point>

[1] http://zeromq.org/

Platform Topics

- platform/<subtopic>

5.2 MODBUS Based Device Interface

The MODBUS-based device interface allows users to define data and actuation points in a configuration file for devices with a MODBUS interface, as shown in Figure 4. The driver then periodically publishes data on the message bus, where agents can then subscribe to some or all of the readings. Agent developers only need to specify the topics of interest to receive data instead of developing their own MODBUS interface. This driver also pushes data to sMAP, which provides a historian function for later experimental analysis as well as allowing agents to retrieve non-live data. An actuator agent allows other agents to send control signals to the devices by publishing their directives to its topic after obtaining a lock. This agent also maintains a schedule of device accesses in cases where time must be reserved to take certain actions.

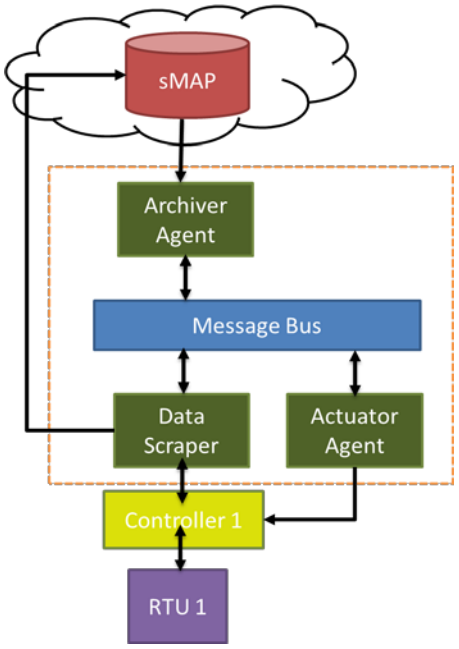

Figure 4: MODBUS device interface details

5.2.1 Catalyst Controller

The HVAC controller used by the Transactional Network project is the Catalyst[2] controller. This controller can be retrofitted onto HVAC units the platform would otherwise be unable to easily interact with. The Catalyst also contains its own algorithms and logic for operating the HVAC when no VOLTTRON agents are performing actions.

The MODBUS driver has been developed to support communication and control of the Catalyst controller, but it has been written to be as generic as possible.

[2] http://transformativewave.com/CATALYST

5.2.2 Discussion Options for Generic Driver

The MODBUS driver has been written to be as generic as possible and can handle most types of data including: shorts, long ints, floating point, double, and string. In the context of the sMAP driver, it is restricted to all variants of integers and floats becaause the historian cannot handle any other types. Certain devices may use a byte-ordering scheme, which is incompatible with this service. For instance, the Dent power meter uses missed Endian types (byte ordering), which cannot be handled in a generic way. Additional code will be required for these special cases.

5.3 Archiver (sMAP Query Service)

The archiver agent allows agents to specify a data point and time range to receive historical data. Agents can use the archiver mechanism to build up a baseline of building sensor readings before making control decisions. VOLTTRON also allows agents to log data to the historian by simply publishing their data to a logging topic. The paths and data points for this logging are dynamic, allowing the agents to use any scheme they wish.

5.4 Logger (Agent Access for Writing to sMAP)

A mechanism for storing data in SMAP has been provided. The data logger is written as a sMAP driver, but receives information in ZeroMQmessage sent to a topic prefixed with datalogger/log. The source name is configured at the time the platform auto-starts and is defined in the MODBUS driver configuration file. The rest of the topic is extracted and used as the path in sMAP for the data point. If the path already exists, the time series items will be added to the end of the series. For example, if the source name is 'test data', and the topic we publish data to is datalogger/log/campus1/building1/testdata, then our data will be posted to the time series under campus1/building1/testdata in the source name 'test data'.

5.5 Actuator (Commands to Devices)

This service allows agents to send control commands to devices in the platform. Agents communicate with the actuator over the message bus by publishing to actuator topics for lock requests and control commands. The steps for an agent to issue a command are as follows:

- Agent requests lock
- Actuator assigns lock or rejects request
- If successful, agent publishes command signals
- Actuator issues command signals from agents with locks
- When agent finishes task, it releases the lock

The actuator works on points that have been set as read/write in the MODBUS configuration file. The following is an example of some available points:

```
Point Name,PNNL Point Name,Units,Units Details,MODBUS IO Type,Read / Write,Point Address,Notes

CO2Stpt,ReturnAirCO2Stpt,PPM,1000.00 (default),Holding Register Float,Read/Write,1011,Setpoint to enable demand control ventilation

Cool1Spd,CoolSupplyFanSpeed1,%,0.00 to 100.00 (75 default),Holding Register Float,Read/Write,1005,Fan speed on cool 1 call

Cool2Spd,CoolSupplyFanSpeed2,%,0.00 to 100.00 (90 default),Holding Register Float,Read/Write,1007,Fan speed on Cool2 Call
```

5.6 Scheduler

The actuator agent also serves as a scheduling agent allowing agents to reserve devices according to a set schedule. This ensures that agents that need access to a device for an experiment will not be locked out by other devices during the duration of the test.

Currently, the schedule is expressed in the actuator agent’s configuration file and allows a set of agents to be given access to a device’s lock for the time period specified. This schedule is repeated daily and can be modified by switching to a different file.

This simple scheduler is still under development and more functionality will be added in the future to increase flexibility.

5.7 Weather

This simple weather service publishes Weather Underground based weather data to the platform and also serves as an example of a Cloud agent. It retrieves data based on a zip code setting in its configuration file and publishes data to a weather topic on the platform.

6.0 Agent Support

The VOLTTRON Lite software provides a number of features to help agents with various functions, those features are described in this section.

6.1 PythonBased Agent Description

The base agent class handles all the default functionality (such as responding to platform commands) and provides hooks for additional functionality. In addition to easing development, it also helps agents behave correctly in the platform. For simple agents, all that may be needed is a few lines of code for a function triggered on an event of interest and that produces an event.

6.2 Topics Utilities

The topics.py file contains constants for common topics used by the platform. These include topics for platform messages, the archiver, weather, and actuator agent. It also contains templates that can be used to fill out topics with the specifics of paths for the platform the agent is executing on. This allows topics to be constructed from agent configuration information and prevents needing to change code to run on a different platform.

The rtu_path dictionary contains the campus, building, and unit information. When this is passed to the topics.ARCHIVER_RESPONSE method with the point information, a specific topic path is constructed that the agent can then be used for subscriptions.

topics.ARCHIVER_RESPONSE(point='OutsideAirTemperature', **rtu_path)

6.3 Headers

Messages published in VOLTTRON contain a list of header name/value pairs. These headers can contain any information agents wish to use to identify the content and topics of the message to allow other agents to filter their subscriptions. Several basic headers are recommended to help the platform work with them. These are: From, To, Content-Type, RequesterID (for ActuatorAgent).

6.4 Utility Decorators

VOLTTRON Lite makes use of Python "decorators"[1] to enable agents to specify when certain methods are executed simply by adding some markup text to their method. These decorators greatly decrease the amount of code that needs to be written and provides a flexible mechanism for introducing additional support features in the future.

6.4.1 Timing Decorators

Periodic (@periodic) and one-shot timer (@timer) decorators are provided that can declare a method to be run once after a set delay or recurring every period of time.

[1] https://wiki.python.org/moin/PythonDecorators

6.4.2 Subscription Matching Decorators

The set of decorators in matching.py are a shortcut to using subscriptions in VOLTTRON Lite. Instead of manually setting up each subscription, developers can instead decorate methods with matching decorators, which specify what messages will trigger this method. Agents can match on any part of the topic path and any header they wish. For instance:

```
@matching.match_headers({headers_mod.TO: agent_id})

 @matching.match_exact(topics.ARCHIVER_RESPONSE(point='OutsideAirTemperature', **rtu_path))

def on_temp_response(self, topic, headers, message, match):
```

These decorators specify that the “on_temp_response” method is to be called whenever a message is published on the Archiver Reponse topic for “OutsideAirTemperature” and the message headers contain a name value pair of “To” and the id of this agent.

Messages can be matched by full or partial topics with wildcard matching and can also be filtered by header information. The module also allows topics to be based on the contents of the agent’s configuration file, which allows the same agent code to be utilized across different instances of the platform by changing only the configuration file.

6.5 Other Utilities

There are several utilities for building topics for subscriptions, as well as a collection of constants to simplify agent development. Additionally, utilities for scheduling agent actions are provided.

7.0 Building and Deploying Agents

This section describes how to build and deploy agents on the Transactional Network platform.

7.1 Building an Agent Egg

Python agents executed on the platform are compiled into an "egg," which is an executable archive file. This is then used by the platform (along with an agent configuration file) to launch the agent.

7.2 Launch File

Agents are launched using the contents of a launch file. This can actually support executing non-Python applications by filling out the "exec" portion of the file. These files enable multiple instances of an agent to be launched with different configuration parameters. An agent can be written to work with RTU HVAC units individually. An instance will be launched for each HVAC with only the identification of the device different in the launch files. As much configuration information should be in the launch file as possible to minimize code changes during deployments.

For instance, this is the launch configuration for the listener agent, which is a sample agent provided with the platform. It will execute the listener agent's egg and pass the command line options for a configuration file (itself) and the subscription and publication urls to tie into the platform's message bus.

```
{
   "agent": {
      "exec": "listeneragent-0.1-py2.7.egg --config \"%c\" --sub \"%s\" --pub \"%p\""
   },
   "agentid": "listener1",
   "message": "hello"
}
```

7.3 Deployment using Commands

The VOLTTRON Lite command capability can be used to deploy agents into the platform

- Build the agent egg using the build agent script
  - volttron/scripts/build-agent.sh AFDDAgent
- Make the egg executable
  - chmod +x agents/afddagent-0.1-py2.7.egg
- Install the executable into the platform
  - bin/volttron-ctrl install-executable agents/afddagent-0.1-py2.7.egg
- Load an instance of the agent by loading the launch file
  - bin/volttron-ctrl load-agent agents/AFDDAgent/afddagent_twt1.launch.agent
- Start the agent manually
  - bin/volttron-ctrl start-agent afddagent_twt1.launch.agent
- Enable the agent to autostart when the platform starts
  - bin/volttron-ctrl enable-agent afddagent_twt1.launch.agent

8.0 Future Plans

The VOLTTRON Lite platform is a work in progress and several new features are planned for development.

8.1 BACnet Support

BACnet device communication support will be added to the VOLTTRON Lite software. This will follow the model of the MODBUS driver in that the agents will publish lock requests and commands on the message bus and not interact directly with the devices.

8.2 Multi-Building Coordination

Agents will be able to communicate with agents on other VOLTTRON platforms. This will enable agents to coordinate activities between buildings and also enable a hierarchal arrangement of platforms. A platform could coordinate a neighborhood of houses coordinating a collection of appliances, for instance.

9.0 Additional Information

The VOLTTRON Lite platform maintains a wiki at: https://svn.pnl.gov/RTUNetwork
This will be updated as features are added or modified. Please see this for additional details or contact the VOLTTRON Lite development team.

9.1 Listener Agent

The listener agent 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.

9.1.1 Explanation of Listener Agent

Listeneragent publishes a heartbeat message so it will use the PublishMixin. It also extends base agent to get the default functionality. When creating agents, Mixins should be first in the class definition.

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

ListenerAgent subscribes to all topics by using volttron.lite.matching. This package contains decorators for simplifying subscriptions. Listener agent uses match_all to receive all messages:

```
    @matching.match_all
    def on_match(self, topic, headers, message, match):
        '''Use match_all to receive all messages and print them out.'''
        print "Topic: {topic}, Headers: {headers}, Message:
{message}".format(
                topic=topic, headers=headers, message=message)
```

Listener agent uses the @periodic decorator to execute the pubheartbeat method every HEARTBEAT_PERIOD seconds where HEARTBEAT_PERIOD is specified in the settings.py file. To publish, it creates a header object to set the content type of the message, the time the event was created, and the identification of the agent sending it. This allows other agents to filter messages of a certain type or from a certain agent. It also allows them to interpret the content appropriately. The message is then published out on the heartbeat topic.

```
    # Demonstrate periodic decorator and settings access
    @periodic(settings.HEARTBEAT_PERIOD)
    def publish_heartbeat(self):
        '''Send heartbeat message every HEARTBEAT_PERIOD seconds.

        HEARTBEAT_PERIOD is set and can be adjusted in the settings module.
        '''
        now = datetime.utcnow().isoformat(' ') + 'Z'
        headers = {
            'AgentID': self._agent_id,
            headers_mod.CONTENT_TYPE: headers_mod.CONTENT_TYPE.PLAIN_TEXT,
            headers_mod.DATE: now,
        }
        self.publish('heartbeat/listeneragent', headers, now)
```

Appendix A

The following section contains additional content taken from the Transactional Network wiki. For up to date versions, please go to the wiki directly: https://svn.pnl.gov/RTUNetwork/wiki/

Definition of Terms

- JsonRPC: JSON-encoded remote procedure call
- JSON: JavaScript? object notation is a text-based, human-readable, open data interchange format, similar to XML, but much better
- 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
- ZeroMQ or ØMQ: A library used for interprocess and intercomputer communicationc
- Modbus: Communications protocol for talking with industrial electronic devices
- SSH: Secure shell is a network protocol providing encryption and authentication of data using public-key cryptography
- SSL: Secure sockets layer is a technology for encryption and authentication of network traffic based on a chain of trust
- TLS: Transport layer security is the successor to SSL

Download in other formats:
Plain Text

Basic requirements

- Eggsecutable file launchable by platform
- Subscribe and publish to ZMQ topics
  - Shut down when shutdown message received on platform topic
  - React to messages, send out commands
- BaseAgent
  - Handles subscribing and reacting to mandatory topics
  - Provides a pattern to follow
  - Provides hooks for logic reacting to messages
- ExampleAgents
  - Illustrates usage of platform services
  - Can be modified for specific applications

Required Sofware: Linux

The following packages will need to be installed if they are not already:

- Install Mercurial: sudo apt-get install mercurial
- Install Python DevTools: sudo apt-get install python-dev
- Install g++: sudo apt-get install g++
- Install libevent-dev: sudo apt-get install libevent-dev

Download in other formats:
Plain Text

RTU Network Development in Eclipse

The Eclipse IDE has a Python plugin which is already installed in the RTU VM. In order to setup Eclipse outside the VM, download the IDE from: http://www.eclipse.org/ or through the Software Manager in Linux.

The RTU Network code is stored in a Mercurial repository. There is a plugin available for Eclipse that makes development more convenient (note: you must have Mercurial already installed on the system and have built the project):

- Help -> Install New Software
- Click on the "Add" button
- For name use: Mercurial
- For location: http://mercurialeclipse.eclipselabs.org.codespot.com/hg.wiki/update_site/stable
- After hitting OK, check the box for: MercurialEclipse? Stable Releases
- Click through Next, Agree to Terms, then Finish
- Allow Eclipse to restart

After installing Eclipse, you must add the PyDev? plugin to the environment. In Eclipse:

- Help -> Install New Software
- Click on the "Add" button
- For name use: PyDev?
- For location: http://pydev.org/updates
- Check the box for PyDev?
- Click through Next, Agree to Terms, Finish
- Allow Eclipse to restart

The project can now be checked out from the repository into Eclipse.

- Window -> Show View -> Other -> Mercurial -> Mercurial Repositories
- In the Mercurial Repositories View, click on the New Repositories button
- For Repository location, enter hg clone https://bitbucket.org/berkeleylab/rtunetwork
- Enter your username and password, hit ok
- Right-click on the repository, select Clone
- Enter username/password if not filled in
- Change clone directory name if desired (wiki assumes use of the default)
- Next, Finish
- The project must now be built outside Eclipse. Please follow the directions here and skip the hg clone portion.
- After changing the filesystem outside Eclipse, right-click on the project name and select Refresh
- PyDev? must now be pointed at the correct version of Python.

    - Window-> Preferences
    - Expand PyDev?
    - Select Interpreter-Python
    - Hit New
    - Hit Browse and browse to the Python in the bin directory of the rtunetwork project. Then hit Ok
    - Select All, then Hit Ok
    - You may need to redo this after a platform update and buildout
- In the Project/PackageExplorer view on the left, right-click on the project, PyDev?-> Set as PyDev? Project
- Switch to the PyDev? perspective (if it has not already switched), Window -> Open Perspective -> PyDev?

Eclipse should now be configured to use the project's environment. To test the installation:

- Setup a run configuration for the platform
    - In the Package Explorer view, open the bin folder
    - Righ-click on volttron-lite and select Run As -> Python Run (this will create a run configuration but fail)
    - On the menu bar, pick Run -> Run Configurations...
    - Under Python Run pick "rtunetwork volttron-lite"
    - Click on the Argument tab
    - Change Working Directory to Default
    - In the Program arguments box, add: "-c dev-config.ini" This sets up running the platform in a development mode
    - Click Run, this launches the platform. If the run does not succeed, click the all stop icon (two red boxes overlaid) on the console and then retry.
- Setup a run configuration for the ListenerAgent
    - In the Package Explorer view, open Agents -> ListenerAgent --> listener
    - Righ-click on listeneragent.py and select Run As -> Python Run (this will create a run configuration but fail)
    - On the menu bar, pick Run -> Run Configurations...
    - Under Python Run pick "rtunetwork listeneragent.py"
    - Click on the Argument tab
    - Change Working Directory to Default
    - In the Program arguments box, add: "--config Agents/ListenerAgent/listeneragent.launch.json --pub ipc:///tmp/volttron-lite-agent-publish --sub ipc:///tmp/volttron-lite-agent-subscribe" This launches the agent with the dev settings the platform is using
    - Click Run, this launches the agent
    - You should see the agent start to publish and receive its own heartbeat message

Download in other formats:

Plain Text

Building the Project

The Volttron Lite project includes scripts which automatically pull down dependencies and build the libraries: bootstrap is a one-time use script. Use it after the initial checkout, then use "bin/buildout -N" after that.

Ensure you have installed the required packages before proceeding. Especially if you intend to develop in Eclipse, we recommend creating a directory: ~/workspace In this directory:

- hg clone https://bitbucket.org/berkeleylab/rtunetwork
- cd rtunetwork
- ./bootstrap
  - If bootstrap fails before finishing (for instance from a timeout), run: bin/buildout -N

Note: If bootstrap or buildout fails, try "bin/buildout -N" again. Also, some packages (especially numpy) can be very verbose when they install. Please wait for the wall of text to finish.

To test that installation worked, start up the platform:

- Edit the dev-config.ini file to ensure the paths match up to your installation
- bin/volttron-lite -c dev-config.ini
- If it starts with no errors then your setup is correct
- If you are developing in Eclipse, then you should update the Python path at this point. See: EclipseDevEnvironment

To test agent deployment and messaging, build and deploy ListenerAgent: From the rtunetwork directory

- volttron/scripts/build-agent.sh ListenerAgent
- chmod +x Agents/listeneragent-0.1-py2.7.egg
- cp Agents/listeneragent-0.1-py2.7.egg bin/
- bin/volttron-ctrl run_agent Agents/ListenerAgent/listeneragent.launch.json

Download in other formats:
Plain Text

ListenerAgent

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 ListenerAgent

ListenerAgent publishes a heartbeat message so it will use the PublishMixin?. It also extends BaseAgent to get the default functionality. When creating agents, Mixins should be first in the class definition.

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

ListenerAgent subscribes to all topics by using volttron.lite.matching?. This package contains decorators for simplifying subscriptions. ListenerAgent uses match_all to receive all messages:

```
    @matching.match_all
    def on_match(self, topic, headers, message, match):
        '''Use match_all to receive all messages and print them
out.'''
        print "Topic: {topic}, Headers: {headers}, Message:
{message}".format(
                topic=topic, headers=headers, message=message)
```

ListenerAgent uses the @periodic decorator to execute the pubheartbeat method every HEARTBEAT_PERIOD seconds where HEARTBEAT_PERIOD is specified in the settings.py file. In order to publish, it creates a Header object to set the ContentType? of the message, the time the event was created, and the id of the agent sending it. This allows other agents to filter out messages of a certain type or from a certain agent. It also allows them to interpret the content appropriately. The message it then published out on the heartbeat topic.

```
    # Demonstrate periodic decorator and settings access
    @periodic(settings.HEARTBEAT_PERIOD)
    def publish_heartbeat(self):
        '''Send heartbeat message every HEARTBEAT_PERIOD seconds.

        HEARTBEAT_PERIOD is set and can be adjusted in the settings
module.
        '''
        now = datetime.utcnow().isoformat(' ') + 'Z'
        headers = {
            'AgentID': self._agent_id,
            headers_mod.CONTENT_TYPE:
```

```
headers_mod.CONTENT_TYPE.PLAIN_TEXT,
              headers_mod.DATE: now,
         }
         self.publish('heartbeat/listeneragent', headers, now)
```

Download in other formats:

Plain Text

Weather Agent Topics

Topics used by the WeatherAgent with example output.

['weather/all', '{"temperature": {"windchill_f": "NA", "temp_f": 69.1, "heat_index_f": "NA", "heat_index_string": "NA", "temp_c": 20.6, "feelslike_c": "20.6", "windchill_string": "NA", "feelslike_f": "69.1", "heat_index_c": "NA", "windchill_c": "NA", "feelslike_string": "69.1 F (20.6 C)", "temperature_string": "69.1 F (20.6 C)"}, "cloud_cover": {"visibility_mi": "10.0", "solarradiation": "", "weather": "Clear", "visibility_km": "16.1", "UV": "6"}, "location": {"display_location": {"city": "Richland", "full": "Richland, WA", "elevation": "121.00000000", "state_name": "Washington", "zip": "99352", "country": "US", "longitude": "-119.29721832", "state": "WA", "country_iso3166": "US", "latitude": "46.28490067"}, "local_tz_long": "America/Los_Angeles", "observation_location": {"city": "Richland, Richland", "full": "Richland, Richland, Washington", "elevation": "397 ft", "country": "US", "longitude": "-119.304375", "state": "Washington", "country_iso3166": "US", "latitude": "46.285866"}, "station_id": "KWARICHL21"}, "time": {"local_tz_offset": "-0700", "local_epoch": "1368724778", "observation_time": "Last Updated on May 16, 10:18 AM PDT", "local_tz_short": "PDT", "observation_epoch": "1368724692", "local_time_rfc822": "Thu, 16 May 2013 10:19:38 -0700", "observation_time_rfc822": "Thu, 16 May 2013 10:18:12 -0700"}, "pressure_humidity": {"relative_humidity": "40%", "pressure_mb": "1014", "pressure_trend": "-"}, "precipitation": {"dewpoint_string": "44 F (7 C)", "precip_1hr_in": "0.00", "precip_today_in": "0.00", "precip_today_metric": "0", "precip_today_string": "0.00 in (0 mm)", "dewpoint_f": 44, "dewpoint_c": 7, "precip_1hr_string": "0.00 in (0 mm)", "precip_1hr_metric": " 0"}, "wind": {"wind_degrees": 3, "wind_kph": 2.7, "wind_gust_mph": "3.0", "wind_mph": 1.7, "wind_string": "From the North at 1.7 MPH Gusting to 3.0 MPH", "pressure_in": "29.94", "wind_dir": "North", "wind_gust_kph": "4.8"}}']

['weather/temperature/all', '{"windchill_f": "NA", "temp_f": 69.1, "heat_index_f": "NA", "heat_index_string": "NA", "temp_c": 20.6, "feelslike_c": "20.6", "windchill_string": "NA", "feelslike_f": "69.1", "heat_index_c": "NA", "windchill_c": "NA", "feelslike_string": "69.1 F (20.6 C)", "temperature_string": "69.1 F (20.6 C)"}']

['weather/temperature/windchill_f', 'NA']

['weather/temperature/temp_f', '69.1']

['weather/temperature/heat_index_f', 'NA']

['weather/temperature/heat_index_string', 'NA']

['weather/temperature/temp_c', '20.6']

['weather/temperature/feelslike_c', '20.6']

```
['weather/temperature/windchill_string', 'NA']

['weather/temperature/feelslike_f', '69.1']

['weather/temperature/heat_index_c', 'NA']

['weather/temperature/windchill_c', 'NA']

['weather/temperature/feelslike_string', '69.1 F (20.6 C)']

['weather/temperature/temperature_string', '69.1 F (20.6 C)']

['weather/cloud_cover/all', '{"visibility_mi": "10.0", "solarradiation": "", "weather":
"Clear", "visibility_km": "16.1", "UV": "6"}']

['weather/cloud_cover/visibility_mi', '10.0']

['weather/cloud_cover/solarradiation', ]

['weather/cloud_cover/weather', 'Clear']

['weather/cloud_cover/visibility_km', '16.1']

['weather/cloud_cover/UV', '6']

['weather/location/all', '{"display_location": {"city": "Richland", "full": "Richland, WA",
"elevation": "121.00000000", "state_name": "Washington", "zip": "99352", "country":
"US", "longitude": "-119.29721832", "state": "WA", "country_iso3166": "US",
"latitude": "46.28490067"}, "local_tz_long": "America/Los_Angeles",
"observation_location": {"city": "Richland, Richland", "full": "Richland, Richland,
Washington", "elevation": "397 ft", "country": "US", "longitude": "-119.304375",
"state": "Washington", "country_iso3166": "US", "latitude": "46.285866"}, "station_id":
"KWARICHL21"}']

['weather/location/display_location/all', '{"city": "Richland", "full": "Richland, WA",
"elevation": "121.00000000", "state_name": "Washington", "zip": "99352", "country":
"US", "longitude": "-119.29721832", "state": "WA", "country_iso3166": "US",
"latitude": "46.28490067"}']

['weather/location/display_location/city', 'Richland']

['weather/location/display_location/full', 'Richland, WA']

['weather/location/display_location/elevation', '121.00000000']

['weather/location/display_location/state_name', 'Washington']

['weather/location/display_location/zip', '99352']

['weather/location/display_location/country', 'US']

['weather/location/display_location/longitude', '-119.29721832']

['weather/location/display_location/state', 'WA']

['weather/location/display_location/country_iso3166', 'US']

['weather/location/display_location/latitude', '46.28490067']
```

['weather/location/local_tz_long', 'America/Los_Angeles']

['weather/location/observation_location/all', '{"city": "Richland, Richland", "full": "Richland, Richland, Washington", "elevation": "397 ft", "country": "US", "longitude": "-119.304375", "state": "Washington", "country_iso3166": "US", "latitude": "46.285866"}']

['weather/location/observation_location/city', 'Richland, Richland']

['weather/location/observation_location/full', 'Richland, Richland, Washington']

['weather/location/observation_location/elevation', '397 ft']

['weather/location/observation_location/country', 'US']

['weather/location/observation_location/longitude', '-119.304375']

['weather/location/observation_location/state', 'Washington']

['weather/location/observation_location/country_iso3166', 'US']

['weather/location/observation_location/latitude', '46.285866']

['weather/location/station_id', 'KWARICHL21']

['weather/time/all', '{"local_tz_offset": "-0700", "local_epoch": "1368724778", "observation_time": "Last Updated on May 16, 10:18 AM PDT", "local_tz_short": "PDT", "observation_epoch": "1368724692", "local_time_rfc822": "Thu, 16 May 2013 10:19:38 -0700", "observation_time_rfc822": "Thu, 16 May 2013 10:18:12 -0700"}']

['weather/time/local_tz_offset', '-0700']

['weather/time/local_epoch', '1368724778']

['weather/time/observation_time', 'Last Updated on May 16, 10:18 AM PDT']

['weather/time/local_tz_short', 'PDT']

['weather/time/observation_epoch', '1368724692']

['weather/time/local_time_rfc822', 'Thu, 16 May 2013 10:19:38 -0700']

['weather/time/observation_time_rfc822', 'Thu, 16 May 2013 10:18:12 -0700']

['weather/pressure_humidity/all', '{"relative_humidity": "40%", "pressure_mb": "1014", "pressure_trend": "-"}']

['weather/pressure_humidity/relative_humidity', '40%']

['weather/pressure_humidity/pressure_mb', '1014']

['weather/pressure_humidity/pressure_trend', '-']

['weather/precipitation/all', '{"dewpoint_string": "44 F (7 C)", "precip_1hr_in": "0.00", "precip_today_in": "0.00", "precip_today_metric": "0", "precip_today_string": "0.00 in (0 mm)", "dewpoint_f": 44, "dewpoint_c": 7, "precip_1hr_string": "0.00 in (0 mm)", "precip_1hr_metric": " 0"}']

```
['weather/precipitation/dewpoint_string', '44 F (7 C)']

['weather/precipitation/precip_1hr_in', '0.00']

['weather/precipitation/precip_today_in', '0.00']

['weather/precipitation/precip_today_metric', '0']

['weather/precipitation/precip_today_string', '0.00 in (0 mm)']

['weather/precipitation/dewpoint_f', '44']

['weather/precipitation/dewpoint_c', '7']

['weather/precipitation/precip_1hr_string', '0.00 in ( 0 mm)']

['weather/precipitation/precip_1hr_metric', ' 0']

['weather/wind/all', '{"wind_degrees": 3, "wind_kph": 2.7, "wind_gust_mph": "3.0",
"wind_mph": 1.7, "wind_string": "From the North at 1.7 MPH Gusting to 3.0 MPH",
"pressure_in": "29.94", "wind_dir": "North", "wind_gust_kph": "4.8"}']

['weather/wind/wind_degrees', '3']

['weather/wind/wind_kph', '2.7']

['weather/wind/wind_gust_mph', '3.0']

['weather/wind/wind_mph', '1.7']

['weather/wind/wind_string', 'From the North at 1.7 MPH Gusting to 3.0 MPH']

['weather/wind/pressure_in', '29.94']

['weather/wind/wind_dir', 'North']

['weather/wind/wind_gust_kph', '4.8']
```

Download in other formats:
Plain Text

Exampler Controller Agent

This agent listens for outdoor temperature readings then changes the cool fan speed. It demonstrates pub/sub interaction with the RTU Controller.

Requirements for running this agent (or any agent wishing to interact with the RTU:

- Edit the driver.ini file to reflect the sMAP key, uuid, and other settings for your installation
- Activate the project Python from the project dir: . bin/activate
- Launch the smap driver by starting (from the project directory): twistd -n smap your_driver.ini
- Launch the ActuatorAgent just as you would launch any other agent

With these requirements met, the

- Subscribe to the outside air temperature topic.
- If the new reading is higher than the old reading then
    - Request the actuator lock for the rtu
- If it receives a lock request success it randomly sets the coolsupply fan to a new reading.
- If it does not get the lock, it will try again the next time the temperature rises.
- If the set result is a success, it releases the lock.

Agent Creation Walkthrough

It is recommended that developers look at the ListenerAgent before developing their own agent. That agent expresses the basic functionality this example with walk through and being familiar with the concepts will be useful.

Created Folders

- In the Agents directory, create a new folder TestAgent
- In TestAgent, create a new folder tester, this is the package where our python code will be created

Create Agent Code

- In tester, create a file called "__init__.py" which tells Python to treat this folder as a package
- In the tester package folder, create the file testagent.py
- Create a class called TestAgent
  - Import the packages and classes we will need:

```
import sys

from volttron.lite.agent import BaseAgent, PublishMixin
from volttron.lite.agent import utils, matching
from volttron.lite.messaging import headers as headers_mod
```

- This agent will extend BaseAgent to get all the default functionality
- Since we want to publish we will use the PublishMixin?. Mixins should be put first in the class definition

```
class TestAgent(PublishMixin, BaseAgent):
```

- We don't need to add anything to the BaseAgent init method so we will not create our own

Setting up a Subscription

We will set our agent up to listen to heartbeat messages (published by ListenerAgent). Using the matching package, we declare we want to match all topics which start with "heartbeat/listeneragent". This will give us all heartbeat messages from all listeneragents but no others.

```
    @match_start('heartbeat/listeneragent')
    def on_heartbeat_topic(self, topic, headers, message, match):
            print "TestAgent got\nTopic: {topic}, {headers},
Message: {message}".format(topic=topic, headers=headers,
message=message)
```

Argument Parsing and Main

Our agent will need to be able to parse arguments being passed on the command line by the agent launcher. Use the utils.default_main method to handle argument parsing and other default behavior. Create a main method which can be called by the launcher.

```
def main(argv=sys.argv):
    '''Main method called by the eggsecutable.'''
    utils.default_main(TestAgent,
                       description='Test Agent',
                       argv=argv)


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

Create Support Files for Agent

Volttron-Lite agents need some configuration files for packaging, configuration, and launching.

Packaging Configuration

In the TestAgent folder, create a file called "setup.py" (or copy the setup.py in ListenerAgent) which we will use to create an eggsecutable. This file sets up the name, version, required packages, method to execute, etc. for the agent. The packaging process will also use this information to name the resulting file.

```
from setuptools import setup, find_packages

packages = find_packages('.')
package = packages[0]

setup(
    name = package + 'agent',
    version = "0.1",
    install_requires = ['volttronlite'],
    packages = packages,
    entry_points = {
        'setuptools.installation': [
            'eggsecutable = ' + package + '.agent:main',
        ]
    }
)
```

Launch Configuration

In TestAgent, create a file called "testagent.launch.json". This is the file the platform will use to launch the agent. It can also contain configuration information for the agent.

For TestAgent,

```
{
    "agent": {
        "exec": "testeragent-0.1-py2.7.egg --config \"%c\" --sub
\"%s\" --pub \"%p\""
```

```
    },
    "agentid": "Test1",
    "message": "hello"
}
```

Packaging Agent

The agent code must now be packaged up for use by the platform. The build-agent.sh script will build the eggsecutable package using the setup.py file we defined earlier.

From the rtunetwork directory: "scripts/build-agent.sh TestAgent".

This creates an egg file in the Agents directory which, along with the launch configuration file, would be sent to a deployed platform for installation. For local testing, you may need to change permissions on the file: "chmod 777 Agents/testeragent-.1-py2.7.egg"

Then copy this to the rtunetwork/bin directory.

Testing the Agent

From the Command Line

- Ensure the egg has been copied to the bin directory
- Start the platform by running "bin/volttron-lite -n -c dev-config.ini"
- Launch the agent by running "bin/volttron-ctrl run_agent Agents/TestAgent/testagent.launch.json"

In Eclipse

- If you are working in Eclipse, create a run configuration for TestAgent based on the ListenerAgent configuration in EclipseDevEnvironment.
- Launch the platform
- Launch the TestAgent
- Launch the ListenerAgent TestAgent should start receiving the heartbeats from ListenerAgent

Download in other formats:
Plain Text

Messaging

Agents in VOLTTRON Lite communicate with each other using a publish/subscribe mechanism built on the Zero MQ Python library. This allows for great flexibility as topics can be created dynamically and the messages sent can be any format as long as the sender and receiver understand it. An agent with data to share publishes to a topic, then any agents interested in that data subscribe to that topic.

While this flexibility is powerful, it also could also lead to confusion if some standard is not followed. The current conventions for communicating in the VOLTTRON Lite are:

- Topics and subtopics follow the format: topic/subtopic/subtopic
- Subscribers can subscribe to any and all levels. Subscriptions to "topic" will include messages for the base topic and all subtopics. Subscriptions to "topic/subtopic1" will only receive messages for that subtopic and any children subtopics. Subscriptions to empty string ("") will receive ALL messages. This is not recommended.
- All agents should subscribe to the "platform" topic. This is the topic the VOLTTRON Lite will use to send messages to agents, such as "shutdown".

Agents should set the "From" header. This will allow agents to filter on the "To" message sent back. This is especially useful for requests to the ArchiverAgent so agents do not receive replies not meant for their request.

Topics

In VOLTTRON Lite

- platform - Base topic used by the platform to inform agents of platform events
- platform/shutdown - General shutdown command. All agents should exit upon receiving this. Message content will be a reason for the shutdown
- platform/shutdown_agent - This topic will provide a specific agent id. Agents should subscribe to this topic and exit if the id in the message matches their id.

RTU Controller Agent Topics

These topics are used by the RTU Controller Agent to publish readings from the RTU as well as send commands to it.

A complete list of available RTU Controller data points is here

Catalyst Data

- [RTU/campus1/building1/fakecatalyst1/FanFaultCode, '0.0']
- ['RTU/FanFaultCode/fakecatalyst1/building1/campus1', '0.0']

Catalyst Actuator

- ['RTU/actuators/set/campus1/building1/fakecatalyst1/ActClgSetPoint', 'My unique ID' , '75.5']
- ['RTU/actuators/get/campus1/building1/fakecatalyst1/ActClgSetPoint', 'My unique ID']

Locking

- ['RTU/actuators/lockget/campus1/building1/fakecatalyst1', 'My unique ID']
    - Returns lock result: RTUactuator/lockresult['ActClgSetPoint?','My unique ID', 'SUCCESS/FAILURE']
- ['RTU/actuators/lockrelease/campus1/building1/fakecatalyst1/ActClgSetPoint', 'My unique ID']
    - Returns lock result: RTUactuator/lockresult['ActClgSetPoint?','My unique ID', 'RELEASE/FAILURE']

Listen for actuators being set

- Can return actuator error (string)
- Depth first ['RTU/actuators/value/campus1/building1/fakecatalyst1/ActClgSetPoint', <set topic>, <requestor ID>, <value>]
- Breadth first ['RTU/actuators/value/building1/fakecatalyst1/campus1/ActClgSetPoint',<set topic>, <requestor ID>, <value>]

Platform Commands

To startup the platform, specify the config file with -c <config file>. To specify a log file, use: -l <filename>

```
bin/volttron-lite -c config.ini -l volttron.log
```

Full options:

```
Volttron Lite agent platform daemon

optional arguments:
  -b, --background      background (daemonize) the process
  -c FILE, --config FILE
                        read configuration from FILE
  --gid GID             change group to given GID; only used with
-b
  -l FILE, --log FILE   send log output to FILE instead of stderr
  -L FILE, --log-config FILE
                        read logging configuration from FILE
  -p FILE, --pid-file FILE
                        write process ID to FILE; only used with -
b
  -q, --quiet           decrease logger verboseness; may be used
multiple
                        times
  -s SECTION.NAME=VALUE, --set SECTION.NAME=VALUE
                        specify additional configuration
  --uid UID             change user to given UID; only used with -
b
  -v, --verbose         increase logger verboseness; may be used
multiple
                        times
  --help                show this help message and exit
  --version             show version information and exit
```

The platform can accept commands during operation using bin/volttron-ctrl <command>

```
usage: volttron-ctrl command [options]

Control volttron and perform other related tasks

list of commands:
  disable-agent       prevent agent from starting automatically
  enable-agent        enable agent to start automatically
  help                display help about commands
  install-executable  install agent executable
  list-agents         list agents
  list-executables    list agent executables
  load-agent          install agent launch file
  remove-executable   remove agent executable
  run-agent           run agent(s) defined in config file(s)
  shutdown            stop all agents
  start-agent         start installed agent
  stop-agent          stop running agent
  unload-agent        remove agent launch file
Use `volttron-ctrl -v` to show aliases and global options.
```

Download in other formats:

Plain Text

Platform Startup

With volttron-lite running, you need to perform the following commands:

- Install the agent executable: volttron-ctrl install-executable <path to .egg file>
- Install agent launch file: volttron-ctrl load-agent <path to .json launch file> [<new agent name>]
- Enable automatic starting of agent: volttron-ctrl enable-agent <agent name>
- Test start the agent: volttron-ctrl start <agent name>

Then restart volttron-lite and the agent should start automatically. Autostart can be skipped using the --skip-autstart command-line option.

By convention, agents should have either a .service or .agent suffix. .service agents are considered essential to the platform and are started before other agents. Below is an example.

```
[rtunetwork]$ . bin/activate
(volttron)[rtunetwork]$ cd Agents/ListenerAgent
(volttron)[ListenerAgent]$ python setup.py bdist_egg
...
creating 'dist/listeneragent-0.1-py2.7.egg' and adding
'build/bdist.linux-x86_64/egg' to it
...
(volttron)[ListenerAgent]$ volttron-ctrl install-executable
dist/listeneragent-0.1-py2.7.egg
(volttron)[ListenerAgent]$ volttron-ctrl load
listeneragent.launch.json listener.agent
(volttron)[ListenerAgent]$ volttron-ctrl list-executables
listeneragent-0.1-py2.7.egg
(volttron)[ListenerAgent]$ volttron-ctrl list-agents
AGENT            AUTOSTART   STATUS
listener.agent   disabled
(volttron)[ListenerAgent]$ volttron-ctrl enable-agent
listener.agent
(volttron)[ListenerAgent]$ volttron-ctrl list-agents
AGENT            AUTOSTART   STATUS
listener.agent    enabled
(volttron)[ListenerAgent]$ volttron-ctrl start listener.agent
(volttron)[ListenerAgent]$ volttron-ctrl list-agents
AGENT            AUTOSTART   STATUS
listener.agent    enabled    running
(volttron)[ListenerAgent]$ volttron-ctrl stop listener.agent
(volttron)[ListenerAgent]$ volttron-ctrl list-agents
AGENT            AUTOSTART   STATUS
listener.agent    enabled         0
```

Full ExampleSetup

Download in other formats:
Plain Text

Example Setup

- Modify the driver.ini file
  - Add your SMAP Key to the url
  - Name your collection source
  - Give your collection a uuid
  - Enter your collection paths and metadata
- If you are deploying the WeatherAgent update settings.py with your Weather Underground key
- By default the Catalyst registers are setup to work with a 372 with the latest changes from TWT, if you have an older unit use the catalystreg.csv.371 file

Start the smap driver

```
. bin/activate
twistd -n smap driver.ini
```

With the platform already running:

- Build agent eggs
- Make egg executable
- Install egg into bin (must not already exist there)
- Load agent config file
- Enable agent for autostart

```
volttron/scripts/build-agent.sh ArchiverAgent
chmod +x Agents/archiveragent-0.1-py2.7.egg
bin/volttron-ctrl install-executable Agents/archiveragent-0.1-
py2.7.egg
bin/volttron-ctrl load-agent Agents/ArchiverAgent/archiver-
deploy.service
bin/volttron-ctrl enable-agent archiver-deploy.service
bin/volttron-ctrl list-agents
AGENT                      AUTOSTART   STATUS
archiver-deploy.service     enabled
```

```
volttron/scripts/build-agent.sh ActuatorAgent
chmod +x Agents/actuatoragent-0.1-py2.7.egg
bin/volttron-ctrl install-executable Agents/actuatoragent-0.1-
py2.7.egg
bin/volttron-ctrl load-agent Agents/ActuatorAgent/actuator-
deploy.service
bin/volttron-ctrl enable-agent actuator-deploy.service
```

Do the same things for WeatherAgent if you plan to deploy it.

Control Application Example Install one executable but multiple launch configuration files. Each instance of this agent will work with a different RTU.

```
volttron/scripts/build-agent.sh SMDSAgent
```

```
chmod +x Agents/SMDSagent-0.1-py2.7.egg
bin/volttron-ctrl install-executable Agents/SMDSagent-0.1-py2.7.egg
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-lbnl1.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-lbnl2.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-lbnl3.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-lbnl4.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-lbnl5.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-lbnl6.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-lbnl7.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-twt1.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-twt2.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-twt3.agent
bin/volttron-ctrl load-agent Agents/SMDSAgent/smds-twt4.agent

bin/volttron-ctrl enable-agent smds-lbnl1.agent
bin/volttron-ctrl enable-agent smds-lbnl2.agent
bin/volttron-ctrl enable-agent smds-lbnl3.agent
bin/volttron-ctrl enable-agent smds-lbnl4.agent
bin/volttron-ctrl enable-agent smds-lbnl5.agent
bin/volttron-ctrl enable-agent smds-lbnl6.agent
bin/volttron-ctrl enable-agent smds-lbnl7.agent
bin/volttron-ctrl enable-agent smds-twt1.agent
bin/volttron-ctrl enable-agent smds-twt2.agent
bin/volttron-ctrl enable-agent smds-twt3.agent
bin/volttron-ctrl enable-agent smds-twt4.agent
```

Restart platform and agents you have enable for autostart should start up.

Platform Service Agents

- ActuatorAgent
- ArchiverAgent
- Scheduling
- LoggerAgent

Download in other formats:
Plain Text

Actuator Agent

This agent is used to access the control points of the controller. Agents may request a lock in order to send commands to the RTU.

This is handled via pub/sub. The available points are detailed here.

See the ExampleControllerAgent for an example of using the ActuatorAgent.

This agent will also be handling the scheduling of agent access to devices. This will set out an hour by hour schedule specifying which agents are eligible for a lock on a device.

This agent also handles locking access the RTUs and scheduling of device access.

The ActuatorAgent also sends a heartbeat message to the Catalyst controller to indicate the platform is still running and should be given control. If the heartbeat signal is not sent the Catalyst takes back over. Note: this requires that the Catalyst controller has been setup to do this. Check with TWT if you're not sure.

```
"heartbeat_interval": 30,
        "points":
        {
                "lbnl/building46/fakecatalyst":
                {
                        "heartbeat_point":"PlatformHeartBeat",
                        "schedule":
                        {
                                "17:00": ["foo1", "bar1"],
                                "17:03": ["foo2", "bar2"],
                                "17:06": ["foo3", "bar3"],
                                "17:09": ["foo4", "bar4"]
                        }
                },
```

Download in other formats:
Plain Text

Archiver Agent

The SMDSAgent illustrates working with the ArchiverAgent.

publish('archiver/request/campus1/building1/realcatalyst1/CoolCall1',{},'(now -1h, now)')

publish('archiver/request/campus1/building1/realcatalyst1/CoolCall1',{},'(1374192541000.0, 1374193541000.0)')

The agent listens for messages on "archiver/request". The message that comes after is extracted and used in the Archiver query and represents the path to the value desired. There is also a source name that needs to be specified. This is currently part of the launch config. The data is returned as a list of lists:

[[1371851254000.0,1.0],[1371851314000.0,1.0],[1371851374000.0,1.0],[1371851434000.0,1.0],
[1371851494000.0,1.0],[1371851554000.0,1.0],[1371851614000.0,1.0],[1371851674000.0,1.0],
[1371851734000.0,1.0],[1371851794000.0,1.0],[1371851854000.0,1.0],[1371851914000.0,1.0],
[1371851974000.0,1.0],[1371852034000.0,1.0],[1371852094000.0,1.0],[1371852154000.0,1.0],
[1371852214000.0,1.0],[1371852274000.0,1.0],[1371852334000.0,1.0],[1371852394000.0,1.0],
[1371852454000.0,1.0],[1371852514000.0,1.0],[1371852574000.0,1.0],[1371852634000.0,1.0],
[1371852694000.0,1.0],[1371852754000.0,1.0],[1371852814000.0,1.0],[1371852874000.0,1.0],
[1371852934000.0,1.0],[1371852994000.0,1.0],[1371853054000.0,1.0],[1371853114000.0,1.0],
[1371853174000.0,1.0],[1371853234000.0,1.0],[1371853294000.0,1.0],[1371853354000.0,1.0],
[1371853414000.0,1.0],[1371853474000.0,1.0],[1371853534000.0,1.0],[1371853594000.0,1.0],
[1371853654000.0,1.0],[1371853714000.0,1.0],[1371853774000.0,1.0],[1371853834000.0,1.0],
[1371853894000.0,1.0],[1371853954000.0,1.0],[1371854005000.0,1.0]]

Currently, the format for the time is (start time, end time) as a string. In the example above, that requests data for the last hour. For each item in the list, the first element is the time stamp of the observation, and the second is the value. The smap archiver page indicates that you can use a variety of ways to specify the time, but I've only used unix time, and the now -1h constructs. This is what the smap page says:

You can select the time region queried using a range query, or a query relative to a reference time stamp. In all these cases, the reference times must either be a timestamp in units of UNIX milliseconds. The reference may be modified by appending a relative time string, using unix “at”-style specifications. You can for instance say now + 1hour or now -1h -5m for the last 1:05. Available relative time quantities are days, hours, minutes, and seconds.

Controller Access

There are configuration files you must edit to use keys generated for your lab. Do not check these files into the repository. You should keep them private.

The example smap driver ini file is "driver.ini" at the base of the project directory. Please edit this file to reflect your setup.

ReportDeliveryLocation?: requires an admin key from sMAP Metadata/SourceName: The name of the source for your catalyst uuid: A unique identifier (can be any generated by any means) ip_address: IP address of the catalyst box.

```
[report 0]
#Insert your SMAP key after add
ReportDeliveryLocation = http://smap.lbl.gov/backend/add/<INSERT
YOUR KEY HERE>

[/]
type = Collection
Metadata/SourceName = <PUT YOUR NAME HERE> Catalyst Data 2
uuid = <PUT YOUR UUID HERE>

[/campus1]
type = Collection
Metadata/Location/Campus = Campus Number 1

[/campus1/building1]
type = Collection
Metadata/Location/Building = Building Number 1

[/campus1/building1/catalyst1]
type = volttron.drivers.catalyst.Catalyst
ip_address = <PUT YOUR CATALYST IP HERE>
#see volttron/drivers/catalystreg.csv
#for an example of a catalyst register config file
#register_config = <PUT YOUR REGISTER CONFIG HERE>

[/campus1/building1/logger]
#Currently this will only write to the file specified.
#Hopefully logging to historian is forthcoming.
type = volttron.drivers.smap_logging.Logger
file = 'test.log'
```

Once the ini file is configured:

- In a terminal from the rtunetwork directory
- Activate the project python
  - . bin/activate (note the space)
- The sMAP driver relies on Twistd. Start this now:
  - twistd -n smap <your config file's name>

Data should now be getting scraped from the Catalyst controller, published to the platform, and stored in the sMap historian. If you launch the ListenerAgent, you should see RTU data being published once a minute.

In order to set points, you must start the ActuatorAgent. Launch this as you would launch any other agent. In Eclipse, setup a run configuration with the following Program Arguments: {{--config Agents/ActuatorAgent/actuatoragent.launch.json --pub ipc:///tmp/volttron-lite-agent-publish --sub ipc:///tmp/volttron-lite-agent-subscribe}}.

Please see the ExampleControllerAgent for making use of the ActuatorAgent.

Data Logging

A mechanism for storing data in SMAP has been provided. The data logger is written as an SMAP driver, but receives information in a zero MQ message sent to a topic prefixed with **datalogger/log**. The source name is configured at the time the logger starts up and is defined in the driver.ini file. The rest of the topic is extracted and used as the path in SMAP for the data point. If the path already exists, the time series items will be added to the bottom of the list. For example, if our source name is 'test data', and the topic we publish data to is datalogger/log/campus1/building1/testdata, then our data will be posted to time series under campus1/building1/testdata in the source name 'test data'.

The Logger should be added to the sMAP configuration file alongside the Catalyst configuration. The location listed in the configuration file will not be used unless no path is given after datalogger/log.

```
[/datalogger]
type = volttron.drivers.data_logger.DataLogger
interval = 1
```

Data Logging Format

Data sent to the data logger should be sent as a JSON object that consists of a dictionary of dictionaries. The keys of the outer dictionary are used as the points in SMAP to store the data items. The inner dictionary consists of 2 required fields and 1 optional. The required fields are "Readings" and "Units". Readings contains the data that will be written to SMAP. It may contain either a single value, or a list of lists which consists of timestamp/value pairs. Units is a string that identifies the meaning of the scale values of the data. The optional entry is data_type, which indicates the type of the data to be stored. This may be either long or double.

```
{
    "test3": {
        "Readings": [[1377788595, 1.1],[1377788605,2.0]],
        "Units": "KwH",
        "data_type": "double"
        },
    "test4": {
        "Readings": [[1377788595, 1.1],[1377788605,2.0]],
        "Units": "TU",
        "data_type": "double"
    }
}
```

Data Logger Response

The data logger reports the status of the storage request to topic called datalogger/status. The response will either be success, or error, and the message will contain information on what failed if there was an error. In addition, the data logger looks in the message headers for a field called "from" to determine who should be the recipient of this message. The status message contains a header called "to" that uses the value retrieved from "from" so that agents may filter messages sent to datalogger/status using the match_headers decorator in a base agent derived agent.

Example Code

```
        headers[headers_mod.FROM] = self._agent_id
        headers[headers_mod.CONTENT_TYPE] =
headers_mod.CONTENT_TYPE.JSON

        mytime = int(time.time())

        content = {
            "listener": {
                "Readings": [[mytime, 1.0]],
                "Units": "TU",
                "data_type": "double"
            },
            "hearbeat": {
                "Readings": [[mytime, 1.0]],
                "Units": "TU",
                "data_type": "double"
            }
        }



        self.publish('datalogger/log/', headers,
json.dumps(content))
```

Download in other formats:

Plain Text