# GridAPPS-D

***Release 2021_05.0***

**The GridAPPS-D Team and Community**

**Jul 04, 2021**

# INSTALLATION RUNTIME


This will be on the main page here!

CHAPTER

ONE

# WINDOWS 10 INSTALLATION

This section contains detailed installation instructions and runtime environment tips for running GridAPPS-D and its dependencies on a Windows 10 machine.

## 1.1 Virtual Machine & Docker Setup

### 1.1.1 Table of Contents

A typical Windows 10 installation does not include several of the tools needed to run the GridAPPS-D Platform Several software packages need to be installed prior to installing GridAPPS-D in the next step

Installation Steps:



### 1.1.2 1. Verify System Requirements

- **OS:**
  - Windows 10, Version 2004 or higher, with Build 19041 or higher
- **RAM:**
  - 8GB (*absolute minimum* for 13 and 123 node models, *may encounter memory overload during installation* );
  - 16GB (preferred for small models, minimum for 8500/9500 node models);
  - 32GB (recommended for application development)
- **Disk Space:**
  - 15GB required for installation

**Note:** The download size is quite large, so it is recommended to use a fiber or ethernet interent connection, rathered than a metered hotspot to avoid excessive data usage charges.

---

### 1.1.3 2. Verify OS Build

To check your OS build, type `winver` in the Cortana seach bar:

**|win_setup_run_winver.png|**

Check to see if your OS is

- For x64 systems: **Version 1903 or higher, with Build 18362 or higher.**
- For ARM64 systems: **Version 2004 or higher, with Build 19041 or higher.**

**|win_setup_goodbad_winver.png|**

If not, run **Windows Update** to get the latest verion of Windows 10 available for your machine. It may take some time for the new OS to download. Multiple restarts are typical while upgrading the windows version.

---

### 1.1.4 3. Install Windows Subsystem for Linux

GridAPPS-D and the associated docker containers will run using the Windows Subsystem for Linux (WSL), which is a new feature to Windows 10 that enables linux code to run natively in Windows without a separate virtual machine. The steps in this section are also available on the Microsoft website

#### 3.1. Enable WSL

Open Windows PowerShell as an administrator:

**|win_setup_open_powershell.png|**

Enable WSL by entering

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /
norestart
```

**|win_setup_enable_wsl2.png|**

Then, without restarting, enable the virtual machine platform by entering

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

**|win_setup_enable_VM.png|**

When completed, restart your machine. It may take a few minutes for the new settings to be applied while restarting.

### 3.2. Upgrade to WSL2

Download the latest WSL2 package .msi installer from the Microsoft repository

Run the update package to install WSL2 using the wizard:

**|win_setup_WSL_wizard.png|**

Open Windows PowerShell again and update the settings to use WSL2 by entering

```
wsl --set-default-version 2
```

---

### 3.3. Install Linux Ubuntu OS

Open the Microsoft Store app, and search for `Ubuntu` and install the desired version (available versions are 16.04, 18.04, and 20.04)

**|win_setup_ubuntu_store.png|**

When it has finished downloading, click `Launch`.

**|win_setup_ubuntu_launch.png|**

### 3.4. Set up Ubuntu in WSL

Wait for the Ubuntu OS to install.

**|win_setup_ubuntu_setup.png|**

Select a username and password. These do not need to be the same as your Windows or Microsoft Account login.

**|win_setup_ubuntu_username2.png|**

## 1.1.5 4. Install Docker for Windows

Download and run **Docker Desktop for Windows** from Docker Hub

Be sure to select "**Install required components for WSL2**"

**|win_setup_docker_wizard.png|**

After restarting your machine, Docker should start automatically, and you will see a notification stating "**Linux WSL2 containers are starting**"

**|win_setup_containers_starting.png|**

---

## 1.2 Installing GridAPPS-D

### 1.2.1 1. Clone the GridAPPS-D Docker repository

Disconnect from your corporate/laboratory VPN (if applicable) and open the Ubuntu terminal:

**|win_setup_open_ubuntu.png|**

Clone the GridAPPS-D repository:

```
git clone https://github.com/GRIDAPPSD/gridappsd-docker
```

**|win_setup_clone_gapps.png|**

---

### 1.2.2 2. Install the GridAPPS-D Docker Containers

Change directories into the **gridappsd-docker** folder and start the latest stable release of the GridAPPS-D platform.

- `cd gridappsd-docker`
- `./run.sh`

It is possible to specify a particular release tag using the -t option and the release tag

- `./run.sh -t develop` - use the `develop` branch with latest beta features
- `./run.sh -t releases_2021.03.0` - use the March 2021 release
- `./run.sh -t releases_2020.09.0` - use the September 2020 release

A complete set of release is available in the associated readthedocs page

**|win_setup_gapps_run.sh.png|**

Wait for the platform to download the required docker containers. This is a very large package and will take several minutes.

**|win_setup_pulling_containers.png|**

After the containers have finished downloading, they will automatically be created and then launched:

**|win_setup_containters_pulled.png|**

---

### 1.2.3 3. Launch the GridAPPS-D Platform

When all the containers are running, the terminal will move inside the docker enviroment, which has its own internal directories and path.

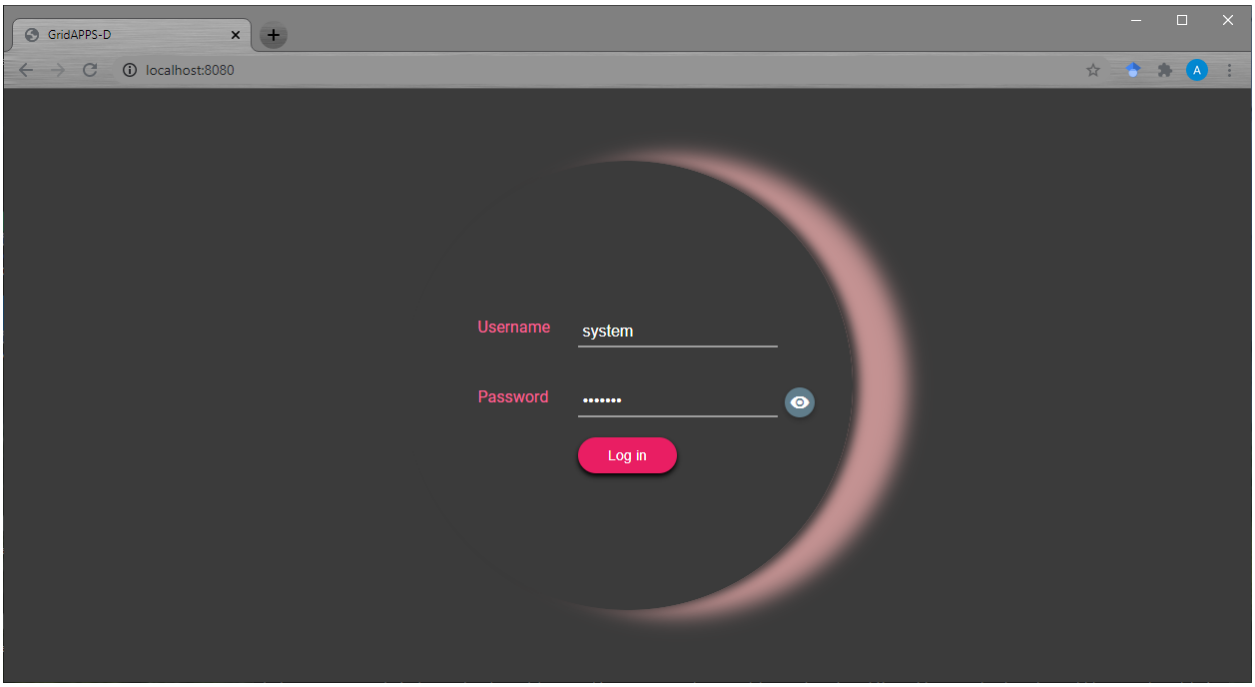
Start the GridAPPS-D platform inside the docker container by running

```
./run-gridappsd.sh
```

**|win_setup_start_platform.png|**

The GridAPPS-D platform is now installed and running.

To confirm, open localhost:8080 to access the GridAPPS-D Visualization App:




***Congratulations! You have successfully installed the GridAPPS-D Platform, and the GridAPPSD-Python development environment!***

---

## 1.3 Running GridAPPS-D

### 1.3.1 1. Starting the Platform

If you are accessing this module after completing the installation steps in the previous procedure, then the GridAPPS-D Platform is already running.

When you start your machine next time, you will need to start the GridAPPS-D Platform again. To do this, change directories into `gridappsd-docker` and run the `./run.sh` script

- `cd gridappsd-docker`
- `./run.sh` or `./run.sh -t release_tag`

### 1.3.2 2. Stopping the Platform

### 1.3.3 3. Restarting the Platform

### 1.3.4 4. Pulling Updated Containers

## 1.4 Installing Python Tutorials

### 1.4.1 1. Install Git for Windows

Install git for windows. This package is required to download and run the python notebooks.

Open gitforwindows.org and download the latest version.

Use the installation wizard with the recommended settings to complete installation.

**|win_setup_install_git.png|**

[*Return to Top*]

### 1.4.2 2. Install Anaconda or Miniconda

Download the latest version of the Miniconda from the Conda.io website:

- Python 3.8 for 64-bit Windows
- Python 3.8 for 32-bit Windows

Use the installation wizard with the recommended settings to complete installation.

**|win_setup_miniconda.png|**

After installation is complete, launch the **Anaconda Prompt (Miniconda3)** from the Start Menu or by typing `anaconda` in the Cortana toolbar

**|win_setup_launch_miniconda.png|**

The miniconda terminal window will open

**|win_setup_miniconda_terminal.png|**

[*Return to Top*]

### 1.4.3 3. Install Jupyter Lab

In the miniconda terminal window, run

```
pip install jupyterlab
```

to install the Jupyter environment for executing the python notebooks. It may take a couple minutes to collect and install all the required packages.

**|win_setup_install_jupyter.png|**

[*Return to Top*]

### 1.4.4 4. Install GridAPPSD-Python

In the Miniconda terminal window, download and install GridAPPSD-Python by running

```
pip install git+https://github.com/GRIDAPPSD/gridappsd-python.git@develop#egg=gridappsd
```

to download the GridAPPSD-Python library and required packages.

**|win_setup_install_gapps_python.png|**

GridAPPSD-Python and all dependencies should have been automatically added to your anaconda path after completion.

[*Return to Top*]

### 1.4.5 5. Download Python Training Notebooks

In the miniconda terminal window, clone the python notebooks by running `git clone https://github.com/GRIDAPPSD/gridappsd-hackathon` to download the python training notebooks.

**|win_setup_install_notebooks.png|**

By default, the notebooks will be saved in the directory `C:\Users\username\gridappsd-hackathon`

Close the miniconda terminal

[ ]:

If running on a remote server (e.g. AWS cloud or university / laboratory server farm), start the notebooks by running

```
jupyter notebook --port 8890 --no-browser --ip='0.0.0.0'
```

[ ]:

CHAPTER

TWO

# USING THE GRIDAPPS-D VIZ

CHAPTER

THREE

# DOCKER SHORTCUTS

CHAPTER

FOUR

# GRIDAPPS-D INTRODUCTION

## 4.1 1. What is GridAPPS-D?

GridAPPS-D™ is an open-source platform that accelerates development and deployment of portable applications for advanced distribution management and operations. It is built in a linux environment using Docker, which allows large software packages to be distributed as containers. Docker tools will be discussed in Lesson

The GridAPPS-D™ project is sponsored by the U.S. DOE's Office of Electricity, Advanced Grid Research. Its purpose is to reduce the time and cost to integrate advanced functionality into distribution operations, to create a more reliable and resilient grid.

GridAPPS-D enables standardization of data models, programming interfaces, and the data exchange interfaces for:

- devices in the field
- distributed apps in the systems
- applications in the control room

The platform provides

- robust testing tools for applications
- distribution system simulation capabilities
- standardized research capability
- reference architecture for the industry
- application development kit

## 4.2 2. GridAPPS-D Platform Characteristics

### 4.2.1 2.1. Vendor / Vendor Platform Independent

The GridAPPS-D Platform and application development environment is independent of any specific vendor or vendor platform, in other words vendor neutral. The results of this effort are intended to be useful and available to any vendor or application developer who wishes to apply them or incorporate them into existing or future products.

### 4.2.2 2.2. Standards-based Architecture

GridAPPS-D is the first platform for energy and distribution management systems that is designed with standards for data integration, including data models, programming interfaces, and data exchange interfaces between grid devices in the field, distributed applications in utility systems, and applications in utility control rooms. This means that the applications developed using GridAPPS-D make them broadly applicable and interchangeable across utility systems, reducing the cost and time for utilities to integrate new functionality.

To the greatest extent possible, the GridAPPS-D Platform incorporates and supports industry standards, in particular interoperability standards, including the power system model representation using the Common Information Model (CIM) and communications with other platforms / physical equipment through DNP3, IEEE 2030.5, and the open field messaging bus (OpenFMB)

### 4.2.3 2.3. Replicable

As a reference implementation of a standards-based architecture, advanced applications and services developed with GridAPPS-D Platform should be replicable, with the ability to be deployed at multiple locations on different distribution feeders with almost no code customization.

### 4.2.4 2.4. Flexible Distribution Simulation

The GridAPPS-D Platform enables users to run real-time quasi-static simulations of large distribution network models with real-time load data, thermal co-simulation of houses, real-time weather data, and real-time operation of switches, DERs, and volt-var control equipment. The platform supports multiple distribution simulators through a co-simulation bridge that abstracts the simulation configuration details to a simple API.

---

## 4.3 3. Data Representation & Management

A key to GridAPPS-D is providing the distribution system application developer with a standardized approach to data. The intent is to allow the developer to make logical references to data referencing standard data models and interfaces without concern for how the data is physically made available. This standardized, logical data interface is based on existing standards to the greatest extent possible.

### 4.3.1 3.1. Standards-based Data Representation

The Common Information Model (CIM) is used for all power system models, which enables rapid exchange of power system models across compliant applications and services. Using the set of standardized model queries provided by the PowerGrid Models API, a GridAPPS-D application is able to scale seamlessly across different network models with no modifications to the application code.

### 4.3.2 3.2. Standards-based Data Interfaces

The GridAPPS-D Platform and GridAPPS-D APIs provide a standardized method for interfacing with power system model data, real-time simulation data, historical data, and log data. Each of these APIs abstract the database specifics, and enable simple queries through a set of standardized messages formatted as JSON strings.

### 4.3.3 3.3. Data Translation to Non-standardized Elements

CIM Hub and the Configuration File API allow conversion of the power system model data from the standards-based CIM XML format used by the GridAPPS-D Platform to model formats used by other software packages, such as GridLAB-D and OpenDSS. This model conversion process can be performed with a simple set of standardized API calls.

### 4.3.4 3.4. Available Distribution Feeders

The GridAPPS-D platform comes pre-configured with a combination of IEEE Test Feeders, PNNL Taxanomoy feeders, and other realistic synthetic models. Additional models and actual utility feeder data can be uploaded easily as CIM XML files into the GridAPPS-D Platform, which can then be used for application testing and real-time simulation.

---

## 4.4 4. Real-Time Distribution Simulation

The GridAPPS-D Platform inlcudes a robust real-time distribution simulator with comparable capabilities to a Dispatcher Training Simulator. This environment enables application developers to test algorithms and application code on both the standard realistic sythetic feeders pre-configured in the GridAPPS-D Platform download and any other power system models that the user can upload through the CIM Hub package.

The distribution simulator is the source of data to the distribution system application developer enabling them to evaluate the performance of their application with ideal or realistic noisy data under different operating and performance conditions.

The GridAPPS-D platform currently supports only quasi-static simulation (i.e. simulation of electromechanical / electromagnetic transients, variable microgrid island frequency, synchro-check relays, etc. are not supported currently). These types of simulations can be performed with GridLAB-D outside of the the GridAPPS-D Platform and application development environment.

### 4.4.1 4.1. Real-Time & Faster-than-Real-Time Simulation

Simulations can be run in two modes:

1) Real-time mode: one second of computer clock time corresponds to one second of simulation time. The GridAPPS-D Platform runs the simulation in each time and publishes simulation data and sensor measurements every three seconds.

2) Faster-than-real-time mode: The GridAPPS-D runs the simulation as fast as possible and does not wait for three seconds of computer clock time to pass before it publishes the simulation data from the current time step. This mode is very useful for creating historical training data sets for AI/ML applications.

### 4.4.2 4.2. Controllable Power System Equipment

All of the power system equipment can be controlled in real-time through the Simulation API, allowing applications to open/close switches, dispatch DGs / DERs, adjust setpoints of rooftop PV, adjust regulator taps, and turn capacitor banks on or off.

### 4.4.3 4.3. Noisy / Bad Data Injection & Communication Failures

The GridAPPS-D Platform supports the Sensor Simulator Service, which is able to inject noise, bad measurements, and data packet losses into the simulation output. The frequency at which sensors publish can also be adjusted and aggregated, allowing realistic representation of real sensors, such as AMI meters that publish data every 15 minutes, rather than at each simulation time step. This allows the user to train and evaluate applications with realistic measurement for meters and sensors, rather than "pure" data created by the power flow solver.

The GridAPPS-D Platform also supports simulation of communication failures through the Test Manager during which data is not received from sensors, control commands are delivered to selected equipment, or both. This enables application developers to test algorithm performance under realistic conditions, during which physical equipment might not respond to control commands.

### 4.4.4 4.4. Reconfigurable Power System Topologies

The GridAPPS-D Platform supports simulation of both meshed and radial power system topologies, as well as reconfiguration of the power system network in real-time by opening / closing / tripping of various switching devices, such as breakers, reclosers, sectionalizers, and fuses. These switches can be controlled by an application through the Simulation API or through the GridAPPS-D Viz GUI

### 4.4.5 4.5. Real-Time Simulation Visualization

The GridAPPS-D Platform includes the Viz GUI application, which presents a simple graphic user interfaces with some of the basic functionalities found in an Dispatcher Training Simulator, inlcuding a one-line diagram of the feeder, colorized switch positions, outage locations, alarm messages, and customizable stripcharts of power flow, node voltage, and tap position.

---

## 4.5 5. Using the GridAPPS-D Platform

GridAPPS-D currently runs in a Linux virtual machine (VM). Although it can be built from sources, the primary form of distribution is as a set of Docker containers. Users can install the Docker infrastructure on their computer and then download the Docker containers. Several platform usage scenarios are then feasible:

1. Start and run the application through its browser interface. Utilities could use the platform this way to evaluate new applications, or to evaluate applications on their own circuits. The App Hosting Manager allows a user to install and configure new applications to run in the platform, by modifying configuration files but without having to write new code. GridAPPS-D will also be able to ingest any distribution circuit provided in CIM format.
2. Write scripted scenarios and responses using the Test Manager, and run those through GridAPPS-D. This mode can be used for a more rigorous evaluation, and also for operator training.
3. Write a new application, using one of the open-source examples as a template. This mode should provide a faster on-ramp for application developers to develop a standards-compliant product.

4. DMS vendors can use the platform to develop and test their own standards-compliant interfaces. Any GridAPPS-D code may be incorporated into a commercial product, pursuant to its BSD license terms. The goal is for an application to be deployable from one platform to another, simply by moving the program file(s) and updating local configuration files.

CHAPTER

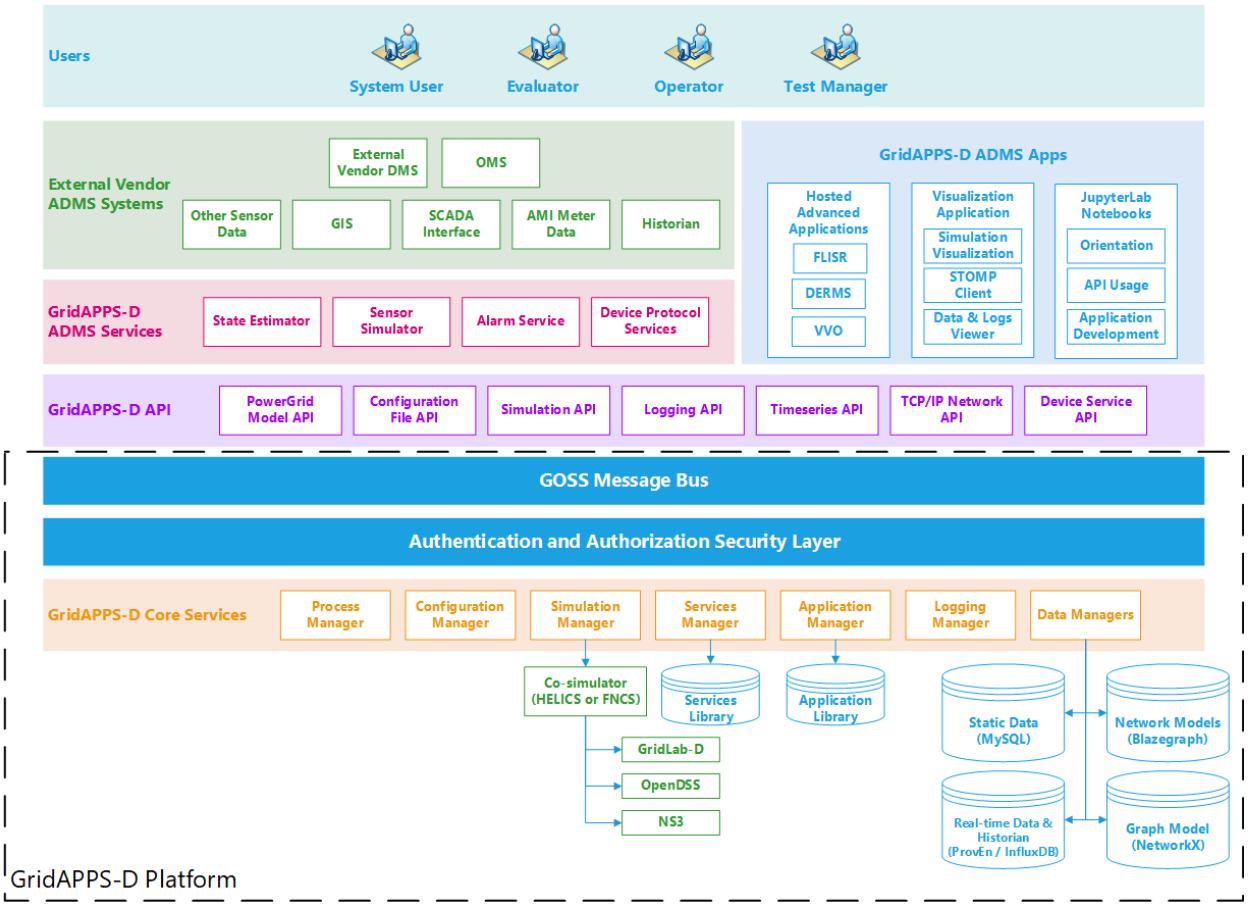
FIVE

# GRIDAPPS-D ARCHITECTURE

## 5.1 1. GridAPPS-D Architecture

GridAPPS-D offers a standards-based, open-source platform that enables rapid integration of advanced applications and services through a robust application programming interface (API).

The architecture of the development ecosystem is illustrated below.



## 5.2 2. GridAPPS-D User Roles

The GridAPPS-D platform contains several user roles with different permissions.

[[GET DESCRIPTION OF USER ROLES AND PERMISSION FROM TARA]]

- System
  - This role is used by XXX to do XXX
  - Permission inlcude
    - *
    - *
    - *
- Evaluator
  - This role is used by XXX to do XXX
  - Permission inlcude
    - *
    - *
    - *
- Operator
  - This role is used by XXX to do
  - Permission inlcude
    - *
    - *
    - *
- Test Manager
  - This role is used by XXX to do
  - Permission inlcude
    - *
    - *
    - *

## 5.3 3. Integration with External Vendor Systems

External vendor systems are able to interface with GridAPPS-D compliant applications and services through two means.

The first is direct integration through the standards-based API and message bus. This enables products that comply with the GridAPPS-D™ platform to * reduce utility time and cost to integrate new functionality * give utilities more choice in technology providers * scale up or down for any size utility * expand market opportunities for developers and vendors

The second method is through the standards-based services, such as the DNP3 service, IEEE 2030.5 service, etc. that enable communication between GridAPPS-D compliant applications and external vendor systems through SCADA and other control center protocols.

---

## 5.4 4. GridAPPS-D Applications

The GridAPPS-D platform and API enable rapid development of advanced power applications that are able to operate in a real-time environment and interface with external software and systems. Multiple power applications have already been developed on the platform, including

- Volt-Var Optimization (VVO)
- Fault Location Isolation and Service Restoration (FLISR)
- Distributed Energy Resource Dispatch and Management (DERMS)
- Solar Forecasting, Load Forecasting, etc.
- and more

Applications can be containerized in Docker for direct integration into the platform or interface through the API. Applications can be written in any programming language, but API libraries are currently available in only Python and Java.

---

## 5.5 5. GridAPPS-D Services

The GridAPPS-D platform can host a multitude of services for processing both real-time simulation and control center data. These services can be called by any application through the GridAPPS-D API.

Some of the available services include

- **State Estimator**
- **Sensor Simulator**
- **Alarm Service**
- **DNP3 Protocol Service**
- **IEEE 2030.5 Protocol Service**

---

## 5.6 6. GridAPPS-D Application Programming Interface

GridAPPS-D offers a unique standards-based application programming interface (API) that will be the focus of the lessons in this set of tutorials. The API enables any application, service, or external vendor product to interface with each other, access control center data, run a real-time simulation, and issue equipment control commands.

GridAPPS-D has several APIs to serve different needs and objectives, inlcuding * **Powergrid Models API** – Allows apps and services to access the power system model data * **Configuration File API** – Allows apps to set equipment statuses and system conditions * **Simulation API** – Allows apps to start a real-time simulation and issue equipment commands * **Timeseries API** – Allows apps to pull real-time and historical data * **Logging API** – Allows apps to access logs and publish log messages

Additional APIs are currently under development to enable communication and control of field devices, as well as cyber-physical network co-simulation.

---

## 5.7 7. GOSS Message Bus

One of the unique features of GridAPPS-D is the GOSS Message Bus, which enables integration and communication between applications, services, and external software on a publish-subscribe basis.

The GridAPPS-D platform publishes SCADA and simulation data, alarms, and other real-time data. Applications subscribe to the types of messages relevant to their objectives and publish equipment commands and control settings.

---

## 5.8 8. GridAPPS-D Core Services

"Under the hood" of the GridAPPS-D platform are the core services and managers.

An application developer should not need a detailed understanding of the core services, as all interaction is performed through the various APIs, which will be dicussed in detail in the upcoming tutorial lessons.

The core services provide the key functionality offered by the GridAPPS-D platform, inlcuding database access, processing API calls, handling equipment commands, and running simulations.

Some of the core services included in the GridAPPS-D platform are * **Platform Manager** – Coordinates all of the other managers * **Process Manager** – Coordinates platform component interactions * **Application Manager** – Manages application registration, execution, and status reporting * **Configuration Manager** – Manages the setup and configuration of real-time simulations * **Simulation Manager** – Allows users and apps to create, start, stop, and pause co-simulations * **Data Manager** – Coordinates the integrated repository of model, timeseries data, and metadata * **Model Manager** – Loads and checks CIM-based power system models * **Logging Manager** – Supports logging for application development and execution * **Services Manager** – Coordinates all services available for users and apps * **Test Manager** – Enables creation of simulation events, faults, and network outages

---

## 5.9 9. Co-Simulation Framework

The co-simulation framework serves as the simulation context for the rest of GridAPPS-D. When a simulation is requested through the GridAPPS-D plaform the simulation manager instantiates a FNCS or HELICS co-simulation federation consisting of two applications. The first application is a powerflow simulator which can be either GridLAB-D or OpenDSS that simulates real world distribution feeder or feeders. The second is a custom application that serves as bridge between the FNCS/HELICS message bus and the GOSS message bus. The data that travels between the co-simulation federation and the rest of the platform are SCADA measurement, SCADA control, and simulation status and control messages.The bridge application subscribes to the simulation input topic to recieve any SCADA control, simulation control, and simulation event messages. The bridge forwards SCADA control commands and simulation events like faults and outages to the powerflow simulator. The bridge publishes SCADA measurements from the powerflow simulator on a simulation output topic that GridAPPS-D applications and other parts of the GridAPPS-D platform subscribe to.

---

## 5.10 10. Database Structures

Default installation of GridAPPS-D comes with following data stores:

- **MySQL:** It is used to store log data from platform, applications and services. For details, please see Logging API, which is covered in detail in Lesson 2.7.
- **Blazegraph:** It is used to store power grid model data. The data contains equipments, properties and their initial measurement values. It is a triplestore that supports complex graph representation and class structure for CIM standard data model.
- **InfluxDB:** InfluxDB is a time series data store and is used to store simulation output, simulation input, weather and load data. It also store output from services line sensor service and alarms service. For the purposes of the GridAPPS-D project, InfluxDB is managed by Proven. Proven is a database software suite supporting disclosure, collection, and management of modeling and simulation data.

For the purpose of developing applications, the data stores used should be transparent to the application as long the data model and standardized API is used.

---

*Return to Top*

---

CHAPTER

SIX

# GRIDAPPS-D PYTHON LIBRARY

## 6.1 Getting Started

Before running any of the sample routines in this tutorial, it is first necessary to start the GridAPPS-D Platform and establish a connection to this notebook so that we can start passing calls to the API.

*Open the Ubuntu terminal and start the GridAPPS-D Platform if it is not running already:*

```
cd gridappsd-docker
```

~/gridappsd-docker$ `./run.sh -t develop`

*Once containers are running,*

gridappsd@[container]:/gridappsd$ `./run-gridappsd.sh`

---

## 6.2 1. A First Course in GridAPPSD-Python

GridAPPSD-Python is a Python library that can wrap API calls and pass them to the various GridAPPS-D APIs through the GOSS Message Bus

The library has numerous shortcuts to help you develop applications faster and interface them with other applications, services, and GridAPPS-D compatible software packages.

*Return to Top*

---

---

## 6.3 2. Building Blocks of an Application

This section is going to provide an overview of some of the key building blocks of a GridAPPS-D application.

### 6.3.1 2.1. Import Required Python Libraries

The first step is to import the required libraries.

Below is a list of some of the additional libraries that you may need to import.

You may not need all of these additional libraries, depending on the needs of your application

- `argparse` – This is the recommended command-line parsing module in Python.(Online Documentation)
- `json` – Encoder and decoder for JavaScript Object Notation (JSON). (Online Documentation)
- `logging` – This module defines classes and functions for event logging. (Online Documentation
- `sys` – Python module for system specific parameters. (Online Documentation)

- `time` – Time access and conversions. (Online Documentation)
- `pytz` – Library to enable resolution of cross-platform time zones and ambiguous times. (Online Documentation
- `stomp` – Python client for accessing messaging servers using the Simple Text Oriented Messaging Protocol (STOMP). (Online Documentation)

[ ]:
```
import argparse
import json
import logging
import sys
import time
import pytz
import stomp
```

---

*Return to Top*

### 6.3.2 2.2. Import Required GridAPPS-D Libraries

The GridAPPS-Python API contains several libraries, which are used to query for information, subscribe to measurements, and publish commands to the GOSS message bus. These inlcude

`GridAPPSD` – This is primary library that contains numerous methods and tools that will be dicussed in detail in the subsequent lessons.

`utils` – A set of utilities to assist with common commands, inlcuding

Function Call |

Usage ——————|———

*utils.validate_gridappsd_uri()* |

Checks if GridAPPS-D is hosted on the correct port

*utils.get_gridappsd_address()* |

Returns the platform address such that response can be passed directly to a socket or the STOMP library

*utils.get_gridappsd_user()* |

Returns the login username

*utils.get_gridappsd_pass()* |

Returns the login password

*utils.get_gridappsd_application_id()* |

Only applicable if the environment variable 'GRIDAPPSD_APPLICATION_ID' has been set

utils.get_gridappsd_simulation_id()_ |

retrieves the simulation id from the environment.

[ ]:
```
from gridappsd import GridAPPSD, utils
```

---

### 6.3.3 2.3. Establish a Connection to the GridAPPS-D Platform

The next step is to establish a connection with the GridAPPS-D platform so that API calls can be passed and processed.

This can be done by 1) manually specifying the connection and port or 2) using the GridAPPS-D utils to automatically determine the port

#### Option 1: Manually specify connection parameters

By default, the GridAPPS-D API communicates with the platform on port 61613.

```
[ ]: gapps = GridAPPSD("('localhost', 61613)", username='system', password='manager')
```

#### Option 2: Use GridAPPS-D utils to determine connection

The GridAPPS-D utils include several functions to automatically determine the location of the platform and security credentials for passing API commands

```
[ ]: gapps = GridAPPSD(address=utils.get_gridappsd_address(),
                       username=utils.get_gridappsd_user(), password=utils.get_gridappsd_pass())
```

*Return to Top*

---

### 6.3.4 2.4. Pass a Simple API Call

There are three generic API call routines:

- *send(self, topic, message)* –
- *get_response(self, topic, message)* –
- *subscribe(self, topic, callback)* –

For this example, we will use a very short query to request the MRIDs of the models available in the GridAPPS-D Platform. We will explain how to make various kinds of queries in the upcoming lessons on how to use each API.

The first step is to define the topic, which specifies the channel on which to communicate with the API. The concept of the GridAPPS-D Topic will be introduced in the next lesson. The specific topic definitions and their purposes will be discussed in greater detail in the lessons on each GridAPPS-D API.

```
[ ]: topic = "goss.gridappsd.process.request.data.powergridmodel"
```

Next, we need to create the message the will be passed. The message must be a valid Python Dictionary or JSON-formated string. The way a message is created, structured, formatted, and parsed is discussed in detail in

If it is a short query, we can write it as a single line.

```
[ ]: message = {"requestType": "QUERY_MODEL_NAMES", "resultFormat": "JSON"}
```

If it is a long query, we can break up the lines of the python dictionary object to improve readability:

```
message = { "key1": "value1", "key2": "value2" }
```

[ ]:
```
message = {
    "requestType": "QUERY_MODEL_NAMES",
    "resultFormat": "JSON"
}
```

The GridAPPSD-Python Library then wraps that string and passes it as a message to the API through the GOSS Message Bus.

[ ]:
```
gapps.get_response(topic, message)
```

*Return to Top*

---

---

CHAPTER

SEVEN

# GRIDAPPS-D APPLICATION STRUCTURE

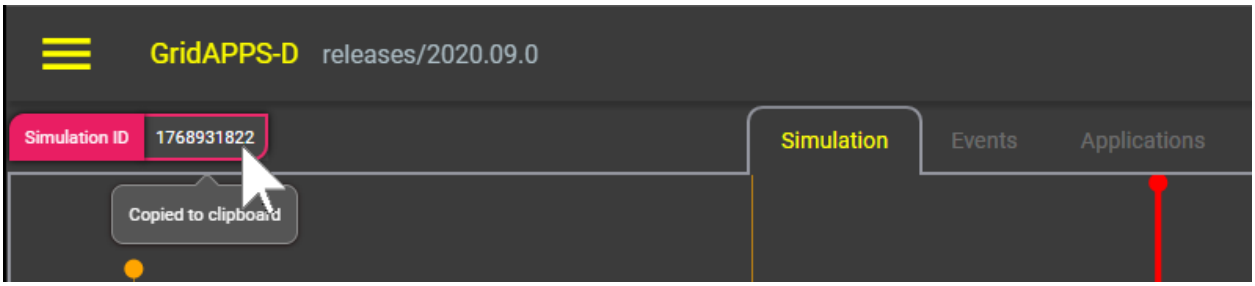
## 7.1 1. Application Structure

**2) Start a simulation in the GridAPPS-D Viz:**

The Simulation API calls covered in this lesson need to be passed to an active simulation. For the purposes of this tutorial, we will use the GridAPPS-D Viz at localhost:8080 to start a simulation of the IEEE 123 Node model with a 3600 sec simulation time.

The steps for starting a simulation were covered in Lesson 2.5, Section 3.

After starting the simulation, paste the simulation_id into the code block below by clicking on the simulation_id. This will automatically copy the simulation_id to your computer's clipboard.




When your application is containerized in Docker and registered with the GridAPPS-D Platform using the docker-compose file, the simulation_id and feeder model mRID as passed as part of the application start call. For this notebook, that information needs to be copied and pasted into the first code block below.

[ ]:
```
# Import GridAPPSD-Python Library:
from gridappsd import GridAPPSD

# Paste Simulation ID into this variable:
viz_simulation_id = "1093527122"

# Simulation running on IEEE 123 node model:
model_mrid = "_C1C3E687-6FFD-C753-582B-632A27E28507"

# Establish connection to GridAPPS-D Platform:
gapps = GridAPPSD(viz_simulation_id, "('localhost', 61613)", username='system', password=
↪'manager')
assert gapps.connected
```

```
[ ]: # Set environment variables - when developing, put environment variable in ~/.bashrc␣
     ↪file or export in command line
     # export GRIDAPPSD_USER=system
     # export GRIDAPPSD_PASS=manager

     import os # Set username a
     os.environ['GRIDAPPSD_USER'] = 'tutorial_user'
     os.environ['GRIDAPPSD_PASS'] = '12345!'

     # Connect to GridAPPS-D Platform
     gapps = GridAPPSD(viz_simulation_id)
     assert gapps.connected
```

## 7.2 2. Querying for the Power System Model

The first portion of a GridAPPS-D application is series of queries to the PowerGrid Models API to obtain information about the power system model.

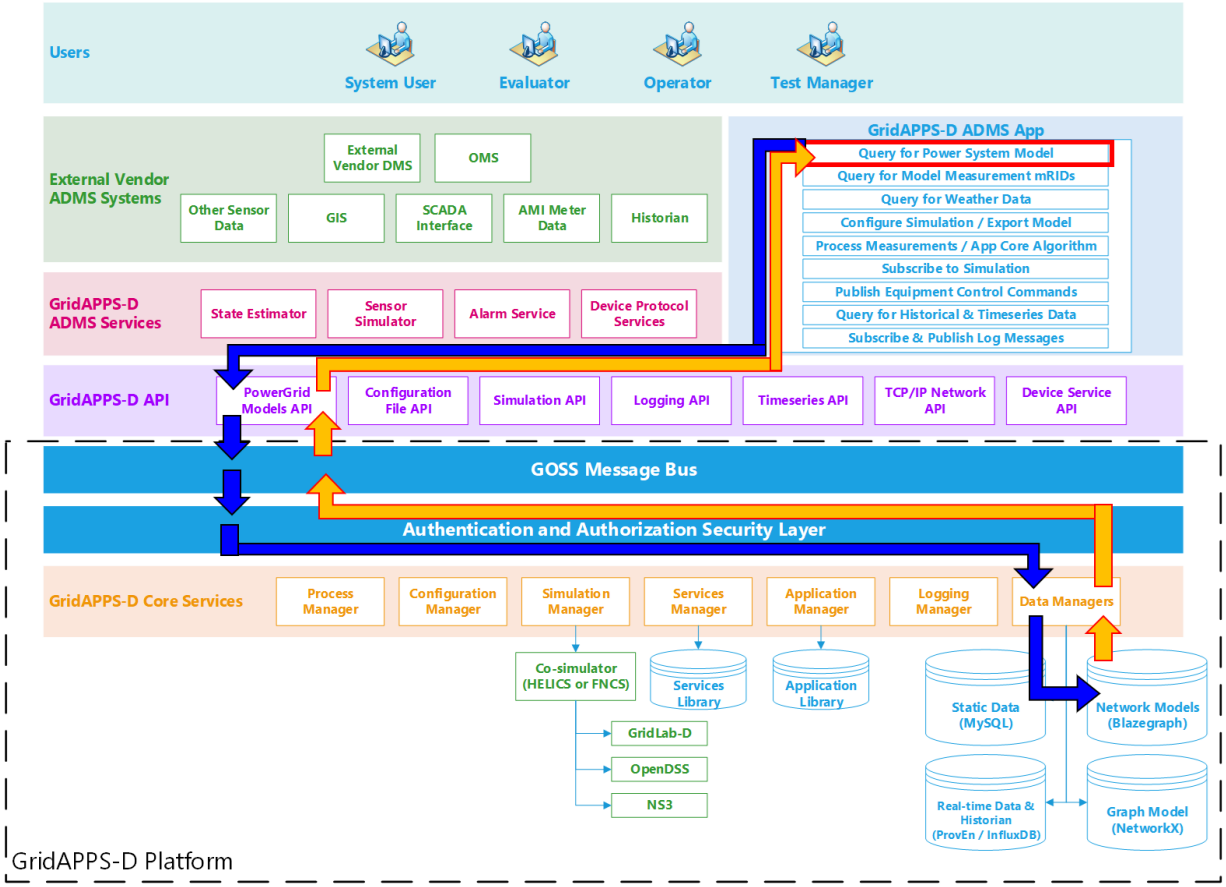
Because GridAPPS-D applications are designed to be portable across numerous power system models without any code modification, the application must query the Blazegraph database and create a set of local variables that contain the information needed by the app to run its internal code.

An application will query for the various pieces of power system equipment relevant to its objective (e.g. a VVO app will be interested in regulators and capacitors, while a FLISR app will be interested in switches and reclosers present in the model). The query will typically include requests for information about the names, location, mRIDS, and electrical parameters for the various pieces of equipment needed by the application..

### 7.2.1 2.1. Information flow

The figure below shows the information flow involved in making a query for the power system model.

The query is sent using `gapps.get_response(topic, message)` on a queue channel (explained in API Communication Channels) with a response expected back from the platform within the specified timeout period.




**Application passes query to GridAPPS-D Platform**

First, the application creates a query message for requesting information about the desired power system components in the format of a JSON string or equivalant Python dictionary object. The syntax of this message is explained in detail in Using the PowerGrid Models API.

The application then passes the query through the PowerGrid Models API to the GridAPPS-D Platform, which publishes it to a queue channel on the GOSS Message Bus. If the app is authenticated and authorized to pass queries, the query message is delivered to the data managers, which obtain the desired information from the Blazegraph Database.

**GridAPPS-D Platform responds to Application query**

The data managers then publish the response from the Blazegraph Database to the appropriate queue channel. The PowerGrid Models API then returns the desired information back to the application as a JSON message or equivalant Python dictionary object.

### 7.2.2 2.2. Sample App code

Below is a sample query of how the application will use the PowerGrid Models API to query for the details associated for all the switches in the feeder.

```
[ ]: from gridappsd import topics as t

message = {
    "modelId": model_mrid,
    "requestType": "QUERY_OBJECT_DICT",
```

(continues on next page)

(continued from previous page)

```
    "resultFormat": "JSON",
    "objectType": "LoadBreakSwitch"
}

response_obj = gapps.get_response(t.REQUEST_POWERGRID_DATA, message)
switch_dict = response_obj["data"]

# Filter to get mRID for switch SW2:
for index in switch_dict:
    if index["IdentifiedObject.name"] == 'sw2':
        sw_mrid = index["IdentifiedObject.mRID"]

print(switch_dict[0]) # Print dictionary for first switch

print('mRID of sw2 is ',sw_mrid)
```

## 7.3 3. Querying for Measurement mRIDs

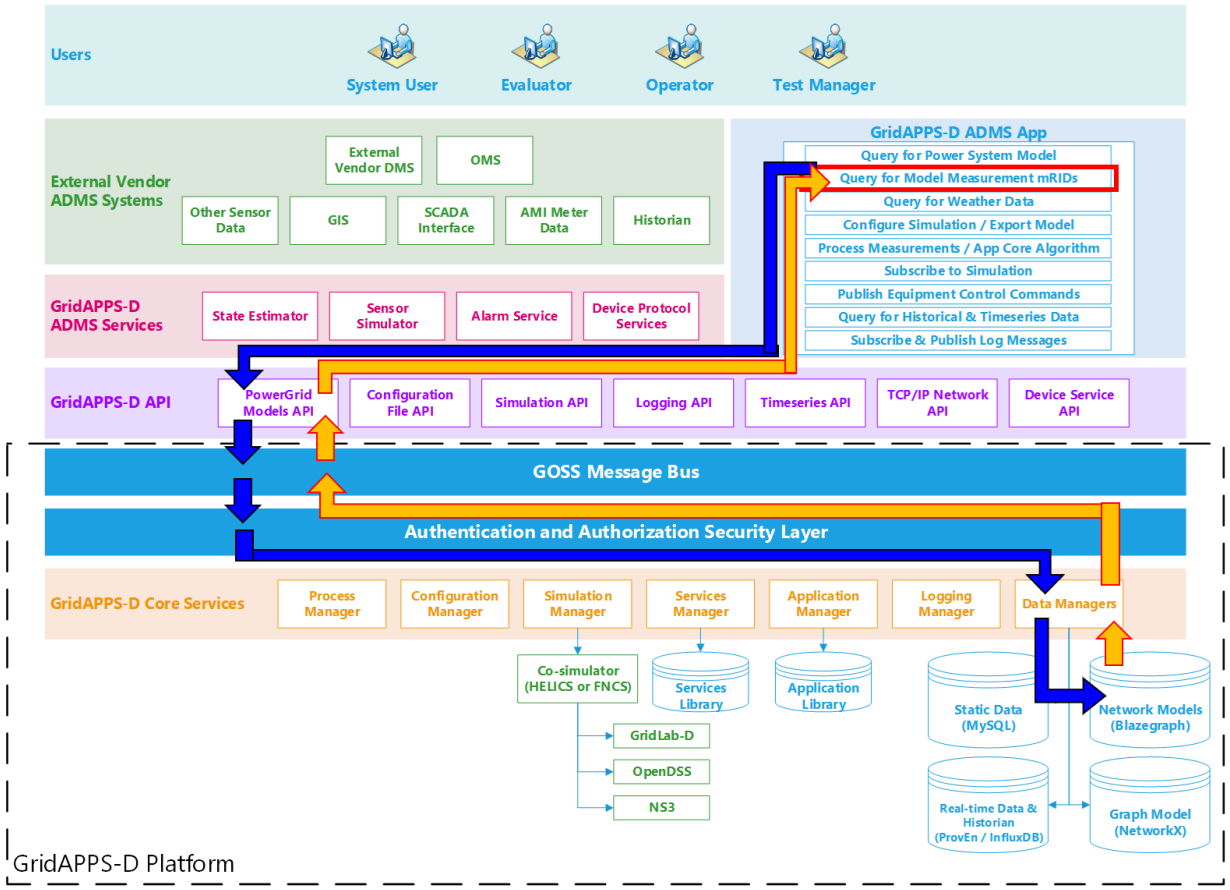
The next portion of a GridAPPS-D application is series of queries to the PowerGrid Models API to obtain information about the measurements associated with various pieces of equipment the application is interested in. Due to structure of the Common Information Model (introduced in Lesson 2.6), there exist a separate set of objects associated with the positive-neutral-voltage (PNV), volt-ampere (VA), and position measurements (POS) for each line, transformer, switch, etc.

Because GridAPPS-D applications are designed to be portable across numerous power system models without any code modification, the application must query the Blazegraph Database and create a set of local variables that contain the unique mRIDS of each measurement needed by the app to run its internal code. In a subsequent step, the app will use these measurement mRIDs to subscribe to the live streaming data issued by the simulation.

### 7.3.1 3.1. Information Flow

The figure below shows the information flow involved in making a query for the power system model.

The query is sent using `gapps.get_response(topic, message)` on a queue channel (explained in Lesson 3.1) with a response expected back from the platform within the specified timeout period.




The figure below shows the information flow involved in making a query for the power system model.

**Application passes query to GridAPPS-D Platform**

First, the application creates a query message for requesting information about the desired power system components in the format of a JSON string or equivalant Python dictionary object. The syntax of this message is explained in detail in Lesson 3.3.

The application then passes the query through the PowerGrid Models API to the GridAPPS-D Platform, which publishes it to a queue channel on the GOSS Message Bus. If the app is authenticated and authorized to pass queries, the query message is delivered to the data managers, which obtain the desired information from the Blazegraph Database.

**GridAPPS-D Platform responds to Application query**

The data managers then publish the response from the Blazegraph Database to the appropriate queue channel. The PowerGrid Models API then returns the desired information back to the application as a JSON message or equivalant Python dictionary object.

Below is a sample query of how the application will use the PowerGrid Models API to query for the measurement mRIDs of all switches in the power system model

[ ]:
```
message = {
    "modelId": model_mrid,
    "requestType": "QUERY_OBJECT_MEASUREMENTS",
    "resultFormat": "JSON",
    "objectType": "LoadBreakSwitch"
}
```

(continues on next page)

(continued from previous page)

```
response_obj = gapps.get_response(t.REQUEST_POWERGRID_DATA, message) # Pass query to␣
↪PowerGrid Models API
measurements_obj = response_obj["data"]

global Pos_obj # Define global python dictionary of position measurements
Pos_obj = [k for k in measurements_obj if k['type'] == 'Pos'] # Filter measurements to␣
↪just switch positions

print(Pos_obj[0]) # Print switch position measurement mRID for first switch
```

## 7.4 4. Querying for Weather Data

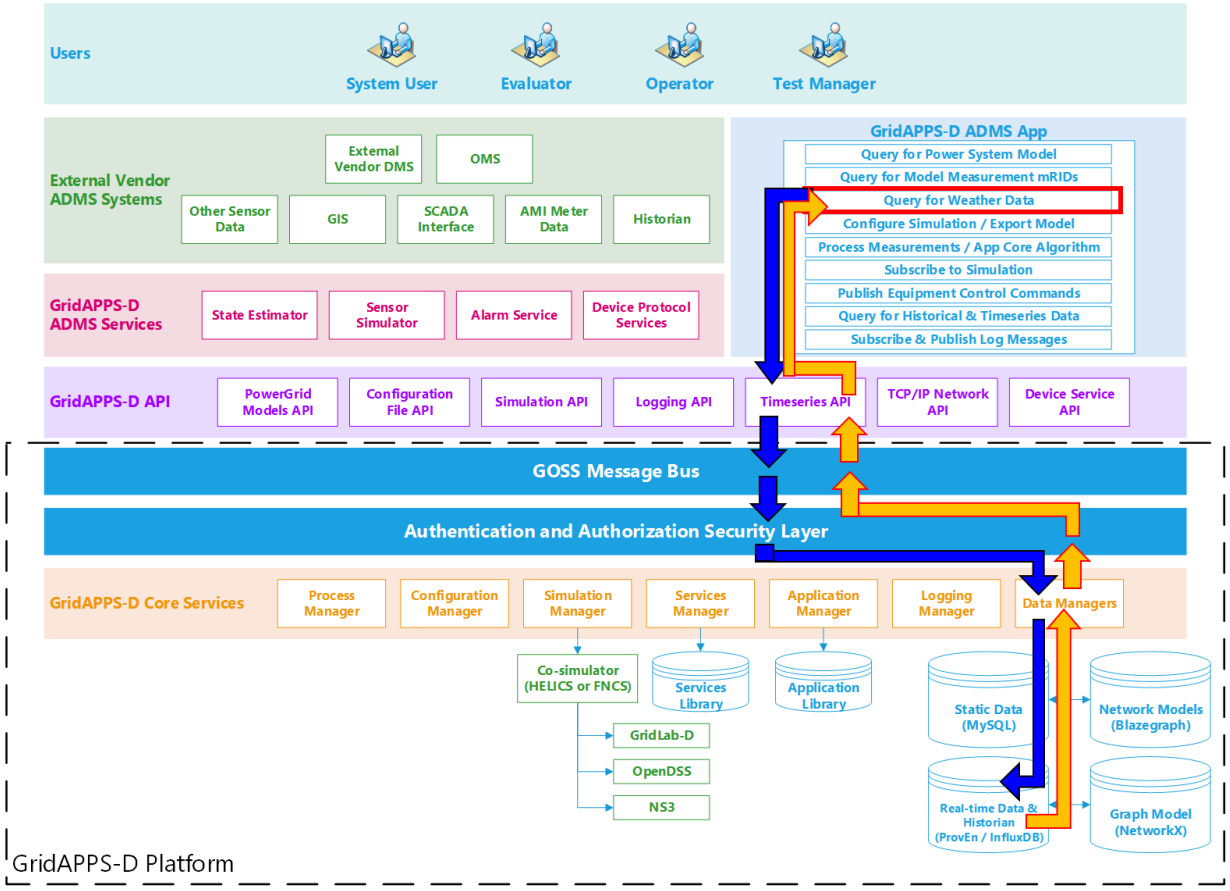
The next portion of a GridAPPS-D application is series of queries to the Timeseries API to obtain information about the weather data for the current time, including irradiation, temperature, etc. This information can be used for solar forecasting, load forecasting, etc.

Because GridAPPS-D applications are designed to be portable across numerous power system models without any code modification, the application must query the Timeseries Influx Database and create a set of local variables that contain the weather data needed by the app to run its internal code.

### 7.4.1 4.1. Information Flow

The figure below shows the information flow involved in making a query for the power system model.

The query is sent using `gapps.get_response(topic, message)` on the Timeseries queue channel (explained in Lesson 3.1) with a response expected back from the platform within the specified timeout period.




**Application passes query to GridAPPS-D Platform**

First, the application creates a query message for requesting information about the desired power system components in the format of a JSON string or equivalant Python dictionary object. The syntax of this message is explained in detail in Lesson 3.7.

The application then passes the query through the Timeseries API to the GridAPPS-D Platform, which publishes it to a queue channel on the GOSS Message Bus. If the app is authenticated and authorized to pass queries, the query message is delivered to the Data Managers, which obtain the desired information from the Timeseries Influx Database.

**GridAPPS-D Platform responds to Application query**

The Data Managers then publish the response from the Timeseries Influx Database to the appropriate queue channel. The Timeseries API then returns the desired information back to the application as a JSON message or equivalant Python dictionary object.

### 7.4.2 4.2. Sample App Code

Below is a sample query to the Timeseries API requesting all weather data between a certain startTime and endTime (given in unix absolute time). The application can then use that weather data to feed its internal forecasting algorithm.

```
[ ]: # Use queryFilter of "startTime" and "endTime"
message = {
    "queryMeasurement":"weather",
    "queryFilter":{"startTime":"1357048800000000",
                   "endTime":"1357048860000000"},
```

(continues on next page)

(continued from previous page)

```
    "responseFormat":"JSON"
}

response_obj = gapps.get_response(t.TIMESERIES, message) # Pass query to Timeseries API
weather_obj = response_obj["data"]

print(weather_obj[1]) # Print first line of weather data
```

## 7.5 5. Configuring a Parallel Simulation

Some applications may choose to run parallel simulations (similar to a digital twin), either within the GridAPPS-D platform or by exporting the model to OpenDSS, GridLAB-D, etc. This is accomplished through one or more queries to the Configuration File API to create a simulation configuration file and/or exported power system model.

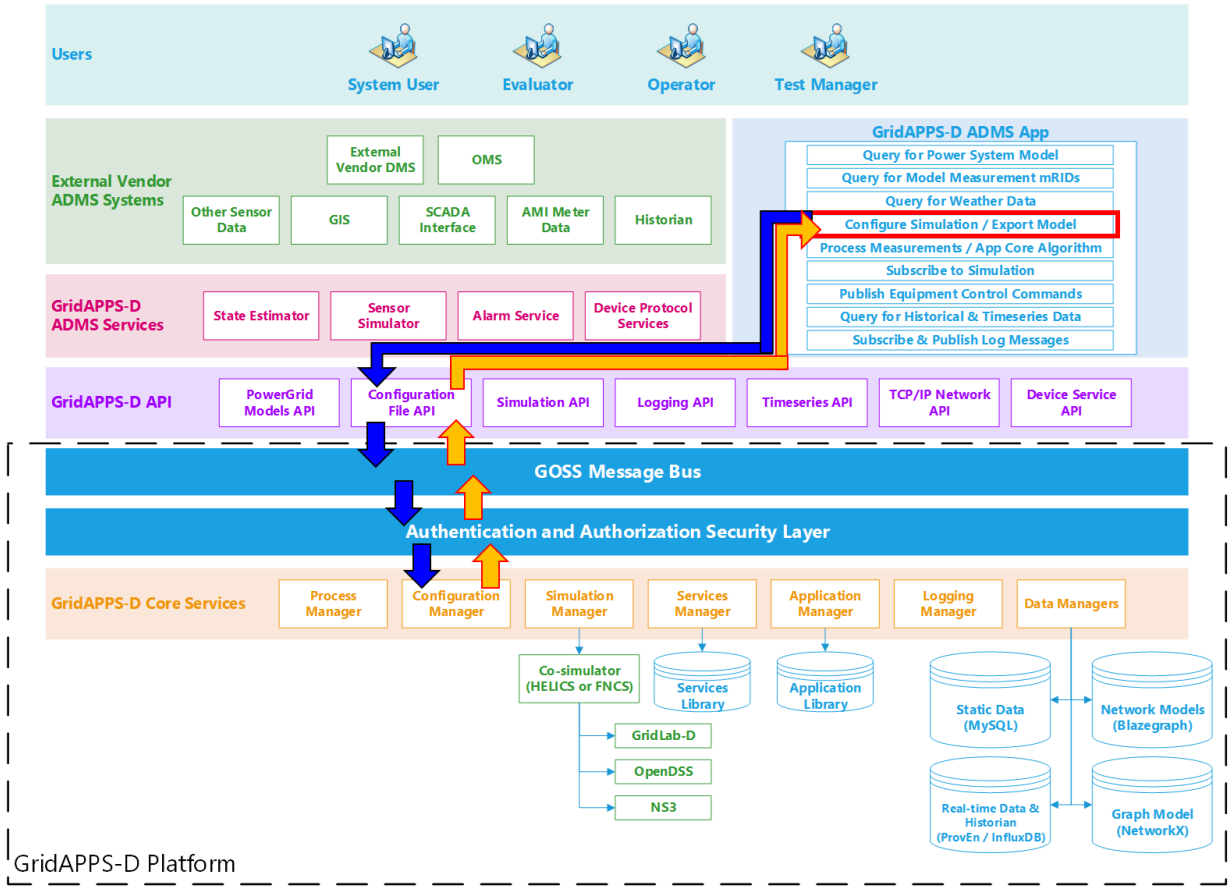
The simulation configuration file contains all the necessary info to create a new simulation, including the power system model, date/time, and variations from the default basecase (i.e. re-dispatched DERs and switches that have been opened/closed).

The exported power system model is the entire model as a set of GLM or DSS that can be saved to an external file and then solved with a different power flow solver outside of the GridAPPS-D Platform.

### 7.5.1 5.1. Information Flow

The figure below shows the information flow involved in making a query for the power system model.

The query is sent using `gapps.get_response(topic, message)` on the Configuration File queue channel (explained in Lesson 3.1) with a response expected back from the platform within the specified timeout period.




**Application passes query to GridAPPS-D Platform**

First, the application creates a query message for requesting information about the desired power system configuration in the format of a JSON string or equivalant Python dictionary object. The syntax of this message is explained in detail in Using the Configuration File API

The application then passes the query through the Configuration File API to the GridAPPS-D Platform, which publishes it to a queue channel on the GOSS Message Bus. If the app is authenticated and authorized to pass queries, the query message is delivered to the Configuration Manager.

**GridAPPS-D Platform responds to Application query**

The Configuration Manager obtains the CIM XML file for the desired power system model and then converts it to the desired output format with all of the requested changes to the model. The Configuration File API then returns the desired information back to the application as a JSON message (for Y-Bus or partial models) or export the files to the directory specified in the

### 7.5.2 5.2. Sample App Code

Below is a sample query showing how an application would make a query through the Configuration File API to change all loads to constant current loads, convert the power system model to a set of OpenDSS files, and export them to the directory /tmp/dsssimulation.

[ ]:
```
topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "DSS All",
    "parameters": {
        "directory": "/tmp/dsssimulation/",
        "model_id": model_mrid,
        "simulation_id": "12345678",
        "simulation_name": "ieee13",
        "simulation_start_time": "1518958800",
        "simulation_duration": "60",
        "simulation_broker_host": "localhost",
        "simulation_broker_port": "61616",
        "schedule_name": "ieeezipload",
        "load_scaling_factor": "1.0",
        "z_fraction": "0.0",
        "i_fraction": "1.0",
        "p_fraction": "0.0",
        "solver_method": "NR" }
}

gapps.get_response(topic, message)
```

## 7.6 6. Processing Measurements & App Core Algorithm

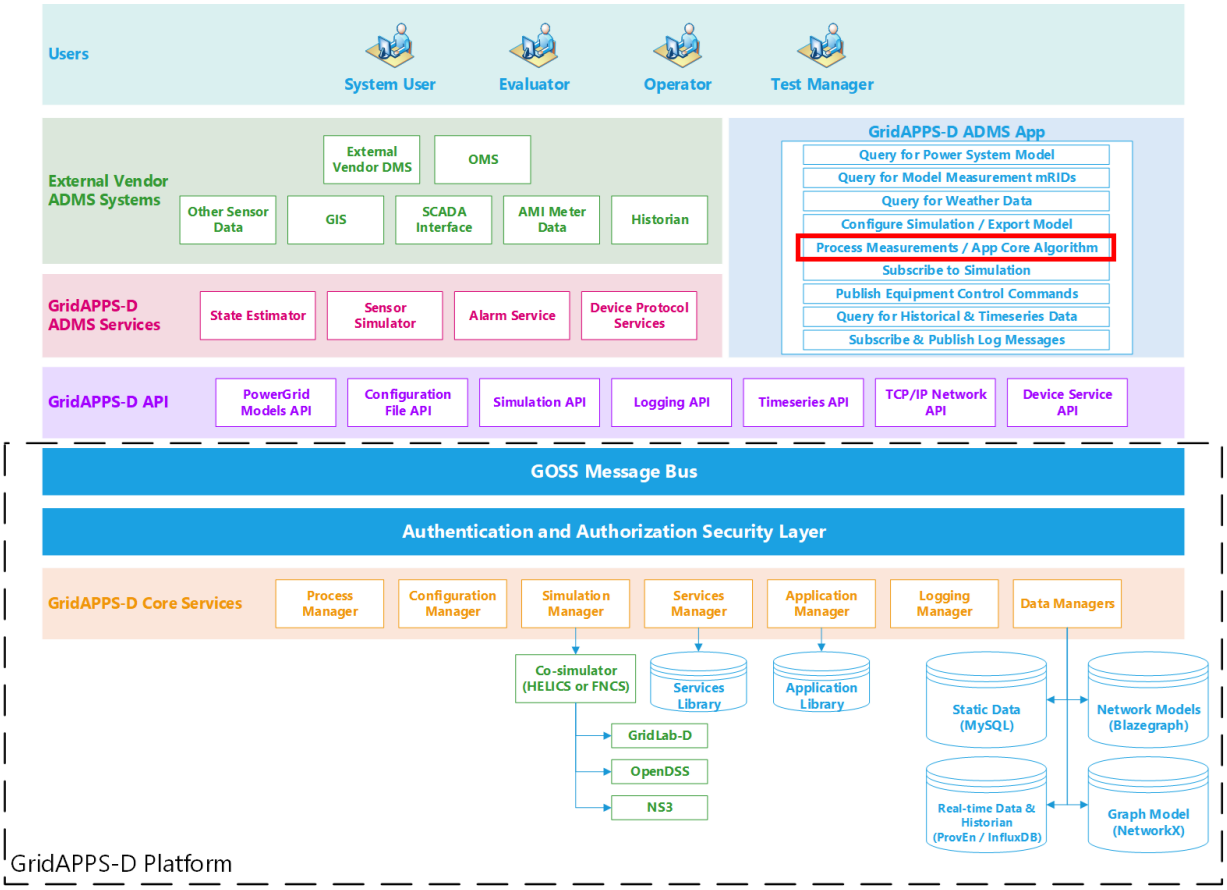
The next portion of a GridAPPS-D application is the measurement processing and core algorithm section. This section is built as either a class or function definition with prescribed arguments. Each has its advantages and disadvantages:

- The function-based approach is simpler and easier to implement. However, any parameters obtained from other APIs or methods to be used inside the function currently need to be defined as global variables.
- The class-based approach is more complex, but also more powerful. It provides greater flexibility in creating additional methods, arguments, etc.

### 7.6.1 6.1 Information Flow

This portion of the application does not communicate directly with the GridAPPS-D platform.

Instead, the next part of the GridAPPS-D application (*Subscribing to Simulation Uuptut*) delivers the simulated SCADA measurement data to the core algorithm function / class definition. The core algorithm processes the data to extract the desired measurements and run its optimization / control agorithm.




**No message from core algorithm to GridAPPS-D Platform**

The core algorithm does not send any API messages to the platform

**No response to core algorithm from GridAPPS-D Platform**

The core algorithm receives its measurement data and other imputs from the subscription object defined next, rather than directly from the GridAPPS-D platform.

### 7.6.2 6.2. Sample App Code

Below is a very simple core algorithm that determines the number of open switches in the model and prints the result for each simulation timestep. The syntax of the function / class definition is described in detail in

[ ]:
```
def demoSubscription1(header, message):
    # Extract time and measurement values from message
    timestamp = message["message"]["timestamp"]
    meas_value = message["message"]["measurements"]

    meas_mrid = list(meas_value.keys()) #obtain list of all mrid from message

    # Filter to measurements with value of zero
    open_switches = []
    for index in Pos_obj:
        if index["measid"] in meas_value:
            mrid = index["measid"]
```

(continues on next page)

(continued from previous page)

```
            power = meas_value[mrid]
            if power["value"] == 0:
                open_switches.append(index["eqname"])

    # Print message to command line
    print(".............")
    print("Number of open switches at time", timestamp, ' is ', len(set(open_switches)))
```

## 7.7 7. Subscribing to Simulation Output

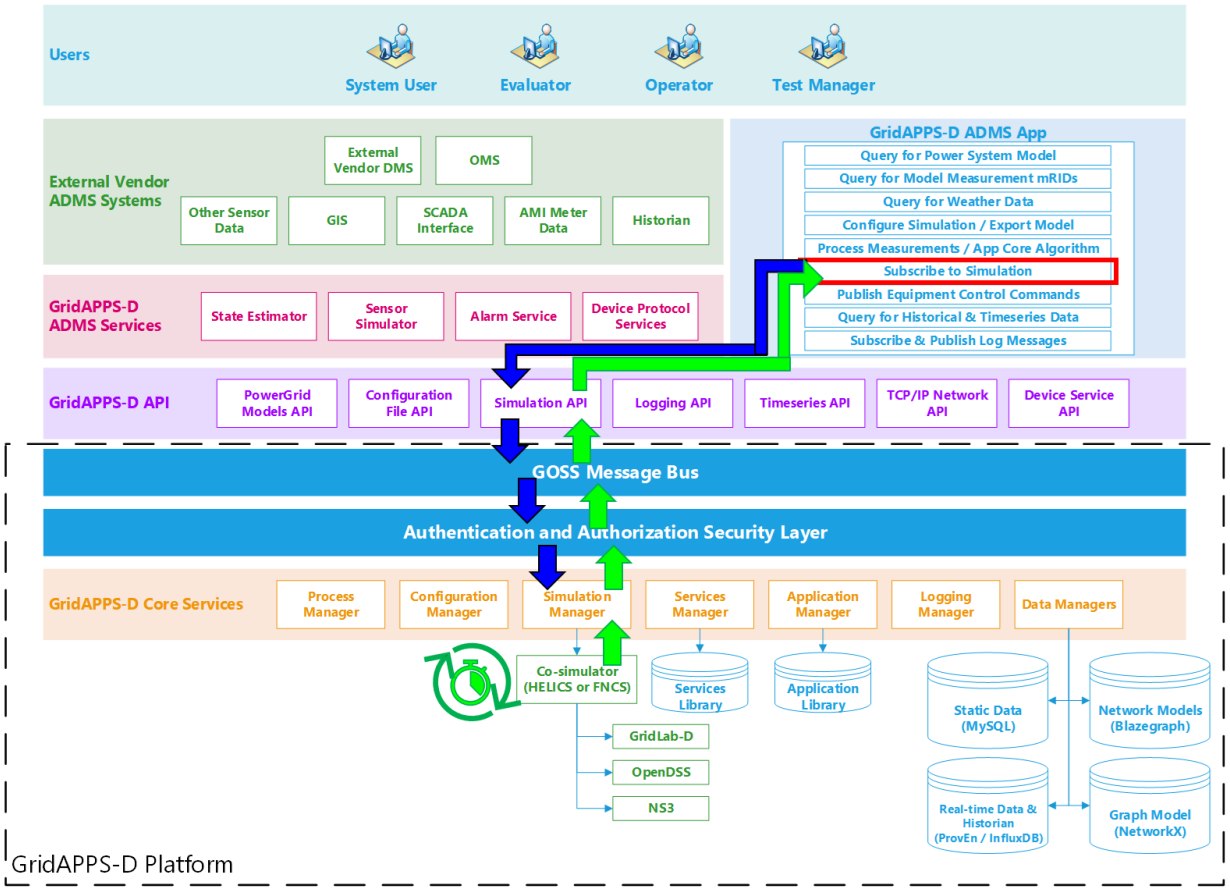
The next portion of a GridAPPS-D application is series of queries to the Timeseries API to obtain information about the weather data for the current time, including irradiation, temperature, etc. This information can be used for solar forecasting, load forecasting, etc.

Because GridAPPS-D applications are designed to be portable across numerous power system models without any code modification, the application must query the Timeseries Influx Database and create a set of local variables that contain the weather data needed by the app to run its internal code.

### 7.7.1 7.1. Information Flow

The figure below shows the information flow involved in subscribing to the simulation output.

The subscription request is sent using `gapps.subscribe(topic, class/function object)` on the specific Simulation topic channel (explained in API Communication Channels). No immediate response is expected back from the platform. However, after the next simulation timestep, the Platform will continue to deliver a complete set of measurements back to the application for each timestep until the end of the simulation.




**Application passes subscription request to GridAPPS-D Platform**

The subscription request is perfromed by passing the app core algorithm function / class definition to the `gapps.subscribe` method. The application then passes the subscription request through the Simulation API to the topic channel for the particular simulation on the GOSS Message Bus. If the application is authorized to access simulation output, the subscription request is delivered to the Simulation Manager.

**GridAPPS-D Platform delivers published simulation output to Application**

Unlike the previous queries made to the various databases, the GridAPPS-D Platform does not provide any immediate response back to the application. Instead, the Simulation Manager will start delivering measurement data back to the application through the Simulation API at each subsequent timestep until the simulation ends or the application unsubscribes. The measurement data is then passed to the core algorithm class / function, where it is processed and used to run the app's optimization / control algorithms.

### 7.7.2 7.2. Sample App Code

Below is an example of how an application subscribes to the GridAPPS-D simulation output using the function or class definition created as part of the *Measurement Processing / App Core*

```
[ ]: from gridappsd.topics import simulation_output_topic

output_topic = simulation_output_topic(viz_simulation_id)

gapps.subscribe(output_topic, demoSubscription1)
```

## 7.8 8. Publishing Equipment Commands

The next portion of a GridAPPS-D App is publsihing equipment control commands based on the optimization results or objectives of the app algorithm.

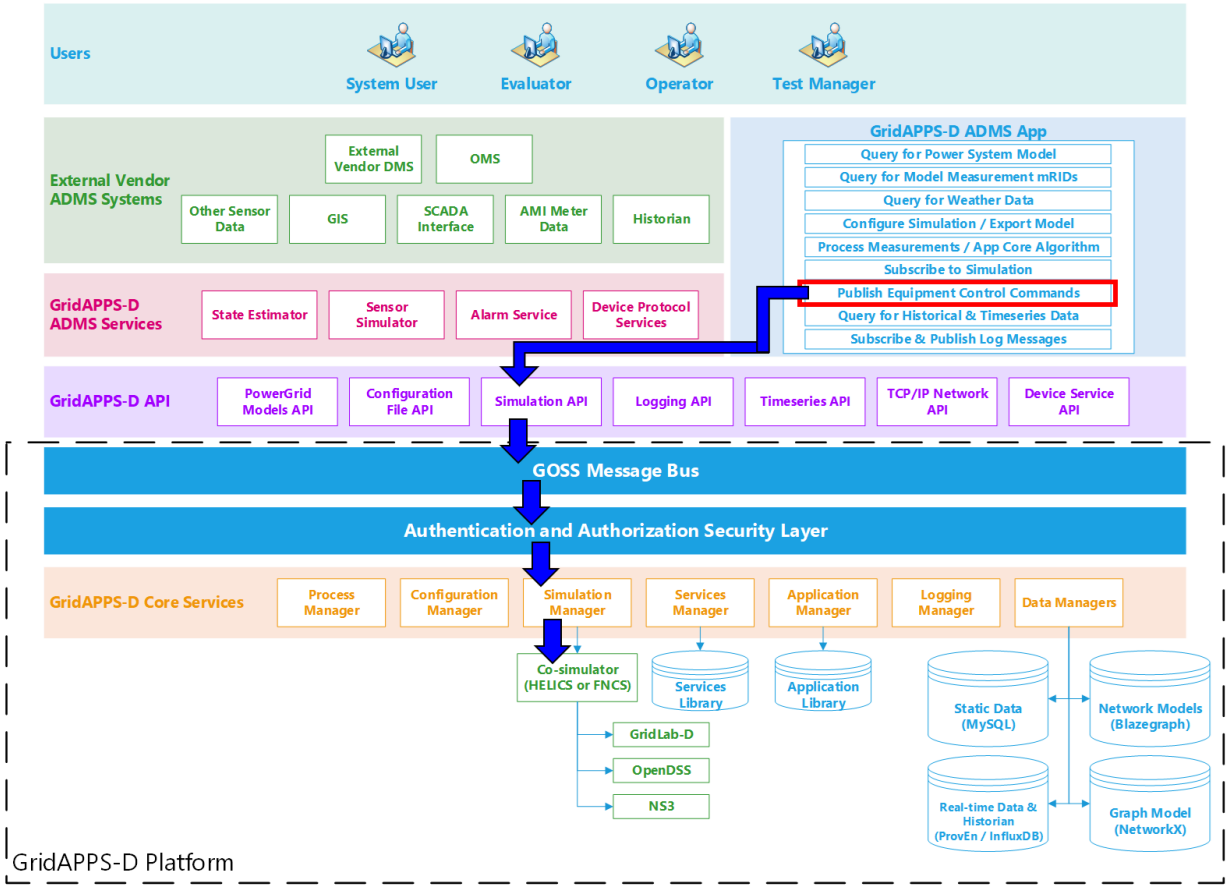
Depending on the preference of the developer, this portion can be a separate function definition, or included as part of the main class definition as part of the *Measurement Processing / App Core* class definition described earlier.

### 7.8.1 8.1. Information Flow

The figure below outlines information flow involved in publishing equipment commands to the simulation input.

Unlike the various queries to the databases in the app sections earlier, equipment control commands are passed to the GridAPPS-D API using the `gapps.send(topic, message)` method. No response is expected from the GridAPPS-D platform.

If the application desires to verify that the equipment control command was received and implemented, it needs to do so by 1) checking for changes in the associated measurements at the next timestep and/or 2) querying the Timeseries Database for historical simulation data associated with the equipment control command.




**Application sends difference message to GridAPPS-D Platform**

First, the application creates a difference message containing the current and desired future control point / state of the particular piece of power system equipment to be controlled. The difference message is a JSON string or equivalant Python dictionary object. The syntax of a difference message is explained in detail in **`Publishing Equipment Commands <>`__**.

The application then passes the query through the Simulation API to the GridAPPS-D Platform, which publishes it on the topic channel for the particular simulation on the GOSS Message Bus. If the app is authenticated and authorized to control equipment, the difference message is delivered to the Simulation Manager. The Simulation Manager then passes the command to the simulation through the Co-Simulation Bridge (either FNCS or HELICS).

**No response from GridAPPS-D Platform back to Application**

The GridAPPS-D Platform does not provide any response back to the application after processing the difference message and implementing the new equipment control setpoint.

### 7.8.2 8.2. Sample App Code

Below is an example of an app code block

[ ]:
```
import time
from gridappsd import DifferenceBuilder
from gridappsd.topics import simulation_input_topic

input_topic = simulation_input_topic(viz_simulation_id)

my_open_diff = DifferenceBuilder(viz_simulation_id)
my_open_diff.add_difference(sw_mrid, "Switch.open", 1, 0) # Open switch given by sw_mrid
open_message = my_open_diff.get_message()

my_close_diff = DifferenceBuilder(viz_simulation_id)
my_close_diff.add_difference(sw_mrid, "Switch.open", 0, 1) # Close switch given by sw_
↪mrid
close_message = my_close_diff.get_message()

while True:
    time.sleep(5)
    gapps.send(input_topic, open_message)
    time.sleep(5)
    gapps.send(input_topic, close_message)
```

### 7.8.3 8.3. Viewing Application Results in GridAPPS-D Viz

Return to the browser tab in which the GridAPPS-D Simulation is currently running. Switch `sw5` will now be opening and closing every 5 seconds, with the downstream portion of the feeder being de-energized and reconnected with each switch operation.

The core application algorithm will also reflect this with the printed response alternating between two and three open switches every few timesteps.

---

## 7.9 9. Querying Historical & Timeseries Data

The next portion of a GridAPPS-D application is querying historical data from the current and/or previous simulations.

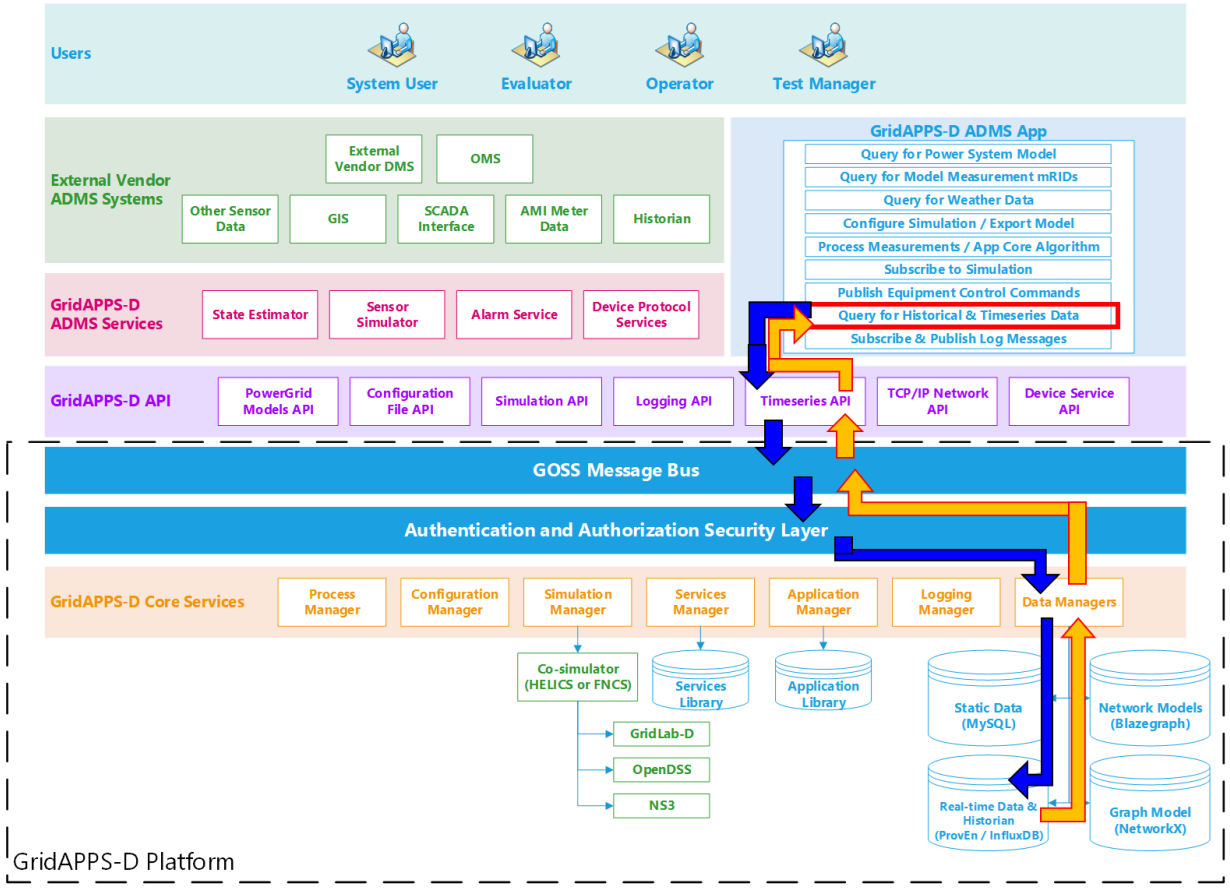
All simulation output and commands from the current and previous simulations are stored in the Timeseries Database, and can be queried to provide AI/ML training data, verify processing of equipment commands, or

Note that Timeseries Database data is cleared when the GridAPPS-D Platform is shut down with the ./stop.sh script. It is recommended to copy historical / training data to an external persistent directory using the `docker cp` command, as given in the [Docker Shortcuts] section.

### 7.9.1 9.1. Information Flow

The figure below outlines the information flow involved in querying for historical and timeseries data.

The query is sent using the `gapps.get_response(topic, message)` method on the Timeseries queue channel with a response expected back from the GridAPPS-D platform within the specified timeout period.




**Application passes query to GridAPPS-D Platform**

First, the application creates a query message for requesting information about the desired power system components in the format of a JSON string or equivalant Python dictionary object. The syntax of this message is explained in detail in Querying Timeseries Data.

The application then passes the query through the Timeseries API to the GridAPPS-D Platform, which publishes it to a queue channel on the GOSS Message Bus. If the app is authenticated and authorized to pass queries, the query message is delivered to the Data Managers, which obtain the desired information from the Timeseries Influx Database.

**GridAPPS-D Platform responds to Application query**

The Data Managers then publish the response from the Timeseries Influx Database to the appropriate queue channel. The Timeseries API then returns the desired information back to the application as a JSON message or equivalant Python dictionary object.

### 7.9.2 9.2. Sample App Code

[ ]:
```
import time

start_time = str(int(time.time())-10) # Start query from 10 sec ago
end_time = str(int(time.time()))

# Query for a particular set of measurments
message = {
    "queryMeasurement": "simulation",
    "queryFilter":{"simulation_id": simulation_id,
                   "startTime": start_time,
                   "endTime": end_time,
                   "measurement_mrid": pos_obj},
    "responseFormat":"JSON"
}

gapps.get_response(t.TIMESERIES, message) # Pass API call
```

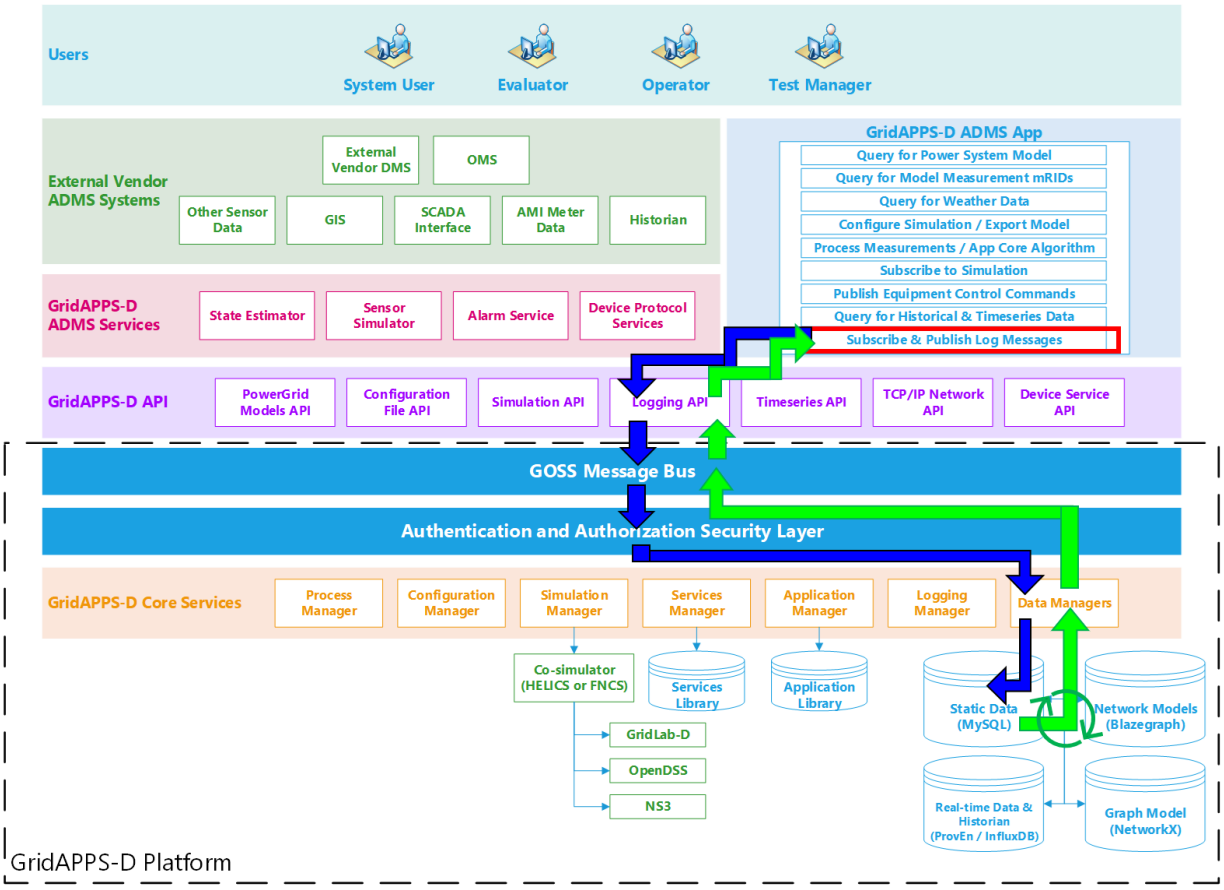
## 7.10 10. Subscribing and Publishing to Logs

The last portion of an application is subscribing and publishing to logs. This step is extremely useful for 1) informing end users of application behavior and 2) application debugging during development and demonstration.

The The GridAPPS-D Logging API provides an extension of the standard Python logging library and enables applications to subscribe to real-time log messages from a simulation, query previously logged messages from the MySQL database, and publish messages to their either own log or their GridAPPS-D logs.

### 7.10.1 10.1. Information Flow Diagram

The figure below shows the information flow involved in subscribing and publishing to logs.




## 7.10.2 10.2 Sample App Code

[ ]:
```
from gridappsd.topics import simulation_log_topic
log_topic = simulation_log_topic(viz_simulation_id)

def demoLogFunction(header, message):
    timestamp = message["timestamp"]
    log_message = message["logMessage"]

    print("Log message received at timestamp ", timestamp, "which reads:")
    print(log_message)
    print(".......................")

gapps.subscribe(log_topic, demoLogFunction)
```

CHAPTER

# EIGHT

# GRIDAPPS-D SERVICE STRUCTURE

[ ]:

CHAPTER

NINE

# INTRODUCTION TO THE COMMON INFORMATION MODEL

This section introduces the CIM as a model format that is used for power system data and information exchange across applications, platforms, and services. The CIM is used for all power system models in GridAPPS-D, and it is important to have an understanding of the concepts and implementation of CIM for describing power systems using unique mRIDs for each piece of equipment and associated modeling objects.

## 9.1 1. Introduction

### 9.1.1 1.1. What is the Common Information Model?

The Common Information Model (CIM) is an abstract information model that can be used to model an electrical network and the various equipment used on the network.

CIM is widely used for data exchange of bulk transmission power systems, and is now beginning to find increasing use for distribution modeling and analysis.

By using a common model, utilities, vendors, and researches from both academia and industry can reduce the effort and cost of data integration, and instead focus on developing increased functionality for managing and optimizing the smart grid of the future.

### 9.1.2 1.2. Why is Data Integration Important?

In a typical distribution utility there are hundreds and even in some cases thousands of software solutions and applications that are managed by the IT department. These applications are used and operated independently by the various groups, departments, and organizations within the utility. Whenever a business process requires data from one system or application to be transferred to another system or application, the data needs to be manually extracted from the first database and then converted to the format of the other application's database.

Two strategies exist for dealing with extreme level of effort needed to manage, update, export, convert, and import data formats between different applications and databases.

1) Reduce the number of databases by purchasing a large software suite from a single vendor using a single proprietary data format that is internally-integrated and compatible with all the applications needed by utility
2) Adopt a common data integration platform that allows external integration between multiple software packages using a shared data format

### 9.1.3 1.3. What does CIM Provide?

CIM is an information model, that is an abstract, formal representation of objects, their attributes, the relationships between them, and the operations that can be performed on them. It is NOT a database structure or physical data store. It is a technology-agnostic model for describing the properties of physical power system equipment, power flow data, and messages that can be exchanged between various platforms and applications.

To describe various power system objects, CIM uses **Class Diagrams** and **Sequence Diagrams** created using the Unified Modeling Language (UML). It also uses the Resource Description Framework (RDF) to describe classes and attributes in an eXtensible Markup Language (XML) file format. The details of what is covered in each part of the CIM is described in detail below.

---

## 9.2 2. Background and Structure of the CIM

### 9.2.1 2.1. UML Class Diagrams

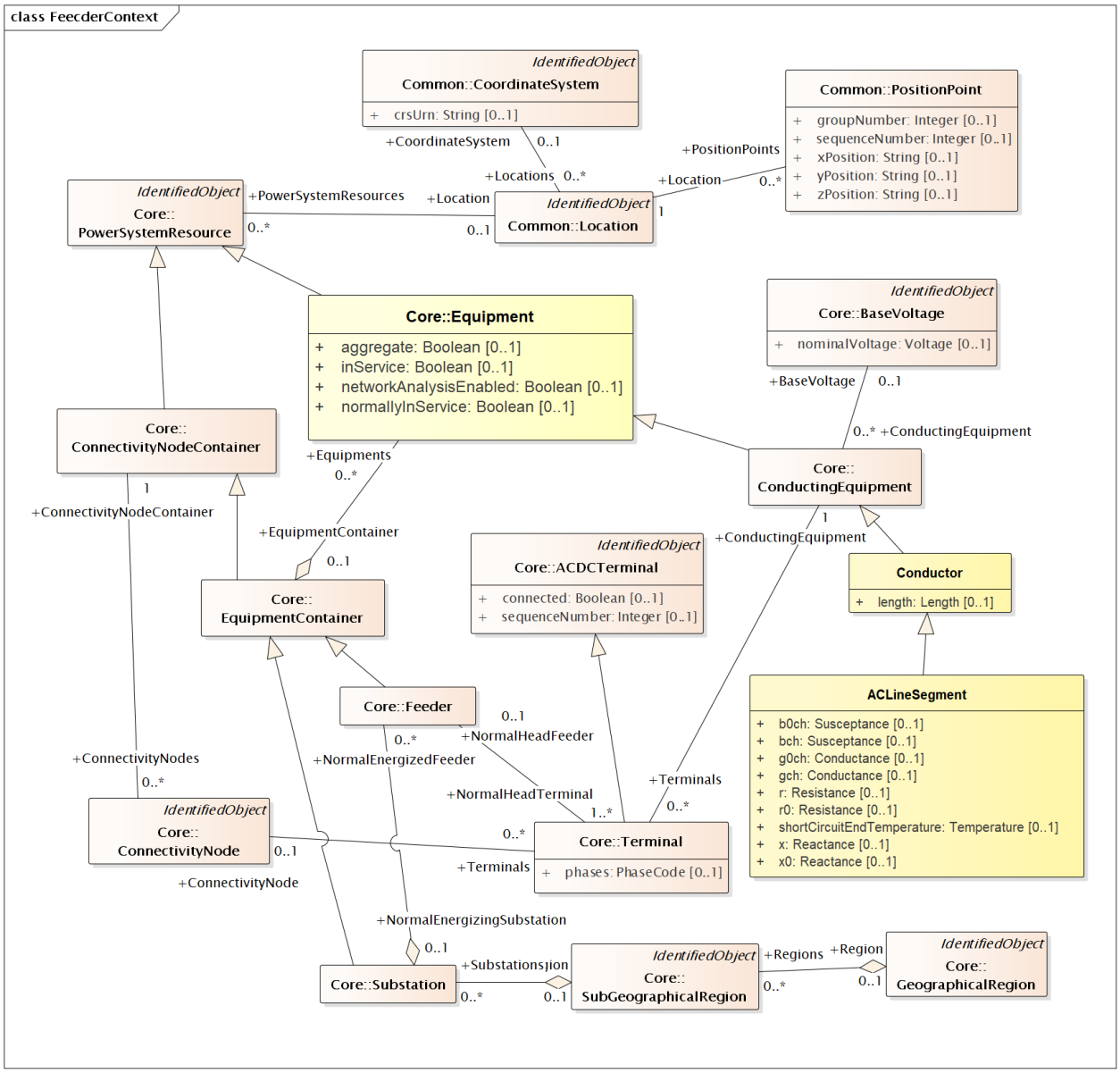
The Unified Modeling Language (UML) provides 13 types of diagrams to define software architecture. One of the is the **UML Class Diagram**, which visually represents object hierarchies and relationships.

First a review of basic concepts and terminology related to class diagrams:

- An **object** is any thing that we want to describe.
- A **class** represents a specific type of object.
- A **class hierarchy** is a model of the system showing every component as a separate class. The class hierarchy should represent the real-world structure of the system.
- A **package** is a group of classes. Think of folders in a computer file explorer.
- **Inheritance** allows us to define very general "parent classes" and very specific "child classes".
- **Attributes** are the properties that describe what type of thing the class represents.
- **Associations** are the relationships between various objects and how they are connected to each other.

Class diagrams show all the attributes and associations of various classes in a particular package in a single picture. To read a class diagram, remember that

- Lines with an arrowhead indicate class inheritance. For example, in the figure below, *ACLineSegment* inherits from *Conductor*, *ConductingEquipment*, *Equipment* and then *PowerSystemResource*. *ACLineSegment* inherits all attributes and associations from its ancestors (e.g., length), in addition to its own attributes and ancestors.
- Lines with a diamond indicate composition. For example, *Substations* make up a *SubGeographicalRegion*, which then make up a *GeographicRegion*.
- Lines without a terminating symbol are associations. For example, *ACLineSegment* has (through inheritance) a *BaseVoltage*, *Location* and one or more *Terminals*.
- Italicized names at the top of each class indicate the ancestor (aka superclass), in cases where the ancestor does not appear on the diagram. For example, *PowerSystemResource* inherits from *IdentifiedObject*.




A complete set of UML Class Diagrams is provided in the Advanced CIM Modeling section. This section contains class diagrams for all the objects used in GridAPPS-D and tables of properties to help you create and pass your own custom SPARQL queries to the Blazegraph Database.

### 9.2.2 2.2. UML Sequence Diagrams

UML sequence diagrams are used to model the flow of messages, events, and actions between the entities of a system. Time is represented vertically—showing the time sequence of interactions in the system. Displayed horizontally at the top of the diagram are the applications or entities in the system.

CIM uses UML diagrams to represent work flow, operations processes, and other utility use-cases. For the purposes of application development within GridAPPS-D, a detailed understanding of UML sequence diagrams is not required.

### 9.2.3 2.3. Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a method of defining information models that is specified by the World Wide Web Consortium (the W3C). Detailed documentation is available on the W3C website.

RDF focuses on making statements about objects in a subject-predicate-object expression. Each expression is commonly called a "triple" in RDF terminology. The subject is defined by naming a resource, the object denotes traits or attributes associated with the subject, and the predicate expresses the relationship between the subject and the object.

The subject, or resource, in an RDF model is expressed as a Uniform Resource Identifier (URI). URIs are similar to the Uniform Resource Locators (URLs) used as web addresses but are more general because they are not limited to accessible data on the web. The predicate and object are also technically URIs and so also are just identifiers. The subject-predicate-object triplets takes the form of expressing syntactical constructs like "a substation has a name".

RDF Schema (RDFS) files describe the classes, attributes, and relationships of an information model and typically use an .rdfs file format. RDF instance files describe object instances and typically use an .xml extension. RDF incremental files describe changes to a set of object instances as described by an instance file, and typically use an .xml extension.

CIM uses RDF instance files to define power system models with unique master resource identifier (**mRID**) issued by a model authority. The mRID is globally unique within an exchange context. Global uniqeness is easily achived by using a UUID for the mRID. It is strongly recommended to do this. For CIM XML data files in RDF syntax, the mRID is mapped to rdf:ID or rdf:about attributes that identify CIM object elements.

#### 2.3.1 Key Concepts & Terminology from RDF

- **URI References** – CIM and GridAPPS-D use two URI references to identify properties and resources. These identify the RDF format and the CIM classes used.
  - `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
  - `<http://iec.ch/TC57/CIM100#>`
- __

## 9.3 3. Summary of CIM XML Classes

This section provides a brief look at the classes of equipment modeled in CIM XML and used in GridAPPS-D.

Details of each package, the class diagram, and attributes of each class are provided in the relevant sections of the reference guide to this lesson.

### 9.3.1 3.1. Names, Nodes, Terminals

The *Core* package provides very high level information of the distribution feeder model

#### 3.1.1. IdentifiedObject

The *Core* package contains a class called *IdentifiedObject*. This class is very abstract and only contains attributes used to reference the object either by a user or in software. The attributes of *IdentifiedObject* include ***mRID***, which is the master resource identifier that should be a globally 3-18 unique identifier of objects; the *mRID* does not have to be human-readable. This identifier is generally intended to be used by software systems.

The attributes *name*, *description*, *aliasName*, and *pathName* are intended for providing identifiers that are human-readable. It is common for names of objects within a utility to not be unique due to historical naming conventions, the results of mergers and acquisitions, and the inability of other software systems to manage uniqueness. For these reasons, there are no constraints on these names requiring them to be unique.

#### 3.1.2. PowerSystemResource

The *PowerSystemResource* class inherits from *IdentifiedObject* and provides another relatively abstract class used in the CIM. The *PowerSystemResource* class supports an association to a *Company* class. This relationship identifies the company that operates the resource.

#### 3.1.4. ConnectivityNode

The *ConnectivityNode* class has a relationship to the *Terminal* class. Each *ConductingEquipment* object has *Terminals,* which are then connected to *ConnectivityNodes.* The terminals can be thought of as being closely related to the conducting equipment, and the connectivity nodes are the glue that defines what equipment is connected to what other equipment.

CIM also includes the *TopologicalNode* class, which is used to convert breaker-switch oriented power system models to bus-branch models. This object is not used in GridAPPS-D, which does not feature transmission substation configurations (e.g. breaker-and-a-half, main-and-transfer-bus, ring-bus, etc.) that require topological processing of breaker and switch positions to determine network topology and line connectivity.

### 9.3.2 3.2. Power System Equipment

CIM XML provides a number of classes for defining physical power system equipment, including lines, switches, transformers, regulators, capacitors, and reactors.

#### 3.2.1. *Equipment* and *ConductingEquipment*

The *ConductingEquipment* class inherits from an *Equipment* class which inherits from *PowerSystemResource*. This is the parent class for most of the physical equipment that are used to model the power system.

#### 3.2.2. *Conductor* and *ACLineSegment*

Directly inheriting from *ConductingEquipment* is the *Conductor* class. This class specifies the length of the conductor.

Each segment of a distribution line is defined in a CIM model as an *ACLineSegment*. This class contains the electrical attributes commonly associated with a line needed for steady state analysis, including the positive-sequence and zero-sequence resistance, reactance, conductance, and susceptance.

More details are available in the **`LineModel class diagram <>`__** and **`list of attributes <>`__**

#### 3.2.4. *PowerTransformer, TransformerWindings,* and *TapChanger*

These three classes specify the portions of a step-down transformer and regulator.

The *PowerTransformer* class inherits from *Equipment* (not ConductingEquipment) and has associations to the TransformerWinding class.

The majority of the electrical characteristics associated with the transformer are actually associated with the *TransformerWinding* objects.

An association from the *TransformerWinding* class to the *TapChanger* class is used when the transformer has a tap changer. The *TapChanger* class has as attributes for things like the tap steps and nominal setting. The *TapChanger* class inherits from the *PowerSystemResource* class instead of the *Equipment* class, so it has few inherited attributes and associations.

## 9.4 References

Portions of this tutorial have reproduced verbatim text and information from the EPRI report An Introduction to the CIM for Integrating Distribution Applications and System and the CIM Ontology Diagrams

[ ]:

CHAPTER

TEN

# API COMMUNICATION CHANNELS

## 10.1 1. What are Channels in GridAPPS-D?

When communicating with the GridAPPS-D Platform through API, it is necessary to specify a communication channel, which tells the GridAPPS-D platform on which channel to communicate with the application and through which API the message should be directed.

## 10.2 2. `/queue/` vs `/topic/`

GridAPPS-D uses two types of communication channels to determine the visibility of the API call to other applications and services.

## 10.3 2.1. Queue Channels

`/queue/` is used for communication channels where only the GridAPPS-D Platform is listening to the API call. These API calls are processed on a first-in, first-out basis. There is only one subscriber to the communication channel.

API calls to the Blazegraph database, Logs, Timeseries database, Config files, and Platform status are all queue channels. All the GridAPPS-D Topics for queue channels typically do not change over the course of an application or simulation run.

In the GridAPPSD-Python library, it is assumed that a topic is a queue channel if not otherwise specified. These two GridAPPS-D Topic definitions are equivalent:

```
topic = '/queue/goss.gridappsd.process.request.data.powergridmodel'
```

```
topic = 'goss.gridappsd.process.request.data.powergridmodel'
```

## 10.4 2.2. Topic Channels

`/topic/` is used for communication channels where the API call is to broadcast to all subscribers through the GOSS Message Bus, inlcuding other applications, services, FNCS Bridge, etc.

API calls to the Simulation, services, and active applications use topic channels to communicate and typically need to the specify the Simulation ID, Service ID, and Application ID. The particular topic for such an API call will change between simulations and instances, and so shortcut functions are provided in GridAPPSD-Python library to assist in generating the correct Topic.

In GridAPPSD-Python, it is necessary to specify if a GridAPPS-D Topic is a `/topic/` channel broadcasting to all subscribers:

```
topic = "/topic/goss.gridappsd.simulation.input."+simulation_id
```

---

## 10.5 3. Static GridAPPS-D Topics

Below are a list of the most common topics and where they are used. The appropriate topic for each API call will also be listed again in the subsequent lessons on each GridAPPS-D API. The list below can serve as an additional convenient reference.

These topics remain the remain the same between platform, application, and simulation instances. The GridAPPSD-Python Library shortcuts use all uppercase naming to indicate that these are static topic names.

### 10.5.1 Importing the Topics Library

When using topics in GridAPPSD-Python, it is recommended to import the `topics` library from `gridappsd`. This enables you to rapidly call the correct topic without needing to search for the correct topic string. This also protects your code from any changes inside the GridAPPS-D Platform if particular topic strings are deprecated or replaced – the python library names will stay persistent between all Platform releases.

For static GridAPPS-D topics, import the library by running

[ ]:
```
from gridappsd import topics as t
```

### 10.5.2 3.1. Request PowerGrid Model Data

This `/queue/` channel is used to communicate with PowerGrid Model API to pull power system model info from the the Blazegraph Database. The PowerGrid Model API is covered in detail in Lessons 2.2 and 2.3.

The base static string used is `goss.gridappsd.process.request.data.powergridmodel`, which can be called using the `.REQUEST_POWERGRID_DATA` or `.BLAZEGRAPH` methods from the topics library

A sample message that would be passed with this topic is

[2]:
```
from gridappsd import topics as t

# Sample PowerGrid Model message, explained in Lesson 2.2.
message = '{"requestType": "QUERY_MODEL_NAMES", "resultFormat": "JSON"}'

gapps.get_response(t.REQUEST_POWERGRID_DATA, message)
```

[2]:
```
{'data': {'modelNames': ['_204AC68D-C4B3-4D93-A2B5-B1C195C49954',
   '_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62',
   '_4F76A5F9-271D-9EB8-5E31-AA362D86F2C3',
   '_503D6E20-F499-4CC7-8051-971E23D0BF79',
   '_5B816B93-7A5F-B64C-8460-47C17D6E4B0F',
   '_67AB291F-DCCD-31B7-B499-338206B9828F',
   '_77966920-E1EC-EE8A-23EE-4EFD23B205BD',
   '_9CE150A8-8CC5-A0F9-B67E-BBD8C79D3095',
   '_AAE94E4A-2465-6F5E-37B1-3E72183A4E44',
   '_C1C3E687-6FFD-C753-582B-632A27E28507',
   '_E407CBB6-8C8D-9BC9-589C-AB83FBF0826D']},
 'responseComplete': True,
 'id': '365753873'}
```

[ ]:
```
from gridappsd import topics as t

# Sample PowerGrid Model message, explained in Lesson 2.2.
message = '{"requestType": "QUERY_MODEL_NAMES", "resultFormat": "JSON"}'

gapps.get_response(t.BLAZEGRAPH, message)
```

---

### 10.5.3 3.2. Request Timeseries Data

This /queue/ channel is used to communicate with the Timeseries API and Timeseries database, which stores real-time and historical data, such as weather information and AMI meter readings. The Timeseries database is covered in detail in Lesson 2.XX. A sample message that would be passed with this topic is

**Text String:** The topic can be specified as a static string:

- `topic = "goss.gridappsd.process.request.data.timeseries"`
- `gapps.get_response(topic, message)`

**GridAPPSD-Python Library Method:** The correct topic can also be imported from the GridAPPSD-Python topics library:

- `from gridappsd import topics as t`
- `gapps.get_response(t.TIMESERIES, message)`

---

### 10.5.4 3.3. Request Platform Status

This topic is used to check that status of the GridAPPS-D Platform.

**Text String:** The topic can be specified as a static string:

- `topic = "/queue/goss.gridappsd.process.request.status.platform"`
- `gapps.get_response(topic, message)`

**GridAPPSD-Python Library Method:** The correct topic can also be imported from the GridAPPSD-Python topics library.

- `from gridappsd import topics as t`
- `gapps.get_response(t.PLATFORM_STATUS, message)`

---

### 10.5.5 3.4. Querying Log Data

This topic is used to query log data in the MySQL Database using the Logging API

**Note:** This topic is different from the one used to subscribe to real-time log data being published by an ongoing simulation. This topic is used for querying data already stored in the database.

**Text String:** The topic can be specified as a static string:

- `topic = "goss.gridappsd.process.request.data.log"`
- `gapps.get_response(topic, message)`

**GridAPPSD-Python Library Method:** The correct topic can also be imported from the GridAPPSD-Python topics library:

- `from gridappsd import topics as t`
- `gapps.get_response(t.LOGS, message)`

---

### 10.5.6 3.5. Subscribing to Platform Logs

This topic is used to subscribe the to logs created by the GridAPPS-D Platform, such as which managers and core services have been started and are running.

**Text String:** The topic can be specified as a static string:

- `topic = "goss.gridappsd.process.request.data.timeseries"`
- `gapps.get_response(topic, message)`

**GridAPPSD-Python Library Function:** The correct topic can also be imported from the GridAPPSD-Python topics library. Note that this is a python function similar to the dynamic topics presented in the next section.

- `from gridappsd.topics import platfor_log_topic
- `topic = platform_log_topic()`
- `gapps.get_response(topic, message)`

[*Return to Top*]

---

## 10.6 4. Dynamic GridAPPS-D Topics

Several GridAPPS-D topics are unique to each application, simulation, or service instance. These topics are dynamic and will change from instance to instance.

The GridAPPS-D Platform will require that the topic specify the particular instance so that the API call can be delivered to the correct simulation or service.

To assist with the task of creating a dynamic topic that automatically updates between instances, several function are available in the GridAPPSD-Python topics library.

The available GridAPPSD-Python functions for dynamic topics are

- `simulation_input_topic(simulaton_id)` – Gets the topic to write data to for the simulation
- `simulation_output_topic(simulation_id)` – Gets the topic for subscribing to output from the simulation
- `simulation_log_topic(simulation_id)` – Topic for the subscribing to the logs from the simulation
- `service_input_topic(service_id, simulation_id)` – Utility method for getting the input topic for a specific service
- `service_output_topic(service_id, simulation_id)` – Utility method for getting the output topic for a specific service
- `application_input_topic(application_id, simulation_id)` – Utility method for getting the input topic for a specific application
- `application_output_topic(application_id, simulation_id)` – Utility method for getting the output topic for a specific application

---

### 10.6.1 4.1. Subscribe to Simulation Output

This topic is used to communicate with the Simulation API, which is covered in detail in Lesson XX. The Simulation Output Topic is used to subscribe to the simulation output, enabling applications to listen to switching actions, obtain equipment measurements, and so on.

The GridAPPSD-Python shortcut function for generating the correct topic is

`simulation_output_topic(simulation_id)`

There are two ways to use the function. The first is to call the library function directly. The second is to use it as part of a class definition.

**1) Call the topic function directly**

[ ]:
```
# Import GridAPPS-D Topic Function:
from gridappsd.topics import simulation_output_topic

# Call GridAPPSD-Python Topic Function
topic = simulation_output_topic(simulation_id)

# Print to Notebook Kernel:
print(topic)
```

**2) Use the topic function in a class definition**

```
[ ]: # Import GridAPPS-D Topic Function:
from gridappsd.topics import simulation_output_topic

# Define Subscription Class
class MySubscription(object):
    def __init__(self,simulation_id):
        self._subscribe_to_topic = simulation_output_topic(simulation_id)

# Define Main Function:
def _main():
    subscription = MySubscription(simulation_id)
    print(subscription._subscribe_to_topic)

# Call Main Function:
_main()
```

### 10.6.2 4.2. Publish to Simulation Input

This topic is used to communicate with the Simulation API, which is covered in detail in Lesson XX. The Simulation Input Topic is used to publish commands to the GOSS Message Bus, which are then broadcast to all applications, services, and simulations that are listening. Examples of actions that will use this topic include taking switching actions, adjusting DER setpoints, and changing regulator taps.

The GridAPPSD-Python shortcut function for generating the correct topic is

`simulation_input_topic(simulation_id)`

There are two ways to use the function. The first is to call the library function directly. The second is to use it as part of a class definition.

**1) Call the topic function directly**

```
[ ]: # Import GridAPPS-D Topic Function:
from gridappsd.topics import simulation_input_topic

# Call GridAPPSD-Python Topic Function
topic = simulation_output_topic(simulation_id)

# Print to Notebook Kernel:
print(topic)
```

**2) Use the topic function in a class definition**

```
[ ]: # Import GridAPPS-D Topic Function:
from gridappsd.topics import simulation_input_topic

# Define Subscription Class
class MySimulationPublisher(object):
    def __init__(self,simulation_id):
        self._publish_to_topic = simulation_input_topic(simulation_id)

# Define Main Function:
```

(continues on next page)

(continued from previous page)

```
def _main():
    subscription = MySimulationPublisher(simulation_id)
    print(subscription._publish_to_topic)

# Call Main Function:
_main()
```

---

### 10.6.3 4.3. Subscribe to Simulation Logs

This topic is used to communicate with the Simulation API, which is covered in detail in Lesson XX. The Simulation Output Topic is used to subscribe to the simulation output, which applications use to * Listen to switching actions * Obtaining equipment measurements * ***GET FULL LIST***

The GridAPPSD-Python shortcut function for generating the correct topic is

`simulation_output_topic(simulation_id)`

There are two ways to use the function. The first is to call the library function directly. The second is to use it as part of a class definition.

**1) Call the topic function directly**

[ ]:
```
# Import GridAPPS-D Topic Function:
from gridappsd.topics import simulation_output_topic

# Call GridAPPSD-Python Topic Function
topic = simulation_output_topic(simulation_id)

# Print to Notebook Kernel:
print(topic)
```

**2) Use the topic function in a class definition**

[ ]:
```
# Import GridAPPS-D Topic Function:
from gridappsd.topics import simulation_output_topic

# Define Subscription Class
class MySubscription(object):
    def __init__(self,simulation_id):
        self._subscribe_to_topic = simulation_output_topic(simulation_id)

# Define Main Function:
def _main():
    subscription = MySubscription(simulation_id)
    print(subscription._subscribe_to_topic)

# Call Main Function:
_main()
```

CHAPTER

# ELEVEN

# API MESSAGE STRUCTURE

This tutorial introduces the format used for passing messages to the GridAPPS-D API and how to wrap those messages using the GridAPPSD-Python Library.

## 11.1 1. Python Dictionaries VS JSON Strings

One of the confusing aspects of passing messages to and from the GridAPPS-D Platform and APIs is the difference between Python Dictionaries and JSON scripts, which look identical.

**JSON** is a *serialization format*. That is, JSON is a way of representing structured data in the form of a textual string.

A **Python Dictionary** is a *data structure*. That is, it is a way of storing data in memory that provides certain abilities to the code: in the case of dictionaries, those abilities include rapid lookup and enumeration.

It is possible to convert between the two by importing the JSON library: `import json`. Full documentation of JSON-Python interoperability and usage is available in Python Docs.

Use the `json.dumps()` method to serialize a dictionary as a JSON string. Use the `json.loads()` to import a JSON file and convert it into a dictionary. But the two are not the same: dictionaries are for working with data in your program, and JSON is for storing it or sending it around between programs.

With the GridAPPSD-Python Library, it is possible to pass query arguments as either a python dictionary or as a string. Both approaches will provide the same results.

**1) Format API call message as a dictionary**

This is the most direct approach, and will be used most often throughout this set of notebook tutorials. The format and structure of the python dictionary is explained in the next section.

[ ]:
```
model_mrid = "_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62" # IEEE 13 Node used for all example queries

# Format message as python dictionary
message = {
    "requestType": "QUERY_OBJECT_IDS",
    "resultFormat": "JSON",
    "modelId": model_mrid,
    "objectType": "LoadBreakSwitch"
}
```

```
[ ]: # Specify correct topic
topic = "goss.gridappsd.process.request.data.powergridmodel"

# Pass API Call to GridAPPS-D Platform
gapps.get_response(topic, message)
```

**2) Format API call message as a string**

This approach uses quotations (either ' ' or " ") to wrap the API call (identical to the python dictionary) as JSON-formatted text, concatenated into a string.

```
[ ]: model_mrid = "_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62" # IEEE 13 Node used for all example␣
↪queries

# Format message as JSON text wrapped as a string
message = """
{
    "requestType": "QUERY_OBJECT_IDS",
    "resultFormat": "JSON",
    "modelId": "%s",
    "objectType": "LoadBreakSwitch"
}
""" % model_mrid
```

```
[ ]: # Specify correct topic
topic = "goss.gridappsd.process.request.data.powergridmodel"

# Pass API Call to GridAPPS-D Platform
gapps.get_response(topic, message)
```

---

## 11.2 2. Structure of a GridAPPS-D Message

The structure of messages in GridAPPS-D follows that of a Python Dictionary using a data structure that is more generally known as an associative array. An excellent tutorial on advanced usage of the python dictionary structure is available on Real Python.

A dictionary consists of a collection of **key-value pairs**. Each key-value pair maps the **key** to its associated **value**.

- A dictionary is defined by enclosing a comma-separated list of key-value pairs in curly braces ( **{ }** ).
- A colon ( **:** ) separates each key from its associated value.
- Square brackets ( **[ ]** ) are used for a list of values associated to a particular key.
- Additional curly braces ( **{ }** ) can be used for cases where multiple key-value pairs (e.g. equipment setpoints) are associated with a particular key (e.g. an equipment class).

The general dictionary format used for GridAPPS-D messages is

```
message = {
    "key1": "value1",
    "key2": ["value21", "value22"],
    "key3": {
```

(continues on next page)

(continued from previous page)

```
        "key31": "value31",
        "key32": "value32"
        },
      .
      .
      .
    "key": "value"
}
```

**Important**: Be sure to pay attention to placement of commas ( , ) at the end of each line. Commas are placed at the end of each line *except* the last line. Incorrect comma placement will result in a syntax exception.

The particular set of key-value pairs for each GridAPPS-D API is covered in detail in Lessons 2.1 through 2.7.

---

## 11.3 3. Parsing Returned Data

After passing an API call, the GridAPPS-D Platform returns a JSON string that is subsequently converted into a python dictionary by the GridAPPSD-Python Library. This section will outline how to parse the data returned.

For this example, we are going to use a simple query from the PowerGrid Model API (covered in Lesson 2.2.) to obtain the details of a piece of equipment using its unique mRID (introduced in the next lesson).

[ ]:
```
model_mrid = "_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62" # IEEE 13 Node used for all example␣
↪queries

# Specify correct topic
topic = "goss.gridappsd.process.request.data.powergridmodel"

message = {
        "modelId": model_mrid,
        "requestType": "QUERY_OBJECT_DICT",
        "resultFormat": "JSON",
        "objectType": "LinearShuntCompensator",
}

# Pass API Call to GridAPPS-D Platform
response = gapps.get_response(topic, message)

import json
with open("foo.txt", 'w') as out:
    out.write(json.dumps(response, indent=2))
```

The structure of the python dictionary returned by the API is three key-value pairs for the keys of

- `'data'` – this is the data you requested
- `'responseComplete'` – true or false
- `'id'` – unique id associated with the API response dictionary

A typical API response will the structure below:

```
response = {
    'data': [{'key1': 'value1',
        'key2': ['value21', 'value22']},
        {'key1': 'value1',
         'key2': ['value21', 'value22']}],
     'responseComplete': True,
     'id': '12345678'
}
```

The first step is to filter the dictionary to just the data requested: `response['data']`. The result will be a list object.

Note: *some* API calls will also need to additional filters of `[results][bindings]`. The STOMP Client presented in the next section is very helpful for previewing the structure of the dictionary returned by GridAPPS-D.

```
[ ]: response = gapps.get_response(topic, message)
response_obj = response['data']
```

As `response_obj` is of the python type `list` rather than `dict`, it is necessary to use numerical indices instead of keys to access the values. A simple `for` loop is very helpful here.

In this example, we want to filter the results to create a list that contains just the name and mRID of the capacitor banks in the model.

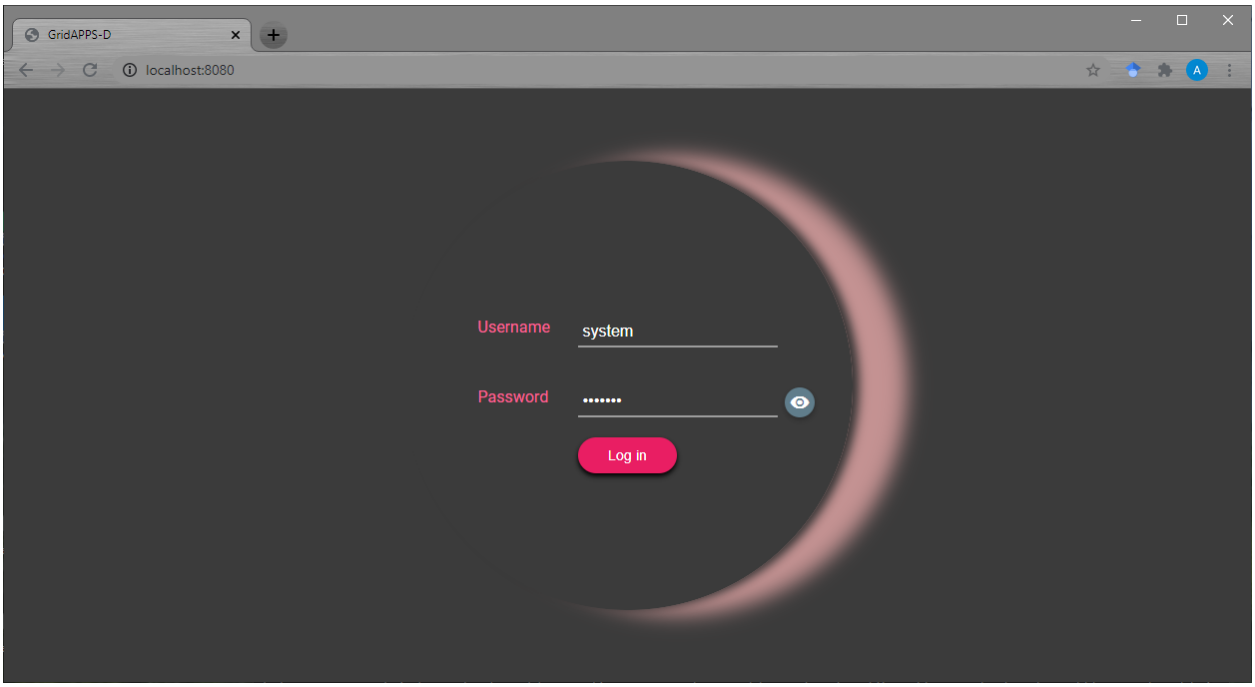
```
[ ]: capacitors = []
for index in response_obj:
    cap_name = index['IdentifiedObject.name']
    cap_mrid = index['id']
    message = dict(name = cap_name,
                   mrid = cap_mrid)
    capacitors.append(message)
```

[*Return to Top*]

## 11.4 4. Using the STOMP Client

The GridAPPS-D Visualization App includes a feature to pass API call messages through the GUI using the Simple Text Oriented Messaging Protocol (STOMP).

Open the Viz App, which is hosted on localhost:8080 (note: cloud-hosted installations will use the IP address of the server).




Log in and click on the menu in the top left corner of the browser window:

**|viz_top_menu.png|**

Select **Stomp Client** from the drop-down menu:

**|viz_menu_stomp.png|**

This opens the STOMP Client, which can be used to pass a message to any of the GridAPPS-D APIs to preview results or debug the API call message.

**|viz_stomp_client.png|**

### 11.4.1 4.1. Specifying the Topic

The appropriate GridAPPS-D topic needs to be copied and pasted into the **Destination Topic** box at the top of the window. The topic specifies on which channel the STOMP Client will communicate with the GridAPPS-D Platform and to which API the message needs to be delivered.

A complete list of GridAPPS-D topics was provided in Lesson 1.4. and will also be provided in context for each of the API calls detailed in subsequent lessons.

**IMPORTANT:** Remember to remove the python wrapping quotations at the beginning and end of the topic. For example, if the python-wrapped topic was

```
topic = "goss.gridappsd.process.request.data.powergridmodel" # Specify the topic
```

then the topic that is entered in the Stomp Client **Destination Topic** box is simply

```
goss.gridappsd.process.request.data.powergridmodel
```

**IMPORTANT:** The GridAPPSD-Python shortcut functions will not work in the STOMP Client. The full text string versions must be used.

### 11.4.2 4.2. Entering the Request Message

The **Request** box accepts an API call message identical to those provided in these notebook lessons.

**IMPORTANT:** Remember to remove the python wrapping at the beginning and end of the message. For example, if the python-wrapped message was

```
message = "{"requestType": "QUERY_MODEL_NAMES", "resultFormat": "JSON"}" # Sample
PowerGrid Model API Call
```

then the message that is entered in the Stomp Client **Request** box is simply

```
{"requestType": "QUERY_MODEL_NAMES", "resultFormat": "JSON"}
```

The STOMP client will automatically flag any errors in the JSON message.

## 11.5 4.3. Submitting a Request

After entering the topic and message, click **Send request** to send the API call to the GridAPPS-D Platform. The response will be displayed in the box below.

**|viz_stomp_send_request-2.png|**

It can be seen that the response from the STOMP Client is identical to that obtained by passing the same topic and message using the GridAPPSD-Python Library:

[ ]:
```
from gridappsd import GridAPPSD # Import Libraries
gapps = GridAPPSD("('localhost', 61613)", username='system', password='manager') #␣
↪Connect to Platform
topic = "goss.gridappsd.process.request.data.powergridmodel" # Specify correct Topic
message = {
    "requestType": "QUERY_MODEL_NAMES",
    "resultFormat": "JSON"
} # Sample PowerGrid Model API message
gapps.get_response(topic, message) # Pass API call to Platform
```

CHAPTER

TWELVE

# USING THE POWERGRID MODELS API

## 12.1 1. Introduction to the PowerGrid Model API

The PowerGrid Models API is used to pull model information from the Blazegraph Database, inlcuding the names, mRIDs, measurements, and nominal values of power system equipment in the feeder (such as lines, loads, switches, transformers, and DERs).

In the Application Components diagram (explained in detail with sample code in GridAPPS-D Application Structure), the PowerGrid Models API is used for querying for the power system model and querying for model measurement MRIDs.

**|power_grid_models_usage.png|**

## 12.2 2. API Syntax Overview

**Application passes query to GridAPPS-D Platform**

First, the application creates a query message for requesting information about the desired power system components in the format of a JSON string or equivalant Python dictionary object. The syntax of this message is explained in detail below.

The query is sent using `gapps.get_response(topic, message)` with a response expected back from the platform within the specified timeout period.

The application then passes the query through the PowerGrid Models API to the GridAPPS-D Platform, which publishes it to the `goss.gridappsd.process.request.data.powergridmodel` queue channel on the GOSS Message Bus. If the app is authenticated and authorized to pass queries, the query message is delivered to the data managers, which obtain the desired information from the Blazegraph Database.

**GridAPPS-D Platform responds to Application query**

The data managers then publish the response from the Blazegraph Database to the appropriate queue channel. The PowerGrid Models API then returns the desired information back to the application as a JSON message or equivalant Python dictionary object.

### 12.2.1 2.1. API Communication Channel

All queries passed to the PowerGrid Models API need to use the correct communication channel, which is obtained using the *GridAPPS-D Topics library*.

The PowerGrid Model API uses a `/queue/` channel to pull power system model info from the the Blazegraph Database. The base static string used is `goss.gridappsd.process.request.data.powergridmodel`, which can be called using the `.REQUEST_POWERGRID_DATA` or `.BLAZEGRAPH` methods from the topics library.

When developing in python, it is recommended to use the `.REQUEST_POWERGRID_DATA` method. When using the STOMP client in GridAPPS-D VIZ, it is necessary to use the base static string.

```
[ ]: from gridappsd import topics as t
     topic = t.REQUEST_POWERGRID_DATA
```

### 12.2.2 2.2. Structure of a Query Message

Queries passed to PowerGrid Models API are formatted as python dictionaries or equivalent JSON scripts wrapped as a python string.

```
message = {
    "requestType": "INSERT QUERY HERE",
    "resultFormat": "JSON",
    "modelId": "OPTIONAL INSERT MODEL mRID HERE",
    "objectType": "OPTIONAL INSERT CIM CLASS HERE",
    "objectId": "OPTIONAL INSERT OBJECT mRID HERE",
    "filter": "OPTIONAL INSERT SPARQL FILTER HERE"
}
```

The components of the message are as follows:

- `"requestType":` – Specifies the type of query. Available requestType are listed in the next section.
- `"resultFormat":` – Specifies the format of the response, can be `"JSON"`, `"CSV"`, or `"XML"`. (CAUTION: the PowerGridModel API uses the key *resultFormat*, while the Timeseries API uses the key *reponseFormat*. Using the wrong key for either API will result in a java.lang error.)
- `"modelID":` – Optional. Used to filter the query to only one particular model whose mRID is specified. Be aware of spelling and capitalization differences between JSON query spelling `"modelId"` and Python Library spelling `model_id`.
- `"objectType":` – Optional. Used to filter the query to only one CIM class of equipment. Speciying the *objectID* will override any values specified for *objectType*.
- `"objectID":` – Optional. Used to filter the query to only one object whose mRID is specified. Specifying the *objectID* will override any values specified for *objectType*.
- `"filter":` – Optional. Used to filter the query using a SPARQL filter. SPARQL queries are covered in the next lesson.

The usage of each of these message components are explained in detail with code block examples below.

**Important**: Be sure to pay attention to placement of commas ( `,` ) at the end of each JSON line. Commas are placed at the end of each line *except* the last line. Incorrect comma placement will result in a JsonSyntaxException.

All of the queries are passed to the PowerGrid Model API using the `.get_response(topic, message)` method for the GridAPPS-D platform connection variable.

### 12.2.3 2.3. Specifying the `requestType`

Below are the possible `requestType` strings that are used to specify the type of each query. Executable code block examples are provided for each of the requests in the subsections below.

The first group of *requestType* key-value pairs are for queries for information related to the entire model or a set of models, such as the model name, mRID, region, and substation:

- `"requestType": "QUERY_MODEL_NAMES"` – *Query for the list of all model name mRIDs*
- `"requestType": "QUERY_MODEL_INFO"` – *Query for the dictionary of all details for all feeders in Blazegraph*

The second group of *requestType* key-value pairs are for queries for a single object or a single class of objects withing a model, such as the object mRID, CIM attributes, or measurement points:

- `"requestType": "QUERY_OBJECT_TYPES"` – *Query for the types of CIM classes of objects in the model*
- `"requestType": "QUERY_OBJECT_IDS"` – *Query for a list of all mRIDs for objects of a CIM class in the model*
- `"requestType": "QUERY_OBJECT"` – *Query for CIM attributes of an object using its unique mRID*
- `"requestType": "QUERY_OBJECT_DICT"` – *Query for the dictionary of all details for an object using either its *objectType* OR its *objectID**
- `"requestType": "QUERY_OBJECT_MEASUREMENTS"` – *Query for all measurement types and mRIDs for an object using either its *objectType* OR its *ObjectID**

The third group of *requestType* key-value pairs are for queries based on SPARQL filters or complete SPARQL queries. The structure of SPARQL was introduced in Lesson 1.7 (to be completed soon). Usage of these two *requestType* will covered separately in the next two lessons.

- `"requestType": "QUERY_MODEL"` – Query for all part of a specified model, filtered by object type using a SPARQL filter.
- `"requestType": "QUERY"` – Query using a complete SPARQL query.

---

## 12.3 3. Querying for Feeder Model Info

This section outlines the pre-built JSON queries that can be passed to the PowerGrid Model API to obtain mRIDs and other information for all models and feeders stored in the Blazegraph Database.

### 12.3.1 3.1. Query for mRIDs of all Models

This query obtains a list of all the model MRIDs stored in the Blazegraph database.

Query requestType:

- `"requestType": "QUERY_MODEL_NAMES"`

Allowed parameters:

- `"resultFormat":` – "XML" / "JSON" / "CSV" – Optional. Will return results as a list in the format selected.

[ ]:
```
message = {
    "requestType": "QUERY_MODEL_NAMES",
    "resultFormat": "JSON"
}
```

```
[ ]: gapps.get_response(topic, message)
```

*Return to Top*

### 12.3.2 3.2. Query for Details Dictionary of all Models

This query returns a list of names and MRIDs for all models, substations, subregions, and regions for all available feeders stored in the Blazegraph database.

Query requestType:

- `"requestType": "QUERY_MODEL_INFO"`

Allowed parameters:

- `"resultFormat":` – "XML" / "JSON" / "CSV" – Will return results as a list in the format selected.

```
[ ]: message = {
    "requestType": "QUERY_MODEL_INFO",
    "resultFormat": "JSON"
}
```

```
[ ]: gapps.get_response(topic, message)
```

---

## 12.4 4. Querying for Object Info

This section outlines the pre-built JSON queries that can be passed to the PowerGrid Model API to obtain mRIDs and other information for a particular object or a class of objects for one or more feeders stored in the Blazegraph Database.

All of the examples in this section use the IEEE 13 node model. The python constructor %s is used for all queries to enable the code block to be cut and paste into any python script without needing to change the model mRID.

```
[ ]: model_mrid = "_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62" # IEEE 13 Node used for all example␣
↪queries
```

### 12.4.1 4.1. Query for CIM Classes of Objects in Model

This query is used to query for a list of all the CIM XML classes of objects present in the Blazegraph for a particular model or all models in the database.

Query requestType is

- `"requestType": "QUERY_OBJECT_TYPES"`

Allowed parameters are

- `"modelId":` "model name mRID" – Optional. Searches only the particular model identified by the given unique mRID
- `"resultFormat":` – "XML" / "JSON" / "CSV" – Will return results as a list in the format selected.

$_1)Query entire Blazegraph database$

Omit the '"modelId"' parameter to search the entire blazegraph database.

[ ]:
```
message = {
    "requestType": "QUERY_OBJECT_TYPES",
    "resultFormat": "JSON"
}
```

[ ]:
```
gapps.get_response(topic, message)
```

**2) Query for only a particular model**

Specify the model MRID as a python string and pass it as a parameter to the method to return only the CIM classes of objects in that particular model.

Be aware of spelling and capitalization differences between JSON query spelling `"modelId"` and Python Library spelling `model_id`.

[ ]:
```
message = {
    "requestType": "QUERY_OBJECT_TYPES",
    "modelId": model_mrid,
    "resultFormat": "JSON"
}
```

[ ]:
```
gapps.get_response(topic, message)
```

### 12.4.2 4.2. Query for mRIDs of Objects in a Feeder

This query is used to obtain all the mRIDs of objects of a particular CIM class in the feeder.

Query responseType is

- `"requestType": "QUERY_OBJECT_IDS"`

Allowed parameters are:

- `"modelId":` "model name mRID" – When specified it searches against that model, if empty it will search against all models
- `"objectType":` "CIM Class" – Optional. Specifies the type of objects you wish to return details for.
- `"resultFormat":` – "XML" / "JSON" / "CSV" – Will return results as a list in the format selected.

Within a particular feeder, it is possible to query for objects of all the CIM classes obtained using `"requestType": "QUERY_OBJECT_TYPES"` (discussed above in *Section 4.1*). Note that the RDF URI is not included in the query, only the name of the class, such as `"objectType": "ACLineSegment"` or `"objectType": "LoadBreakSwitch"`.

[ ]:
```
message = {
    "requestType": "QUERY_OBJECT_IDS",
    "resultFormat": "JSON",
    "modelId": model_mrid,
    "objectType": "LoadBreakSwitch"
}
```

[ ]:
```
response_obj = gapps.get_response(topic, message)
```

[ ]:
```
response_obj['data']
```

```
[ ]: switch_mrids = response_obj['data']['objectIds']
```

### 12.4.3 4.3. Query for CIM Attributes of an Object

This query is used to obtain all the attributes and mRIDs of those attributes for a particular object whose mRID is specified.

Query responseType is

- `"requestType": "QUERY_OBJECT"`

Allowed parameters are:

- `"modelId":` "model name mRID" – When specified it searches against that model, if empty it will search against all models
- `"objectId":` "object mRID" – Optional. Specifies the type of objects you wish to return details for.
- `"resultFormat":` – "XML" / "JSON" / "CSV" – Will return results as a list in the format selected.

Within a particular feeder, it is possible to query for objects of all the CIM classes obtained using `"requestType": "QUERY_OBJECT_TYPES"` (discussed above in *Section 4.1*). Note that the RDF URI is not included in the query, only the name of the class, such as `"objectType": "ACLineSegment"` or `"objectType": "LoadBreakSwitch"`.

```
[ ]: object_mrid = "_2858B6C2-0886-4269-884C-06FA8B887319"
message = """
{
        "requestType": "QUERY_OBJECT",
        "resultFormat": "JSON",
        "modelId": "%s",
        "objectId": "%s"
}
""" % (model_mrid, object_mrid)
```

```
[ ]: message = {
        "requestType": "QUERY_OBJECT",
        "resultFormat": "JSON",
        "objectId": "_4F76A5F9-271D-9EB8-5E31-AA362D86F2C3"
}
```

```
[ ]: gapps.get_response(topic, message)
```

### 12.4.4 4.4. Query for Object Dictionary

This query returns a python dictionary of all the equipment attributes and mRIDs. The query can be for 1) all objects of a particular `objectType` or 2) for those connected to a particular object based on the `objectId`.

If neither `objectType` or `objectId` is provided, the query will provide all measurements belonging to the model. Query responseType is

- `"requestType": "QUERY_OBJECT_DICT"`

Allowed parameters are:

- `"modelId":` "model name mRID" – When specified it searches against that model, if empty it will search against all models

- `"objectId"`: "object mRID" – Optional. Specifies the type of objects you wish to return details for.
- `"objectType"`: "CIM Class" – Optional. Specifies the type of objects you wish to return details for.
- `"resultFormat"`: "XML" / "JSON" / "CSV" – Will return results as a list in the format selected.

Speciying the `objectID` will override any values specified for `objectType`.

**Example 1: Querying for model dictionary for an ``"objectID":``**

[ ]:
```
message = {
    "modelId": model_mrid,
    "requestType": "QUERY_OBJECT_DICT",
    "resultFormat": "JSON",
    "objectId": switch_mrids[1]
}

gapps.get_response(topic, message)
```

**Example 2: Querying for model dictionary for an ``"objectType":``**

[ ]:
```
message = {
    "modelId": model_mrid,
    "requestType": "QUERY_OBJECT_DICT",
    "resultFormat": "JSON",
    "objectType": "TransformerTank"
}

gapps.get_response(topic, message)
```

---

## 12.5 5. Querying for Object Measurements

### 12.5.1 5.1. Object mRIDs vs Measurement mRIDs

A key concept in GridAPPS-D and CIM XML power system models is the difference between the object mRID of a piece of equipment and multiple measurement mRIDs associated with its control settings and power flow values.

Measurements differ from the state variables (e.g. those obtained from State Estimator or a power flow calculation) in that the values are measured here and not calculated or estimated. Each Measurement is associated to a *PowerSystemResource*, and in GridAPPS-D for now, also a Terminal that belongs to the same *PowerSystemResource*. (Non-electrical measurements, for example weather, would not have the Terminal).

The *measurementType* is a string code from IEC 61850, with the following currently suppported:

- **PNV** – Phase to Neutral Voltage
- **VA** – Volt-Amperes (apparent power)
- **A** – Amperes (current)
- **POS** – Position for switches and transformer taps

Each measurement object has a **name**, **mRID**, and **phases**. In GridAPPS-D, each phase is measured individually so multi-phase codes like ABC should not be used.

Pos measurements will be discrete, for such things as tap position, switch position, or capacitor bank position.

The others will be Analog, with magnitude and optional angle in degrees.

Each MeasurementValue will have a timeStamp and mRID inherited from IdentifiedObject, so the values can be traced.

### 12.5.2 5.2. Querying for Measurements

This query returns details for the measurements within a model. The query can be for 1) all objects of a particular `objectType` or 2) for those connected to a particular object based on the `objectId`.

If neither `objectType` or `objectId` is provided, the query will provide all measurements belonging to the model. Query responseType is

- `"requestType": "QUERY_OBJECT_MEASUREMENTS"`

Allowed parameters are:

- `"modelId":` "model name mRID" – When specified it searches against that model, if empty it will search against all models
- `"objectId":` "object mRID" – Optional. Specifies the type of objects you wish to return details for.
- `"objectType":` "CIM Class" – Optional. Specifies the type of objects you wish to return details for.
- `"resultFormat":` "XML" / "JSON" / "CSV" – Will return results as a list in the format selected.

Speciying the `objectID` will override any values specified for `objectType`.

**Example 1: Querying for all measurements for an ``"objectID":``**

[ ]:
```
message = {
    "modelId": model_mrid,
    "requestType": "QUERY_OBJECT_MEASUREMENTS",
    "resultFormat": "JSON",
    "objectId": switch_mrids[1]
}

gapps.get_response(topic, message)
```

**Example 2: Querying for all measurements for an ``"objectType":``**

[ ]:
```
message = {
    "modelId": model_mrid,
    "requestType": "QUERY_OBJECT_MEASUREMENTS",
    "resultFormat": "JSON",
    "objectType": "ACLineSegment"
}

gapps.get_response(topic, message)
```

## 12.6 5.3. Filtering Returned Data

After receiving the python dictionary of measurements, it will be necessary to parse it to inlcude just the desired set of measurements. This is done using the method presented in *Parsing Returned Data*

[ ]:
```
obj_msr_ACline = gapps.get_response(topic, message, timeout=10)

# Filter to just values for 'data' key
obj_msr_ACline = obj_msr_ACline['data']

# Chose specific measurement mrid. Screen out those whose type is not PNV. For example,
obj_msr_ACline = [k for k in obj_msr_ACline if k['type'] == 'Pos']

obj_msr_ACline
```

## 12.7 6. GridAPPSD-Python Shortcut Methods

A small number of simple PowerGrid Model API queries have pre-built Python functions that can be used without specifying the topic and a particular message.

### 12.7.1 6.1. Querying for mRIDs of all Models

The `.query_model_names` method is associated with the GridAPPSD connection object and returns a list of all the CIM XML classes of objects present in the Blazegraph for a particular model or all models in the database.

This method will return identical results to the python dictionary message explained above in *Section 3.1*

[ ]:
```
gapps.query_model_names()
```

### 12.7.2 6.2. Query for CIM Classes of Objects in Model

The `.query_object_types` method is associated with the GridAPPSD connection object and returns a list of all the CIM XML classes of objects present in the Blazegraph for a particular model or all models in the database.

This method will return identical results to the python dictionary message explained above in *Section 4.1*

Allowed parameters are

- model_id (optional) - when specified, it searches only the particular model identified by the given unique mRID

**1) Query entire Blazegraph database**

Leave the arguments blank to search all models in the Blazegraph database

[ ]:
```
gapps.query_object_types()
```

**2) Query for only a particular model**

Specify the model MRID as a python string and pass it as a parameter to the method to return only the CIM classes of objects in that particular model

```
[ ]: model_mrid = "_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62" # IEEE 13 Node used for all example␣
     ↪queries
     gapps.query_object_types(model_mrid)
```

### 12.7.3 6.3. Query for CIM Attributes of an Object

The `.query_object` method is associated with the GridAPPSD connection object and returns a list of all the CIM XML classes of objects present in the Blazegraph for a particular model or all models in the database.

This method will return identical results to the python dictionary message explained above in *Section 4.3*

```
[ ]: model_mrid = "_49AD8E07-3BF9-A4E2-CB8F-C3722F837B62" # IEEE 13 Node used for all example␣
     ↪queries
     object_mrid = "_2858B6C2-0886-4269-884C-06FA8B887319"

     gapps.query_object(model_mrid, object_mrid)
```

---

## 12.8 7. Available Models in Default Installation

### 12.8.1 7.1. IEEE 13 Node Model

This is a very small distribution test feeder operating at 4.16 kV voltage level. It consists of a single voltage regulator at the substation, overhead and underground lines, shunt capacitor, and an in-line transformer. This feeder is relatively highly loaded and provides a good test of the convergence of the problem for a very unbalanced system.

This model is recommended for debugging as the model is small enough that issues can be traced by hand.

### 12.8.2 7.2. IEEE 123 Node Model

This models a medium-sized unbalanced distribution system operating at the nominal voltage of 4.16 kV. It consists of overhead and underground lines with single, two and three-phase laterals, along with step regulators and shunt capacitors for voltage regulation. The feeder model is characterized by the unbalanced loading having all combinations of load types (constant current, impedance, and power). It also includes a few switches to allow for the alternate paths for the power flow via feeder reconfiguration.

This model is recommended for initial app testing and debugging thorugh the first stages of development.

### 12.8.3 7.3. IEEE 123 Node Model with PV

### 12.8.4 7.4. IEEE 8500 Node Model

This is a relatively large and realistic radial distribution feeder consisting of MV and LV (secondary) circuits [21]. Unlike other test systems, this feeder also includes 120/240V center-tapped transformers that are commonly deployed in North American power distribution systems. Thus, it allows for users to interchange between the two versions of loading conditions: balanced (208 V) and unbalanced (120 V) in the secondary transformers. Voltage control is possible using a substation LTC transformer, as well as multiple poletop regulators and capacitor banks. The feeder was created to test scalability and convergence of power flow algorithms on a large unbalanced power distribution system.

- Length: 170 km
- Nominal voltage: 12.47 kV, 120/240V
- Topology: radial
- Service transformers: yes
- Customers: 1177
- Peak load: 11.1 MW
- Normally-open switches: no

### 12.8.5 7.5. 9500 Node Test System

The 9500 Node Test System includes three radial distribution feeders with just over 12 MW of load, consisting of both medium voltage and low voltage equipment each supplied by a different distribution substation. The three distribution feeders are connected to each other through Normally-Open switches, which is representative of the way many utilities operate in North America. One feeder represents today's grid with low penetration of customer-side renewables. The second represents a potential future grid with microgrids and 100% renewable penetration. The third has no customer resources, a district steam plant, and a utility-scale PV farm. All three feeders have customers connected by low-voltage secondary triplex lines.

| DER Name | kW Rating | kVA Rating | Characteristics | Equipment | Feeder | Location |
|---|---|---|---|---|---|---|
| SteamGen1 | 3000 | 4000 | Legacy CHP steam plant | | S3 | Old Town |
| PVFarm | 500 | 750 | Community solar farm | | S3 | Old Town |
| MicroTurb-1 | 200 | 250 | Natural gas microturbine | Capstone C200S [10] | S2 | New Neighborhood Microgrid |
| MicroTurb-2 | 200 | 250 | | | S2 | |
| MicroTurb-3 | 200 | 250 | | | S2 | |
| MicroTurb-4 | 200 | 250 | | | S2 | Central Neighborhood |
| Diesel620 | 620 | 775 | Inverter-connected diesel genset (VAR support when OFF) | Innovus IP CVS 620 [15] | S1 | Hospital Microgrid |
| Diesel590 | 590 | 737 | | Innovus IP CVS 590 | S1 | Central Neighborhood |
| LNGEngine100 | 100 | 125 | Inverter-connect LNG genset | InVerde Ultera 125 [16] | S1 | Shopping Center Microgrid |
| LNGEngine1800 | 1800 | 2250 | LNG reciprocating peaking unit | Cummins HSK78G [13] | S1 | Industrial District Microgrid |
| Battery1 | 250 | 250 | Generic battery storage | | S2 | New Neighborhood Microgrid |
| Battery2 | 250 | 250 | Generic battery storage | | S2 | |

### 12.8.6 7.6. PNNL Taxonomy Feeder

### 12.8.7 7.7. EPRI J1 Feeder

### 12.8.8 7.8. UAF Microgrid

---

**|GridAPPS-D_narrow.png|**

CHAPTER

# THIRTEEN

# USING THE CONFIGURATION FILE API

## 13.1 1. Introduction to the Configuration File API

The Configuration File API is used to generate power system models that can be solved in GridLab-D or OpenDSS based on the original CIM XML model. The load profile and ZIP parameters can be modified from the nominal values prior to model creation and export.

In the Application Components diagram (explained in detail with sample code in GridAPPS-D Application Structure), the Configuration File API is used for configuring parallel simulations and exporting the power system model.

**|04_config_sim_export.png|**

**Application passes query to GridAPPS-D Platform**

First, the application creates a query message for requesting information about the desired power system configuration in the format of a JSON string or equivalant Python dictionary object. The syntax of this message is explained in detail below.

The application then passes the query through the Configuration File API to the GridAPPS-D Platform, which publishes it to a queue channel on the GOSS Message Bus. If the app is authenticated and authorized to pass queries, the query message is delivered to the Configuration Manager.

**GridAPPS-D Platform responds to Application query**

The Configuration Manager obtains the CIM XML file for the desired power system model and then converts it to the desired output format with all of the requested changes to the model. The Configuration File API then returns the desired information back to the application as a JSON message (for Y-Bus or partial models) or export the files to the directory specified in the

---

## 13.2 2. API Syntax Overview

### 13.2.1 2.1. API Communication Channel

All queries passed to the PowerGrid Models API need to use the correct communication channel, which is obtained using the *GridAPPS-D Topics library*.

The PowerGrid Model API uses a `/queue/` channel to pull power system model info from the the Blazegraph Database. The base static string used is `goss.gridappsd.process.request.config`, which can be called using the `.CONFIG` method from the topics library.

When developing in python, it is recommended to use the `.CONFIG` method. When using the STOMP client in GridAPPS-D VIZ, it is necessary to use the base static string.

```
[ ]: from gridappsd import topics as t
     topic = t.CONFIG
```

### 13.2.2 2.2. Structure of a Query Message

Queries passed to Configuration File API are formatted as python dictionaries or equivalent JSON scripts wrapped as a python string.

The accepted set of key-value pairs for the Configuration File API query message is

```
message = {
    "configurationType": "INSERT QUERY HERE",
    "parameters": {
        "key1": "value1",
        "key2": "value2"}
}
```

The components of the message are as follows:

- `"configurationType":` – Specifies the type of configuration file requested.
- `"parameters":` – Specifies any specific power system model parameters. Values depend on the particular configurationType.

The usage of each of these message components are explained in detail with code block examples below.

**Important**: Be sure to pay attention to placement of commas ( `,` ) at the end of each JSON line. Commas are placed at the end of each line *except* the last line. Incorrect comma placement will result in a JsonSyntaxException.

All of the queries are passed to the Configuration API using the `.get_response(topic, message)` method for the GridAPPS-D platform connection variable.

### 13.2.3 2.3. Specifying the `configurationType`

Below are the possible `configurationType` key-value pairs that are used to specify the type of each query. Executable code block examples are provided for each of the requests in the subsections below.

The first group of *configurationType* key-value pairs are for queries for information related to the GridLab-D GLM files and settings:

- `"configurationType": "GridLab-D All"` – *Query for all GridLab-D files*
- `"configurationType": "GridLab-D Base GLM"` – *Query for GridLab-D base GLM file*
- `"configurationType": "GridLab-D Symbols"` – *Query for GridLab-D symbols file*
- `"configurationType": "GridLab-D Simulation Output"` – *Query for available measurement types*

The second group of *configurationType* are for queries for CIM dictionary and feeder index files:

- `"configurationType": "CIM Dictionary"` – *Query for python dictionary of CIM XML model*
- `"configurationType": "CIM Feeder Index"` – *Query for python dictionary of model mRIDs*

The third group of *configurationType* key-value pairs are for queries for OpenDSS model files

- `"configurationType": "DSS All"` – *Query for all OpenDSS model files*
- `"configurationType": "DSS Base"` – *Query for OpenDSS version of power system model*

- `"configurationType": "DSS Coordinate"` – *Query for list of OpenDSS XY coordinates*
- `"configurationType": "YBus Export"` – *Export Y-Bus matrix from OpenDSS*

---

## 13.3 3. Querying for GridLab-D Configuration Files

This section outlines the details of key-value pairs for the possible queries associated with each value of the `queryMeasurement` key listed above.

### 13.3.1 3.1. Query for all GridLab-D Files

This API call generates all the GLM files necessary to solve the power system model in GridLab-D. The query returns a directory where the set of GLM files are located.

Configuration File request key-value pair:

- `"configurationType": "GridLab-D All"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                "directory":                           output directory as string ,
                "simulation_name":                     string ,
                "simulation_start_time":               epoch time number ,
                "simulation_duration":                 number ,
                "simulation_id":                       number ,
                "simulation_broker_host":              string ,
                "simulation_broker_port":              number ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "i_fraction":                          number ,
                "p_fraction":                          number ,
                "z_fraction":                          number ,
                "load_scaling_factor":                 number ,
                "schedule_name":                       string ,
                "solver_method":                       string }
```

The numeric values for the key-value pairs associated with `parameters` can be written as number or as strings. The key-value pairs can be specified in any order.

**Example: Export IEEE 13 node model with constant current loads to GLM files :**

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "GridLAB-D All",
    "parameters": {
        "directory": "/tmp/gridlabdsimulation/",
        "model_id": model_mrid,
        "simulation_id": "12345678",
        "simulation_name": "mysimulation",
        "simulation_start_time": "1518958800",
```

(continues on next page)

(continued from previous page)

```
        "simulation_duration": "60",
        "simulation_broker_host": "localhost",
        "simulation_broker_port": "61616",
        "schedule_name": "ieeezipload",
        "load_scaling_factor": "1.0",
        "z_fraction": "0.0",
        "i_fraction": "1.0",
        "p_fraction": "0.0",
        "solver_method": "NR" }
}

gapps.get_response(topic, message, timeout = 120)
```

**Note:** The output directory is inside the GridAPPS-D Docker Container, not in your Ubuntu or Windows environment. To access the files, it is necessary to change directories to those inside the docker container.

Open a new Ubuntu terminal and run: * `docker exec -it gridappsd-docker_gridappsd_1 bash` * `cd /tmp/gridlabdsimulation` * `ls -l`

To copy the files to your regular directory, use the `docker cp` command. For help using docker, see Docker Shortcuts on working with Docker containers.

**|2_7_config_file_docker_directory.png|**

---

## 13.4 3.2. Query for GridLab-D Base GLM File

This API call generates a single GLM file that contains the entire power system model that can be solved in GridLab-D. The query returns the entire model GLM file wrapped in a python dictionary.

Configuration File request key-value pair:

- `"configurationType": "GridLab-D Base GLM"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "i_fraction":                          number ,
                "p_fraction":                          number ,
                "z_fraction":                          number ,
                "load_scaling_factor":                 number ,
                "schedule_name":                       string }
```

The numeric values for the key-value pairs associated with `parameters` can be written as number or as strings. The key-value pairs can be specified in any order.

**Example 1: Create GLM base file using nominal load values:**

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
```

(continues on next page)

(continued from previous page)

```
    "configurationType": "GridLAB-D Base GLM",
    "parameters": {"model_id": model_mrid}
}

gapps.get_response(topic, message, timeout = 60)
```

**Example 2: Create GLM base file using all constant current loads and hourly load curve:**

[ ]:
```
topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "GridLAB-D Base GLM",
    "parameters": {
        "model_id": model_mrid,
        "load_scaling_factor": "1.0",
        "z_fraction": 0.0,
        "i_fraction": 1.0,
        "p_fraction": "0.0",
        "schedule_name": "ieeezipload"}
}

gapps.get_response(topic, message, timeout = 60)
```

---

## 13.5 3.3. Query for GridLab-D Symbols File

This API call generates a file with all the XY coordinates used by GridLab-D when running a simulation.

Configuration File request key-value pair:

- `"configurationType": "GridLab-D Symbols"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "simulation_id":                        number }
```

The key-value pairs can be specified in any order.

[ ]:
```
topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "GridLAB-D Symbols",
    "parameters": { "model_id": model_mrid }
}

gapps.get_response(topic, message)
```

---

## 13.6 3.4. Query for GridLab-D Measurement Types

This API call returns a list of device names and types of available measurement in the GridLab-D format (**not** CIM classes or measurement mRIDs)

Configuration File request key-value pair:

- `"configurationType": "GridLab-D Simulation Output"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "simulation_id":                       number }
```

The key-value pairs can be specified in any order.

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "GridLAB-D Simulation Output",
    "parameters":{"model_id": model_mrid}
}

gapps.get_response(topic, message)
```

---

## 13.7 4. Querying for CIM Dictionary Files

### 13.7.1 4.1. Query for Model Dictionary

This API generates a python dictionary which maps the CIM mRIDs of objects in the power system model to names of model objects used in other simulators.

Configuration File request key-value pair:

- `"configurationType": "CIM Dictionary"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "simulation_id":                       number }
```

The key-value pairs can be specified in any order.

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "CIM Dictionary",
    "parameters":{"model_id": "_AAE94E4A-2465-6F5E-37B1-3E72183A4E44"}
```

(continues on next page)

(continued from previous page)

```
}

gapps.get_response(topic, message, timeout = 30)
```

### 13.7.2 4.2. Query for CIM Feeder Index

This API call returns a python dictionary of the available feeders in the Blazegraph database of power system models.

Configuration File request key-value pair:

- `"configurationType": "CIM Feeder Index"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "simulation_id":                       number }
```

The key-value pairs can be specified in any order.

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "CIM Feeder Index",
    "parameters":{"model_id": model_mrid}
}

gapps.get_response(topic, message)
```

## 13.8 5. Querying for OpenDSS Configuration Files

### 13.8.1 5.1. Query for all OpenDSS Files

This API call generates all the OpenDSS files necessary to solve the power system model in OpenDSS. The query returns a directory where the set of DSS files are located.

Configuration File request key-value pair:

- `"configurationType": "DSS All"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                   REQUIRED VALUES
                "model_id":                            mRID as string ,
                "directory":                           desired output directory as string␣
↪,
                "simulation_name":                      string ,
                "simulation_start_time":               epoch time number ,
```

(continues on next page)

(continued from previous page)

```
                "simulation_duration":                number ,
                "simulation_id":                      number ,
                "simulation_broker_host":             string ,
                "simulation_broker_port":             number ,
                    OPTIONAL KEYS                  OPTIONAL VALUES
                "i_fraction":                         number ,
                "p_fraction":                         number ,
                "z_fraction":                         number ,
                "load_scaling_factor":                number ,
                "schedule_name":                      string ,
                "solver_method":                      string }
```

The numeric values for the key-value pairs associated with `parameters` can be written as number or as strings. The key-value pairs can be specified in any order.

**Example: Export IEEE 13 node model with constant current loads to DSS files :**

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "DSS All",
    "parameters": {
        "directory": "/tmp/dsssimulation/",
        "model_id": model_mrid,
        "simulation_id": "12345678",
        "simulation_name": "ieee13",
        "simulation_start_time": "1518958800",
        "simulation_duration": "60",
        "simulation_broker_host": "localhost",
        "simulation_broker_port": "61616",
        "schedule_name": "ieeezipload",
        "load_scaling_factor": "1.0",
        "z_fraction": "0.0",
        "i_fraction": "1.0",
        "p_fraction": "0.0",
        "solver_method": "NR" }
}

gapps.get_response(topic, message)
```

**Note:** The output directory is inside the GridAPPS-D Docker Container, not in your Ubuntu or Windows environment. To access the files, it is necessary to change directories to those inside the docker container.

Open a new Ubuntu terminal and run: * `docker exec -it gridappsd-docker_gridappsd_1 bash` * `cd /tmp/dssdsimulation` * `ls -l`

To copy the files to your regular directory, use the `docker cp` command. For help using docker, see Docker Shortcuts on working with Docker containers.

### 13.8.2 5.2. Query for OpenDSS Base File

This API call generates a single DSS file that contains the entire power system model that can be solved in OpenDSS. The query returns the entire model DSS file wrapped in a python dictionary.

Configuration File request key-value pair:

- `"configurationType": "DSS Base"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "i_fraction":                          number ,
                "p_fraction":                          number ,
                "z_fraction":                          number ,
                "load_scaling_factor":                 number ,
                "schedule_name":                       string }
```

The numeric values for the key-value pairs associated with `parameters` can be written as number or as strings. The key-value pairs can be specified in any order.

**Example 1: Create GLM base file using nominal load values:**

[ ]:
```
topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "DSS Base",
    "parameters": {"model_id": model_mrid}
}

gapps.get_response(topic, message)
```

**Example 2: Create GLM base file using all constant current loads and hourly load curve:**

[ ]:
```
topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "DSS Base",
    "parameters": {
        "model_id": model_mrid,
        "load_scaling_factor": "1.0",
        "z_fraction": 0.0,
        "i_fraction": 1.0,
        "p_fraction": "0.0",
        "schedule_name": "ieeezipload"}
}

gapps.get_response(topic, message)
```

### 13.8.3 5.3. Query for OpenDSS Coordinate File

This API call generates a file with all the XY coordinates used by OpenDSS when plotting the feeder.

Configuration File request key-value pair:

- `"configurationType": "DSS Coordinate"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                    REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OPTIONAL KEYS                   OPTIONAL VALUES
                "simulation_id":                       number }
```

The key-value pairs can be specified in any order.

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "DSS Coordinate",
    "parameters": {"model_id": model_mrid}
}

gapps.get_response(topic, message)
```

---

### 13.8.4 5.4. Query for Y-Bus Matrix

This API call generates a Y-Bus matrix from **either** the model mRID or the simulation id.

**Note:** The GridAPPS-D platform currently does not have an in-built topology processor, so the Y-Bus matrix is NOT updated during the simulation to reflect switching actions or transformer tap changes that happen in real time.

Configuration File request key-value pair:

- `"configurationType": "YBus Export"`

The `parameters` key has a set of required values as well as some optional values:

```
"parameters": {     REQUIRED KEYS                   REQUIRED VALUES
                "model_id":                            mRID as string ,
                    OR
                "simulation_id":                       number }
```

The key-value pairs can be specified in any order.

**Example 1: Request Y-Bus for IEEE 13 node model using model mRID:**

```
[ ]: topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "YBus Export",
    "parameters": {"model_id": model_mrid}
}

gapps.get_response(topic, message)
```

**Example 2: Request Y-Bus for IEEE 13 node model with all loads set as constant current using model mRID:**

[ ]:
```
topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType": "YBus Export",
    "parameters": {
        "model_id": "_C1C3E687-6FFD-C753-582B-632A27E28507",
        "load_scaling_factor": "2.0",
        "schedule_name": "ieeezipload",
        "z_fraction": "0.4",
        "i_fraction": "0.3",
        "p_fraction": "0.3" }
}

gapps.get_response(topic, message)
```

**Example 3: Obtain Y-Bus from simulation_id:**

[ ]:
```
viz_simulation_id = "paste id here"
```

[ ]:
```
topic = "goss.gridappsd.process.request.config"

message = {
    "configurationType":"YBus Export",
    "parameters":{"simulation_id": viz_simulation_id}
}

gapps.get_response(topic, message)
```

---

**|GridAPPS-D_narrow.png|**

[ ]:
```

```

CHAPTER

FOURTEEN

# INDICES AND TABLES

- genindex
- modindex
- search