International Journal of Smart Grid and Clean Energy

A software based smart grid framework for automated decision on resource allocation, topological control using OMNET++

Eric Horton, Prakash Ranganathan*

Electrical Engineering Department, University of North Dakota, 243 Centennial Drive Stop 7165, Grand Forks, North Dakota, USA

Abstract

The futuristic smart grid require creative architectures for Information and Communication technologies (ICT) networks. One such architecture must need some sort of topological control for the next generation grid. To assess architectural impact and effectiveness, simulation models are important. Communication networks are expected to be tightly coupled-integrated element for smart grid systems. The co-existence of communication and control and their joint operation necessitate accurate modeling of communication events. While at the first glance, it is tempting to model the communication network as a black box that introduces delay/s between its input and output, the complex interactions among grid network and its components and between data sources and the network make it less tractable. The paper focuses a software-based smart grid architecture modeling using an OMNET++, a discrete event simulator. It consists of layers of independent management modules for communication, and control events that represent real-world cases using generators, circuit breakers, switches or relays, transmission lines and loads. A topology sorting algorithm is presented using modified Dijkstra's Algorithm, and several contingency scenarios for an IEEE 14 bus system was carried out emulating real electric test-bed conditions.

Keywords: Smart grid, topology control, resource allocation, OMNETT++

1. Introduction

Discrete-event simulation is a trusted platform for modeling and simulating a variety of systems. OMNeT++ [1] is a public open source, component-based and discrete event simulation environment and is becoming a very popular simulation platform especially in sensor networks, communication and networking community. Its primary application area covers the simulation of communication networks, IT systems, queuing networks and business processes as well. The authors in [2] has shown that OMNeT++ is very suitable for simulating wireless sensor networks owing to its modular structure and using NED language for ease of simulation configuration. This paper will focus on smart grid networks and environment. The model described in the paper is a real-world smart grid, and therefore takes on many assumptions and simplifications. Only real power is considered for simplicity for modeling. These assumptions aside, the model attempts to establish a clear set of commination principles between generators and loads in order to make smart, priority based decisions to reduce costs and down times. We believe that the grid framework presented would serve as a point of reference for futuristic distributed decision models.

1.1 Smart grid module creation using OMNET++

A project in OMNET++ requires a NED and INI configuration file [1]. Any behaviors associated with a module in the NED file must be defined in a corresponding C++ class. The layout and topology of all of

*Manuscript received July 11, 2017; revised October 27, 2017.
Corresponding author. *E-mail address*: prakash.ranganathan@engr.und.edu.
doi: 10.12720/sgce.6.4.233-251

the network configurations were established using the NED language unique to the OMNET++ simulation environment. The NED language is also used as a base for defining the modules within each network and the connections between each module. Networks are created using the 'network' type of the NED language and are populated with unique modules. The NED language allows for the creation of networks consisting of any number of unique modules with any number of interconnections. The NED language also encompasses the initial display state of the network and all modules/connections.

The modules within grid network can contain modules of their own (compound module) therefore becoming, in a sense, sub-networks within the over-arching network. Each module is defined in the .ned files (NED language file type) as having a discrete number of gates through which it can connect to other modules as well as parameters that can take on a variety of different types (*int, double, bool,* etc.). These modules can that be given specific behaviors by associating them with a C++ class. Gates and module parameters can be extracted from the .ned files associated with the modules within the C++ classes. Along with the ability to create modules within networks and compound modules, the NED language also supports the ability to make custom connection types to link the in/out/inout gates of each module. These connection types exhibit many characteristics similar to modules within the .ned file such as the ability to define parameters and give specific behaviors via a C++ class. By defining custom (or using built in) connection types it is possible to implement any number of connection behaviors (delay, data-rate, data-loss, or open/closed connection). Many functions and code segments within the C++ associated with each .ned file are drawn from the inclusion of the omnetpp.h library. Some of these functions will be explained within the paper, within reason. All parameters of the NED files can be manipulated in the associated INI configuration file without affecting the base code. This allows for multiple different scenarios to be set up and also allows for parameters to be changed on the fly to evaluate the effect on any given network. The authors in [3] is a flexible smart grid co-simulation framework. It allows to combine several simulators for different requirements.Using mosaik, it is possible to combine simulators for photovoltaic, power plants, households etc. to a complex smart grid simulation scenario. However, mosaik assumes a perfect link for the communication between the individual entities like houses, power plants and electric vehicles. The authors in [4] simulate the influence of cyber-attacks on a Smart Grid using Matlab/Simulink in combination with OPNET. In [5], OMNeT++ and OpenDSS(electric power Distribution System Simulator) are combined and OMNeT++ takes control of OpenDSS. In [6], OMNeT++ and OpenDSS are run in parallel and the events are synchronized at certain time slots. Both solutions are limited to OpenDSS for the grid simulation and not easily extendible for additional simulators.In [7], the authors use OMNeT++ to analyze measurements from a real testbed to evaluate the communication effort caused by using electric vehicles for stabilizing the power grid.This shows a good example of the flexibility of OMNeT++ regarding combination with other information sources.

2. Modeling IEEE 14 Bus Grid Configuration

The grid configuration referenced throughout this paper is an IEEE 14 bus system Fig. 1. The topology is captured and hard coded as seen in the NED file associated with this configuration below. Each node in the 14 bus network is referred as Process Bus (PB). There are 14 PB's. The length of transmission line (TLines) also included in the model.

Within the IEEE configuration network, 14 Process-Bus (PB) submodules are defined along with a single Control Module. The PB modules represent decision making busses within the 14 bus system. The control module can exhibit control over the PB's and their connections (TLines) however, if it becomes compromised the network will revert to a predefined, independent PB state. These modules will be further elaborated on in their specific sections of the paper. All line connections exhibit both length and status (disabled/not disabled).This network configuration ran in the Tkenv GUI is displayed in Fig. 2. PB's containing generators in the IEEE 14 Bus OMNET++ system are represented in green while loads only are represented in black. Due to the simplicity of the model, synchronous condensers are represented as Loads. The same components used to realize this IEEE 14 Bus model, can be used for many diverse topologies.

```
package Networks;
//IMPORT .ned FILE ASSOCIATED WITH SUBMODULES
import PB.PB;
import TLines.TLines;
import Control_Module.Control_Module;

network IEEE_CONFIG
{
    @display("bgb=1000,1000"); //NETWORK SIZE

    submodules:
         PB[14]: PB; //14 PROCESS-BUSSES
        Control_Module: Control_Module { //CREATE CONTROL MODULE
            @display("p=845,93");
        }
        //HARD CODED IEEE 14 BUS CONNECTIONS
    connections:
        PB[0].gate++ <--> TLines {  Length = 112; disabled = false; } <--> PB[1].gate++;
        PB[0].gate++ <--> TLines {  Length = 144; } <--> PB[4].gate++;
        PB[1].gate++ <--> TLines {  Length = 167; } <--> PB[4].gate++;
        PB[1].gate++ <--> TLines {  Length = 172; } <--> PB[3].gate++;
        PB[1].gate++ <--> TLines {  Length = 124; } <--> PB[2].gate++;
        PB[2].gate++ <--> TLines {  Length = 100; disabled = false; } <--> PB[3].gate++;
        PB[3].gate++ <--> TLines {  Length = 148; disabled = false; } <--> PB[4].gate++;
        PB[3].gate++ <--> TLines {  Length = 20; disabled = false; } <--> PB[8].gate++;
        PB[3].gate++ <--> TLines {  Length = 50; disabled = false; } <--> PB[6].gate++;
        PB[4].gate++ <--> TLines {  Length = 50; } <--> PB[5].gate++;
        PB[5].gate++ <--> TLines {  Length = 150; disabled = false; } <--> PB[10].gate++;
        PB[5].gate++ <--> TLines {  Length = 94; } <--> PB[11].gate++;
        PB[6].gate++ <--> TLines {  Length = 114; disabled = false; } <--> PB[7].gate++;
        PB[8].gate++ <--> TLines {  Length = 34; disabled = false; } <--> PB[9].gate++;
        PB[8].gate++ <--> TLines {  Length = 56; } <--> PB[13].gate++;
        PB[9].gate++ <--> TLines {  Length = 8; disabled = false; } <--> PB[10].gate++;
        PB[11].gate++ <--> TLines {  Length = 86; } <--> PB[12].gate++;
        PB[12].gate++ <--> TLines {  Length = 56; } <--> PB[13].gate++;
        //LENGTH AND STATUS FOR TLINES DEFINED WITHIN BRACKETS (disbled default = false)
```

Fig. 1. IEEE 14 Bus .ned file.

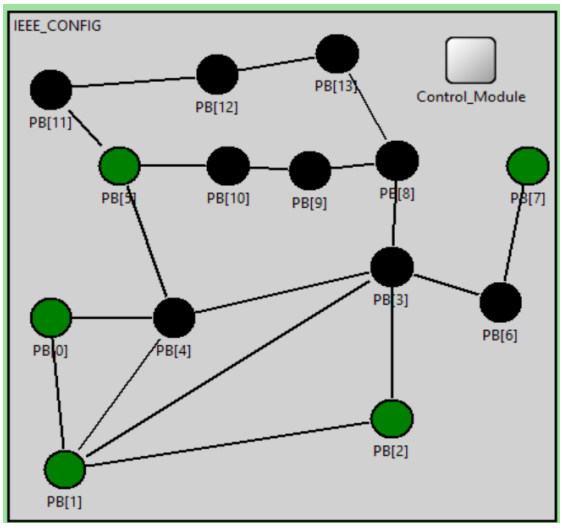

Fig. 2. OMNETT model of 14 bus network.

2.1 The Process-Bus

The Process-Bus (PB) Fig. 3 is the complex module used to represent smart, decision making busses within this grid model. The PB complex-module has no *C++ class* associated with it. The simple modules that compose the PB module are *MessageHub*, LoadProcessing, GenProcessing, Facilities, and Generator. A brief description of each simple module's purpose within the PB:

- The *Generator* module represents a single generator within each PB.
- The *Facilities* module represents a Load within the each PB
- The GenProcessing and *LoadProcessing* modules take in requests or offers from each Facility or Generator module and then make decisions about which *Gens/Loads* to service/choose.
- The MessageHub module is the avenue for external communication with other PBs in the grid. All offers and requests pass through the *MessageHub* before being sent to other PBs.Each simple module

that composes the behavior of the PB are covered in detail in their specific sections of the report.

The submodules in each PB are connected with standard connections (not TLines). This is due to the fact that the localized geographical position as well as abundant interconnections would make TLines failures/delays much less substantia.

```
import Load.Facilities;
import Load.LProcessing;
import Gen.Generator;
import Gen.GProcessing;

package PB;

module PB
{
    parameters:
        int numFacil = default(1); //Number of Facilities within PB
        int numGen = default(0); //number of generators within PB
        string color = default("black"); //default PB color
        int xpos; //x position
        int ypos; //y position
        @display("p=$xpos,$ypos;bgb=546,440"); //PB position display
        @display("b=80,,oval,$color"); //PB shape and color


    gates:
        inout gate[]; //External Gate

    submodules:
        MessageHub: MessageHub {
            @display("p=297,35");
        }
        Facilities[numFacil]: Facilities {
            @display("p=397,374,r");
        }
        LoadProcessing: LProcessing if numFacil > 0 {
            @display("p=436,140");
        }
        Generator[numGen]: Generator {
            @display("p=103,299,r");
        }
        GenProcessing: GProcessing if numGen > 0 {

            @display("p=157,154");
        }
    connections: //CONNECTIONS OF SUBMODULES WITHIN THE PB
        for i=0..numFacil-1 {
            Facilities[i].INToutrequest --> LoadProcessing.INTfacil++ if numFacil > 0;
            LoadProcessing.INTfacilPWR++ --> Facilities[i].INTpowergate if numFacil > 0
        }

        LoadProcessing.INTout --> MessageHub.LoadINTin++ if numFacil > 0;
        MessageHub.LoadINTout++ --> LoadProcessing.INTin if numFacil > 0;

        for i=0..numGen-1 {
            Generator[i].powerOut --> GenProcessing.GenInputPower++ if numGen > 0;
        }

        GenProcessing.INTout --> MessageHub.GenINTin++ if numGen > 0;
        MessageHub.GenINTout++ --> GenProcessing.INTin if numGen > 0;

        for i=0..sizeof(gate)-1 {
            MessageHub.EXTgate++ <--> gate[i];
        }
```

Fig. 3. PB .ned file.

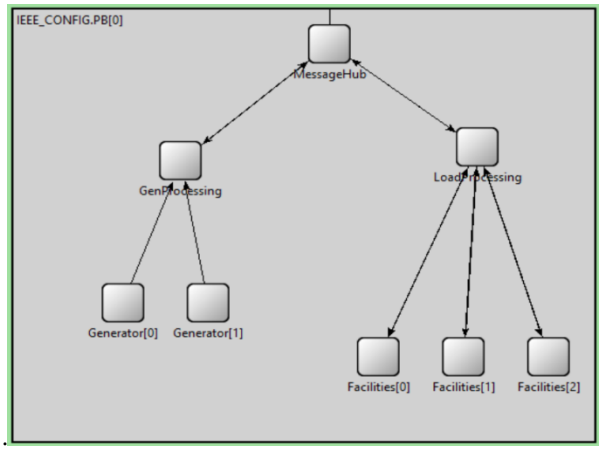

Fig. 4. Showing a process bus with two generator modules and three demand modules in the Tkenv GUI.

The .ned file associated with the PB complex module is depicted in Fig. 3 above. The parameters of the PB define the following attributes:

- *int numFacil, numGen* – Number of Facilities & Generator modules within the PB

- *string color* – Defines the color of the PB
- *int xpos, ypos* – Defines the x & y position of the PB within the Tkenv GUI

The four sub-modules within the PB (discussed previously) are declared under 'submodules'. This indicates that the Facilities and Generators are both vector submodules of size *numFacility* and *numGen* respectively. There is logic within the declaration of both *LoadProcessing* and *GenProcessing* to either include or exclude them from the PB (*e.g. if the number of facility modules = 0, then there is no need to have a LoadProcessing* submodule). The interconnections of each submodule are defined in 'connections:' and utilize the gate names already defined in the .ned files associated with each of the four simple module types. These connections are regulated by logic that prevents connection attempts to modules that do not exist in any particular PB.

2.1.1. Modeling facilities (demand) branches of Process Bus

The Facilities module only appears within PBs and never exists as an independent entity. Each PB can have any number of Facilities greater than or equal to zero. A single facility module may be used to represent the total load within a given PB or multiple facilities can be utilized to distinguish between load types (residential, medical, commercial, etc.). The .ned file for facilities serves to establish parameters and gates for the module. These parameters are similar to the variables defined in the C++ class described in next section. The .ned file and C++ code is not shown in this paper.

An important feature of the Facilities module is the priority parameter. The facility Priority is a way to layer the priority levels associated with distance, topological, relative priority *sorting* features. This is necessary because, for example, a facility with larger demand should always have a higher priority than other low level request. All priority levels used throughout the model are characterized by a higher priority equals to a lower number (e.g. priority 0 > priority 1).

2.1.2. Load Processing module

The *Load Processing* module is a significant decision maker of the PB. It has the ability to make decisions about which PB's to send offers to and which generator offers to accept. The decision making and tracking of Facility requests by the *Load Processing* module's C++ behavioral code is accomplished by working closely with an instance of the *LoadJobs* class which is owned by the *LProcessing* class. This *LoadJobs* object stores all information about a given Job, including which Generators are servicing it and Generator offers. A Job is defined as a power request from a given Facility. The .ned file & actual C++ code for the *LProcessing* module is not discussed in this paper.

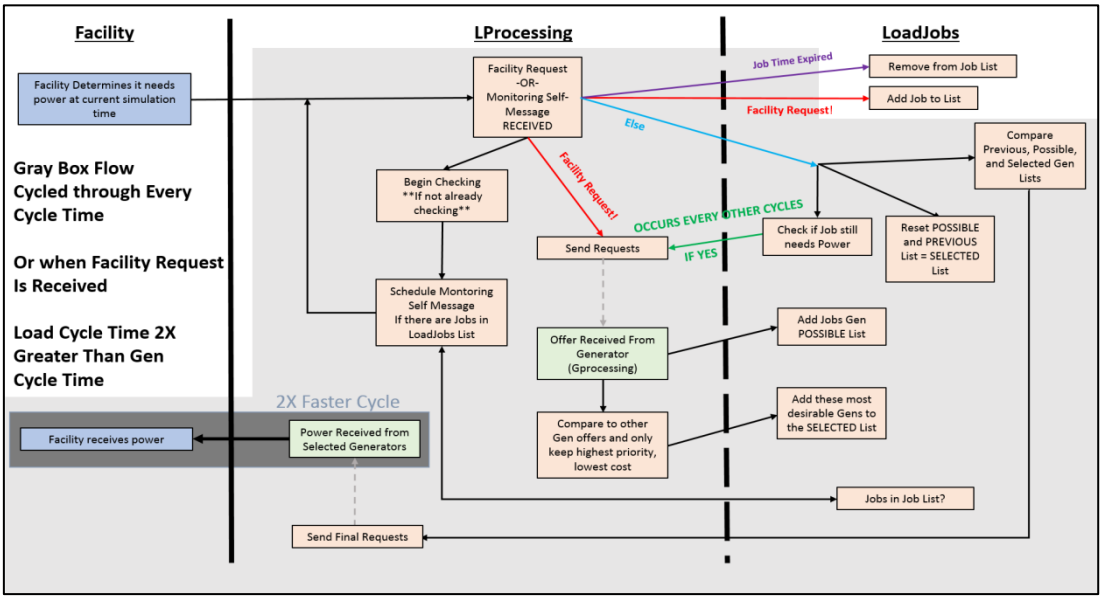

Fig. 5. Load Processing and flow diagram.

Fig. 5 shown below is a block diagram that indicate how demands are processed by load processing engine. In other words, it is a flow diagram for the requests, forwarding, and decision making capabilities of the *Facilities-LProcessing-LoadJobs* link within each PB. One of the most important characteristics of the load interactions is that while there are Jobs in the Job Lists, the *LProcessing* Module is constantly monitoring itself. Every time a monitoring cycle occurs, the *LProcessing* works with *LoadJobs* class to identify any jobs that still need power, send new requests (every other cycle), and send final requests.Decision making about offers occur as they are received.

2.1.3. Tracking Load Jobs in LProcessing module

The LoadJobs class instances are objects owned by the LProcessing module to assist with Job data keeping and decision *making*. The actual C++ code for LoadJobs will not be shown in this paper.

3. Generator Processing Functions and Module

Each PB can have any number of *Generator* modules greater than or equal to zero. One generator can be used for the entire power supply of any PB, or multiple can be used to represent different types or locations. Each generator is given a maximum amount of power that it is capable of supplying, a base cost per power unit, and an environmental priority. These attributes are included in any power offer. The generators are capable of distributing their power supply among any number of different Jobs associated with different facilities, as long as total power distributed remains less than max power. The .ned file & actual C++ code for the Generator module will not be shown in this paper

The Generator module is much less passive than the Facilities module as it owns both the Jobs and Queue object as well as checking itself every cycle and relaying required amounts of power for each Job.

The *Gen Processing* module serves as the power Generation for PB. It assists each generator in determining which jobs should be selected for servicing and which jobs should be placed in the queues as well as being the point of origin for all generator offers. The .ned file & actual C++ code for the *GProcessing* module will not be shown in this report,

3.1 JobsList and Queue Processing

In both the *Generator* and *GProcessing* sections of this paper, *JobsList* and *Queue* have been referred to by their pointer names *GJobs* and *GQueue* respectively. The nature of the *JobsList* and *Queue* classes are similar, they will be discussed in parallel. The *JobsList* class stores information about the *Jobs* that a *Generator* is servicing while the Queue class stores the Jobs that are in the queue. Both are objects owned by each *Generator* module.

It is possible for a job to be in both the *Jobs* and *Queue* lists, with some of the requested power being serviced while the rest is in queue. The changes made to the power being provided to a Job in *Jobs* list updates the future power. The *Generators* only act on the current power. At the beginning of every new cycle, current power equals future power. The actual C++ code for both the *JobsList* and *Queue* classes will not be shown in this paper.

4. Topology Control Module Using Dijkstra's Algorithm and Generator Processing Cycle

The next diagram Fig.6 depicts the flow of events for the receiving of requests, monitoring of jobs/queues, and sending of offers and power messages for the Generator-JobsList-Queue-GProcessing half of the PB. An important feature of this flow is that while there are Jobs in the Job List or Queue, the cycle will repeat (2X faster than the Load cycle). This allows Generators to keep sending out power messages while the GProcessing, Joblist, and Queue classes continually update the Job in the servicing list and Queue.

In order to proceed with the discussion of the remaining modules, it is necessary to understand the topology map the other modules use to find paths and generate priorities. This map is set up by the Topology Module located within the Control Module complex module. The Topology module utilizes the Topology class from the OMNET++ library to create a topology map of the grid. This map contains the

locations of each PB as well as the TLines linking them. The PB's are represented as nodes within the map and the TLines are represented as weighted connections. The connection weights correspond to the length of each TLine. The .ned file and actual C++ code associated with the Topology module is not included in this paper.

4.1 Relaying through messages hub

The workings of the message hub serve to relay offers, requests, and power from the *LProcs* and *GProcs* of the local and external PBs. All messages passing from the *Load branch* or *Gen Branch* of a given PB to either the local opposite branch or external PBs must pass through the *Message Hub*. The functions and parameters of the *Message Hub* serve this sole purpose (relaying data) and use communicate with the *Topology Module* to determine the routing of these messages. Because all functions of the *Message Hub* have the same basic premise (relay data/create new message/forward power), this module will not be discussed in detail in this paper.

4.2 Tracking priorities of generators and loads

The *SingularPriority* class keeps track of the individual priorities list for each *GProcessing* and *LProcessing* module. Individual priorities list refers to the combination of the Distance Only Priorities and Topology Sort Priorities without taking into account the effect of other similar *GProc* or *LProc* modules in the grid. The Distance Priorities and Topo Sort Priorities are discussed in detail in the *DistancePriority* and *TopoSort* class sections of the report. For more information about what defines a "non-individual" priority list, refer to the *RelativePriority* section of this report.

The *SingularPriority* class is pointed to by "SingPrio" in both *GProcessing* and *LProcessing* modules as each instance of these modules creates and owns their own unique SingularPriority object. The actual C++ code for the SingularPriority class is not displayed in this paper. There is no .ned file for the *SingularPriority* class (*la module*).

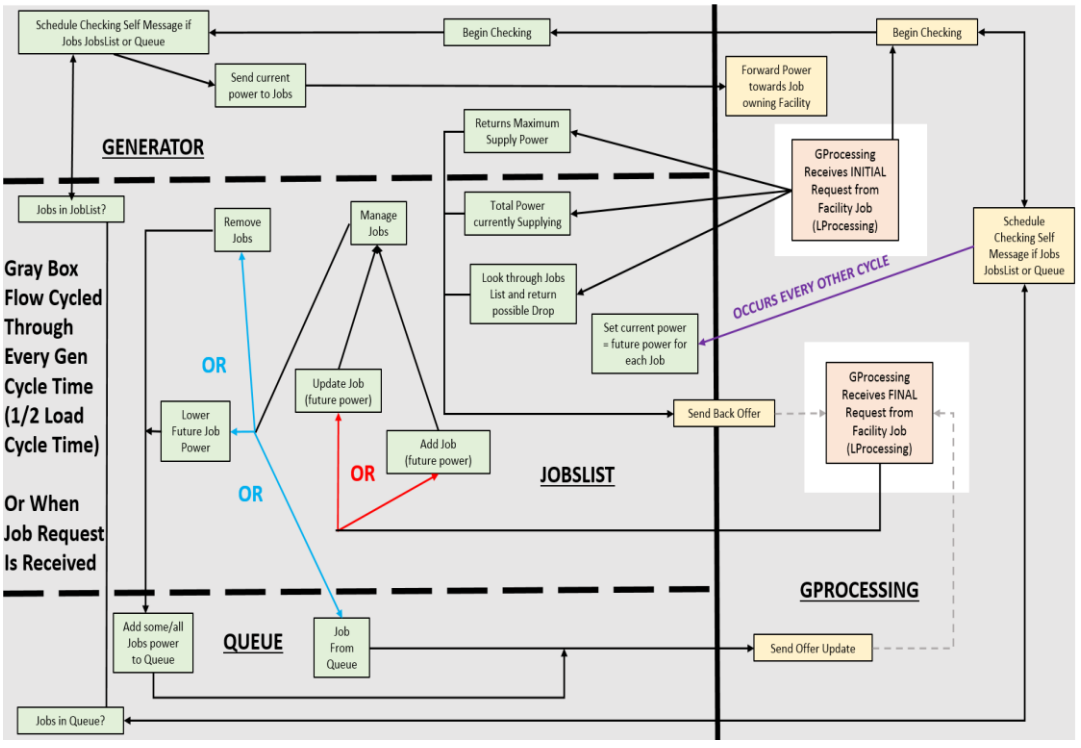

Fig. 6. Generator half of PB flow cycle.

4.3 Tracking topological Distance Priorities

The *DistancePriority* class keeps track of individual distance priorities. It obtains the distance to each PB that contains the opposite module type (*LProc* for *GProc*, *GProc* for *LProc*) and then sorts these PB's based on these varying distances. Each DistancePriority object is owned by the SingularPriority object owned by either a GProc or LProc module.

5. Topological Sorting

The topo sort class owned by each SingularPriority class essentially orders the PB's in the network by level of abstraction away from the originating PB and by number of connections. It works essentially the same as a normal topological sort except that topological sorts require a noncyclical graph, this topo sort class gets around that by assuming that all connections branch outward from the originating PB from least connections to more connections (eliminating the cyclic problem).

6. Specific Modules Forms

6.1 Control modules

The Control Module NED file stores default configuration data for the grid model including the cycle times for the generators (*base cycle time*) as well as multipliers for the other cycle times. It stores parameters (within Priority Control) to determine the type of PB priority scheme to be used (Relative, Singular, Distance Only, Topo Sort only). Along with all other NED parameters, the Control Module owns three critical modules: Topology, Connection Modifier, and Priority Control. The Topology Module was already discussed previously. Priority Control and Connection Modifier will be discussed next section. The Control Module also has a parameter to dictate whether or not *LoadProcessing* modules will use generator environmental priorities in their decision making.

6.2 Priority Control modules

The Priority Control Module will not be discussed in detail as its main purpose is to relay the priorities from other classes to the requesting LProc/GProc. It takes in the parameters from its NED file in order to conclude which type of priority to send. The logic is fairly simple and if using singular priorities the Priority Control class simply relays the priorities lists from either SingularPriority, TopoSort, or DistancePriorities. If however, the Priority Control Module parameters are set to use relative priorities. Then it will take the singular versions of either the SingularPriority, TopoSort, or DistancePriorities and add to it the inverses priorities list from all other PB's containing the same type. It will then reduce this priority list and return that to the module.

In essence this relative priority takes the priority list of the given PB and the culmination of the inverse priorities of all other similar modules (LProc/GProc against it). This however results in a very little weight on the individual priorities list in the overall summation and does not work very well for most network configurations (including the IEEE configuration). A better version would sum the inverse priorities, reduce them, and then add those to the original independent priorities list, thereby placing a larger weight on the individual priority. This implementation is described below:

STEPS FOR RETURNING RELATIVE PRIORITY

1. Determine whether to use Topo Priorities, Distance Priorities, or Combined (SingularPriorities)
2. Sum all Inverse Priorities Lists of all similar module types (LProc/GProc) other than the requesting module
3. Sort this summed list
4. Reduce the summed list
5. Add this list to the Individual Priorities List (Topo, Distance, or Combined) of the requesting module
6. Sort & Reduce this list

7. return to the requesting module

6.3 Connection modifier module

The connection modifier currently only has the capabilities to disable connections based on its NED parameters (can be specified in each INI configuration file). This allows it to disable connections between any two PB's in the network at any time. This connection disable must be between two predetermined PBs at a predetermined time prior to starting the simulation. This disabling is accomplished by calling the makeFail() function of TLines for the given connection at the given time. The Connection Modifier identifies the connecting TLine by searching through the Topology Map and finding the connection between the two specified PBs. It then schedules a self-message to be received at the specified time. When the self-message is received, the makeFail() function is called to disable the TLine.

6.4 TLines

The TLines connection type is used to create the connections between any two separate PBs within each grid configuration. The length of each TLine must be established before runtime in the NED file for the given grid configuration. This length is used to set the message delay of each TLine. Each TLine can be disabled (no messages will pass through) either before starting the simulation in the NED file of the grid configuration or by using the Connection Modifier module to disable during runtime. If a connection is disabled during runtime, the makeFail () function of TLines is called. This function sets the connection to disabled and updates the display. A disabled TLines connection is displayed as a broken black line while enabled connections are simple black lines. NOTE. When a TLine is disabled, it does not allow any messages to pass through it (they are deleted), however the animation of the GUI still shows the message go through the TLine, but it is deleted at the very end of its travel. This can be confusing as it appears the message went through, but it did not.

6.5 PB status

This class is owned by each *Message Hub* Module (simply because each PB must have a *Message Hub*). It updates the tool tips of the PB, GProc, LProc, Facilities, and Generator modules. These tooltips can be accessed by hovering the mouse over any of these modules during runtime.

The tool tip display updates relay the following information for each module:

- PB: Total Number of Generators and Facilities as well as total power statistics
- GProcessing: All Generators with power stats
- *Generato*r: All jobs in service and in queue with power stats and facility statistics
- LProcessing: Each Job and power stats
- *Facilities*: Jobs associated with that individual facility as well as power stats and servicing generators info

The updating of the display strings (tooltips) during runtime requires string streams and reading of parameters from a variety of variables from all involved modules. Despite this, it does not affect the actual functioning of the grid model and so will not be discussed in detail. The continual updating of the tooltips for every module significantly reduces the performance of the model. If the desire is to run in real-time (real-time scheduler) or run at optimal speed. Then turn off the tool tip updating by setting Control Module display update parameter to false. This can be done directly in the NED file or configuration to configuration in the INI file.

7. Contingency Cases on Generator, Load Selections and Decision Schemes

The following scenarios show some of the capabilities of the grid model. They are presented in a very specific manner with small custom networks built for each scenario, however each of these example behaviors apply to any network with any level of complexity. Many times throughout the case explanations the Facility/LoadProcessing combination is simply referred to as load. Despite this

nomenclature it is still a facility obtaining the power and LProcessing making the decisions.

7.1 Simple power surplus case

(A) Load selects best gen

This scenario shows a basic grid network with two generator containing PBs (PB 1 & 2) and one facility containing PB (PB 0). Both generators have more than enough capability to service the facility's job, however the generator at PB 1 has better environmental priority (environmental consideration is turned on) and lower cost.

Since there is a surplus of available power the facility at PB 0 simply chooses the more desirable of the two generators. PB 1 generator begins supplying power and PB 2 generator remains idle. This scenario is described in the following Fig. 7, Fig.8 and Fig. 9.

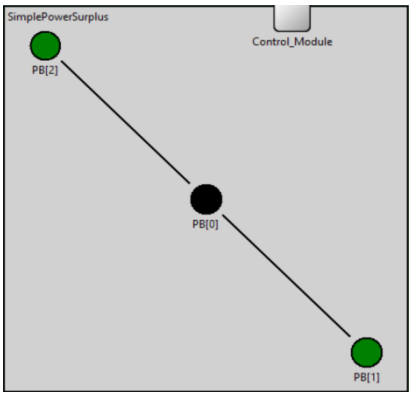

Fig. 7. Grid configuration at t = 0.

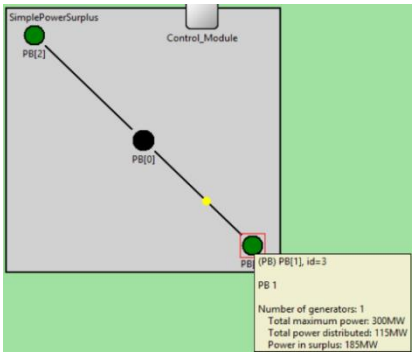

Fig. 8. PB 0 obtaining power from PB 1 (better gen).

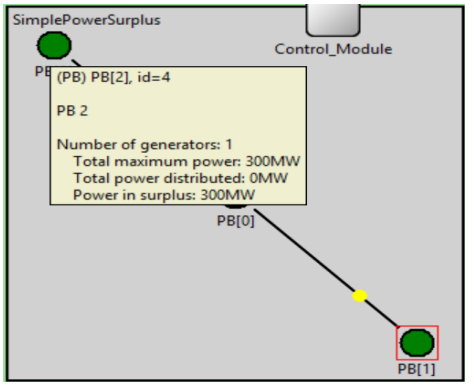

Fig. 9. PB 2 gen is idle (tooltip).

(B) Load splits power amongst multiple gens

This scenario is very similar to the previous scenario (see previous explanation), except that neither generators are capable of servicing the total power need of the facility at PB 0 alone (they each have half that capacity). In response the Load (facility/LProc combo) at PB 0 chooses both generators at PB1 & PB2 to obtain the necessary power. The following Fig. 10 and Fig. 11 showcase these events.

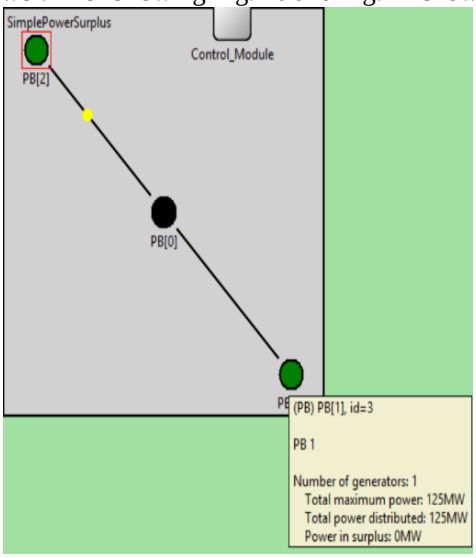

Fig. 10. PB 0 obtaining power from both PB 1.

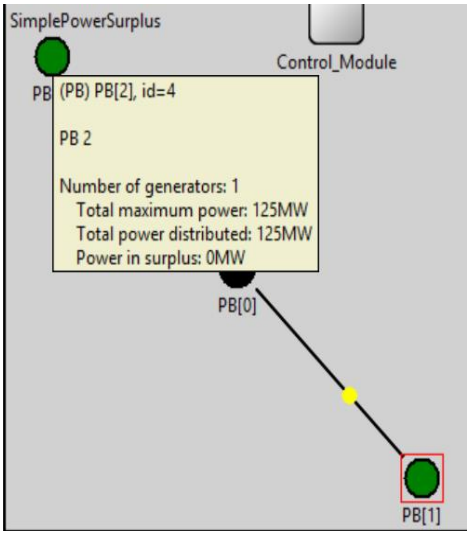

Fig. 11. Tooltip for PB 2.

7.2 Simple power deficit cases

(A) Gen chooses best load

This scenario shows a basic grid network with two facility containing PBs (PB 1 & 2) and one generator containing PB (PB 0). The single generator does not have enough capability to service more than one of the facility jobs at a time. In this scenario both facilities have identical parameters except PB 1 is much closer to PB 0 resulting in a lower PB singular priority (relative does not applicable; only one generator PB). Therefore PB 0 Gen chooses the job associated with PB 1 facility and places the other job in queue. Fig. 12 and Fig.13 display these events.

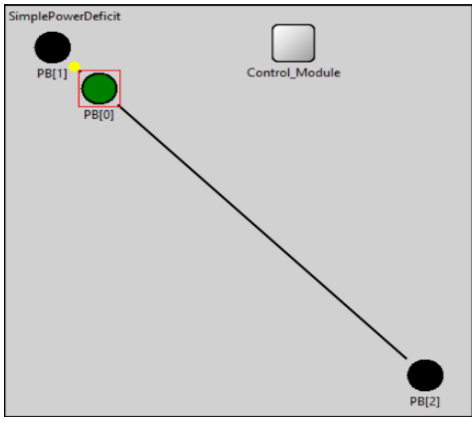

Fig. 12. PB 0 gen chooses PB 1 (higher priority) to supply power.

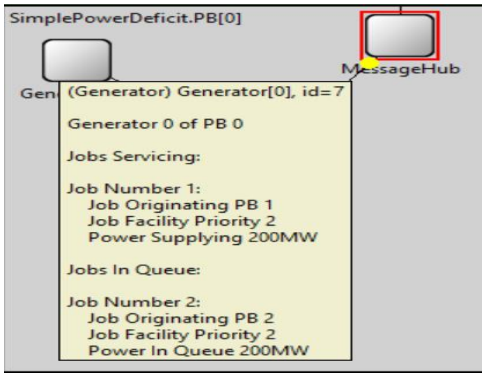

Fig. 13. PB 0 generator tooltip.

It can also be seen from the Generator tool tip (Fig. 31) that the generator is servicing the job from PB1 and has the job from PB2 in queue.

(B) Gen splits total available power amongst multiple loads

This scenario is very similar to the last (see previous scenario) except that now the generator at PB 0 has enough available power (300 total) to supply all of one of the jobs (200) and part of the other. The generator still chooses PB 1 to fulfill the total need, except now it also supplies 100 to PB 2 and only places 100 of that job into the queue. This shows that the Generators are capable of servicing multiple jobs from multiple PBs/Facilities. Fig. 14- Fig. 17 showcase these events.

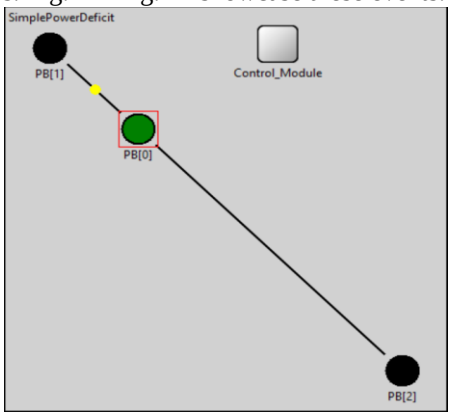

Fig. 14. PB 0 Supplies 200 to PB 1 (higher priority).

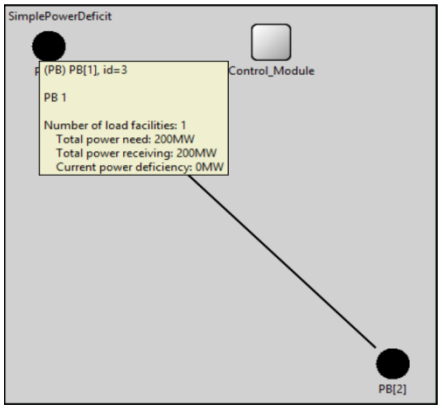

Fig. 15. PB 1 tooltip.

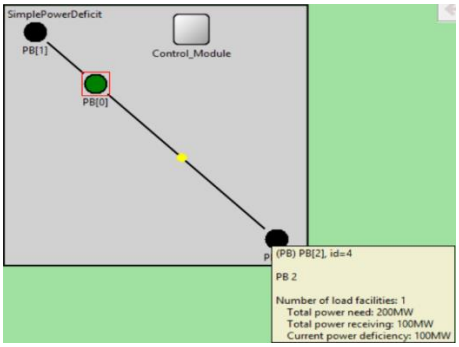

Fig. 16. PB 0 gen supplies 100 to PB 0.

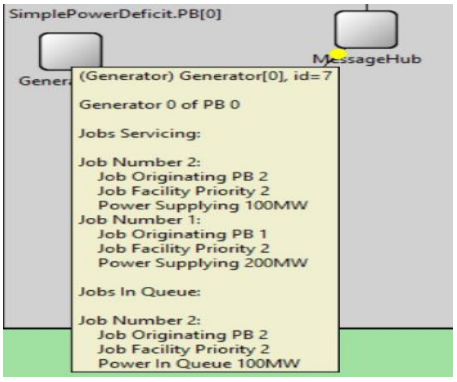

Fig. 17. Tooltip for PB 0 generator.

7.3 Time out case for job requests

(A) Gen gets job from queue

This scenario involves one PB (PB 0) containing a generator (max power = 200MW) and two PBs (PB 1 & PB2) containing facilities (power need = 200). The facility in PB 1 has a facility priority of 0 (highest priority, say a hospital) while the other (in PB2) has a facility priority of 3. Because of the difference in facility priority, the single generator must choose PB1's job regardless of other parameters. However, PB1's facility's job has a very short need time. Once the need time expires and the job terminates, the generator searches its queue and sees the job associated with PB2's facility and begins servicing. This scenario is depicted in the following Fig. 18- Fig.21:

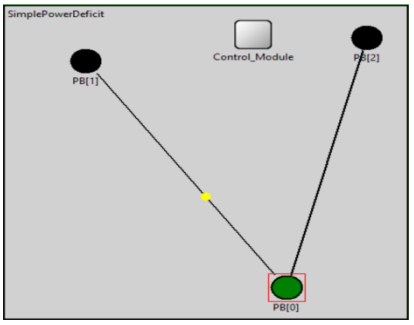

Fig. 18. PB 0 supplies power to PB 1.

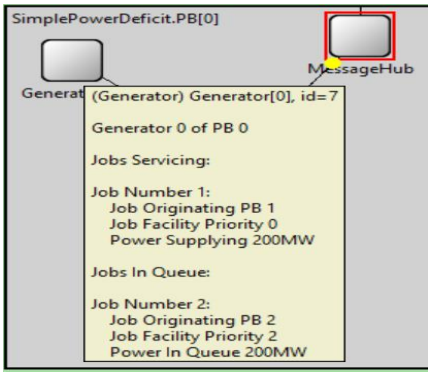

Fig. 19. PB 0 gen tooltip (prior to job termination.

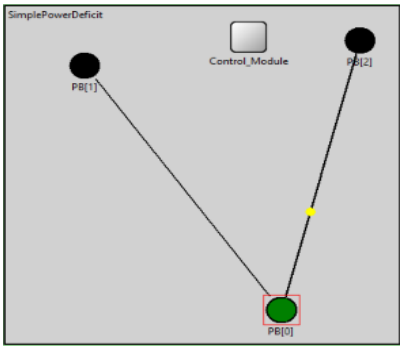

Fig. 20. PB 1 Job terminated, PB 0 selects queued job PB 2.

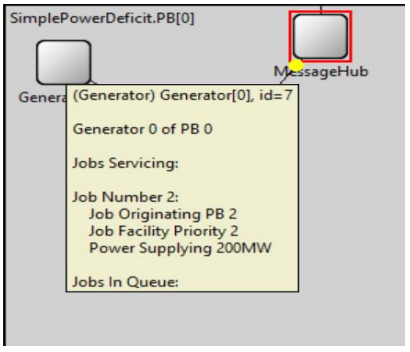

Fig. 21. PB 0 gen tooltip (after termination).

(B) Load re-evaluates best gen

This scenario involves two generator (max power = 200) containing PBs (PB 0 & 1) and two facility (power need = 200) containing PBs (PB 2 & 3). The generator at PB 0 has much better parameters than the one at PB 1, hence loads want to choose that generator to fulfill their jobs. However, the facility at PB 2 has a facility priority of 0 (critical) while PB3's facility is only 2. Because of this PB 0's generator chooses the job from PB 2. PB 3's Load Processing settles for the generator at PB 1 to fulfill its need. The facility at PB 2 has a short need time and therefore the job quickly expires. When this happens, the generator at PB 0 (better generator) looks into its queue and sees PB 3's job. It sends an offer update and PB 3 re-evaluates the generators servicing it and switches to PB 0's generator from PB 1's generator. The following Fig. 22- Fig. 26 summarize the scenario in OMNET++ simulation.

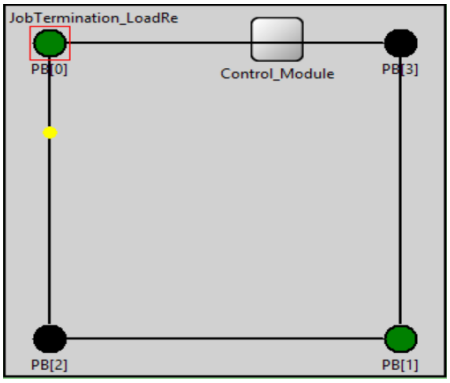

Fig. 22. PB 0 sends to better load PB2.

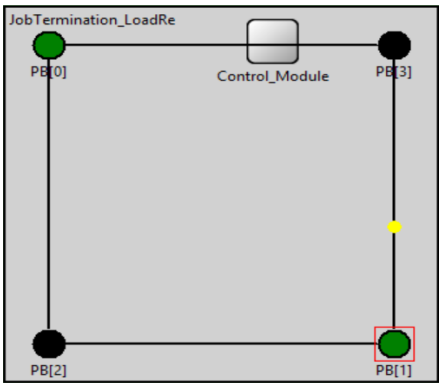

Fig. 23. PB 1 supplies worse load PB 3.

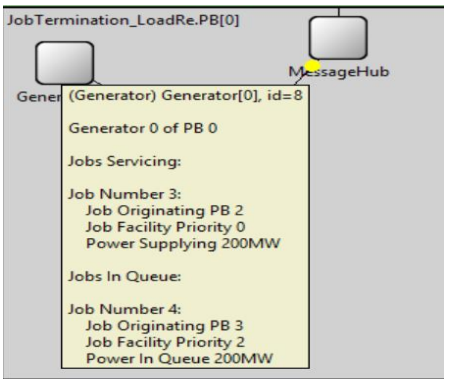

Fig. 24. PB 0 gen tooltip.

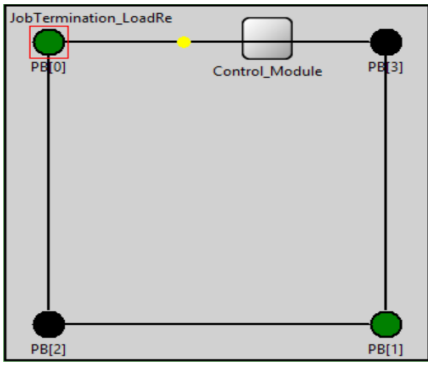

Fig. 25. PB 0 supplies PB 3 after job termination.

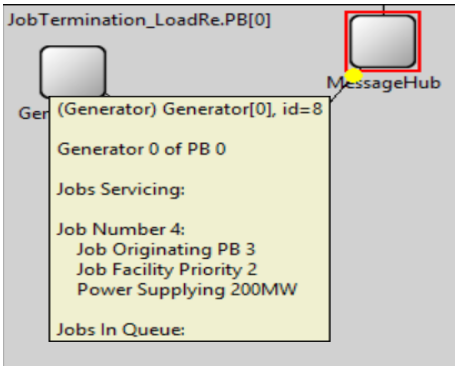

Fig. 26. PB 0 gen tooltip.

7.4 Connection failures

(A) Load acknowledges failure & automated alternative selection

In this scenario, there are two PB's with generators (PB 1 & PB2) and one PB with a facility (PB 0) with a demand of 115 MW of power. One of the generators (at PB 2) has much better power capabilities, environmental priority, lower pricing and with closer proximity to PB0. The other generator (at PB 1) is far less desirable. The *Load Processing* of PB 0 decides to choose the better generator (as it should), however at time t = 2s, the connection between PB 0 and PB 2 fails. After the selected generator fail to send the committed power amount for *three* cycles, the *Load Processing* realizes that something has gone wrong and begins to search for a different source of power. It then selects the readily available generator at PB 1 as this is now the sole source. The following Fig. 27-Fig. 30 summarize the scenario in OMNET++ simulation.

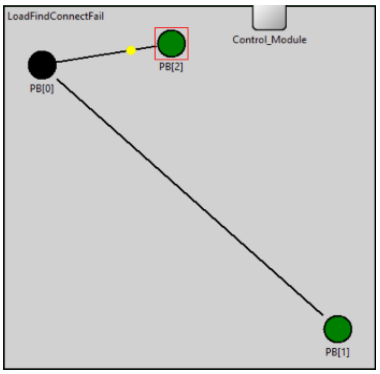

Fig. 27. Time t = 0.

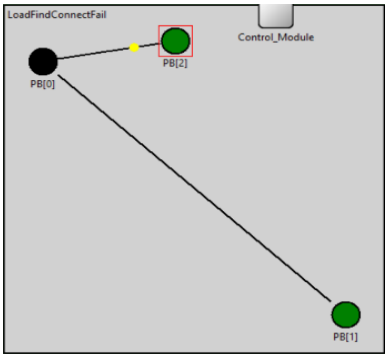

Fig. 28. PB2 sending power.

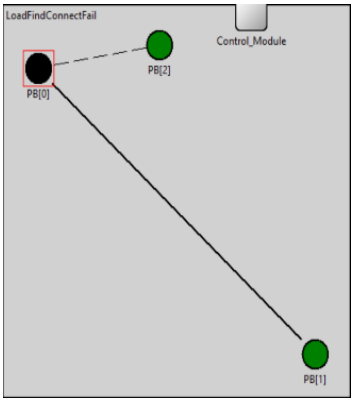

Fig. 29. Connection failure at t=2.

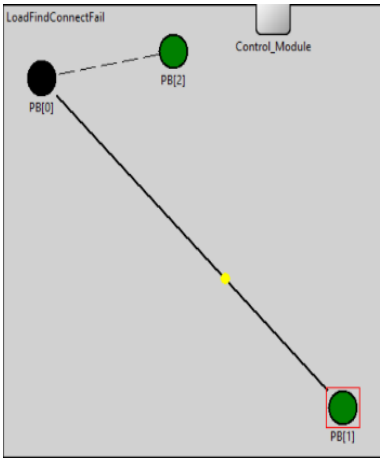

Fig. 30. Automated power transfer from PB1.

(B) Gen acknowledges failure

The other scenario is similar to the previous case, except roles of the facility containing generator and PBs are switched. There is a single generator (PB 0) capable of supplying enough power for one of the two loads (PB 1 & PB2). When the connection to the better facility (PB 2) fails, the generator realizes that it is not sending power through after three missed cycles and looks into its queue to switch to the only job associated with a facility it can reach. The following Fig. 31- Fig. 33 summarize the scenario in OMNET++ simulation.

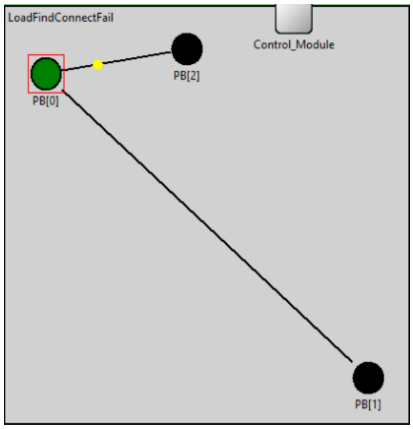

Fig. 31. PB 0 Gen supplies to better load PB 2.

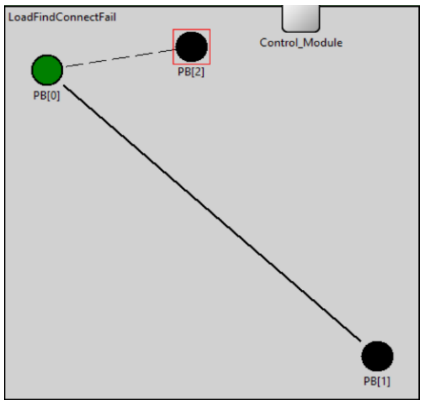

Fig. 32. Connection breaks.

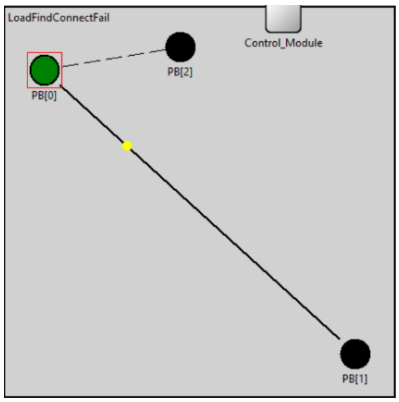

Fig. 33. PB 0 gen switches to supply to PB 1.

7.5 Other cases

(A) All power needs satisfied localy

The following scenario demonstrates the ability of a PB with both a generator and facility to satisfy all power distribution/consumption locally. There are three PBs in the network, all of which contain one facility (power need = 150) and one generator (max power = 100), and each of these PBs is not sending or receiving any power from external PB sources. The following Fig. 34- Fig. 35 summarize the scenario in OMNET++ simulation.

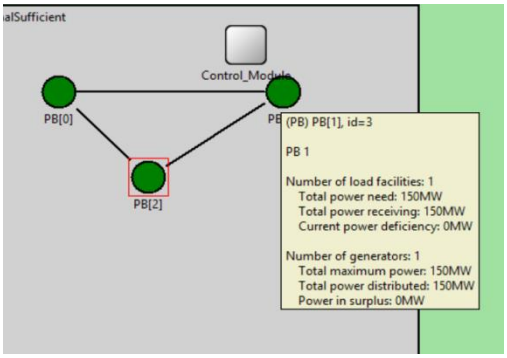

Fig. 34. 3 Self Sufficient PBs in grid.

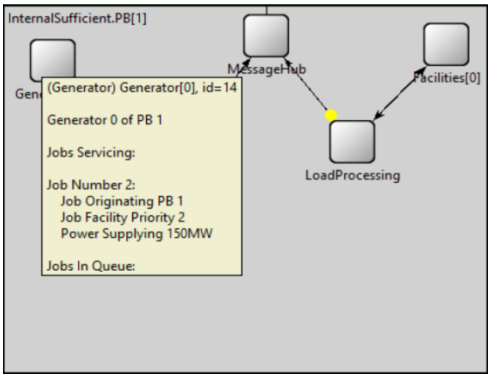

Fig. 35. Tooltip for PB 1 gen.

8. Conclusion

The paper investigated a software-based framework for smart grid modeling and control. A topology-sorting algorithm is successfully evaluated for an IEEE 14 bus system for routing supply to demand sites under several contingency cases. Our model provides a vast amount of parameters adjustable for conducting a comprehensive study under IEEE 802.15.4. The default values for the majority protocol parameters are stored in a single C++ header file. The parameters and characteristics of nodes in the OMNET++ network can easily be modified during the simulation. This is done via corresponding NED file for each of the modules (e.g., transmitterPower, and sensitivity), so that users can change them conveniently in the simulation configuration file omnetpp.ini.

References

[1] OMNeT++. [Online]. Available: http://www.omnetpp.org

[2] Mallanda C, Suri A, Kunchakarra V, Iyengar SS, Kannan R, Durresi A, Sastry S. Simulating wireless sensor networks with omnet+. *IEEE Computer*, 2005.

[3] Mosaik is a flexible smart grid co-simulation framework. [Online]. Available: http://mosaik.offis.de.

[4] Sadi MAH, Ali MH, Dasgupta D, Abercrombie RK. Opnet/simulink based testbed for disturbance detection in the smartgrid. In: *Proc. of the 10th Annual Cyber and Information Security Research Conference*, 2015:17:1–17.

[5] Levesque M *et al.*. Communications and power distribution network co-simulation for multidisciplinary smart grid experimentations. In: *Proc. of the 45th Annual Simulation Symposium*, 2012:2:1–2:7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2331751.2331753

[6] Bhor D, Angappan K, Sivalingam K. A co-simulation framework for smart grid wide-area monitoring networks. In *Proc. International Conference on Communication Systems and Networks*, 2014:1–8.

[7] Bocker S, Lewandowski C, Wietfeld C, Schluter T, Rehtanz C. Ict based performance evaluation of control reserve provision using electric vehicles. In: *Proc. Innovative Smart Grid Technologies Conference Europe*, 2014:1–6.