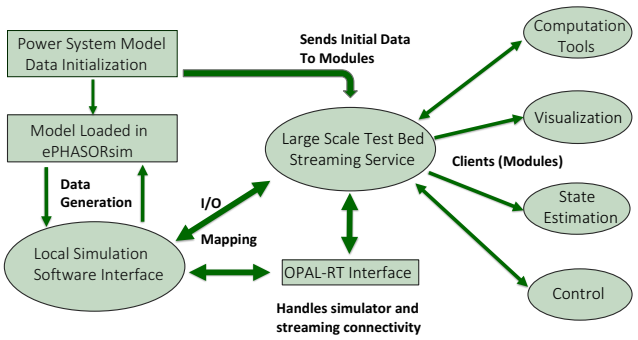
Overview

OpalApiControl is an application interface for real-time and offline simulations of North American grid system models developed at CURENT. The purpose of the interface is to connect various components of a power system network for visualization, control, data acquisition and state estimation. Through the use of this interface, data acquisition is organized and maintained for requesting modules through a live network server. Data generation and extraction takes place through ePHASORsim model's using RT-LAB's API functions[1]. Particularly, this interface is designed to organize power system device data for initialization and corresponding mapping for data transfer of system device variables. This interface utilizes the real-time capability of RT-LAB's hardware by acquiring data from running models in real-time at the users defined sampling intervals. Data collection is retrieved and served to multiple requesting modules, as it is the key hub for the communication from outside network components, to the running model itself. The system structure is generalized below.

Interface Structure

As seen, in the network graph, the OpalApiControl interface is used as the arbiter between running models in ePHASORsim and CURENT's Large Scale Test Bed, containing the network of modules for visualization, state estimation and control. The interface also performs the task of sending a model's initialization parameters to the server according to a requesting module's needs.



Interface Python Modules Descriptions

parser

A PSS/E 32.0 file parser gathers the system devices and data for a chosen model. This initialization data is then sent to a requesting server. A list of available devices can be seen in parser/settings.py. The settings.py file contains the Settings object for a systems device initialization storage. Any devices that are not defined in the parser will be removed from the system parameters structure prior to sending to the server.

The parser/device_vars.py file is used to create excel pin files which ePHASORsim requires for model I/O routing. Since FMU files are utilized for device model's, device_vars.py first parses the model paths FMU folder for user-defined devices and variables. After extraction, an FMU pins excel sheet is created in the user's model folder. Additional I/O settings as defined by ePHASORsim can be appended to the excel sheet. Currently, the excel file created does not affect the referenced excel file by ePHASORsim, so the user can pick and choose which rows from the FMU pins file they would like to add to the model.

[1]The interface should be compatible with any software using the RT-Lab environment for data acquisition, but testing has only taken place in ePHASORsim with I/O ports defined according to the ePHASORsim manual. It is believed only small changes would be necessary to be used with alternative RT-LAB software

Lastly, parse/bus_coords.py takes a specified bus coordinate file, and creates a new PSS/E file with updated bus parameters. If the user wishes to send bus coordinate data to requesting modules, this file should be chosen as the interface's main PSS/E file. This file should only be used for the local system parameter settings, and not within ePHASORsim. The excel file parser parse/excel.py is in development, as it still needs to be interfaced with the Settings object.

utils

Contains files for system parameter constants as well as objects for setup, control and streaming using the interface. utils/ltbsetup.py contains functions to gather system initialization data, as well as variable header and indexing storage for routing from an ePHASORsim model to requesting modules. utils/simcontrol.py is an object which arbitrates any requests to send or receive data from ePHASORsim models. This object contains the methods for controlling RT-LAB's state by loading, executing and resetting models. The variable header list is also created from this module to be called by the LTBsetup object. In summary, utils/simcontrol.py is the only object that directly interacts with ePHASORsim. For data streaming, module requests, and input events, utils/simstreaming.py contains an object for facilitating server connections, and managing requesting modules, as well as streaming initialization data, and data acquisition values at the sampling rate of 30 $\frac{samples}{sec}$[2]. The utils/run.py file is the compilation of the above objects for a full integration of the LTB server connection, RT-LAB control and running ePHASORsim models.

examples

OpalApiControl/examples implements working examples for setup, running, device variable FMU parsing, and bus coordinate extraction. All `run_model()` instantiations behave as follows,

```
run_model(project=None, model=None, raw=None, dyr=None, xls=None,
↪  path=None, server='tcp://127.0.0.1:<####>'))
```

A more detailed explanation of the interface implementation is described in Program Structure. An example for FMU variable parsing in examples/device_var_parse181.py shows how to use the Device object methods for handling collected devices and variables. examples/examp_event.py is a simple file that shows the ability to send events from a python script on the local machine. Events can also be sent through connected Matlab modules on a local or remote machine connected to the server[3]. examples/examp_se.py is a script for connecting a requesting module via python, where as ltb_se_test.m contains the Matlab script for setting up a connected module in a Matlab environment[4].

[2]User's can determine custom sampling intervals, however, they must be greater than or equal to the model step size. The rate which is chosen for acquistion currently sets the streaming interval as well considering the interface only sends data when new data is available

[3]Requires pymatbridge and dime python packages

[4]Requires CURENT's ltb package

Program Structure

Not all streaming and acquisition needs will be the same, so the OpalApiControl package is designed to allow for custom workflows. Within the application itself, all of the necessary tools for a full implementation are organized for modularity. In fact, the interface itself is operational as is, granted that the server connection, RT-Lab licenses, and python package requirements are met. The guidelines required to maintain a working interface are as follows in the order of which the program currently requests. There are three main objects for full functionality plus a fourth for storing settings. LTBSetup, SimControl,Streaming and Settings.

Setup

The ePHASORsim model must be compiled for the interface to connect. Once connected, the output routing for ePHASORsim is available. The SimControl object can take care of all connectivity and model loading as long as it is compiled. It must be created before any other functions which request information from the model are called. A Settings object is available in SimControl, which allows the interface to access system parameter information independent of other objects.

PSS/E .raw and/or .dyr files for initialization to the server for requesting modules must be in the ePHASORsim model or simulink path. The LTBSetup object can then be instantiated to store system parameters as follows,

```
sim_data = LTBSetup(raw=raw, dyr=dyr, xls=xls, path=modelPath, model=model,
 ↪  simObject=sim) #Creates setup object
sim_data.getsysparam()
sim_data.get_varheader_idxvgs() #Calls SimControl
SimControl.set_settings(sim_data.Settings)#Stores system parameters in SimControl
```

A streaming object must be created to broadcast system parameter, variable names, and associated indices to the designated server. A user can send initialization to all modules on the server, or to specific modules. This allows the user to update specific modules system parameters individually if changes are not universally pertinent.

```
send_init(self,recipient= 'all') #Broadcast to all modules
send_init(self,recipient='<specific module>') #Send to one module
```

Data Acquisition

The main purpose of OpalApiControl is to transmit data generated from the RT-LAB sc_master outputs. Acquisition groups are defined in the sc_console OpComm block in RT-LAB. There are multiple ways to implement the acquisition itself, but all calls through RT-LAB must be from a thread connected to the ePHASORsim model. Issues can occur when multiple threads are requesting connectivity, so the SimControl object is currently setup to handle the data acquisition when called by the simulator from the local machine. The current default acquisition time is 30 $\frac{samples}{sec}$ but can be changed when initializing a SimControl object. The acquisition function is called by the simulator which then makes a call through the API to retrieve simulation time and data as shown below,

```
sigVals, monitorInfo, simTimeStep, endFrame =
 ↪  OpalApiPy.GetAcqGroupSyncSignals(self._acqGroup - 1, 0, 0, 1, 1)
missedData, offset, self.simulationTime, _ = monitorInfo
```

Conditions for handling the data for a given interval are then implemented and can be seen in utils/simcontrol.py. The `OpalApiPy.GetAcqGroupSyncSignals()` call takes the following parameters,

Acquisition group number

Synchronization mode (1 Enables, 0 Disables). This synchronizes the communication block(OpComm) with the real-time target

Interpolation (1 Enables, 0 Disables).

Threshold. Sets the maximum wait time between the communication block and the simulation. *If the wait time is too small, the model will crash. Set to 1.*

Acquisition Time Step. 1 sets acquisition to match model time step. This setting cannot be lower than the model time step[5].

If data has successfully been acquired, the `acquire_data()` function will return the simulation time, the simulation step number for the given sampling, and a tuple of the data values. The Streaming object is then used to transfer model information to the server.

Streaming

All interface server requests are handled by the Streaming object. It routinely checks the dime server connection for module requests with `sync_and_handle()`. This gathers module requested devices, variables and indices pertaining to the **Varheader** and **Idxvgs** initialization from setup. Module requests are stored in `Streaming.moduleInfo` and sent as follows,

```
def vars_to_modules(self, t, k, varout)#Simulation time, step, and data from
 ↪  acquire_data()
        varout = array(varout)#Converts data tuple to numpy array
        for mod in self.moduleInfo.keys():
            idx = self.moduleInfo[mod]['vgsvaridx']
            try:
                values = varout[idx]
                Varvgs = {'t': t,
                          'k': k,
                          'vars': array(values),
                          'accurate': array(values),
                          }
                self.dimec.send_var(mod, 'Varvgs', Varvgs)
            except:
                #module request out of bounds, remove module
```

Varvgs is a dictionary sent to each module according to its requests that contains the simulation time, simulation step, and the requested data values as a numpy array. **vars_to_modules()** Takes care of all module requests at the same time. Current implementation requires all modules to be requesting for sampling which is greater than or equal to the interval specified in setup. Multiple modules with different sampling requests will require multiple data acquisition threads which both handle connectivity to ePHASORsim cooperatively.

[5]The acquisition time step returns an endFrame value. This value is related to the buffering settings in RT-LAB's probe control menu for an acquisition group. The endFrame returns true when a complete sample has been finished. For the purposes of the OpalApiControl Interface, no buffering is needed. The acquisition time step actually sets the samples for buffering during one acquisition, and is therefore not needed for an interface which only needs one value per signal at a requested interval. However, models which require simulink memory for data initilization will encounter undersampling issues when the buffer is set to low. Set the samples per signal as low as possible in the RT-LAB probe control settings. It is advised to move all ePHASORsim computation to the sc_master block to lessen the demands of the sc_console

Events

Streaming stores module events in an event queue which is checked with the simulation time at every iteration. Multiple events with the same time are sent as one signal event to ePHASORsim. Events are categorized by device and take a device id, action, duration of action and time of occurence (Example in examples/examp_event.py). The following events are currently available for use,

Bus Faults - Events with name 'Bus'. Action 1 sets a bus fault, and action 0 disables the fault.

Generator Status - Events with name 'Syn'. Action 0 sets status to off, 1 sets status to on.

Line Status - Events with name 'Line'. Action 0 sets status to off, 1 sets status to on.

In order for events to be sent to the proper input ports in ePHASORsim, the port line connections must be labeled 'bus_fault', 'syn_status'[6], and 'line_status'. The interface searches for these port names when sending a signal to the sc_console. The id specifies which device to set. Input ports for each device are automatically labeled as 'bus_fault(1)', 'bus_fault(2)',... 'bus_fault(n)' etc, when multiple devices are found in the model. If only one device is in the model, the port line connection must be named as 'bus_fault(1)', or there will be an input name error.

Comments

The OpalApiControl interface is designed to be a research tool for power system model analysis using ePHASORsim. Basic implementation for data visualization and state estimation has been utilized. Increasing complexity of modules that require the use of the interface may need to implement alternate techniques for data acquisition, streaming handling and events. Current versions run with few undersamples on a 181 and 300 bus system model. User's will need to adjust RT-LAB parameters accordingly to ensure real-time requirements for the interface are met. Very simple local machine scripts can be written to collect and analyze a models data without interacting with a real-time target or server, and may find great uses in organizing a user's data collection and computations during model execution. Just as easily as data values can be linked and sent to a server, graphing utilities and databases on a local machine can be interfaced with the OpalApiControl package for more efficient model analysis.

[6] Tripping large generators will cause ePHASORsim to crash.