ME Journal

Sean Keene (Template by Tylor Slay)

last updated: December 31, 2021

Contents

List of Errors

List of Warnings

Fall 2020

hyperref

August 20, 2021 – Reorganization and Journal Update

OBJECTIVE:

I need to establish mise en place for development of the ME. **OUTLINE:**

- Reorganize my directory, put old files in an Outdated folder.
- Restart this journal.
- Update github README.md.
- Create ModelController.py

PROCEDURE:

Should all be self explanatory.

PARAMETERS:

The point is to get ready to work without fumbling over old files or struggling to get my journal in order. The directory reorganization is pretty straight forward. Restarting this journal starts with this entry, but also organizing my journal update plan (which should be: update the journal every time you push, not commit.) I've already created ModelController.py and added the class outline to it in class form; most of the daily work from here on out will be adding classes and methods one by one, along with testing.

The README needs refactored, and some documentation should be made public. I need to decide what gets links and what, if anything, gets added to the repository directly.

OBSERVATIONS:

I did it. Needs more documentation, but I'll figure out the most useable way of handling that in the future.

DATA:

N/A

RESULTS:

I'm done enough to be ready to move on with programming next week.

Aug 21, 2021 – EdmCore and Callback Classes: step 1

OBJECTIVE:

Get started on EDMCore. See if I can get a simulation running.

OUTLINE:

- Retrieve useful code from the Log Test.
- Start filling in the methods.
- Add startup calls to the program's run routine.

PROCEDURE:

Code them.

PARAMETERS:

N/A

OBSERVATIONS:

Got it done. Everything worked smoothly. The MC is now able to initiate and run simulations from GridAPPS-D. It has the old issue where nothing stops, I'll have to hard code that in. Bigger issue: I started working with callback classes. The commands don't work. The sim runs, but nothing happens. The callbacks aren't called when expected. Need to talk to PNNL about this.

DATA:

All recorded data, or where it can be found.

RESULTS:

EDMCore is a bit more done than I was expecting, since I was able to do things like add config.txt. Callbacks are the next thing to tackle. I knew they'd be a problem, so this is fine. Consult with PNNL. Will report back in next entry.

Aug 23, 2021 – Callback Class Troubleshooting/Repair

OBJECTIVE:

TS&R Callback classes

OUTLINE:

- Contact PNNL
- Fix EdmMeasurementProcessor
- Fix EdmTimeKeeper
- Fix EdmOnClose

PROCEDURE:

Currently using the old callback functions to call the `on_message` method of each callback class. Frankly, this works perfectly well and is invisible to the user and developers in most cases. It does require function definitions and global variables and weird stuff that I'd prefer to remove. So, after contacting PNNL, figure out the issue and fix the classes.

PARAMETERS:

N/A

OBSERVATIONS:

The issue was in the way I was gathering the simulation ID. I reused code that I'd never actually USED in the old script, and this code turns out not to have been working this whole time. Turns out there's an easier way to do it anyways; I just have to start the simulation first, and then get the ID, then use that to instantiate everything. It works fine.

The EdmMeasurementProcessor works fine. The EdmTimekeeper also works fine, but differently from the old callback function: it uses a subscription to the log topic and therefore I have to filter out the messages that include timestep incrementation manually. This is done, and works more or less fine. The EdmOnClose isn't yet working since there's no obvious subscription topic for it. I may have to settle for the callback function on this one, but I'm contacting PNNL to be sure.

May be a timekeeping bug. The timestep increments before the measurement processor; but, the measurement processor uses its own timestamp anyways. Timecodes seem to be 2 off. Will need to do some rigorous testing at the end of Phase 1 to fully verify what's going on and make sure that inputs are being reflected in the outputs at the proper time (which should be the timestep after they're injected.)

DATA:

N/A

RESULTS:

EdmMeasurementProcessor and EdmTimeKeeper working as intended. EdmTimeKeeper filtering function needs cleaned up. EdmOnClose not yet implemented. Awaiting PNNL support.

Sep 2, 2021 – Phase 1 completion

OBJECTIVE:

Determine a data scheme for internal data (I.E. grid states and association data passed to the GO and logger.) Implement MCOutputLog Run a simulation and get output logs

OUTLINE:

- Select data scheme (DICT OF DICTS)
- Add processing to EDMMeasurementProcessor
- Add accessor function to EDMMeasurementProcessor
- Write MCOutputLog
- Add accessor-logger routine to timekeeper
- Add log writing routine to timekeeper
- Add file closeout function to timekeeper closeout
- Test
- Add DER-EMs to model
- Add McInputInterface with a test function
- Add DERSHistoricalDataInput with a test function
- Test (Phase 1 closeout)

PROCEDURE:

NOTE: I accidentially overwrote a bunch of information about previous steps, but it sums up to "it works". Info starts at "Add DER-EMs to model" for this sprint.

PARAMETERS:

N/A

OBSERVATIONS:

Had a bunch of issues adding DER-EMs. A lot of them came from the fact that I was jumping back and forth between CIMHub and the Powergrid-Models repository. USE POWERGRID-MODELS FOR EVERYTHING. It grabs stuff from CIMHub, but the measurement stuff is in powergrid-models as well. A batch script could automate it, if written.

Here is how the process works:

1. Create a text file containing the feeder ID and information about each DER. (See EGoT13_der.txt in the powergrid-models/platform/DER folder.)
2. Edit powergrid-models/platform/cimhubconfig.json to read "blazegraph_url": "http://localhost:8889/bigdata/sparql".T was an issue for some reason.
3. Edit powergrid-models/platform/DER/insert_der.sh with the new filename. Run it. This will create a uuid file with IDs for the new DERs.
4. Run the sparql query in the Data section below to verify DER-EMs have been added to the system.
5. Edit/run powergrid-models/platform/list_all_measurements.sh
6. Edit/run powergrid-models/platform/insert_all_measurements.sh

I added three test DERs to the model at node 671. There were some initial struggles related to controlling them, but these all turned out to be due to incorrect mRIDs and typos. For results, see /Log Demos/testlog(9-17 binary input demo).csv. The DERs were set to 0 or 1kW, counting up in binary. The results were as expected once we knew what we were looking at: the measurement points all seem to be upstream of the inverter (and thus interconnected) and represent a single phase. So on each phase at node 671, we get 333W of power consumed per DER-EM that's "turned on". When all three are turned on, we get 1kW per phase, or 3kW total. This is as expected.

Once I understood what I was looking at, the rest of the model came into focus. The only other dynamic load in the system is a house model that seems to be disconnected until the third (measurement) timestep. Those together with the DER-EMs had noticeable and consistent effects on current throughout the system. Apart from the house and its components, we have very basic energy consumers, compensators, and node to node line segment measurements, etc.

I spent a long time trying to figure out the data scheme for the test input because I didn't understand that assignment is just a reference and I kept popping the "Time" data from an individual row in my input readout. Once I figured that out, the input manager and historical data DER-S worked pretty well. In this current state, the log columns need to be the mRIDs for the battery inverter control IDs (see query below.) Once per second, that message is ssent to the input handler which automatically changes the values for each mRID to the value in the cell. A single input command is sufficient for this (and will probably be sufficient from now on).

Functional testing completed sat. There is an issue with the timecodes starting 6 seconds later than they should be, but this is minor at this point. I made a card to troubleshoot it later, it won't hold up progress.

The following Test Plans were eligible for Phase 1 Testing:

- MC04
- MC05
- DER02
- DER07
- EDM08
- EDM10

DATA:

```
# Storage - DistStorage
PREFIX r:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX c:  <http://iec.ch/TC57/CIM100#>
SELECT ?name ?bus ?ratedS ?ratedU ?ipu ?ratedE ?storedE ?state ?p ?q ?id ?fdrid (group_concat(di
 ?s r:type c:BatteryUnit.
 ?s c:IdentifiedObject.name ?name.
 ?pec c:PowerElectronicsConnection.PowerElectronicsUnit ?s.
# feeder selection options - if all commented out, query matches all feeders
#VALUES ?fdrid {"_C1C3E687-6FFD-C753-582B-632A27E28507"}  # 123 bus
VALUES ?fdrid {"_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62"}  # 13 bus
#VALUES ?fdrid {"_5B816B93-7A5F-B64C-8460-47C17D6E4B0F"}  # 13 bus assets
#VALUES ?fdrid {"_4F76A5F9-271D-9EB8-5E31-AA362D86F2C3"}  # 8500 node
#VALUES ?fdrid {"_67AB291F-DCCD-31B7-B499-338206B9828F"}  # J1
#VALUES ?fdrid {"_9CE150A8-8CC5-A0F9-B67E-BBD8C79D3095"}  # R2 12.47 3
 ?pec c:Equipment.EquipmentContainer ?fdr.
 ?fdr c:IdentifiedObject.mRID ?fdrid.
 ?pec c:PowerElectronicsConnection.ratedS ?ratedS.
 ?pec c:PowerElectronicsConnection.ratedU ?ratedU.
 ?pec c:PowerElectronicsConnection.maxIFault ?ipu.
```

```
 ?s c:BatteryUnit.ratedE ?ratedE.
 ?s c:BatteryUnit.storedE ?storedE.
 ?s c:BatteryUnit.batteryState ?stateraw.
   bind(strafter(str(?stateraw),"BatteryState.") as ?state)
 ?pec c:PowerElectronicsConnection.p ?p.
 ?pec c:PowerElectronicsConnection.q ?q.
 OPTIONAL {?pecp c:PowerElectronicsConnectionPhase.PowerElectronicsConnection ?pec.
 ?pecp c:PowerElectronicsConnectionPhase.phase ?phsraw.
   bind(strafter(str(?phsraw),"SinglePhaseKind.") as ?phs) }
 bind(strafter(str(?s),"#_") as ?id).
 ?t c:Terminal.ConductingEquipment ?pec.
 ?t c:Terminal.ConnectivityNode ?cn.
 ?cn c:IdentifiedObject.name ?bus
}
GROUP by ?name ?bus ?ratedS ?ratedU ?ipu ?ratedE ?storedE ?state ?p ?q ?id ?fdrid
ORDER by ?name
```

RESULTS:

All tests completed sat. Phase 1 of the ME is now complete.

Nov 8, 2021 – Phase 2 completion

OBJECTIVE:

Complete, test, and refactor phase 2.

OUTLINE:

- Establish locational association data (EDM topology, node MRIDs most likely)
- https://miro.com/app/board/o9J$_l uL3bbE=/$
- Establish DERIdentificationManager
- Establish DERAssignmentHandler
- Write a test Historical Data input log using locational data
- Write DERSHistoricalDataInput locational data function (standard)
- Modify MCInputInterface to use DERIdentificationManager association tables by default
- Functional asynchronous test of MC: Logs-Assignment-Association-Realtime data-convertion-Input-Sim-Output-Logs
- Add association data feedthrough to EDMMeasurementProcessor
- Test EDMMeasurementProcessor
- Attempt to add 2-phase battery inverters to buses.

PROCEDURE:

Establish Topological Processor

Topology refers to the location or orientation of components in the model in relation to one another. The most basic locational "units" in the model are Buses; each DER-EM is connected to a Bus, and that connection and therefore bus would be located somewhere in the world.

However, the Grid Operator (and any contracted DERMS or GSP) wouldn't necessarily view the grid in terms of individual buses. The GSP's server/client program, for instance, expects information in "Groups". These Groups may be concerned with one bus, groups of buses, information coming from particular topological branches, etc. This topology is something that needs to be decided ahead of time in accordance with the GO's needs; or, in our case, with the parameters of the testing we're doing with the GSP.

To this end, we developed a topological processing class, GOTopologyProcessor. The topology is stored within an .xml file, in the repository as "topology.xml". This xml file is used to generate a topology lookup "dictionary", with each "Group" key containing a number of "Bus" values. This can be used to determine which buses are within a Group, so that the proper data is being provided to the GSP by the GO (later in development). The dictionary can be reversed so that each bus can be queried for whatever groups it is in.

Currently, we're concerned with functional development and testing of our respective systems; so, each Group is associated one-to-one to each bus in the 13-node model. Group 1 is Bus 650 only, Group 2 is Bus 646, etc.

Establish DERIdentificationManager and DERAssignmentHandler

The input to the MC will come from a variety of sources via DER-S classes. One (conceptual) requirement of each input is that it needs to be located somewhere in the model; without that locational data, DERs would be assigned arbitrarily to buses throughout the model and topology would become meaningless. Each input DER also needs some sort of name or unique identifier so that the system can keep track of which data is going to which DER-EM. Finally, the DER inputs need "operating data" such as power consumed,

importing/exporting, etc; this can be a lot of things, but the idea is that it is the data that is being used to determine the updated grid states of each DER.

Each DER-EM has an mRID associated with it's inputs and controls. This is internal to the CIM model and not known to external inputs. What is known is the bus each input will be assigned to. So, it is possible to assign each input to a DER-EM mRID on the proper bus automatically, provided a DER-EM exists on that bus. So the first step in the assignment development was to automate DER-EM addition.

CIMHub

PNNL provides a set of tools and scripts that allow modifications to an existing model in the blazegraph database. IMPORTANT NOTE: fully stopping the docker container with './stop.sh -c' or an updated version of GridAPPS requires the process to be rerun.

These scripts look at a user-generated text file containing a list of all the DERs we want added to the model, their Bus location (by bus name), and label plate information. This text file is used to drop existing DERs, add the new DER-EMs, and generate the required mRIDs as well as measurement mRIDs for each. The process by which it does so is beyond the scope of our project; suffice to say that the mRIDs can be easily gleaned by database queries either using the web app or within the MC. So, in order to add the proper amount of DER-EMs, we modify the text file and run the $\text{Initialise}_{D}ER_{E}Ms.batbatchscripttorunallthenecessaryroutines.$

This must be accomplished before running the MC simulation (though later on it will likely be integrated with the MC via the batch script.) Because of this, we can't simply add DER-EMs dynamically for each input. Ensuring the proper number of DER-EMs is added to the text file prior to simulation will need to be part of the test procedure. However, we CAN automate generation of said text file. This is not yet implemented, but the idea is that the GSP will generate this text file during its own registration process; it will then be merged with another text file that will contain extra DER-EMs as required for the input logs (for instance). This will ensure that there are enough DER-EMs on each bus to represent all inputs. If not, the assignment handler throws an exception and closes the MC program.

DERAssignmentHandler

Each DER-S object requires a `self.assign_der_s_to_der_em()` method, which is called once by `DERAssignmentHandler.as` during the MC startup process. This is part of the DER-S API function, since each input will represent and provide locational data in different ways. This customizable method retrieves the names and associated buses for each input in the DER-S via some custom function, uses the Bus to get an mRID for a DER-EM on the proper bus, and associates the name with the mRID. The name and bus information are fed through to the measurement processor, which makes the data easier to compare (since measurement names use the DER-EM name, while the input names are whatever the input uses.)

DERIdentificationManager

This class is a bit more simple; it takes the input name/mRID association data and stores it in a lookup table. The accessor methods return an mRID for a given name, or vice versa. This is used by the MCInputInterface; each timestep it retrieves updated states from each DER-S and stores them in a unified input request, each line of which contains the name and operating states for each DER input this timestep. The MCInputInterface takes the names and looks up the proper mRIDs for each, then uses the mRID/data pairs to generate the proper input requests on the Simulation API using the gridappsd python library.

This method thus allows inputs to be delivered to DER-EMs on the proper bus without EVER needing the inputs or DER-Ss to know any of the model data. As long as the input is on a valid bus with an available DER-EM, the input will be assigned and updated states will be sent properly and automatically.

Update input logs and DERHistoricalDataInput

The previous "test" functionality used mRID names as log headers to forcefully send updated states to DER-EMs. The logs needed to be updated to use a more realistic functionality using names and Bus numbers to utilize the Assignment and Identification classes. The logs and DERHistoricalDataInput were iterated on to create a better test case as follows.

Logs

The logs were modified by doubling the columns. If the first (Timestamp) column is removed, the columns are paired. The first column in each pair contains "data" (in this case, real power consumed) and the second column contains the bus number. The headers are a unique identifier, along the lines of "DER1", "DER2", etc. Any unique identifier can be used. This input method is not optimized since the Bus numbers are only used once in the first timestep for identification, but there's no need to overcomplicate the log input at this point. We're just using it for testing purposes and this was easy to implement.

DERHistoricalDataInput

The biggest addition to this object was the `.assign_der_s_to_der_em()` method. This method goes through the list of DERs read from the (odd) headers in the log file, gets the bus numbers for each DER from the even columns of the first line of the .csv, and uses those name/number pairs in the Assignment handler to generate the association table (name/mRID). All updates to the class are in service of this function. Then each timestep, as before, the class checks to see if there's a line of data corresponding to the current simulation timestep, reads the line of data, organizes it into name/value pairs, and sends those name/value pairs to the Input Processor, which replaces the name with an mRID (see Identification Manager) and generates the input request difference messages.

Modify MCInputInterface

As has been detailed above, the input branch was modified to assign DER-EM mRIDs to DER inputs by looking up DER-EMs available on the proper bus and associating them with input names. The updated states come from the inputs in name/value pairs, with the name being a unique itentifier and the value being a power consumption. Association data between mRIDs and unique identifiers is handled in the simulation startup by assignment, and used during the simulation by looking up mRIDs for a name using the association tables in the Identification Manager. As such, the biggest modification to the Input Interface was to add a step in which the Interface reads an input request from a DER-S, takes the names, looks up the associated mRIDs using the Identification Manager, and proceeds to generate the input request messages GridAPPS-D needs in the proper format (using the difference builder method in the gridappsd library; unchanged from Phase 1)

One important note is that power consumption (and presumably generation) is the ONLY implemented grid state input right now. Any other usage of the DER-EMs (voltage or reactive power, for instance) will require additional methods in the MCInputInterface AS WELL AS an iteration on the input request message format between the DER-S and Input Interface. This iteration will need to include a key/value pair for type of grid state update; power, voltage, etc. This will not require any modifications to the assignment or association objects as those merely tie names to mRIDs, so it's not expected that this will be a challenge. It's just not yet been necessary to implement.

Add association feedthrough data

This turned out to be a major pain, but worked in the end. One major challenge of appending association data to measurements is that there are multiple mRIDs corresponding to a single component in the model: each DER-EM has terminal mRIDs, power connection mRIDs, measurement mRIDs, etc. We needed a way to associate the input mRID with the measurement mRID of each DER-EM.

First, we needed to query GridAPPS-D for a new model dictionary containing all of the possible information for all the loads on our feeder model. This was added to `edmCore.establish_mrid_name_lookup_table()`. Now instead of just getting the mrid/name lookup table used by the measurement processor, it also gets this `"cim_dict"`.

The `cim_dict` is then used in a method in the `EDMMeasurementProcessor.append_names()` and `.append_association_data` The methodology is sloppy and difficult to explain, let alone understand, but in short it takes values from both dictionaries and uses it to determine the name of objects based on the names of the DER-EMs taken from the `cim_dict.` These names need to be parsed and excess characters removed to align data in the `cim_dict` with the input association data. Once that's done, we have inputName/inputmRID/DER-EMname/measurementmRID data all lumped together, which is then added to each measurement's value dictionary each timestep. The end result is that our output logs are MUCH larger, as each cell is a dictionary with a half dozen or so key/value pairs. This is obviously not the most efficient method of maintaining this data, but as it's all present it will suffice for now. It was always the intent that a set of tools would be needed to be developed to unpack and use the logs for analysis, and now the logs are larger but much more useful.

This feedthrough data will also be necessary when the GO is implemented, so feedback can be presented to the DERMS about whether DERs are operating as necessary for the dispatch.

2-phase DER-EM test

In short: this works fine. The 13-node feeder has some 2-phase and 1 phase buses. Using the DER-EM addition script and modifications to the text file to use only BC phases (for instance, see above) inverters can be added that only use and affect two or even one phase of the grid at that bus. This can be used directly right now; however, there is no method to distinguise whether or not a DER-EM is 3 phases or not right now, so that will have to be implemented before we can mix 2-phase and 3-phase DER-EMs on the same bus.

PARAMETERS:

N/A

OBSERVATIONS:

I've done some refactoring, but more can be done. It's just not really worth the time right now. A big thing would be cleaning up the appendage stuff in the Measurement Processor.

I added a new class and object to the system: MCConfiguration. It's mostly a data holding structure for various MC config stuff. For example, instead of having to add a method for each DER-S to the startup methods like assignment and to the on-timestep DER-S read methods, there is now a single attribute where you can list all active DER-S objects and comment out ones that aren't needed.

Functional testing completed sat. I tried phase 2 testing before implementing the feedthrough, which doesn't work becuase a lot of the test plans needed the feedthrough data in the logs to be completed. So in this case I will not provide a list of Test Plans completed. They've been updated in their individual documents.

DATA:

Most of the relevent data can be found in the logs. The DER addition scripts are added to a folder called /DERScripts, and invoked by the `Initialise_DER_EMs.bat` script. Refer to the GitHub repository for more info.

The individual dictionaries communicated from actor to actor are manipulated using complex and often repetitious methods. To see the contents of a dictionary at a given simulation time, the best way to go about it is to add a print() function to the code and read it in the terminal.

RESULTS:

All tests completed sat. Phase 2 of the ME is now complete.

Dec 31, 2021 – Phase 3 completion

OBJECTIVE:

Complete, test, and refactor phase 3. (Note: completed well before this date, just updating the journal)

OUTLINE:

- Establish RWHDERS class
- Write RWHDERS method to access emulator data
- Write RWHDERS assignment method
- Test RWHDERS assignment
- Write RWHDERS processing method
- Test RWHDERS processing to output
- Functional test MC: Emulator changes - RWHDERS input - states output - Conversion - Input - Sim - Output - Logs

PROCEDURE:

Establish RWHDERS class

RWHDERS is the second DER-S class implemented, so it is the first case in which DER-S standardization is used. DER-S standardization is a function of the output of the class, since the DER-S concept allows for a wide array of inputs, processing functionality, etc. The following are the required components of any DER-S class:

- Attribute `self.der_em_input_request`: this attribute contains the data required each time step to update DER-EMs with the most current data from the DER-S. It is a list of dictionaries, with key-value pairs containing `{DER_Name:Power_in_W}` data for each DER handled by the DER-S.
- Method `self.initialize_der_s(self)`: Initializes the DER-S by reading files, connecting to emulators, etc. Called during the MC simulation startup process, and generally speaking, generates some attribute or dictionary containing the names of each DER and their respective locational data for assignment to DER-EMs.
- Method `self.assign_der_s_to_der_em(self)`: Called by the DER assignment handler object for each active DER-S. Uses data from within the DER-S to assign DER-EMs to DERs at the proper topological locations.
- Method `self.get_input_request(self)`: Returns the `self.der_em_input_request`. Generally, will also call a DER-S function to update the input request. Called each timestep by the timekeeper to transfer input request data from the DER-S to the Input Handler to be updated into the EDM.

Further attributes or methods are not standardized between DER-Ss and will make up the "processing" functionality which turns input data to the DER-S into the DER-EM Input Request format for the class (either by parsing it directly or possibly even having to calculate it based on input parameters, label plate data, etc.)

Write RWHDERS method to access emulator data

`rwhders.initialize_der_s()`: calls `rwhders.parse_input_file_names_for_assignment()`. The method we've chosen to use for RWHDERS is to circumvent communications protocol requirements by having the emulators write DER input data to text files, which are formatted with the proper identification, location, and power data. So, to access emulator data, RWHDERS must read these files. The input file names contain both the DER name and location, so a string parser seperates out that data for use by assignment and

the identification dictionary (which keeps the internal DER names associated with the file names for each without requiring parsing every time step.)

Write RWHDERS assignment method

Since the file parsing method has already created an input identification dictionary, this assignment method is routine. It grabs each line of the dictionary (containing the DER name and location), then via the assignment handler associates each name with the mRID of a DER-EM on the proper bus.

Test RWHDERS assignment

Not fully completed. Testing RWHDERS assignment requires the emulator output files to be generated, and the GSP isn't quite there yet. Mock input files work fine, however.

Write RWHDERS processing method

This turned out to be extremely simple, since the input files contain direct power values. It just needs to read the power from the files and update the input request dictionary once per time step. If the emulators were to instead provide "exporting/importing" flags and label plate data, as was expected, the der-em input request method would need to be updated to calculate power based on operating parameters.

Testing

As mentioned, testing is being pushed off until the GSP is more complete. A lot of the test plan tests require more sophisticated inputs than can be generated in mockups. For instance: the input files for RWHDERS are static. Since they can't be modified during the simulation, their values are unchanging, and therefore changes can't be tested during the simulation or seen in the outputs. We tested as much as we could and everything seems to be working as intended, but this won't be confirmed until the system is gearing up for actual testing.

PARAMETERS:

N/A

OBSERVATIONS:

RWHDERS turned out to be simple since we made some comprimises with the GSP team to simplify interactivity. Both systems can easily generate, read, or parse text files, and the nature of the data is so simple that it makes sense to just come up with a scheme for ourselves. This does limit the ME to a new requirement: it must be run on the same system as the GSP program; though, we were largely expecting that anyways and there's not a compelling reason to run them on seperate systems in the first place.

DATA:

Most of the relevent data can be found in the logs. The RWHDERS input files are (and must be) located in the "RWHDERS Inputs" directory in the ME folder, as seen on the GitHub.

The individual dictionaries communicated from actor to actor are manipulated using complex and often repetitious methods. To see the contents of a dictionary at a given simulation time, the best way to go about it is to add a print() function to the code and read it in the terminal.

RESULTS:

RWHDERS complete. Partial functional testing completed sat, full testing pending completion of GSP.

Dec 31, 2021 – Phase 4 completion

OBJECTIVE:

Complete, test, and refactor phase 4. (Note: completed well before this date, just updating the journal)

OUTLINE:

- Determine scheme and method for GO-GSP communications
- Establish GOSensor
- Write sensor method
- Test sensor
- Write decision-making method
- Test decision-making process
- Establish GOOutputInterface
- Write method to gather information for service request/feedback
- Write method to generate service request messages
- Write method to establish/perform communications tasks
- Develop and execute synchronous functional test of system

PROCEDURE:

Determine scheme and method for GO-GSP communications

I was expecting this part in particular to be the most complex part of the system; however, since we'd already limited ourself to requiring the ME and GSP to be run on the same system due to the RWHDERS input, we recognized that we could do the exact same thing here by allowing communications to be handled by the MC writing, and the GSP parsing, text files. Rather than use the customized .csv files, these communications are easily handled by .xml which is easily written and parsed by a multitude of programming language libraries.

The format for each message is based on the messaging requirements as seen in the GSP Implementation Profile. See below.

Establish GOSensor

The GOSensor class was envisioned to function in two modes: a Manual mode in which grid service requests from the GSP would be posted at a specific time by user-configured parameters, and an Automatic Mode in which the requests are determined in a more "realistic" manner based on grid states.

After some thought, we determined that only the Manual Mode would be necessary for the current stage of the project. This is due to the fact that we are already using carefully controlled situations for specific testing. Including the ability to automatically detect and respond to grid events is a function of the Grid Operator; however, the EGoT system is designed to create a DER aggregator, the ME tests the EGoT system, and a sophisticated Grid Operator, while possible, is beyond the scope of current work. As such, only Manual Mode is necessary for the time being, with most logical determinations being made not by the class but by the Test Engineer during system configration and eventual log analysis.

Write and test sensor method

The class contains a `self.update_sensor_states()` method, but this is currently unused since Automatic mode isn't being implemented at this time. It's a placeholder for when that does happen, and can include

on-timestep functionality to grab grid states from the measurement processor and make the service request decisions.

Since this isn't currently being implemented, neither will it be tested at this time.

Write decision-making method

There is a `self.make_service_request_decision()` method that is called once per timestep. This is an encapsulation method that reads the configuration settings to determine if it's in Automatic or Manual mode and calls the according method. In the case of Automatic mode, that method is unwritten and so it's a simple pass function.

For manual mode, a seperate method is called: `self.manually_post_service`. This takes the current simulation time as an argument and reads the contents of a prewritten, preloaded "service request" file. This file is in .xml format and contains all of the data required to post a service, including the designated time and duration.

When the simulation reaches the designated time, the proper service is read and "posted". That is, the parameters of the requested service are packaged in a dictionary and added to the `self.posted_service_list` attribute for use by the GO-GSP communications interface.

In the future, the Automatic mode will read grid states and determine if a service is required and, if so, what parameters are required to address the grid issue. The function will then generate a dictionary with information in a format identical to the manually inputted Posted Services.

Write RWHDERS processing method

This turned out to be extremely simple, since the input files contain direct power values. It just needs to read the power from the files and update the input request dictionary once per time step. If the emulators were to instead provide "exporting/importing" flags and label plate data, as was expected, the der-em input request method would need to be updated to calculate power based on operating parameters.

Test decision-making process

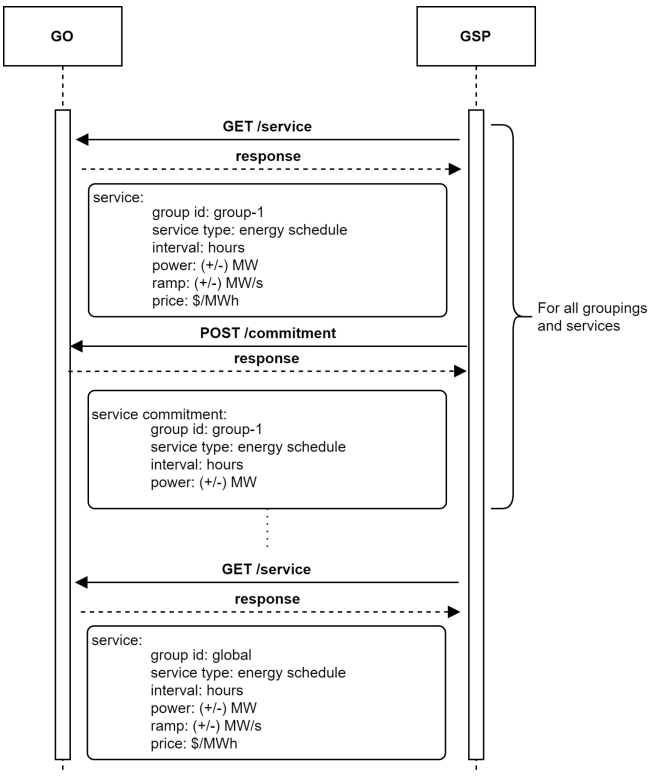
Testing was completed satisfactorily after the GOOutputInterface was built and partially tested. The input and output files corresponded with one another. See below for more details.

Establish GOOutputInterface

The main purpose of the GOOutputInterface class is to provide a custom API between the MC and the DERMS under test. It needs to be specifically designed and customized for the requirements of the DERMS. It could take any number of forms with regards to communications, how they're packaged, etc. But in all cases, they take the Posted Service list, translate it into data readable by the DERMS, and send it over each timestep so the DERMS knows what grid services the GO is requesting at any given time.

In theory, the GO-GSP communications should be bidirectional. The GO sends the list of services to the GSP, which the GSP acknowledges, but it also provides settlement data for use by the Grid Operator for pricing and validation. However, in our tests, validation will be performed by the Test Engineer using human analysis on the logs; therefore, there was no need at all to actually code a system to do this automatically. The only communications necessary are the posted service list from the GO to the GSP.

Because of this simplification and prior requirements, we decided that the most efficient way to provide the posted services to the GSP would be to put them in an .xml file once per time step, updated based on the requirements of the Manual Mode file (or automatically generated services later on.) The services are put in a .xml file that the GSP (which, recall, is located on the same system) can read and respond to by its own requirements. The results end up in the logs.



gogsp[H]

In short, the current architecture of the ME's GO is to read an .xml file containing required service parameters, parse them at the proper time and place them in dictionaries in the Sensor class, read the dictionaries with the Interface class, and repackage the dictionaries back into nearly identical .xml files for the GSP to use. This seemingly roundabout method is necessary both to ensure services are occuring at the proper time, and for modularity purposes: if the output protocol is changed, the only part of the method that needs be changed is the interface message wrapping. No undue coupling has occured.

Testing

As with phase 3, full test plan testing requires a functional GSP for full-loop operations. We tested the system as fully as possible without the GSP, and found that it seems to function as required.

PARAMETERS:

N/A

OBSERVATIONS:

RWHDERS turned out to be simple since we made some comprimises with the GSP team to simplify interactivity. Both systems can easily generate, read, or parse text files, and the nature of the data is so simple that it makes sense to just come up with a scheme for ourselves. This does limit the ME to a new requirement: it must be run on the same system as the GSP program; though, we were largely expecting that anyways and there's not a compelling reason to run them on seperate systems in the first place.

DATA:

Most of the relevent data can be found in the logs. The .xml input and output files are in the ME main folder in the GitHub.

The individual dictionaries communicated from actor to actor are manipulated using complex and often repetitious methods. To see the contents of a dictionary at a given simulation time, the best way to go about

it is to add a print() function to the code and read it in the terminal.

RESULTS:

ME reaches 1.0 pending testing. GSP and ME functional testing will occur in tandem when the GSP is further developed.