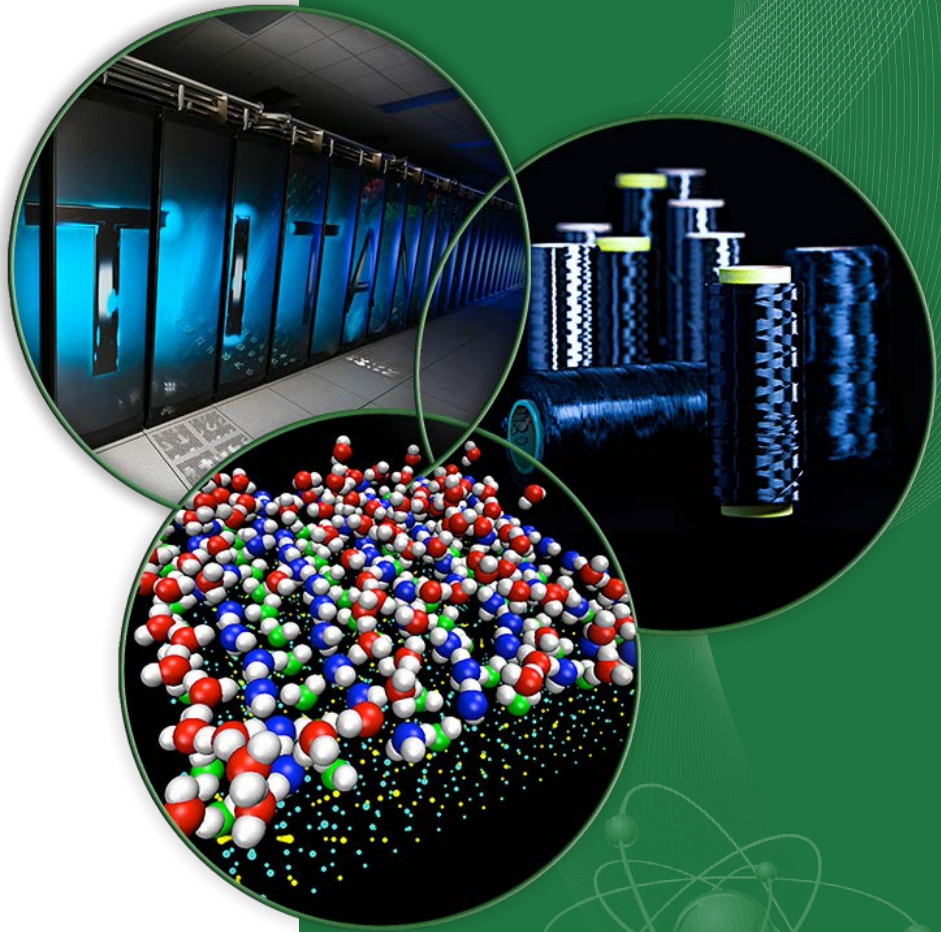
VOLTTRON -- Peak Demand Reduction

Chris Winstead
Oak Ridge National Lab



ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE
National Laboratory

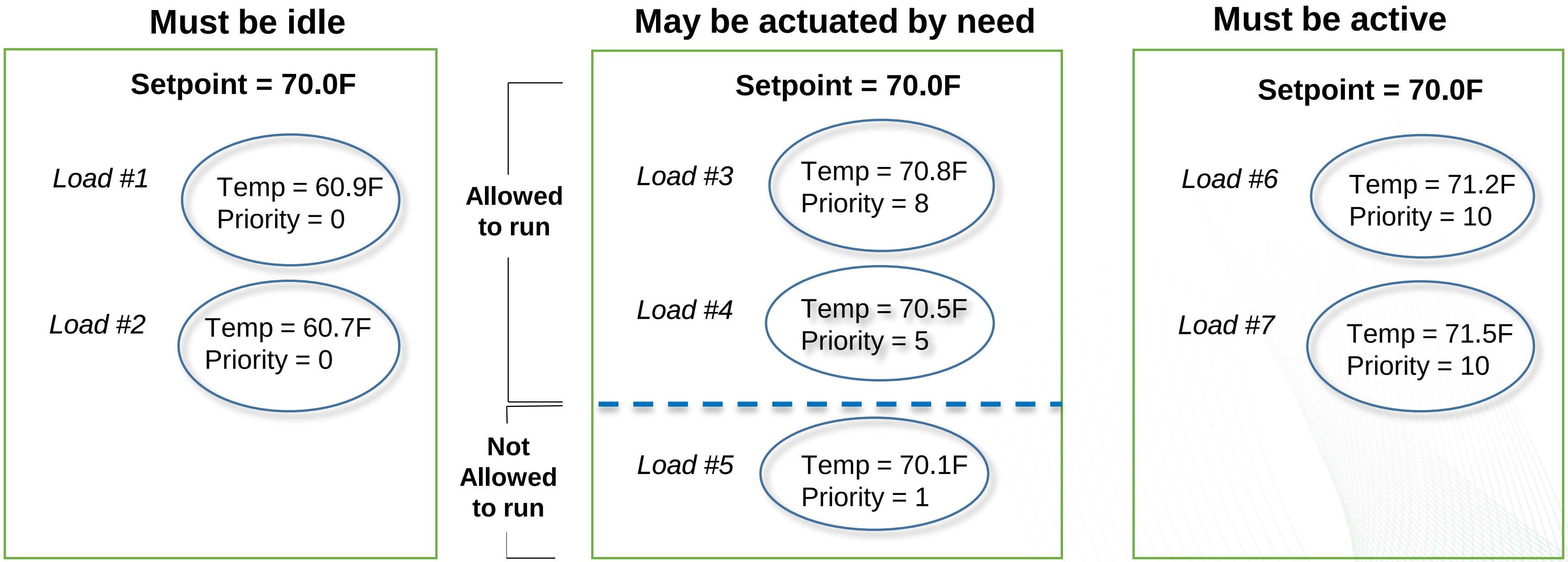
Topics of Discussion

Project Requirements

- Sensor Interface
- Connectivity Across Platforms
- Platform Robustness
- User-Facing Interface
- Application Validation

Priority Based Control – Load Flattening

- The priority based control algorithm seeks to **flatten** electrical loads by quantifying the “need” to operate of particular electrical loads, and then allowing them to compete for permission based on distance from setpoint
- After priority calculations are made, three reservoirs of loads are created
  - Loads that **must** be activated (those at or in excess of maximal priority)
  - Loads that **must** be deactivated (those at zero priority)
  - Loads that may “compete” for activation permission (everything in between)
- **Ex:** HVAC system subject to priority constraints between 1 (min) to 10 (max)
  - 1 priority point per 0.1F from setpoint



Must be idle

Setpoint = 70.0F

Load #1 Temp = 60.9F Priority = 0

Load #2 Temp = 60.7F Priority = 0

Allowed to run

Not Allowed to run

May be actuated by need

Setpoint = 70.0F

Load #3 Temp = 70.8F Priority = 8

Load #4 Temp = 70.5F Priority = 5

Load #5 Temp = 70.1F Priority = 1

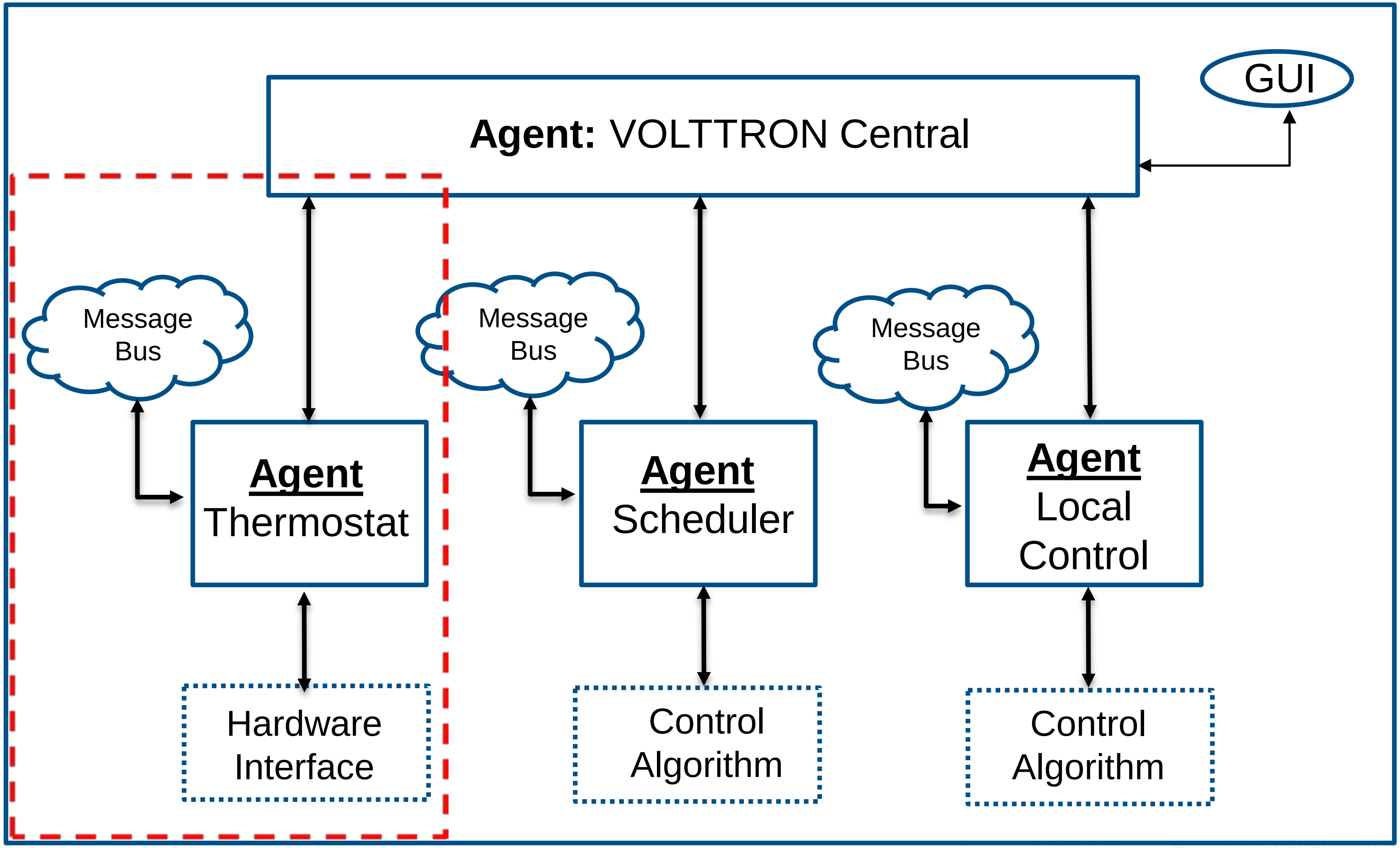
Must be active

Setpoint = 70.0F

Load #6 Temp = 71.2F Priority = 10

Load #7 Temp = 71.5F Priority = 10

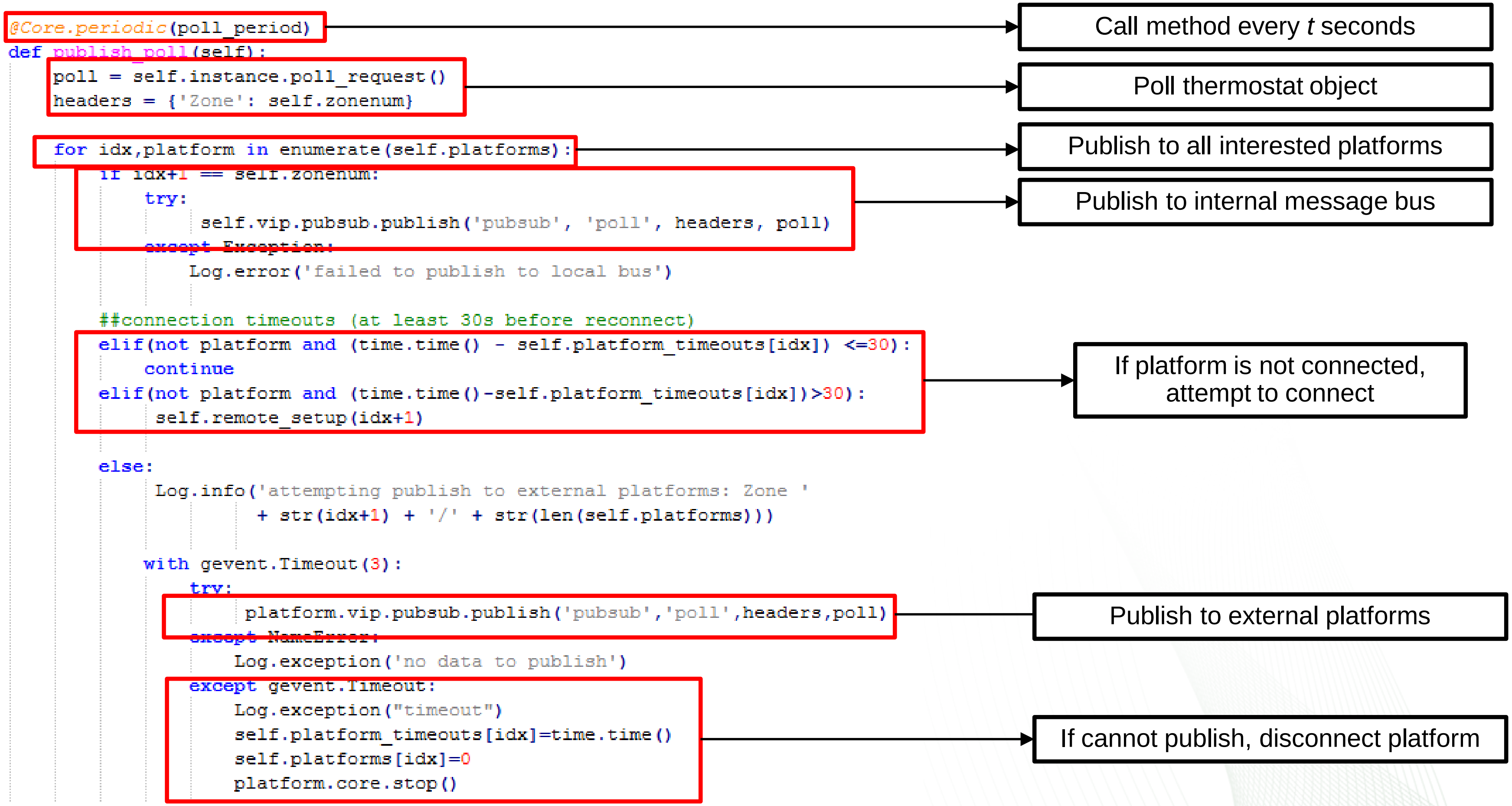
System Infrastructure



Agent
Thermostat

Thermostat agent is responsible for **polling** the sensors and **actuating** relays

Polling the sensors



```
@Core.periodic(poll_period)
def publish_poll(self):
    poll = self.instance.poll_request()
    headers = {'Zone': self.zonenum}

    for idx,platform in enumerate(self.platforms):
        if idx+1 == self.zonenum:
            try:
                 self.vip.pubsub.publish('pubsub', 'poll', headers, poll)
            except Exception:
                Log.error('failed to publish to local bus')

        ##connection timeouts (at least 30s before reconnect)
        elif(not platform and (time.time() - self.platform_timeouts[idx]) <=30):
            continue
        elif(not platform and (time.time()-self.platform_timeouts[idx])>30):
             self.remote_setup(idx+1)

        else:
             Log.info('attempting publish to external platforms: Zone '
                      + str(idx+1) + '/' + str(len(self.platforms)))

            with gevent.Timeout(3):
                try:
                     platform.vip.pubsub.publish('pubsub','poll',headers,poll)
                except NameError:
                    Log.exception('no data to publish')
                except gevent.Timeout:
                    Log.exception("timeout")
                    self.platform_timeouts[idx]=time.time()
                    self.platforms[idx]=0
                    platform.core.stop()
```

Call method every *t* seconds

Poll thermostat object

Publish to all interested platforms

Publish to internal message bus

If platform is not connected, attempt to connect

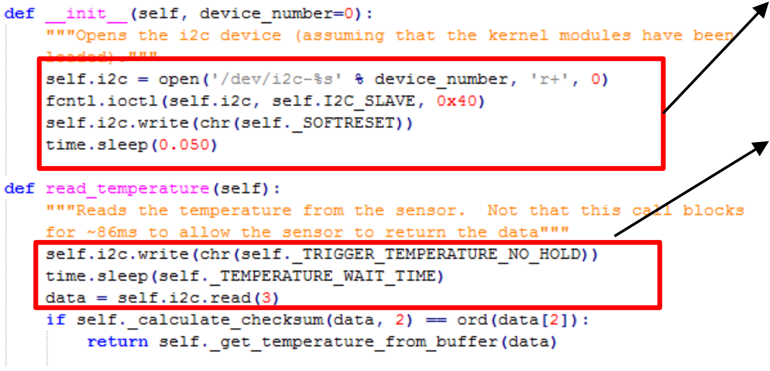
Publish to external platforms

If cannot publish, disconnect platform

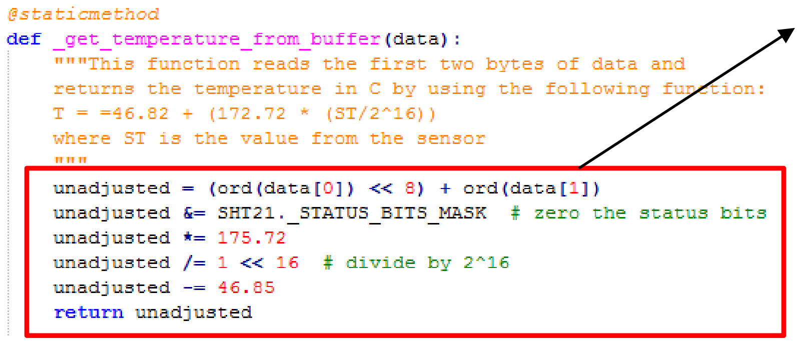
Sensor Interface

Temperature

- Python temperature GPIO interface



```python
def __init__(self, device_number=0):
    """Opens the i2c device (assuming that the kernel modules have been
    loaded)."""
    self.i2c = open('/dev/i2c-%s' % device_number, 'r+', 0)
    fcntl.ioctl(self.i2c, self.I2C_SLAVE, 0x40)
    self.i2c.write(chr(self._SOFTRESET))
    time.sleep(0.050)

def read_temperature(self):
    """Reads the temperature from the sensor.  Not that this call blocks
    for ~86ms to allow the sensor to return the data"""
    self.i2c.write(chr(self._TRIGGER_TEMPERATURE_NO_HOLD))
    time.sleep(self._TEMPERATURE_WAIT_TIME)
    data = self.i2c.read(3)
    if self._calculate_checksum(data, 2) == ord(data[2]):
        return self._get_temperature_from_buffer(data)
```



```python
@staticmethod
def _get_temperature_from_buffer(data):
    """This function reads the first two bytes of data and
    returns the temperature in C by using the following function:
    T = =46.82 + (172.72 * (ST/2^16))
    where ST is the value from the sensor
    """
    unadjusted = (ord(data[0]) << 8) + ord(data[1])
    unadjusted &= SHT21._STATUS_BITS_MASK  # zero the status bits
    unadjusted *= 175.72
    unadjusted /= 1 << 16  # divide by 2^16
    unadjusted -= 46.85
    return unadjusted
```

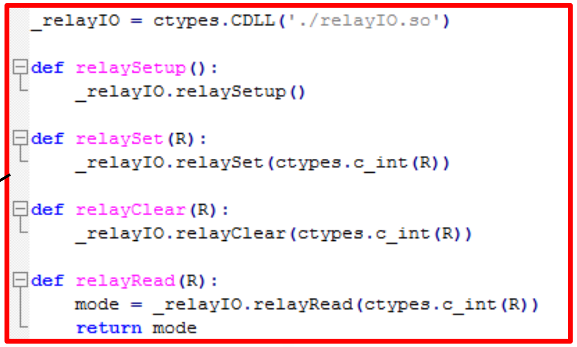
I2C protocol to talk to GPIO

Request temperature from sensor and wait for response

Resolve temp from bytes returned from sensor

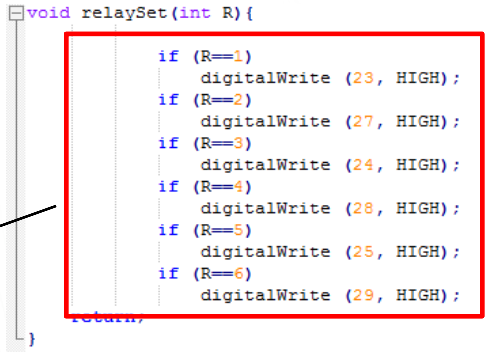
Relays

- Python wrapper for WiringPi C Library



```python
_relayIO = ctypes.CDLL('./relayIO.so')

def relaySetup():
    _relayIO.relaySetup()

def relaySet(R):
    _relayIO.relaySet(ctypes.c_int(R))

def relayClear(R):
    _relayIO.relayClear(ctypes.c_int(R))

def relayRead(R):
    mode = _relayIO.relayRead(ctypes.c_int(R))
    return mode
```

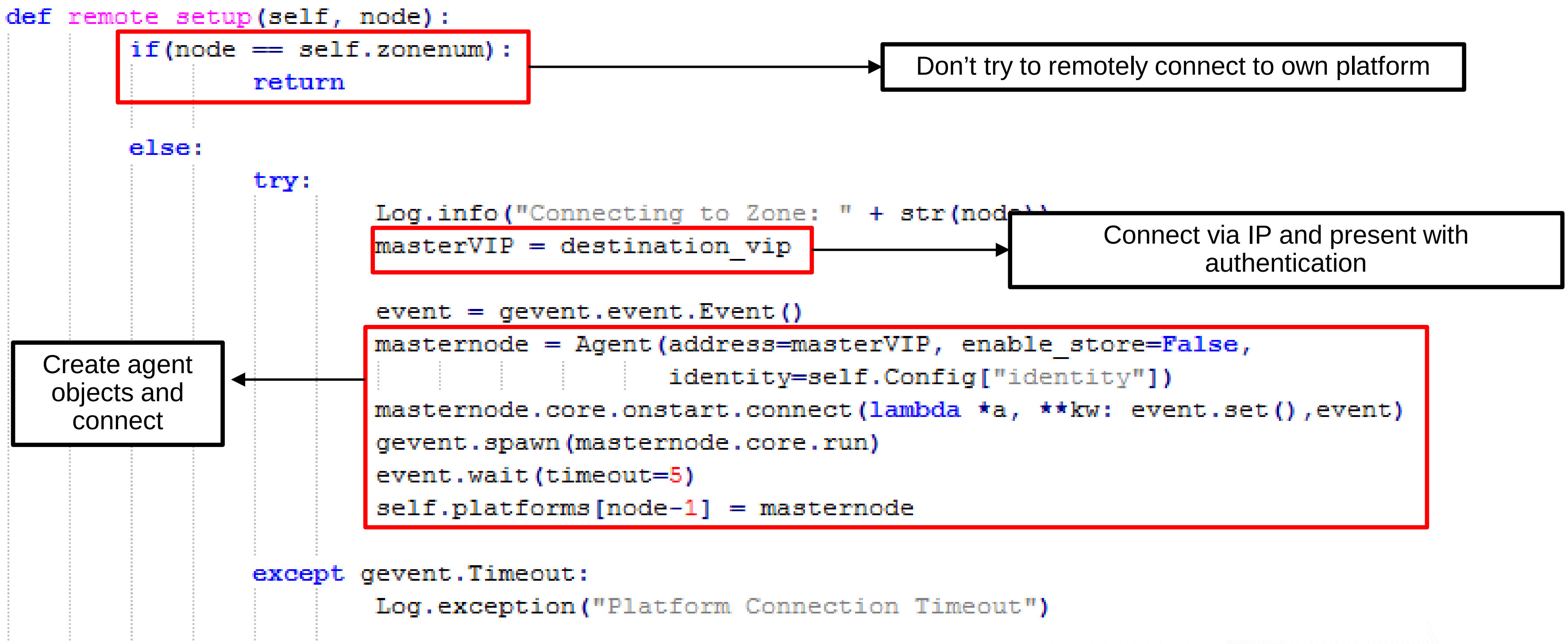
Wrap WiringPi lib in python using ctypes



```c
void relaySet(int R){

            if (R==1)
                digitalWrite (23, HIGH);
            if (R==2)
                digitalWrite (27, HIGH);
            if (R==3)
                digitalWrite (24, HIGH);
            if (R==4)
                digitalWrite (28, HIGH);
            if (R==5)
                digitalWrite (25, HIGH);
            if (R==6)
                digitalWrite (29, HIGH);
    return;
}
```

Example of setting relays using WiringPi lib

Connectivity



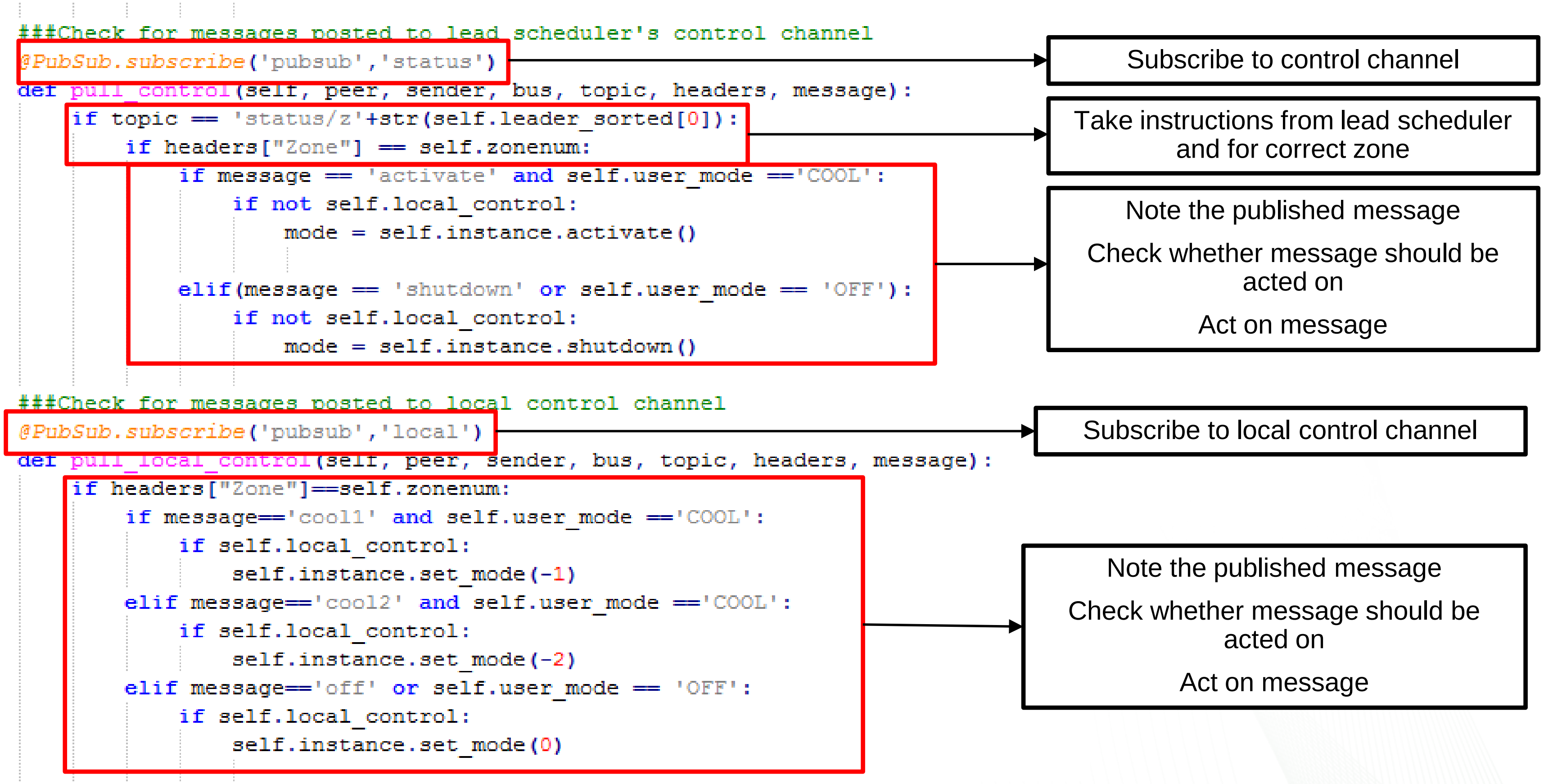
```
def remote_setup(self, node):
        if(node == self.zonenum):
                return

        else:
                try:
                        Log.info("Connecting to Zone: " + str(node))
                        masterVIP = destination_vip

                        event = gevent.event.Event()
                        masternode = Agent(address=masterVIP, enable_store=False,
                                           identity=self.Config["identity"])
                        masternode.core.onstart.connect(lambda *a, **kw: event.set(),event)
                        gevent.spawn(masternode.core.run)
                        event.wait(timeout=5)
                        self.platforms[node-1] = masternode

                except gevent.Timeout:
                        Log.exception("Platform Connection Timeout")
```

Don’t try to remotely connect to own platform

Connect via IP and present with authentication

Create agent objects and connect

Thermostat Agent Subscribing to Control



```
###Check for messages posted to lead scheduler's control channel
@PubSub.subscribe('pubsub','status')
def pull_control(self, peer, sender, bus, topic, headers, message):
    if topic == 'status/z'+str(self.leader_sorted[0]):
        if headers["Zone"] == self.zonenum:
            if message == 'activate' and self.user_mode =='COOL':
                if not self.local_control:
                    mode = self.instance.activate()

            elif(message == 'shutdown' or self.user_mode == 'OFF'):
                if not self.local_control:
                    mode = self.instance.shutdown()

###Check for messages posted to local control channel
@PubSub.subscribe('pubsub','local')
def pull_local_control(self, peer, sender, bus, topic, headers, message):
    if headers["Zone"]==self.zonenum:
        if message=='cool1' and self.user_mode =='COOL':
            if self.local_control:
                self.instance.set_mode(-1)
        elif message=='cool2' and self.user_mode =='COOL':
            if self.local_control:
                self.instance.set_mode(-2)
        elif message=='off' or self.user_mode == 'OFF':
            if self.local_control:
                self.instance.set_mode(0)
```

Subscribe to control channel

Take instructions from lead scheduler and for correct zone

Note the published message

Check whether message should be acted on

Act on message

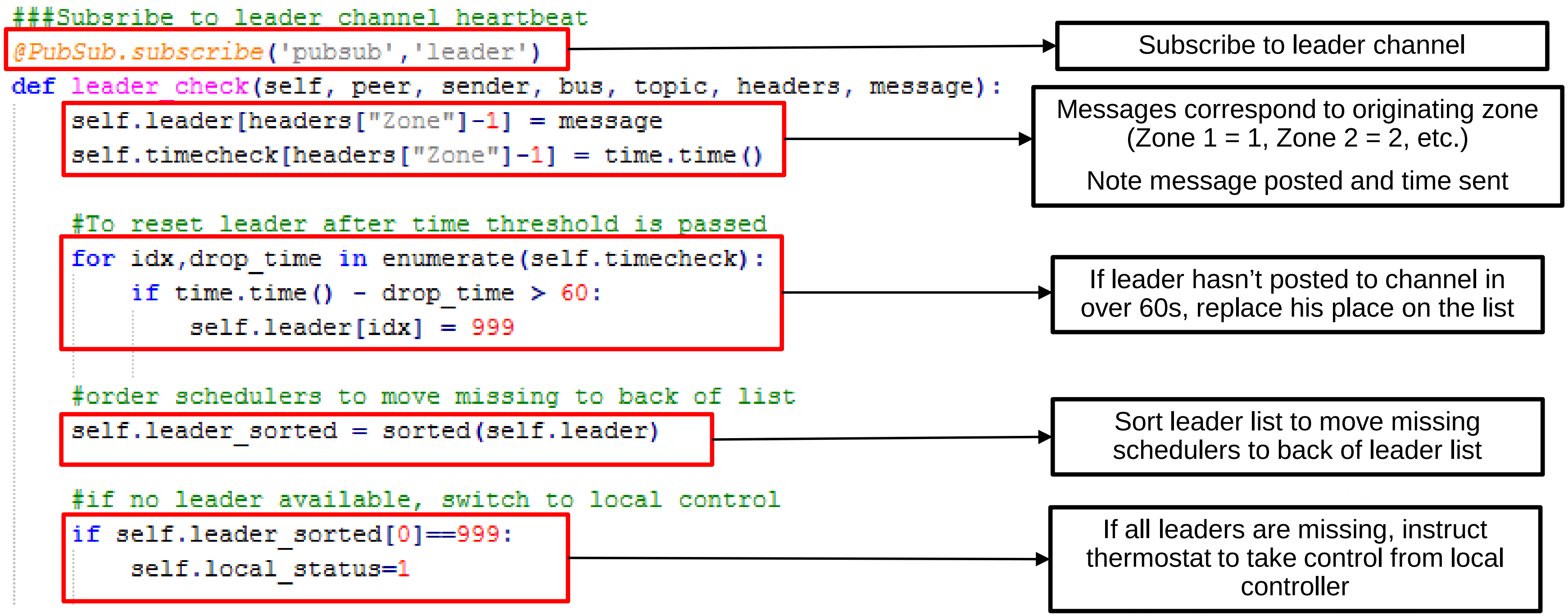
Subscribe to local control channel

Note the published message

Check whether message should be acted on

Act on message

Thermostat Agent
Checking the Leader



```
###Subsribe to leader channel heartbeat
@PubSub.subscribe('pubsub','leader')
def leader_check(self, peer, sender, bus, topic, headers, message):
    self.leader[headers["Zone"]-1] = message
    self.timecheck[headers["Zone"]-1] = time.time()

    #To reset leader after time threshold is passed
    for idx,drop_time in enumerate(self.timecheck):
        if time.time() - drop_time > 60:
            self.leader[idx] = 999

    #order schedulers to move missing to back of list
    self.leader_sorted = sorted(self.leader)

    #if no leader available, switch to local control
    if self.leader_sorted[0]==999:
        self.local_status=1
```

Subscribe to leader channel

Messages correspond to originating zone (Zone 1 = 1, Zone 2 = 2, etc.)

Note message posted and time sent

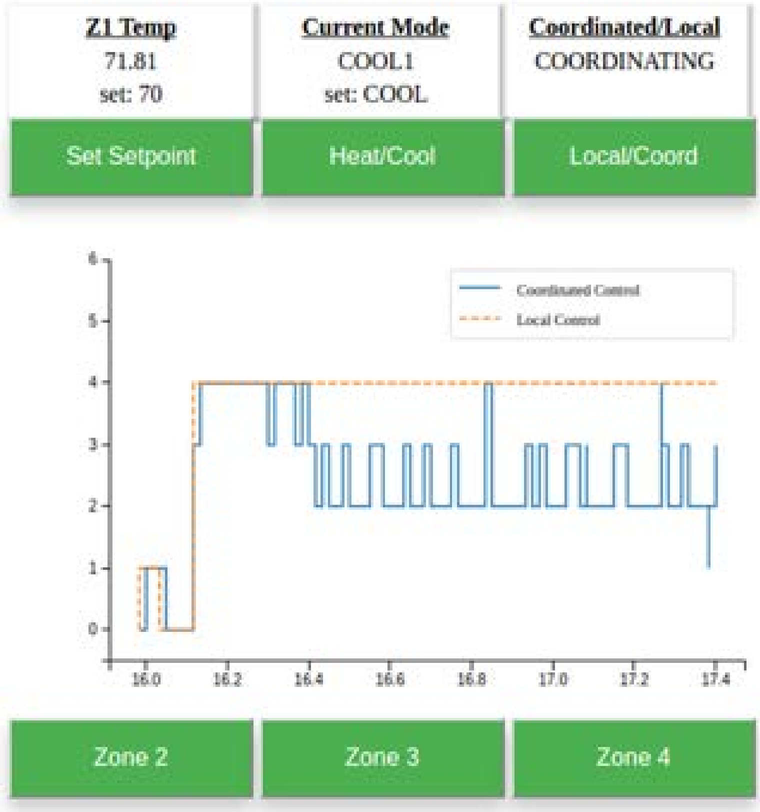
If leader hasn't posted to channel in over 60s, replace his place on the list

Sort leader list to move missing schedulers to back of leader list

If all leaders are missing, instruct thermostat to take control from local controller

Interfacing with the User

- Each thermostat hosts a server for access
- CherryPy backend makes calls to **RPC exposed methods** via **VOLTTRON Central**
- Calls to VOLTTRON Central find exposed methods by parsing platform/agent tags
- Calls made to xxx.xxx.xxx.xxx*/jsonrpc*
- VOLTTRON Central hosted on Intwine



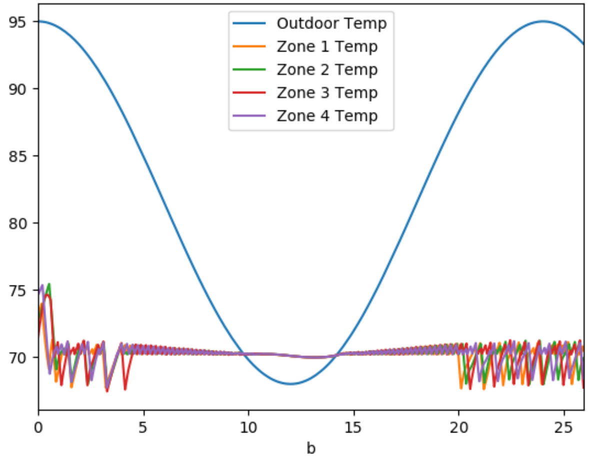
Z1 Temp	Current Mode	Coordinated/Local
71.81 set: 70	COOL1 set: COOL	COORDINATING

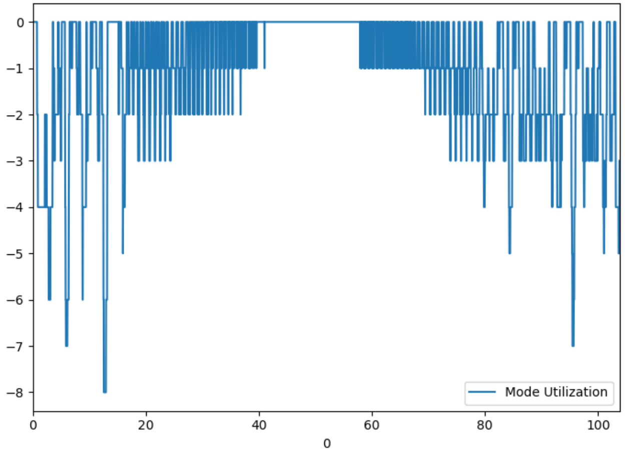
Validating the Model

- Two approaches used:
  - Package agents onto virtual machines and test in discrete event simulator
  - Created Model Agent to be hosted on one of the thermostats

Model Agent

- All thermostat agents made calls to RPC exposed methods within the model agent that conveyed temperature
- Ability to make calls to RPC methods was dependent on successful use of temperature sensor





Discussion

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE
FOR U.S. DEPARTMENT OF ENERGY