VOLTTRON™ Drivers and Historians

CHANDRIKA SIVARAMAKRISHNAN

Pacific Northwest National Laboratory

VOLTTRON^{TM} 2016

Introduction

Two essential VOLTTRON™ services

- Data collection - Driver framework
- Data storage - Historian framework
- Both frameworks are easily extensible

Topics Covered

- Driver framework
  - Configuration for existing driver types
  - Interacting with the device (read & write)
  - Demonstration of a BACnet device setup using VOLTTRON™ utility scripts
  - Development of new drivers

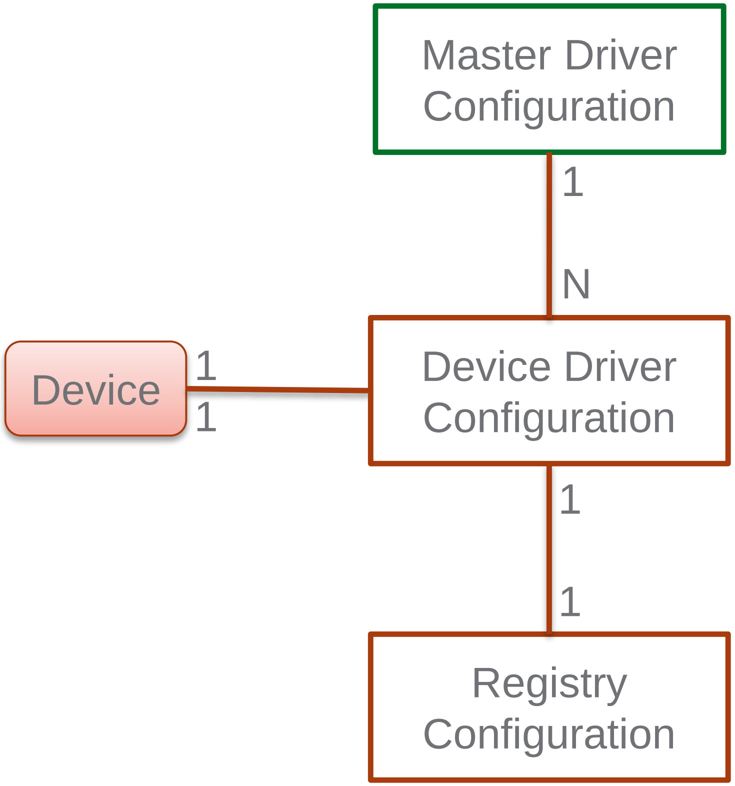
- Historian framework
  - Existing historians
  - Development of new historians

Driver Framework

- Implemented as sub agents of Master Driver Agent
- One driver subagent interfaces with one device
- Currently we have two driver interfaces
  - Modbus
  - BACnet
- Also support a fake driver for development/testing purpose
- On demand read and write are done using Actuator agent

Driver Configuration

- Driver configuration file
  - Driver type, device address, and reference to the registry configuration file
- Register configuration file
  - Settings for each individual data point on the device
  - Specific to driver type
  - Example – point name, point address, units, writeable, index, object type
- Master Driver Agent configuration file
  - Has reference to the list of driver configuration files



Generating BACnet Configuration Files

- VOLTTRON™ provides two scripts to help configure BACnet devices
  - bacnet_scan.py - scan the network for devices
  - grab_bacnet_config.py - creates a CSV register configuration file to use as a starting point
- Uses bacpypes library
- Need a BACpypes.ini configuration file

```
[BACpypes]
objectName: Betelgeuse
address: 10.0.2.15/24       # Address of machine running this script
objectIdentifier: 599
maxApduLengthAccepted: 1024
segmentationSupported: segmentedBoth
vendorIdentifier: 15
```

- Only point with a ‘presentValue’ value property are currently supported

http://bacpypes.sourceforge.net/

Device State Publishes

- Value of each point on a device is published to specific topic on message bus
- Topic name is derived based on campus, building, unit, and path configured in driver configuration file
- Publish one point at a time or all points together
  - [75.2, {"units": "F"}] – to topic - devices/pnnl/isb1/vav1/temperature
  - [{"temperature": 75.2, ...}, {"temperature":{"units": "F"}, ...}] - to topic - devices/pnnl/isb1/vav1/all
- breadth first publish vs depth first
  - devices/temperature/vav1/isb1/pnnl **VS** devices/pnnl/isb1/vav1/temperature
  - devices/all/vav1/isb1/pnnl **VS** devices/pnnl/isb1/vav1/all
- Turn off any of them in your driver configuration

Actuator Agent

Actuator agent

- provides read and write access to device
- agents should schedule a time slot prior to any write operations

Actuator Functions - Read

Get point

- RPC Call:

```
agent.vip.rpc.call(
    'platform.actuator',
    'get_point',
    <device path/point. For example, campus/building/unit/point name>
    ).get(timeout=5)
```

- Alternate method :
  - Publish to - devices/actuators/get/<device path>/<actuation point>
  - Success response @ devices/actuators/value/<device path>/<actuation point>
  - Error response @ devices/actuators/error/<device path>/<actuation point>

Actuator Functions – Write – Step 1

Scheduling a task

- RPC call

```
 publish_agent.vip.rpc.call(
        'platform.actuator',
        ‘request_new_schedule’,
        agent_id,                   # name of requesting agent
        task_id,                    # unique ID for scheduled task.
        priority,                   #('HIGH, 'LOW', 'LOW_PREEMPT').
        message).get(timeout=5)
```

- Input Message:

```
        [
          ["campus/building/device1", "2013-12-06 16:00:00", "2013-12-06 16:20:00”]
       ]
```

- Alternate method:
  - Publish to “devices/actuators/schedule/request”
  - Response @ devices/actuators/schedule/result

Actuator Functions – Write – Step 2

<u>Set point</u>

- RPC call

```
 publish_agent.vip.rpc.call(
        'platform.actuator',          # Target agent
        'set_point',                  # Method
        agent_id,                     # Requestor
        '<device_path>/<point>',      # Point to set
        2.5                           # New value
    ).get(timeout=5)
```

- Alternate:
  - Publish to devices/actuators/set/<device path>/<actuation point>
  - Success response @ devices/actuators/value/<device path>/<actuation point>
  - Error response @ devices/actuators/error/<device path>/<actuation point>

Actuator Functions – Write – Step 3

Cancel a task

- RPC Call:

```
publish_agent.vip.rpc.call(
        'platform.actuator',
        'request_cancel_schedule',
        agent_id,
        taskid).get(timeout=10)
```

- Alternate:
  - Publish to “devices/actuators/schedule/request”
  - Response @ devices/actuators/schedule/result

Actuator Functions – Revert

- Revert implementation is driver specific.
  - Bacnet protocol has built in support for reverting to default value.
  - Modbus protocol does not support this hence volttron Modbus driver implements its own.
- RPC call: revert_point or revert_all

```
publish_agent.vip.rpc.call('platform.actuator',
                           revert_point,
                           agent_id,
                           '<device_path>/<point>' ).get(timeout=5)
```

- Alternate:
  - Publish to actuators/revert/point/<device path>/<actuation point>
  - Success response @ devices/actuators/reverted/point/<device path>/<actuation point>
  - Error response @ devices/actuators/error/<device path>/<actuation point>

Actuator – Notifications

- Task preemption notice - devices/actuators/schedule/response
- Schedule state broadcast - state of all currently used devices to topic devices/actuators/schedule/announce/<full device path>
- Send out heartbeat signal to devices that have a configured heartbeat point

Example:

https://github.com/VOLTTRON/volttron/blob/develop/examples/SchedulerExample/schedule_example/agent.py

Driver Development: Interface Benefits

- Historians will automatically capture data published by the new device driver.
- Device data can be graphed in VOLTTRON™ Central in real time.
- If the device can receive a heartbeat signal the driver framework can be configured to automatically send a heartbeat signal.
- Existing Agents can interact with the device via the Actuator Agent without any code changes.
- Configuration follows the standard form of other devices. Existing and future tools for configuring devices will work with the new device driver.

Driver Development: Interface Development

- Each driver module must create a subclass of **BaseInterface** called Interface.
- Each point on the device must be represented by an instance of **BaseRegister**
- Interface class should be in <driver_type>.py module and should be at services/core/MasterDriverAgent/master_driver/interfaces
- The `Interface` class must implement the following methods:
  - configure
  - scrape_all
  - set_point
  - get_point
  - revert_point
  - revert_all
- revert_point and revert_all can be implemented using BasicRevert mixin.

Driver Development (contd.)

```
{
   "driver_config": {
         "device_address": "130.20.116.13",
         "device_id": 500
                         },
   "campus": "campus",
   "building": "building",
   "unit": "bacnet1",
   "driver_type": "bacnet",
   "registry_config":”/<path>/bacnet.csv",
   "interval": 60,
   "timezone": "UTC"
}
```

On Load

- Master Driver Agent loads Interface class from <**driver_type**>.py
- Calls Interface.configure – passes **driver_config**, contents of registry config file
- Interface should create register object for each point

Runtime operations

- Handled by Master Driver Agent

Historians

- Store and retrieve historical device and analysis data published to the message bus
- Listens to
  - devices/
  - analysis/
  - record/
  - datalogger/

Available Historians

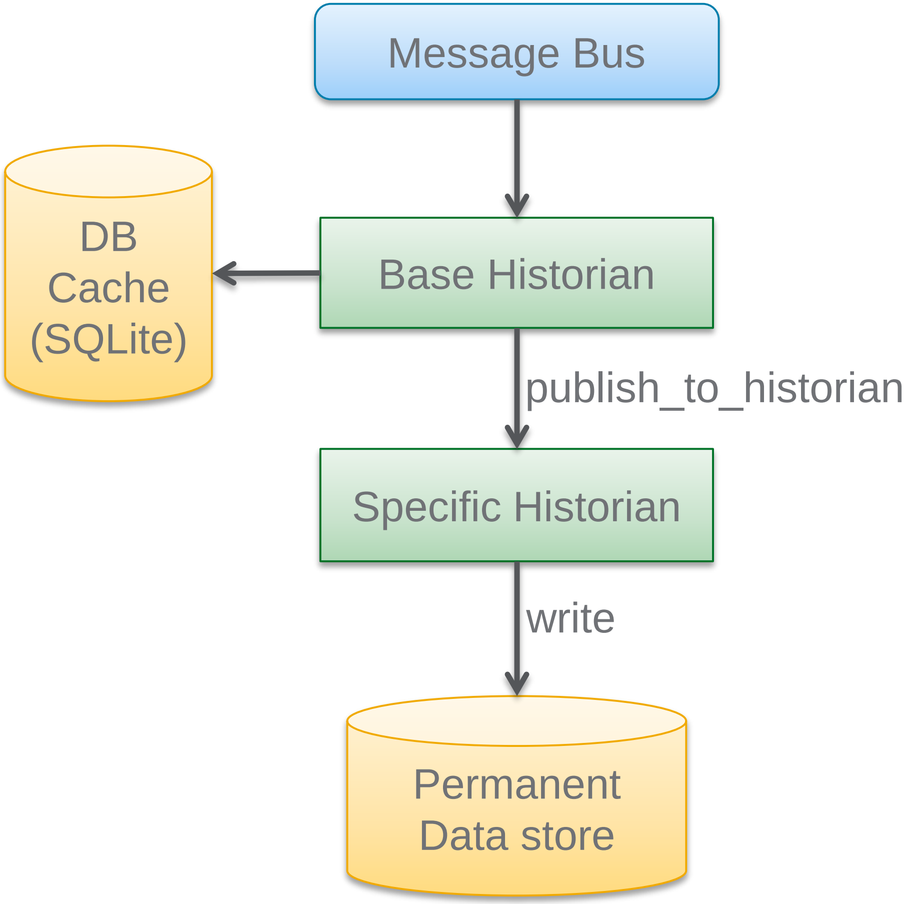
- SQLHistorian – SQLite and MySQL
- MongodbHistorian
- Forward Historian

Platform Historians

- Multiple historians can collect data within a single VOLTTRON™ instance
- The primary historian has the identity ‘platform.historian’
- ‘platform.historian’ a known identity for other agents to easily query historian
- VOLTTRON™ Central queries only the primary historian

Historian Implementation

- Sub class of BaseHistorian
- base_historian.py
  - handles getting device and agent data from the message bus
  - Writes data to local cache until successful write
- Specific implementations should extend this class and implement
  - historian_setup
  - publish_to_historian: store data in db, external service, file, etc.
  - query_historian
  - query_topic_list


Message Bus

Base Historian

DB Cache (SQLite)

publish_to_historian

Specific Historian

write

Permanent Data store

References

Documentation:

- http://volttron.readthedocs.io/en/develop/core_services/drivers/index.html
- http://volttron.readthedocs.io/en/develop/core_services/historians/index.html
- http://volttron.readthedocs.io/en/develop/apidocs/volttron/volttron.platform.agent.html#volttron-platform-agent-base-historian-module

Source code:

- https://github.com/VOLTTRON/volttron/tree/develop/services/core/MasterDriverAgent/master_driver/interfaces
- https://github.com/VOLTTRON/volttron/blob/develop/examples/SchedulerExample/schedule_example/agent.py
- https://github.com/VOLTTRON/volttron/blob/develop/services/core/ActuatorAgent/tests/test_actuator_rpc.py