RelIoT - A Reliability Simulation Framework for ns-3

Introduction

RelIoT is built upon the ns-3 network simulator 3.27 version. It adds 5 modules to the ns-3 source modules to enable power, performance and reliability evaluation: Power Module, Performance Module, Temperature Module, Reliability Module, and CPU Energy Model. The framework is implemented into the ‘src/reliability’ folder, except for the CPU Energy Model in ‘src/energy’ folder.

Note: The first thing one needs to do before starting to look at this documentation is to understand the core concepts and abstractions in the ns-3 system. For this, please refer to:
ns-3 Conceptual Overview
ns-3 Model Library

Getting Started

Supported Platform

The supported platform for ns-3 is listed in its official Installation Guide of ns-3. RelIoT is developed on Ubuntu 18.04 LTS.

Prerequisites

- Git is required to fetch the source code. Use the following command to install:

```
sudo apt install git
```

- **Minimal requirements for C++ users (release):**

```
apt-get install gcc g++ python
```

- **Minimal requirements for Python users (release):**

```
apt-get install gcc g++ python python-dev
```

While our example script is provided in C++, it is also possible to work with python. For developmental goals, extra tools are needed. Please refer to the official Installation Guide of ns-3.

Installation and Building

Clone the repository from github:

```
git clone https://github.com/UCSD-SEELab/RelIoT.git
```

Enter the ./reliabilitysim/ns-allinone-3.27 directory, and build the simulator:

```
cd RelIoT/ns-allinone-3.27
./build.py
```

Once the project has been built you will interact directly with *Waf* and we do it in the ns-3.27 directory and not in the ns-3-allinone directory. You can run the scripts you've placed in the ns-3.27/scratch folder by using ./waf command. For example:

```
cd ns-3.27
./waf --run  'reliability_example'
```

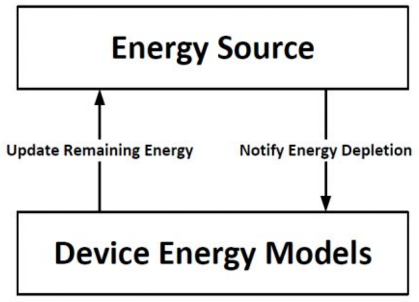
Running the above command in the /ns-3.27 directory will execute the "reliability_example.cc" script.

Our code should successfully complete the building process. The example script will help you get familiar with our proposed reliability framework. For more detailed configuration on ns-3, please refer to the official Installation Guide of ns-3.

Energy Module

The standard energy module consists of 2 major components:

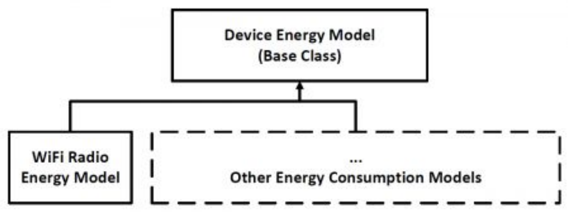
- Energy Source
- Device Energy Model



An energy source on the node represents a battery or a power supply while device energy models denote various energy-consuming components in a system. Multiple device energy models can exist on a single node, e.g. Wi-Fi energy model, CPU energy model, etc. The interaction between energy source and device energy models is bi-directional: each device energy model will notify the energy source of the energy consumed by that portion, and thus update the remaining energy of the source. When energy is completely drained, the energy source will notify all device energy models connected to it.

CPU Energy Model

CPU Energy Model is an extension to the ns-3 Energy Framework. It is implemented as a child class of *DeviceEnergyModel*.


Device Energy Model (Base Class)
WiFi Radio Energy Model
...
Other Energy Consumption Models

The main job of this class is to establish a connection between the energy source and the power model of the CPU. It provides interfaces for updating remaining energy in the energy source based on the power consumption acquired from the power model. It is designed to be state based, where the CPU can take the states *Idle* or *Busy*. In order to determine when a transition will occur between states, a PHY Listener is registered to the Wifi PHY. After a node completes receiving data (i.e. Wifi PHY::RX state ends), PHY Listener notifies CPU Energy Model. Then, an application/program is run by calling PowerModel::RunApp(), and the state is set to *Busy*. The reason behind this choice of mechanism for the CPU Energy Model is that a 'complete' data has to be available at the device for a program/application to be executed. Depending on the input data size of a program, we can wait multiple RX callbacks to collect enough data before starting processing. While the CPU is *Busy* all incoming new packets are queued by PHY layer until application is done executing. Finally, energy depletion is calculated according to the power and execution time values fetched from *PowerModel* and *PerformanceModel* respectively.

Class Implementation (src/energy/model/cpu-energy-model.cc):

- *GetTotalEnergyConsumption*: Returns the total energy consumed since the start of the simulation.
- *GetCurrentState*: Returns the state of the CPU. In the current implementation, the state can be either *Idle* or *Busy*.
- *GetIdlePowerW*: Returns the *Idle* power consumption of the device.

- *GetPhyListener*: Returns a pointer to *PhyListener* object. *PhyListener* tracks the state of PHY Layer and notifies CPU Energy Model when RX state ends.
- *SetEnergySource*: Attaches energy model to the energy source object of the node.
- *SetIdlePowerW*: Sets the *Idle* power consumption of the device.
- *SetPowerModel*: Sets the power model to be used to calculate energy consumption.
- *SetEnergyDepletionCallback*: Sets the callback to be notified of energy depletion at the energy source.
- *ChangeState*: Calculates energy consumption since the last update time and decreases the energy of energy source by that amount. Different amount of energy is consumed depending on the CPU state. This function is called via *callback* each time state change occurs in PHY layer.

CpuEnergyModel Helper

This is a helper class to attach Cpu Energy Model to a node, and associate it with an *EnergySource* and *WifiNetDevice*. **A Wifi device is needed to install it** since *CpuEnergyModel* runs applications based on callbacks from *PhyListener* which listens to the *WifiNetDevice* and gives notifications about the state of the device. Also, **a power model should be set**. Installation of a power model and *PowerModel* class will be discussed in the next section.

Class Implementation (src/energy/helper/cpu-energy-model-helper.cc):

- *Set*: Sets an attribute given the attribute name and attribute value.
- *SetDepletionCallback:* Sets the callback to be invoked when energy is depleted.
- *SetCpuAppRunCallback:* Sets the callback to be invoked when an application is run.
- *SetCpuAppTerminateCallback:* Sets the callback to be invoked when application is terminated..
- *SetPowerModel*: Sets the power model.
- *Install*: Installs the *CpuEnergyModel* on the node.

Reliability Framework

The reliability framework consists of 4 major components:

- Power Model
- Performance Model
- Temperature Model

- Reliability Model

These four models are in a hierarchical relationship:

$$\begin{tikzcd}
\text{Power Model} & \text{Temperature Model} \arrow[l] & \text{Reliability Model} \arrow[l] \\
 & \text{Performance Model} \arrow[ul]
\end{tikzcd}$$

Power model is the head of the rest of the models. Both temperature and performance models are registered to the power model. Reliability model is registered with the temperature model. The update order is the reverse of register relation: power model is always the first to be updated. Then, temperature and performance models will recalculate the output as they take power as a parameter. Finally, reliability of the system will be updated according to temperature changes. Reliability model works on a different update interval from the other models because reliability is a slow changing phenomena. According to some predefined update interval, it asks for the average temperature of the CPU in the last interval, and updates itself.

There is a helper called *ReliabilityHelper* to install these components onto a node all together.

Power Model

Power modeling is implemented with the base class *PowerModel*. It provides interfaces for running an application, getting an estimate of execution time for the application from the performance model, updating temperature, and terminating the application. All child classes of *PowerModel* implement these, but have different models for estimating power. A power model starts outputting power values when *RunApp()* function is called. CPU state is set to *Busy* and the *UpdatePower()* method is scheduled periodically to update power in predefined intervals, until the execution time of the application is completed. Finally, application terminates via *TerminateApp()* function, CPU state is set to *Idle*, and power is adjusted to idle power consumption.

An example power model is implemented in the child class *AppPowerModel*. This model estimates the power consumption of a CPU from its frequency and utilization, given an application to be run on the device. The coefficients of this linear estimation model is configurable through class methods.

Class Implementation (src/reliability/model/app-power-model.cc):

- *RegisterTemperatureModel:* Registers a pointer to the node's temperature model. This pointer is used to call and update the temperature model.

- *RegisterPerformanceModel*: Registers a pointer to the node's performance model. This pointer is used to call and update the performance model.

- *SetDeviceType*: Sets the type of the device (Pi, Arduino etc.).
- *SetA/B/C/D*: Sets the coefficients (A, B, C, D) of the linear equation that calculates the power.
- *SetFrequency*: Sets the frequency of the CPU of the device.
- *SetUtilization*: Sets the CPU utilization.
- *SetState*: Sets the state of CPU, *Busy* or *Idle*.
- *SetIdlePowerW*: Sets the idle power consumption.
- *SetApplication*: Sets the application that will be run on the device. Power model changes depending on the application being run.
- *GetPower*: Returns the power consumption.
- *GetFrequency*: Returns the frequency of the CPU of the device.
- *GetUtilization*: Returns the CPU utilization.
- *GetState*: Returns the state of the CPU.
- *GetIdlePowerW*: Returns the idle power consumption.
- *RunApp*: Executes the application and gets execution time estimate from performance model. Then, it schedules termination of the application based on this execution time.
- *UpdatePower*: CPU power is updated and a new update is scheduled based on the predefined *UpdateInterval* attribute. Temperature model update function is also called through the registered *TemperatureModel* pointer.
- *TerminateApp*: Terminates the application, sets state to *Idle*, changes power consumption value to IdlePowerW.

Performance Model

As there exists many criteria for measuring performance in a networked system, the base class *PerformanceModel* leaves a lot of freedom in implementation for child classes. A child performance class can model metrics such as delay (one way delay or Round Trip Time, RTT), delay variation (jitter), packet delivery rate, end-to-end latency, hop count and throughput, etc. In our current *PerformanceSimpleModel*, a simple performance model estimating execution time based on application and input data size is implemented.

Class Implementation (src/reliability/model/performance-simple-model.cc):

- *SetA/B/C*: Sets the coefficients (A, B, C, D) of the equation that calculates execution time.
- *SetDeviceType*: Sets the type of the device (Pi, Arduino etc.).
- *SetApplication*: Sets the application that will be run on the device. Performance model changes depending on the application being run.
- *SetDatasize*: Sets input data size of the application.
- *GetDatasize*: Returns the input data size.
- *GetExecTime*: Returns the application execution time.

Temperature Model

The goal of the temperature model is to estimate CPU temperature based on its power consumption and ambient temperature. *TemperatureModel* is the base class that provides APIs for interfacing it with *PowerModel* and updating temperature. If the *TemperatureModel* is registered to a *PowerModel*, it can be called inside *UpdatePower* function to subsequently update temperature. Similarly, reliability update is initiated by *TemperatureModel* class.

An example temperature model is implemented in the child class *TemperatureSimpleModel*. This is a simple state space model in the following form:

$$T_{k+1} = A \cdot T_k + B \cdot P_k + C \cdot T_{env} + D$$

T_k, P_k, T_{env} are the CPU temperature, the CPU power consumption and the ambient environment temperature at discrete time k respectively. A, B, C & D are the configurable attributes through methods of this class.

Class Implementation (src/reliability/model/temperature-simple-model.cc):

- *RegisterReliabilityModel:* Registers a pointer to the node's reliability model. This pointer is used to call and update reliability model.
- *SetDeviceType*: Sets the type of the device (Pi, Arduino etc.).
- *SetA/B/C/D*: Sets the coefficients (A, B, C, D) of the state space equation that calculates the temperature.
- *SetTenv*: Sets the ambient temperature.

- *GetTemperature*: Returns the temperature.
- *GetAvgTemperature*: Returns the exponential moving average of temperature.
- *GetTenv*: Returns the ambient temperature.
- *UpdateTemperature*: Updates temperature and calls *UpdateReliability* function through the registered *ReliabilityModel* pointer.

Reliability Model

Reliability model is the last model in the Power Model--> Temperature Model-->Reliability Model hierarchy. Temperature is estimated using power, and reliability is estimated using temperature. The model is implemented in base class *ReliabilityModel* and its structure is very similar to *TemperatureModel*. It provides interface for updating reliability.

An example reliability model is the child class *ReliabilityTDDBModel*. This model implements one of the main degradation mechanisms affecting integrated circuits, Time Dependent Dielectric Breakdown (TDDB). At each *UpdateReliability* call, the reliability of CPU is calculated from the formula of reliability for a single transistor which is:

$$R(t) = e^{-a(\frac{t}{\alpha})^{\beta}}$$

where *t* is the time-to-breakdown, *a* is the device area normalized with respect to the minimum area, and α and β are respectively the shape parameter and scale parameter. The shape parameter β, instead, is a function of the critical defect density, which in turn depends on oxide thickness, temperature and applied voltage. Reliability of the CPU is the sum of reliabilities of transistors that constitute the CPU.

Class Implementation (src/reliability/model/reliability-tddb-model.cc):

- *SetA/B*: Sets the coefficients (A, B) of the equation that calculates reliability.
- *SetArea*: Sets the device area to be used in reliability calculation.
- *UpdateReliability*: Updates reliability based on temperature and some other parameters.

Reliability Helper

This is a helper class to attach models in the reliability stack (power, performance, temperature and reliability) to a node. Users can select which models to use and configure the application and parameters related to the models using this helper.

Class Implementation (src/reliability/helper/reliability-helper.cc):

- *SetPowerModel*: Sets the power model.
- *SetTemperatureModel*: Sets the temperature model.
- *SetReliabilityModel*: Sets the reliability model
- *SetApplication*: Sets the application that will be run on the device.
- *Install*: Installs *PowerModel, TemperatureModel* and *ReliabilityModel* on the node.

Usage

Reliability Module

Module Include:

```
#include "ns3/reliability-module.h"
```

In order to use the reliability framework, the user must install Power Model, Temperature Model, and Reliability Model for the node of interest. Temperature Model and Reliability Model requires the presence of Power Model on the node, whereas Power Model can be installed as a standalone entity. We should note that the Performance Model is automatically installed along with Power Model.

The installation of the reliability stack is simple by using the helper function ReliabilityHelper. The following code snippet configures and installs all four models power, performance, temperature and reliability models.

```
/* Create node */
Ptr<Node> node = CreateObject<Node>();
/* Create reliability stack */
ReliabilityHelper reliabilityHelper;
reliabilityHelper.SetPowerModel("ns3::AppPowerModel");
reliabilityHelper.SetPerformanceModel("ns3::PerformanceSimpleModel");
reliabilityHelper.SetTemperatureModel("ns3::TemperatureSimpleModel");
reliabilityHelper.SetReliabilityModel("ns3::ReliabilityTDDBModel");
reliabilityHelper.SetApplication("LinearRegression",100);
```

```
  reliabilityHelper.Install(node);
```

After creating an instantiation of ReliabilityHelper object, user can configure the models to be used by using Set() methods. The power and performance models change depending on the application, thus user have to also set the target application to be run on the node.

Note that the above example is given for a single node. ReliabilityHelper can be used to install models on multiple nodes simultaneously. For this, a *NodeContainer* input must be provided.

```
  /* Create node container */
  NodeContainer c;
  c.create(10); //creating 10 nodes
...
. // Set functions...
...
  reliabilityHelper.Install(c);
```

Energy Module

Module Include:

```
  #include "ns3/energy-module.h"
```

Similar to reliability module, there are helper functions in energy module to help install energy sources and models. Installing an energy model requires **setting up a energy source and a Wifi net device beforehand**. An example set up may be as follows:

```
  /* Create an energy source */
  BasicEnergySourceHelper basicSourceHelper;
  /* Set initial energy to 1000000 */
  basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue
(100000));
  /* Install the source to the node */
  EnergySourceContainer source = basicSourceHelper.Install (node);
  /* Install CPU energy model to the source */
  CpuEnergyModelHelper cpuEnergyHelper;
  cpuEnergyHelper.Install(device, source);
```

If devices and sources are stored in a container structure for multiple nodes, i.e.

```
  EnergySourceContainer sources;
  NetDeviceContainer devices;
```

Then CPU Energy Model can be installed using the line

```
  DeviceEnergyModelContainer deviceModels = cpuEnergyHelper.Install(devices,
sources);
```

Note:

- `device` is a NetworkDevice container
- Refer to the example script for more details.
- CpuEnergyModel cannot be installed unless a power model is already installed on the node. User should always set up the power model (using ReliabilityHelper or manually) before installing CpuEnergyModel.

Tracing

Energy, power, temperature and reliability values for the a node at different time steps can be traced by 2 different methods:

1. Using Get() functions
2. Using trace sources

1. Each class (power, temperature etc.) has a public Get() function implemented. For example, GetPower() function can be used to obtain power consumption value of the node at any given time. To make access these values in a periodic fashion, one may use Simulator::Schedule as in the example below.

```
void
PrintInfo (Ptr<Node> node)
{
 std::cout << " CPU Power = " << node->GetObject<PowerModel>()->GetPower();
 std::cout << " Temperature = " <<
node->GetObject<TemperatureModel>()->GetTemperature()<<std::endl;
 if (!Simulator::IsFinished ())
 {
   Simulator::Schedule (Seconds (0.5),&PrintInfo,node);
 }
}
```

If we give an initial call to this function with `PrintInfo (node)`, it will schedule itself periodically each 0.5 seconds to print out power and temperature values.

2. Using the tracing system of ns-3 allows us to...

Example Script 1 - Two Node Communication

This script can be found at ns-3.27/scratch/reliability-example.cc

In this section we will go over this script by cutting it into snippets of code and explain how to build a ns-3 simulation. This example program configures two nodes on an 802.11b physical layer, in adhoc mode. All nodes are equipped with power, performance, thermal and reliability models, and the values are traced.

Module Includes

The code proper starts with a number of include statements.

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/energy-module.h"
#include "ns3/internet-module.h"
#include "ns3/reliability-module.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
```

ns-3 Namespace

The next line in the script is namespace declaration

```
using namespace ns3;
```

Main Function

As every C++ program, we declare functions before “main” . However, for the flow of the tutorial we will leave the additional utility functions to the end of the section.

```
int
main (int argc, char *argv[])
```

```
{
 std::string phyMode ("DsssRate1Mbps");
 double Prss = -80;            // dBm
 uint32_t PpacketSize = 200;   // bytes
 bool verbose = false;

 // simulation parameters
 uint32_t numPackets = 10000;  // number of packets to send
 double interval = 1;          // seconds
 double startTime = 0.0;       // seconds
 double distanceToRx = 100.0;  // meters
 double offset = 81;

 CommandLine cmd;
 cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
 cmd.AddValue ("Prss", "Intended primary RSS (dBm)", Prss);
 cmd.AddValue ("PpacketSize", "size of application packet sent", PpacketSize);
 cmd.AddValue ("numPackets", "Total number of packets to send", numPackets);
 cmd.AddValue ("startTime", "Simulation start time", startTime);
 cmd.AddValue ("distanceToRx", "X-Axis distance between nodes", distanceToRx);
 cmd.AddValue ("verbose", "Turn on all device log components", verbose);
 cmd.Parse (argc, argv);
```

This snippet is just the declarations in the main function of your program (script). As in any C++ program, you need to have a main function that will be the first function to be run. The initializations are done as the first thing. In ns-3 command line parsing is usually used to conveniently simulate different scenarios with different parameters.

Node Creation

The next two lines of code in our script creates the ns-3 Node objects that will represent the hosts/computers in the simulation. *NodeContainer* class is a topology helper that provides a convenient way to create and manage multiple *Node* objects together in bulk.

```
 Ptr<Node> node0 = CreateObject<Node>();
 Ptr<Node> node1 = CreateObject<Node>();
 NodeContainer networkNodes = NodeContainer(node0);
 networkNodes.Add(node1);
```

Here, we first create two nodes *node0* and *node1* separately and put them in a container. We didn't need to create them one by one, and could have directly used a NodeContainer. But having pointers to each of them separately will help us in this example later on.

Network Creation

After creating the nodes, we start building the communication elements. *WifiHelper* helps creating WifiNetDevice objects. Using *YansWifiPhyHelper,* we set the PHY Layer attributes of the network. Then, using *YansWifiChannelHelper*, the attributes of communication medium is configured.

```
/** Wifi PHY **/
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
wifiPhy.Set ("RxGain", DoubleValue (-10));
wifiPhy.Set ("TxGain", DoubleValue (offset + Prss));
wifiPhy.Set ("CcaMode1Threshold", DoubleValue (0.0));

/** wifi channel **/
YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel");
// create wifi channel
Ptr<YansWifiChannel> wifiChannelPtr = wifiChannel.Create ();
wifiPhy.SetChannel (wifiChannelPtr);
```

WifiMacHelper helps to configure MAC Layer of the Wifi.

```
/** MAC layer **/
// Add a MAC and disable rate control
WifiMacHelper wifiMac;
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager", "DataMode",
                               StringValue (phyMode), "ControlMode",
                               StringValue (phyMode));
// Set it to ad-hoc mode
wifiMac.SetType ("ns3::AdhocWifiMac");
```

Finally, the PHY and MAC Layer is installed to our nodes' NetDevice using the *WifiHelper*.

```
/** install PHY + MAC **/
// NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, networkNodes);
NetDeviceContainer device0 = wifi.Install (wifiPhy, wifiMac, node0);
NetDeviceContainer device1 = wifi.Install (wifiPhy, wifiMac, node1);
NetDeviceContainer devices = NetDeviceContainer(device0, device1);
```

ns-3 involves models for PHY and MAC layers of Wi-Fi, LTE, WiMax, etc. In this example we create a WiFi network, but detailed information for other models can be found on ns-3 model library.
Wi-Fi Module
Wimax Module
LTE Module
6LoWPaN Module
LR-WPAN Module

Also, there are many routing protocols in ns-3- such as OLSR, AODV, DSDV, DSR, Nix-Vector Routing.

Mobility

The nodes can be stationary or mobile depending on the preference. We configure the mobility of the nodes using *MobilityHelper* as the next step.

```
/** mobility **/
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>
();
positionAlloc->Add (Vector (0.0, 0.0, 0.0));
positionAlloc->Add (Vector (2 * distanceToRx, 0.0, 0.0));
mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (networkNodes);
```

Mobility module documentation can be accessed from Mobility Module.

Energy & Reliability

Next step is to connect an energy source and set an energy model to our nodes.

```
/* energy source */
BasicEnergySourceHelper basicSourceHelper;
// configure energy source
basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue (100000));
// install source
EnergySourceContainer source1 = basicSourceHelper.Install (node1);
```

To be able to install the CpuEnergyModel first, we need to have a power model instantiated first. Now we add power, performance, thermal and reliability models using *reliabilityHelper* class. In this example, we set the *PowerModel* to linear power model and set the application as ’Linear Regression’ with input data size of ’100000’ bytes. *AppPowerModel*, *PerformanceSimpleModel*, *TemperatureSimpleModel* and *ReliabilityTDDBModel* is selected.Finally, the models are installed to nodes via Install function.

```
/* reliability stack */
ReliabilityHelper reliabilityHelper;
reliabilityHelper.SetDeviceType(“RaspberryPi”)
reliabilityHelper.SetPowerModel("ns3::AppPowerModel");
reliabilityHelper.SetPerformanceModel("ns3::PerformanceSimpleModel");
reliabilityHelper.SetTemperatureModel("ns3::TemperatureSimpleModel");
reliabilityHelper.SetReliabilityModel("ns3::ReliabilityTDDBModel");
reliabilityHelper.SetApplication("Adaboost",10000,100);
reliabilityHelper.Install(node1);
/* cpu energy model */
CpuEnergyModelHelper cpuEnergyHelper;
cpuEnergyHelper.Install(device1, source1);
```

Internet Stack and IP Addresses

We now have nodes and devices configured, but we don’t have any protocol stacks installed on our nodes. *InternetStackHelper* will install an Internet Stack (TCP, UDP, IP, etc.) on each of the nodes in the node container. Next we need to associate the devices on our nodes with IP addresses. We provide a topology helper to manage the allocation of IP addresses. The only user-visible API is to set the base IP address and network mask to use when performing the actual address allocation (which is done at a lower level inside the helper).

```
/** Internet stack **/
InternetStackHelper internet;
internet.Install (networkNodes);

Ipv4AddressHelper ipv4;
NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer i = ipv4.Assign (devices);
```

For different configurations and to learn how Internete stack works, please refer to Internet Models. There you can also find routing architecture, TCP and UDP models.

Network Application

We designate a *source* and a *sink* node, and configure them to have a UDP traffic with the following code snippet.

```
TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
Ptr<Socket> recvSink = Socket::CreateSocket (node1, tid);  // node 1, receiver
InetSocketAddress local = InetSocketAddress (Ipv4Address::GetAny (), 80);
recvSink->Bind (local);
recvSink->SetRecvCallback (MakeCallback (&ReceivePacket));

Ptr<Socket> source = Socket::CreateSocket (node0, tid);    // node 0, sender
InetSocketAddress remote = InetSocketAddress (Ipv4Address::GetBroadcast (), 80);
source->SetAllowBroadcast (true);
source->Connect (remote);
```

Simulator

What we need to do at this point is to actually run the simulation. This is done using the global function *Simulator::Run*. Also, a *Simulator::Stop* command is needed if the event queue doesn't automatically run out of events. This finalizes the main function. Note that we also schedule the *GenerateTraffic* function to run at the *startTime*. By scheduling this function, we can generate traffic and send packets from *source* node to *sink* node at desired time points.

```
/** simulation setup **/
// start traffic
Simulator::Schedule (Seconds (startTime), &GenerateTraffic, source,
PpacketSize,networkNodes.Get (0), numPackets, interPacketInterval);
Simulator::Stop (Seconds (10.0));
Simulator::Run ();
Simulator::Destroy ();
```

Additional Utility Functions

The *GenerateTraffic* discussed above is as follows:

```
static void
GenerateTraffic (Ptr<Socket> socket, uint32_t pktSize, Ptr<Node> n,
                uint32_t pktCount, Time pktInterval)
{
 if (pktCount > 0)
   {
     socket->Send (Create<Packet> (pktSize));
     Simulator::Schedule (pktInterval, &GenerateTraffic, socket, pktSize, n,
                          pktCount - 1, pktInterval);
   }
 else
   {
     socket->Close ();
   }
}
```

We use tracer functions to observe power, temperature and reliability values.

```
void
ReliabilityTracer (Ptr<OutputStreamWrapper> stream, double oldValue, double
newValue)
{
  std::cout << "Reliability = " << newValue << std::endl;
  *stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldValue <<
"\t" << newValue;
}

void
TemperatureTracer (Ptr<OutputStreamWrapper> stream, double oldValue, double
newValue)
{
  std::cout << "Temperature = " << newValue << std::endl;
  *stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldValue <<
"\t" << newValue;
}

void
PowerTracer (Ptr<OutputStreamWrapper> stream, double oldValue, double newValue)
{
  std::cout << "Power = " << newValue << std::endl;
  *stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldValue <<
"\t" << newValue;
}
```

These values can be printed using *PrintInfo* function. We schedule this function with *Simulator* to call it intermittently.

```
void
PrintInfo (Ptr<Node> node)
{
 std::cout<<"At time "<< Simulator::Now().GetSeconds()<<", NodeId =
"<<node->GetId();
 std::cout << " CPU Power = " << node->GetObject<PowerModel>()->GetPower();
 std::cout << " Temperature = " <<
node->GetObject<TemperatureModel>()->GetTemperature()<<std::endl;
 std::cout << " Reliability = " <<
node->GetObject<ReliabilityModel>()->GetReliability()<<std::endl;
 if (!Simulator::IsFinished ())
 {
   Simulator::Schedule (Seconds (0.5),&PrintInfo,node);
 }
}
```

Example Script 2 - Ad Hoc Communication Mesh Network

This script can be found at ns-3.27/scratch/wifi-adhoc-grid.cc

This program configures a grid (default 5x5) of nodes on an 802.11b physical layer, with 802.11b NICs in adhoc mode, and by default, sends packets of 1000 bytes to node 1.

The default layout is like this, on a 2-D grid.

```
n20 n21 n22 n23 n24
n15 n16 n17 n18 n19
n10 n11 n12 n13 n14
  n5  n6  n7  n8  n9
  n0  n1  n2  n3  n4
```

Module Includes, Declarations

Similar to our previous program we start with module includes and variable declarations.

```
#include "ns3/core-module.h"
#include "ns3/mobility-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/olsr-helper.h"
#include "ns3/reliability-module.h"
#include "ns3/energy-module.h"
using namespace ns3;

std::string phyMode ("DsssRate1Mbps");
double distance = 500;  // m
uint32_t packetSize = 1000; // bytes
uint32_t numPackets = 100;
uint32_t numNodes = 25;  // by default, 5x5
uint32_t sinkNode = 0;
uint32_t sourceNode = 24;
double interval = 0.5; // seconds
bool verbose = false;
bool tracing = false;
```

Command Line Argument Parsing

To make this script configurable, we add command line argument parsing to give user freedom to change above variables.

```
 CommandLine cmd;

 cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
 cmd.AddValue ("distance", "distance (m)", distance);
 cmd.AddValue ("packetSize", "size of application packet sent", packetSize);
 cmd.AddValue ("numPackets", "number of packets generated", numPackets);
 cmd.AddValue ("interval", "interval (seconds) between packets", interval);
 cmd.AddValue ("verbose", "turn on all WifiNetDevice log components", verbose);
 cmd.AddValue ("tracing", "turn on ascii and pcap tracing", tracing);
 cmd.AddValue ("numNodes", "number of nodes", numNodes);
 cmd.AddValue ("sinkNode", "Receiver node number", sinkNode);
 cmd.AddValue ("sourceNode", "Sender node number", sourceNode);
```

```
cmd.Parse (argc, argv);
```

Node Creation & Network Creation

Different to our previous script, we will work on the nodes in groups, using node containers.

```
NodeContainer c;
c.Create (numNodes);

// The below set of helpers will help us to put together the wifi NICs we want
WifiHelper wifi;
if (verbose)
  {
    wifi.EnableLogComponents ();  // Turn on all Wifi logging
  }

YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
// set it to zero; otherwise, gain will be added
wifiPhy.Set ("RxGain", DoubleValue (-10) );
// ns-3 supports RadioTap and Prism tracing extensions for 802.11b
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel");
wifiPhy.SetChannel (wifiChannel.Create ());

// Add an upper mac and disable rate control
WifiMacHelper wifiMac;
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                               "DataMode",StringValue (phyMode),
                               "ControlMode",StringValue (phyMode));
```

We set WiFi to Ad Hoc mode.

```
// Set it to adhoc mode
wifiMac.SetType ("ns3::AdhocWifiMac");
NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, c);
```

Mobility

```
MobilityHelper mobility;
mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                "MinX", DoubleValue (0.0),
                                "MinY", DoubleValue (0.0),
                                "DeltaX", DoubleValue (distance),
                                "DeltaY", DoubleValue (distance),
                                "GridWidth", UintegerValue (5),
                                "LayoutType", StringValue ("RowFirst"));
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (c);
```

Energy & Reliability

Here, energy sources, energy models and reliability models are installed in bulk to all nodes in container *c*.

```
/** Energy Model **/
/***************************************************************************/
/* energy source */
BasicEnergySourceHelper basicSourceHelper;
// configure energy source
basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue (100000));
// install source
EnergySourceContainer sources = basicSourceHelper.Install (c);
/* reliability stack */
ReliabilityHelper reliabilityHelper;
reliabilityHelper.SetDeviceType(“RaspberryPi”);
reliabilityHelper.SetPowerModel("ns3::AppPowerModel");
reliabilityHelper.SetPerformanceModel("ns3::PerformanceSimpleModel");
reliabilityHelper.SetTemperatureModel("ns3::TemperatureSimpleModel");
reliabilityHelper.SetReliabilityModel("ns3::ReliabilityTDDBModel");
reliabilityHelper.SetApplication("LinearRegression",10000,100);
reliabilityHelper.Install(c);
/* cpu energy model */
 CpuEnergyModelHelper cpuEnergyHelper;
 cpuEnergyHelper.Install(devices, sources);
/***************************************************************************/
```

Routing and Internet Stack Configuration

```
// Enable OLSR
OlsrHelper olsr;
Ipv4StaticRoutingHelper staticRouting;

Ipv4ListRoutingHelper list;
list.Add (staticRouting, 0);
list.Add (olsr, 10);

InternetStackHelper internet;
internet.SetRoutingHelper (list); // has effect on the next Install ()
internet.Install (c);

Ipv4AddressHelper ipv4;
NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer i = ipv4.Assign (devices);
```

Network Application

```
TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
Ptr<Socket> recvSink = Socket::CreateSocket (c.Get (sinkNode), tid);
InetSocketAddress local = InetSocketAddress (Ipv4Address::GetAny (), 80);
recvSink->Bind (local);
recvSink->SetRecvCallback (MakeCallback (&ReceivePacket));

Ptr<Socket> source = Socket::CreateSocket (c.Get (sourceNode), tid);
InetSocketAddress remote = InetSocketAddress (i.GetAddress (sinkNode, 0), 80);
```

```
 source->Connect (remote);
```

Simulator

```
 // Give OLSR time to converge-- 30 seconds perhaps
 Simulator::Schedule (Seconds (30.0), &GenerateTraffic,
                      source, packetSize, numPackets, interPacketInterval);

 // Output what we are doing
 NS_LOG_UNCOND ("Testing from node " << sourceNode << " to " << sinkNode << "
with grid distance " << distance);

 Simulator::Stop (Seconds (60.0));
 Simulator::Run ();
 Simulator::Destroy ();
```

Appendix A: List of Files Included in the Reliability Framework

Energy Module

Model:

'src/energy/model/cpu-energy-model.cc',
'src/energy/model/cpu-energy-model.h',

Helper:

'src/energy/helper/cpu-energy-model-helper.cc',
'src/energy/helper/cpu-energy-model-helper.h',

Reliability Module

Model:

'src/reliability/model/power-model.cc',
'src/reliability/model/power-model.h',
'src/reliability/model/app-power-model.cc',
'src/reliability/model/app-power-model.h',
'src/reliability/model/util-power-model.cc',
'src/reliability/model/util-power-model.h',
'src/reliability/model/temperature-model.cc',
'src/reliability/model/temperature-model.h',
'src/reliability/model/temperature-simple-model.cc',
'src/reliability/model/temperature-simple-model.h',
'src/reliability/model/performance-model.cc',
'src/reliability/model/performance-model.h',
'src/reliability/model/performance-simple-model.cc',
'src/reliability/model/performance-simple-model.h',
'src/reliability/model/reliability-model.cc',
'src/reliability/model/reliability-model.h',
'src/reliability/model/reliability-tddb-model.cc',
'src/reliability/model/reliability-tddb-model.h',

Helper:

'src/reliability/helper/reliability-helper.cc',
'src/reliability/helper/reliability-helper.h',