NetSim®

Accelerate Network R & D

MATLAB & Python Interface for RPL DODAG Visualization

Version 15.0

A Network Simulation & Emulation Software

By

TETCOS LLP

Software: NetSim Standard v15.0 / v15.1, Visual Studio 2022, MATLAB R2019 and higher (MATLAB backend) or Python 3.13 (Python backend)

Project Download Link:

https://github.com/NetSim-TETCOS/RPL_DoDAG_Visualization_v15.0/archive/refs/heads/main.zip

Follow the instructions specified in the following link to download and setup the Project in NetSim:

https://support.tetcos.com/en/support/solutions/articles/14000128666-downloading-and-setting-up-netsim-file-exchange-projects

1 Introduction

RPL (Routing Protocol for Low-Power and Lossy Networks) is a routing protocol for wireless networks with low power consumption and generally susceptible to packet loss. It is a proactive protocol based on distance vectors and operates on IEEE 802.15.4, optimized for multi-hop and many-to-one communication, but also supports one-to-one messages.

This protocol is specified in RFC 6550 with special applications in RFCs 5867, 5826, 5673 and 5548. RPL can support a wide variety of link layers, including those with limitations, with potential losses or that are used in devices with limited resources. This protocol can quickly create network routes, share routing knowledge and adapt the topology in an efficient way.

RPL creates a topology similar to a tree (DAG or directed acyclic graph). Each node within the network has an assigned rank (Rank), which increases as the teams move away from the root node (DODAG). The nodes resend packets using the lowest range as the route selection criteria.

The DODAG formed as part of RPL protocol in NetSim can be visualized by interfacing NetSim with MATLAB or Python, selected at runtime.

2 Choosing a Visualization Backend

The v15 workspace supports two interchangeable ways to visualize the DODAG as it forms: MATLAB (via `PlotDAG.m`) or Python (via `DoDAG_Plot.py`, using matplotlib). Only one backend needs to be set up to run a simulation – install whichever you have available, or both if you want the runtime choice.

The backend is selected once per simulation run, via a console prompt shown when the simulation starts. This works the same way whether `NetSimCore.exe` is run directly from a terminal or launched from the NetSim GUI:

```
Choose DODAG visualization backend:
  1) Python
  2) MATLAB
Enter choice [1]:
```

If no valid choice is entered – for example, in an unattended or scripted run where the console is not interactive – the Python backend is used by default.

3 Setting Up the MATLAB Backend

DODAG Visualization is implemented in NetSim by interfacing with an external plotting process, using either MATLAB (via `PlotDAG.m`) or Python (via `DoDAG_Plot.py`). Only one backend is needed to run a simulation – set up whichever you have available, or both if you want the runtime choice.

The codes required for the DODAG Visualization done in MATLAB are written to a `PlotDAG.m` file and this file is available in the MATLAB folder under `bin_x64` of `RPL_DoDAG_Visualization_IoT_Workspace`.

Steps to simulate:

1. Add the following MATLAB install directory path in the Environment PATH variable
   - `<MATLAB_INSTALL_DIRECTORY>\bin\win64`
   - For eg: `C:\Program Files\MATLAB\R2020b\bin\win64`

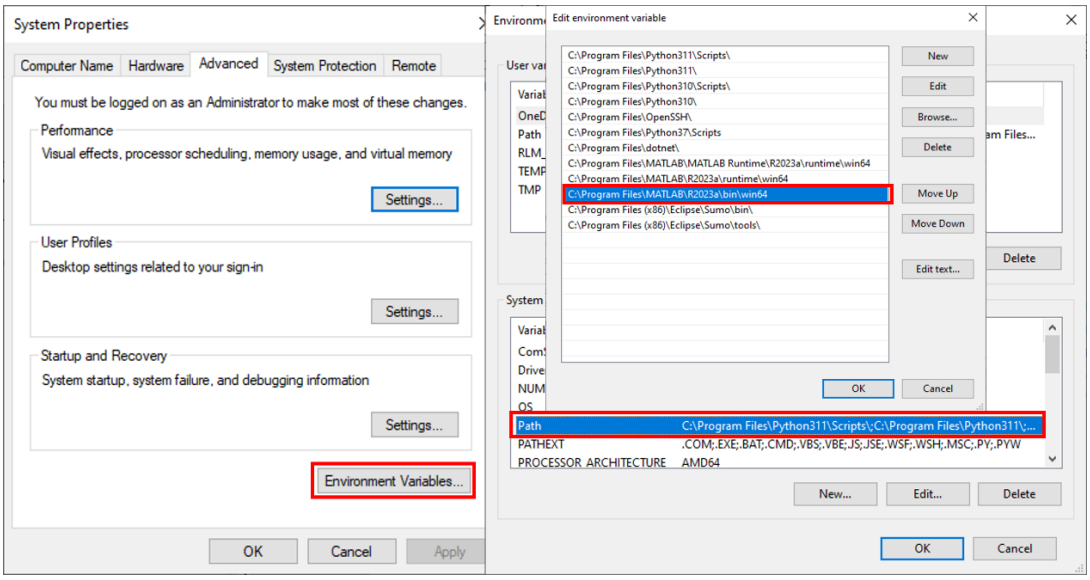

Figure 3-1: *Environment variable PATH.*

If the machine has more than one MATLAB installed, the directory for the target platform must be ahead of any other MATLAB directory (for instance, when compiling a 64-bit application, the directory in the MATLAB 64-bit installation must be the first one on the PATH).

2. Open Command prompt as admin and execute the command "matlab -regserver". This will register MATLAB as a COM automation server and is required for NetSim to start MATLAB automation server during runtime.

4 Setting Up the Python Backend

The Python backend requires Python 3.13 specifically. The bundled `netsim_socket_client.pyd` extension module (in `NetSim_Python_Interface_Client`) is built against `python313.dll` and will not load under other Python versions installed on the same machine.

No manual registration step is required – unlike MATLAB's `matlab -regserver`, NetSim launches `DoDAG_Plot.py` automatically once Python is chosen at the prompt. On first run, the script installs matplotlib via pip if it is not already present.

The plotting script is available at `bin_x64\DoDAG_Plot.py`. It connects back to NetSim over the built-in Python socket interface (`NetSim_Python_Interface.h`) and redraws the DODAG on every update, saving each frame to `bin_x64\DoDAG_Plot_latest.png` as well as showing it live.

5 Example

The `RPL_DoDAG_visualize_IoT_15` workspace comes with a sample network configuration that is already saved. To open this example, go to **Your work** in the Home screen of NetSim and click on **RPL_DODAG_Example** from the list of experiments.

The saved network scenario consists of:

- 5 Wireless Sensor Nodes
- 1 LOWPAN Gateway
- 1 Router
- 1 wired node

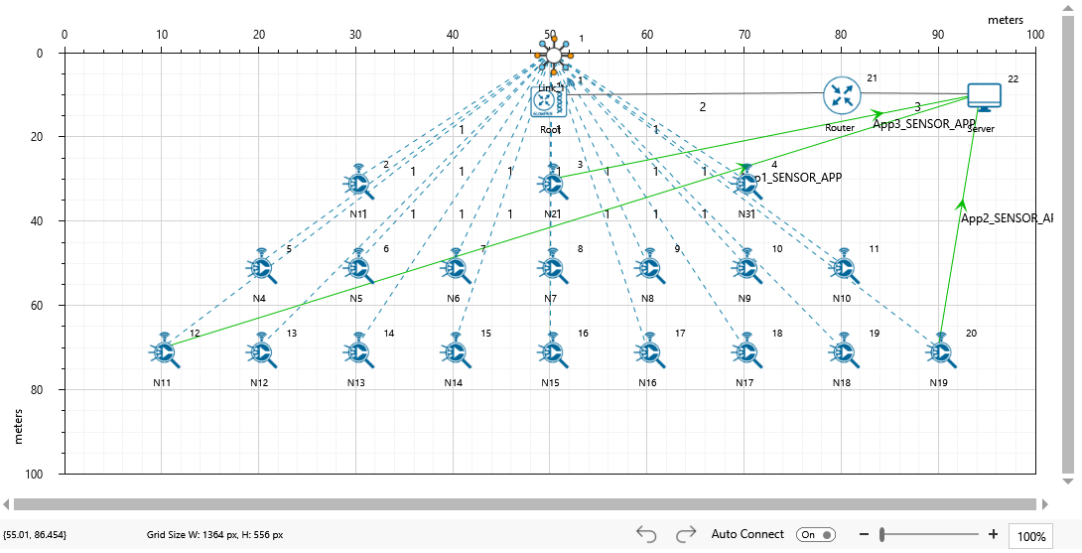

Figure 5-1: *Network Scenario.*

Run Simulation. When prompted in the console, enter 1 to visualize with Python or 2 for MATLAB (default is Python if nothing is entered).

6 Results and Discussion

A plot will open, showing the DODAG when the simulation is started, and the first route is formed between the root device and the sensors. And the DODAG will be dynamically updated. With the MATLAB backend selected, the DoDAG gets plotted in MATLAB as shown below:

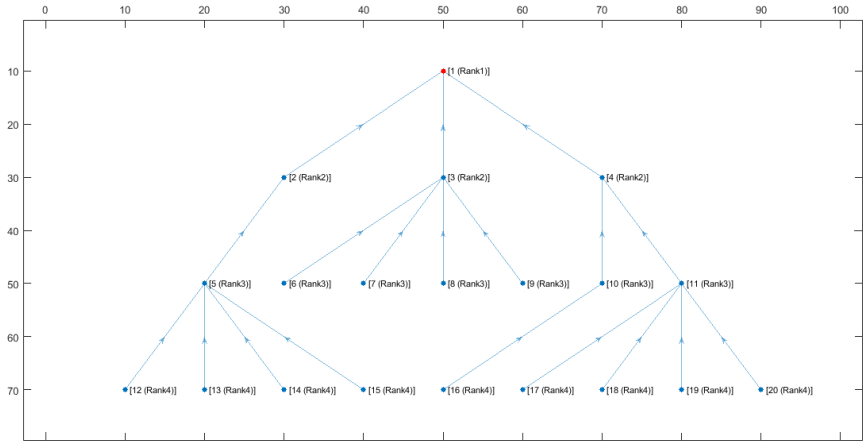

Figure 6-1: *Plot of DODAG Formation in MATLAB.*

With the Python backend selected, the same DODAG is plotted using matplotlib. The root node is highlighted as a red square and labeled ROOT, so it is immediately clear which node the rest of the tree connects back to:

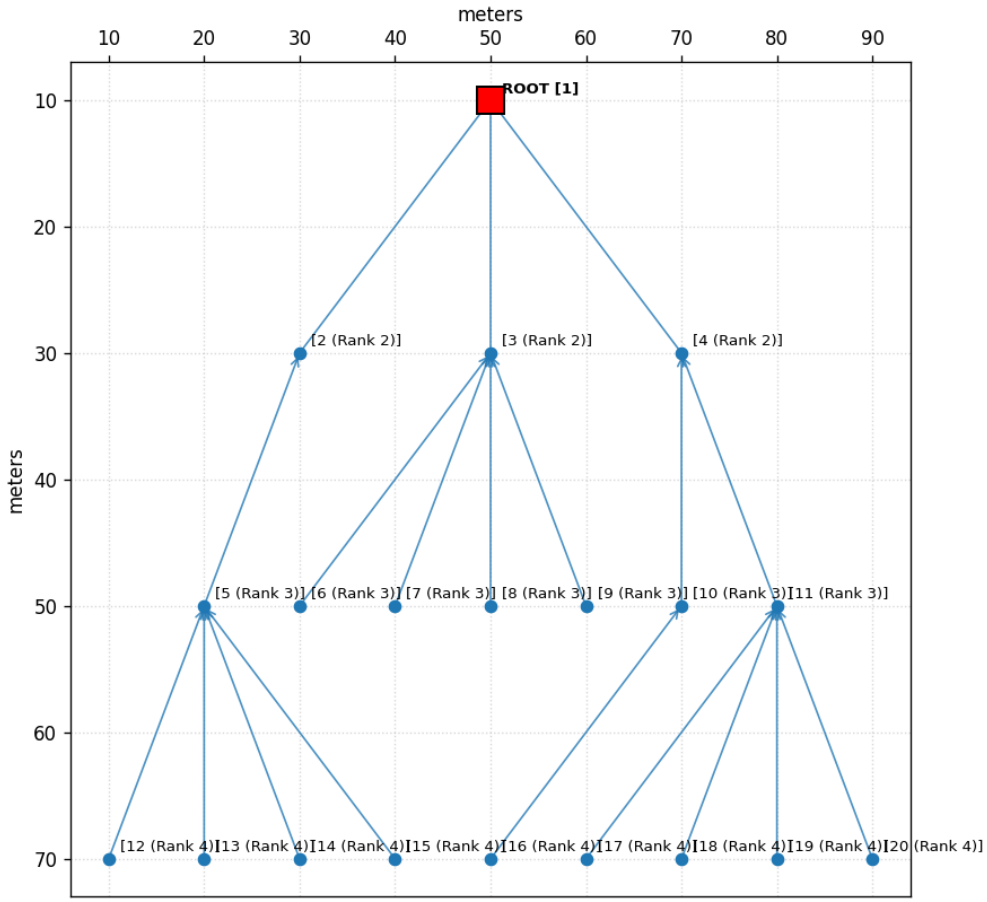

Figure 6-2: *Plot of DODAG Formation in Python (root node highlighted).*

7 Appendix: NetSim Source Code Modifications

The code below implements the runtime backend choice and both plotting paths (Python and MATLAB), replacing the MATLAB-only `DoDAG_Plot.c` from the v14.4 project. Only the parts that changed structurally are shown; routine boilerplate (#includes, the small string-buffer helper used to safely build the MATLAB assignment strings) is omitted for brevity.

7.1 New prototypes added to RPL.h, within RPL project

```
/* New prototypes added to RPL.h, within RPL project */

//DODAG visualization (DoDAG_Plot.c) - backend chosen at runtime (Python or MATLAB)
void fn_netsim_dodag_interface_start();
double fn_netsim_dodag_run();
double fn_netsim_dodag_finish();
```

7.2 DoDAG_Plot.c, within RPL project – backend selection and dispatch

```
typedef enum { DODAG_BACKEND_PYTHON = 1, DODAG_BACKEND_MATLAB = 2 } DODAG_BACKEND;
static DODAG_BACKEND g_backend = DODAG_BACKEND_PYTHON;

void fn_netsim_dodag_interface_start()
{
    int choice = 0;
    fprintf(stderr, "\nChoose DODAG visualization backend:\n 1) Python\n 2) MATLAB\
        nEnter choice [1]: ");
    if (scanf("%d", &choice) != 1)
    {
        choice = 1;
    }

    if (choice == 2)
    {
        g_backend = DODAG_BACKEND_MATLAB;
        /* launch MatlabInterface.exe, netsim_matlab_interface_configure(pszAppPath) */
    }
    else
    {
        g_backend = DODAG_BACKEND_PYTHON;
        /* launch DoDAG_Plot.py, g_python_handle = Init_netsim_python_Interface() */
    }
}

double fn_netsim_dodag_run()
{
    /* ... compute cnt, adjacency, nID, nRank, Xc, Yc, H - same topology logic as
       the v14.4 project, using is_rpl_configured() / rpl_node_is_root() (see note
       below) instead of matching device type strings ... */

    if (g_backend == DODAG_BACKEND_MATLAB)
    {
        send_to_matlab(cnt, adjacency, nID, nRank, Xc, Yc, H);
    }
    else
    {
        send_to_python(cnt, adjacency, nID, nRank, Xc, Yc, H);
    }

    /* ... free buffers ... */
}
```

`send_to_python()` builds a typed message (adjacency, positions, ranks, root index) via `NetSim_Python_Interface.h` and sends it over the socket. `send_to_matlab()` builds the same MATLAB ASCII assignment strings (theArray/Xc/Yc/nID/nRank/H) the v14.4 code did and calls `PlotDAG.m`.

7.3 Changes to fn_NetSim_RPL_Init() and fn_NetSim_RPL_Finish(), in RPL.c, within RPL project

```
/* Changes to fn_NetSim_RPL_Init() and fn_NetSim_RPL_Finish(), in RPL.c, within RPL
    project */
```

```
_declspec (dllexport) int fn_NetSim_RPL_Init(struct stru_NetSim_Network *
    NETWORK_Formal,
                                             NetSim_EVENTDETAILS *pstruEventDetails_Formal,

                                             char *pszAppPath_Formal,
                                             char *pszWritePath_Formal,
                                             int nVersion_Type,
                                             void **fnPointer)
{
    fn_netsim_dodag_interface_start();
    fn_netsim_dodag_run();
    return fn_NetSim_RPL_Init_F();
}

_declspec(dllexport) int fn_NetSim_RPL_Finish()
{
    fn_netsim_dodag_finish();
    return fn_NetSim_RPL_Finish_F();
}
```

7.4 Change to choose_parents_and_siblings(), in Neighbor.c, within RPL project

```
/* Change to choose_parents_and_siblings(), in Neighbor.c, within RPL project */

rpl_add_route_to_parent(d, dodag->pref_parent->nodeId);
fn_netsim_dodag_run();

for (i = 0; i < rpl->neighbor_count; i++)
{
    PRPL_NEIGHBOR neighbor = rpl->neighbor_list[i];
    ...
}
```

Root/member detection uses RPL's own semantics (`is_rpl_configured()` and `rpl_node_is_root()`, both from `RPL.h`) rather than matching device type strings such as "SENSOR"/"SINKNODE". Not every NetSim device configured as the RPL root is necessarily typed SINKNODE – for example, a 6LoWPAN Gateway device is typically typed NODE – so the RPL-level check is what makes this work across different network scenarios.