Automated Network Simulations with ns-3

Kiet Quang Huynh Vo, Quang Nguyen Pham

May 18, 2017

1 Introduction

1.1 Context

Automated Network Simulations module is part of the Evolutionary Network Optimiser project. The project is about building a generic system for network optimization using evolutionary algorithm (EA) where each network instance will be evaluated using simulation. To that end, this module provides the capability to automate the process of setting up and running a network simulation on top of the famous network simulator, ns-3. In addition, users will have access to its output mechanism where data is collected in order to evaluate the network performance.

This document describes the library that offers the automated simulations as part of the network evaluator.

1.2 Requirements

Basic information and requirements:

- Language: C++11
- Platform: Ubuntu 16.4 - 64 bit
- Compiler: GNU C++ compiler - g++ 5.4.0
- Dependencies:
  - ns-3.26: for network simulation (debug build is required to use logging module)
  - Graph/Event Reader library (named AnyGraph from the same project)
  - Boost 1.63 (Boost.Test): for unit testing
- Recommended IDE: Eclipse IDE for C++, version Neon.2

Header files of dependencies must be copied into folder `/usr/local/include`. Shared and static libraries of dependencies must be copied into folder `/usr/local/lib`. All libraries should be compiled with `-fPIC` to generate position independent code.

Environment settings: `LD_LIBRARY_PATH="/usr/local/lib"`

Any project that uses this library must also link the file `libns3.26-core-debug.so` when building (option `-lns3.26-core-debug`).

1.3 Structure

The source code of the library resides in the `src/` folder. Source files are organized into the following folders:

- `components/`: classes that are used to model a network (e.g `Node`, `Edge`).
- `exceptions/`: exception classes relevant to the usage of this library.
- `input/`: sub-module that utilises the AnyGraph library to read files and extracts relevant data for running simulation.
- `nets/`: implementation for several types of physical network models.
- `output/` the output system to record and capture data of a simulation that may interest users.
- `simulation/`: implementation to set up network simulations.
- `simulation/events`: several types of implemented events that generate traffic for a network.
- `test/`: Unit Test component for this library.

And three standalone header files:

- `Common.h`: Define basic name alias and macros. This file is included by "Type.h".
- `Type.h`: Predefine all classes existed in the library and also their pointer types. This file is included internally everywhere in the library and by "Simul.h".
- `Simul.h`: Include all header files of the library. Library users should include only this file.

All classes in the library is contained in the namespace named `simul`. Users should be aware and be familiar with the use of namespace in C++.

2 Usage

This section aims at the basic users of the library, who want to set up network topology and simulate network events. The fundamental steps to set up a system of networks are as follows:

1. Create a set of edges (e.g. `P2PEdge`) that represents physical network connections.
2. Supply the edges to a `Simulation` object.
3. Finally, invoke `SetUpNetworks()` to let the `Simulation` object do its construction work.

The first step above can be achieved in two ways, either supplying a graph file to the library's input module or creating edges manually. These methods will be given detailed descriptions in section 2.3 and 2.4.

2.1 Pointers and Object Passing

This library mainly uses pointers when passing objects around in order to obtain polymorphism and saving resources. Nevertheless, users will not find any traces of raw pointers and keywords such as `new` and `delete` as smart pointers are employed here instead. As a result, users do not have to bother about deleting referenced objects or deallocating them when working with this kind of pointers. For more details on the subject, readers should refer to C++ documentation of `std::shared_ptr` and the two header files `Common.h` and `Type.h` (section 1.3).

2.2 Object Reusability

This library is built on the foundation of the ns-3 and therefore simulations are run by taking in user input and setting up the global ns-3 simulator. The latter action is usually refered to as "configuration". As every class of the library may be connected to the global configuration to some extent, users should never make any assumptions regarding the reusability of each object they have created. Unless they know the behaviour of an implementation well, users should always create new objects when launching a new simulation. The reusability of an object of a specific type should be thoroughly documented by its author in relevant parts of this document. Incidentally, the concept of user input data and configuration mentioned previously will be discussed more in later subsections.

2.3 Using GraphInput Module

The `IGraphInput` interface provides the specification for this usage, precisely the pure virtual method `ReadGraph(string)` that takes in a path to the graph file and returns a set of edges. Users only need to create an object of a derived class (e.g. `GraphMLInput`) and call the overriden method. Nevertheless, it should be borne in mind that whether or not an object of this type is reusable is up to the author of its implementation. As of now, there is only one implementation, namely `GraphMLInput` and the method `ReadGraph(string)` can be invoked as many times as users want in order to produce edges from graphml files.

2.4 Manually Working with Edge

Aside from producing a set of edges automatically from graph files, users can also manually create objects of derived classes of `Edge` such as `P2PEdge`. However, it should be noted that trying to create an edge that connects a node to itself will lead to an error. Having a collection of network edges, users then can set the attributes for them such as `Delay` and `DataRate`. This is however, a fairly advanced usage so the detailed instructions are put off until section 2.8.

2.5 Network Event Simulations

After finishing construct a physical network topology, users may want to set up some events and observe the flow of data inside as this is the goal of the library. In order to do so, a collection of event classes that derive from the interface `Event` is provided and can be found under `simulation/events/`. Just by simply creating an event object with two nodes and calling `SetUp()` then `Schedule(double)`, these two nodes will be ready for their communication according to the specific event (e.g. `UdpTestEvent`).

If the networks are set up from a graph file, users may not have any `Node` objects to provide as input to `Event`. In that case, the method `GetNode(string)` from the `Simulation` object can be used to retrieve a pointer to the found node using its ID. If the node is not found, a null pointer will be returned and it is users' responsibility to check it.

When invoked, the static method `Event::Run()` will activate the global simulator and any event that has been scheduled will begin. In some rare cases, the simulation may run forever as one sub-event generates another. As a consequence, another method with stop time as input parameter is offered, namely `Event::Run(double)`. If users are not sure what value to supply, they may use the recommended stop time, which can be retrieved through `Event::GetRecommendedStopTime()`. This value is calculated from all the invocations of `Schedule(double)` and therefore, it should only be fetched right before launching simulation. In addition, the `defaultEventLength` variable also contributes to the computation of the recommended stop time and can be accessed via appropriate setter and getter. As ns-3 is a discrete event simulator, the simulation does not comply with

the concept of continuous time in reality. It means that if there is no more event until the stop time, the simulation will stop. Therefore users do not need to be reserved when providing the stop time.

If an event is to happen many times in a simulation, `Schedule(double)` must be called many times as well. Moreover, after launching a simulation (`Event::Run()`), an event must be scheduled again if it is to take part in the next run.

Different logging levels are at users' disposal (section 2.9) if they wish to see what happens during the event simulation. Additionally, a query method named `IsSuccessful()` can be used to verify that an event simulation ends successfully. Nonetheless, different types of events may have different definitions of "successful" and implementations for this function. This method is only reliable if users know exactly what an event's implementation does.

2.5.1 TcpAppEvent and UdpAppEvent

`TcpAppEvent` and `UdpAppEvent` are presently not fully reusable. Trying to reuse them with `SetUp()` and `Schedule(double)` many times is possible but new data is appended instead of replacing the old one. As a consequence, memory resource will grow everytime `SetUp()` is called.

These two classes have not provided the implementation for `IsSuccessful()` method.

2.5.2 UdpTestEvent

On the other hand, due to its simplicity, `UdpTestEvent` offers great reusability and stable functionality which are very useful for testing purpose. The typical behaviours of this class are discussed below:

- Everytime `Schedule(double)` is invoked, the input time is also added to the list that will be used later by `ReSchedule()`.
- `ReSchedule()` iterates the list described above and does the scheduling implementation for each of the start time values.
- Calling `SetUp()` will clear the list.

`IsSuccessful()` can only work correctly if there are no two objects of this type such that they contain the same destination node.

2.6 Using EventInput Module

This class provides the method `vector<EventPtr> ReadEvent(string,SimulationPtr )` to read an event file into a existing simulation. The simulation object must be initialised with a graph file before this function can be called. The path variable in the parentheses is the path to the event file that is either in xml or json format. The event file and the graph file must be within the same use case.

The method `vector<EventPtr> ReadEvent(string,SimulationPtr)` will initialise one of the event object (i.e. `TcpAppEvent` or `UdpAppEvent`) according to what is written in file. If the protocol is not recognised, `UdpTestEvent` will be used by default (section 2.8).

2.7 Correct Usage of a Simulation Object

As ns-3, the very foundation of this library, uses one global simulator, having more than one configured `Simulation` object is illegal and only one of them should have the valid configuration. Every time the method `SetUpNetworks()` is called, the respective `Simulation` object is configured and the previous configuration is destroyed. However, users can make

a `Simulation` object valid again by calling its corresponding `SetUpNetworks()` because all the added edges still remain inside. Taking this into consideration, users should now be able to distinguish between the input data and configuration data.

Additionally, each time users configure the global simulator, all of the already set up events also become invalid as their settings are flushed along with the old configuration. In a similar manner, invoking `Event::SetUp()` and they are ready to run again but bear in mind that data traffic can only flow if there is connection between two input nodes (in an `Event` object). The reason is because users may mistakenly use the wrong `Simulation` object (with different set-up of nodes and edges) for an `Event` object.

2.8 Setting Attributes for Simulations

This subsection provides a catalog of available attributes for users to config based on their needs. In case they are not set, the system will use default values. Since these attributes are provided by the ns-3 library and taken as string input, users should study ns-3 documentation on attribute values carefully in order to supply the acceptable string input values. Most of these attributes are user input data and therefore need to be set before configuring `Simulation` and `Event` (section 2.7) so that they can take effect. The respective methods for setting these attributes can be found in the relevant header files of the library.

Class	Attributes	Default Value	Notes
Edge	DataRate (src/dst)	∞	e.g. "5Mbps"
	MTU (src/dst)	1500	
	InterframeGap (src/dst)	"0ns"	
	Delay	"0ms"	
Event	start time		
	data transfer		implementation dependent
	protocol	"udptest"	

Table 1: Simulation Attributes

In case users want to supply these attributes in graph or event files, they are advised to take a look at the "Writing a use case" document of the AnyGraph library.

If there is an already configured `Simulation` object that contains a newly updated edge, users can propagate the new data into the configuration without calling `SetUpNetworks()` again. The `P2PEdge` provides an exclusive method for this, namely `UpdateConfiguration()`.

2.9 Logging

The library employs ns-3's logging module in order to output its logging messages and only uses three verbosity levels at present, namely Debug, Warn, and Error. Logging can be enabled on a component-by-component basis, for the whole library, or globally (including all ns-3 components). To enable a log component, `LogComponentEnable(char*, enum LogLevel)` must be called. More information regarding this function and its parameters can be found in the header file `log.h` (i.e. in `ns-3.26/build/ns3/`), the tutorial, and manual of ns-3. On the other hand, the static method `Simulation::EnableAllLogComponents(enumLogLevel)` provides users a convenient way to enable all log components for this library. Some examples of using the logging module are:

```
ns3::LogComponentEnable("Simul::UdpTestEvent", ns3::LOG_LEVEL_DEBUG);
```

```
simul::Simulation::EnableAllLogComponents(ns3::LogLevel(ns3::LOG_LEVEL_
ALL | ns3::LOG_PREFIX_ALL));
```

2.10 The Output Module

The module is a sub-system responsible for recording and capturing data flowing inside the simulated network. In order to get some interesting data out of simulation, users first need to create some kind of data-capturing object from the output module, configure it, run the simulation and using relevant functions to gather the data.

2.10.1 PacketInfo

This interface is an abstract and simple model for a packet's metadata to store important information such as packet's size, type, and timestamps. It is very crucial to interpret the timestamps sensibly according to a packet's type. For example, the delivered time of a packet with the type `PACKET_SENT` should be disregarded because this type indicates that no nodes have admitted that they received this packet. However, in reality, some nodes may have receive but do not change the packet type to others (e.g. `PACKET_DELIVERED`) so the delivered time is not reliable no matter what value it may have. In general, the interpretation of a `PacketInfo` differs depending on the class that produces it (e.g. `Ipv4PacketInfoOutput`).

2.10.2 Ipv4PacketInfoOutput

This class implements a collection of callback functions on the Ipv4 protocol to collect traffic data inside a network. Callback functions are activated once a set of conditions is met, for instance, a packet has been sent or arrived. When it is invoked, a callback function will start recording the respective packet in the form of `PacketInfo`. In order to capture data, the relevant callback functions must be enabled by users beforehand throught a set of enable and disable methods:

Methods	When callback is activated
`EnableCbTx()`	a node sends a packet
`EnableCbDrop()`	a node drops a packet
`EnableCbSendOutgoing()`	source node generates and sends a packet
`EnableCbUnicastForward()`	a node forwards a packet
`EnableCbLocalDeliver()`	destination node receives a packet

Table 2: Callback configuration explanation

After running the simulation, users can extract captured data using `GetPacketInfo List()`, which returns a list of `PacketInfo` of delivered packets (`PACKET_DELIVERED`) by default. To obtain different lists, simply call `SetListType(Ipv4PIOListType)` to specify it first. When launching a new simulation, users must use `ClearPacketInfoList()` to remove old data so that it will not be mixed with the new one.

Recall from section 2.7 that a `Simulation` object has the right to detroy all of the global configuration and thus the result is similar to the invocation of `DisableCbAll()`. When it happens, users are expected to enable those callback functions again.

2.10.3 IGraphOutput

This interface offers the contrary functionality to that of the `IGraphInput` by producing a graph file from a set of edges.

2.11 Exception Classes

Exception classes are provided as the means for throwing suitable error messages and (in case they are not handled) terminating the program at specific points where the causes of errors are introduced. Exception message can be retrieved through `what()` or `GetMessage()`.

2.12 Scope and Limitations

The library's notable limitations are listed below:

- Optimised build is not supported yet.
- Each `Simulation` object can hold no more than 254×64 `P2PNet` (corresponding to the same number of `P2PEdge`) and 254 `WifiNet` (each of which can only have a maximum of $256^2 - 3$ `WifiEdge`). This will be improved later.
- Port numbers used in events are limited (depending on the specific implementation but usually from 49152 to 65535). This should be taken into account when generating a large number of events.
- `TcpAppEvent` and `UdpAppEvent` have complex and unpredicted behaviour and should only be considered experimental.
- ns-3 does not support multithreading and so does this library.

3 Interfaces

This section provides a more overall view of the system through a list of interfaces which specify the design that the implementations must follow.

3.1 Edge

This is an abstract class for network edges (or connections). Any edge class is expected to derive from this class or its subclasses. There is presently no exact pure virtual method for this class, however having a separate interface facilitates the use of polymorphism as well as leaves room for later extension.

3.2 IGraphInput

A simple interface for reading a graph file and extract the necessary data for simulation. Any subclass should override the method `ReadGraph(string)`. The return value is a pointer to an `std::set` of `EdgePtr`.

3.3 IEventInput

3.4 Network

This interface is the base class for all kinds of network implementation and as a result, derived classes are expected to override `Configure()` and `GetAllNetDevices()`. The former's function is to install the relevant network on the involved nodes using ns-3 while the latter is used to gather all the devices from the installed network. Furthermore, the subclasses must provide some kind of mechanism for `Simulation::Flush()` to reset their local IP address databases (e.g. `P2PNet::ResetNetCount()`) should they calculate those addresses incrementally. Last but not least, it is a good practice to hide these methods and make `Simulation` a friend class as they are critical and should be controled by `Simulation` only.

3.5 Event

The Event interface requires all its subclasses to implement the `DoSetUp()`, `DoSchedule(double)`, and `DoReSchedule()` functions whose behaviours may vary according to different types of network events. Additionally, they must also provide a user-friendly means to check whether or not an event runs successfully through `IsSuccessful()` and allows the successful state to be reset with `ResetState()`. Except for `IsSuccessful()`, the other three are private and are called appropriately by `SetUp()`, `Schedule()`, and `ReSchedule()`, which are defined in the `Event` class.

3.6 PacketInfo

This interface models a packet's metadata that is usually of interest to users. Aside from the provided pure virtual methods, the header file also defines a new enum type, `PacketType` that is used to specify the current state of a packet.

3.7 IPacketInfoOutput

This class represents an output system that can records the data flow inside a network. The pure virtual methods `GetPacketInfoList()` and `ClearPacketInfoList()` offer users basic features to retrieve data as well as remove it before running a new simulation so that new data can be fetched without being mixed in with the old one.

3.8 IGraphOutput

The only method that needs to be overridden is `WriteGraph(SetPtr<EdgePtr>, string)`, which should utilise the AnyGraph library to produce a graph file from the input set of edges. Moreover, this function returns a boolean value that indicates whether or not the serialisation has succeeded.

4 Implementation Details

This section documents the implementation of the library as well as presents the rationale behind it. In addition, the expected behaviours of the main modules will also be described carefully.

4.1 Node

This class can be considered as a wrapper for `ns3::Node` and each `simul::Node` object has an `ns3::Node` object attached to its as a member variable. This makes the process of managing an `ns3::Node`, its devices, and their attributes easier and more automated. The reason is because the `ns3::Node` is part of the configuration data and becomes un-usabled when `ns3::Simulator::Destroy()` is called, whereas the user input data still remains as member variables of `simul::Node` and can be used for the next configuration.

When `GetNs3Node()` is used to fetch the `ns3::Node`, it will first check if there is already an object of this type. If not, it must call `CreateNs3Node()` to create one before returning a pointer to it. In current implementation, the `ns3::Node` is installed with an internet stack accompanied by Nix-Vector routing after being created. From experimentation, the simulation runs much faster with Nix-Vector routing than global routing in ns-3.

`FlushNs3Node()` simply sets the pointer to null so that the next time `GetNs3Node()` is invoked, a new `ns3::Node` can be created. Moreover, this method is private as only `Simulation` should have the right to touch it (section 4.3).

4.2 Edge

This model is an abstraction of a network's physical link that connects two nodes. The only constructor requires users to supply two nodes (as pointers) which both must be not null and of course different. Aside from `Delay`, which is a real property of a link, this class also holds other attributes of the two connected devices (from two nodes) such as `DataRate`, `MTU`, and `interframeGap`.

4.2.1 P2PEdge

This class has an exclusive method named `UpdateConfiguration()` that provides the capability to propagate the newly changed edge properties into the global configuration without invoking `SetUpNetworks()` again. This is very useful as it exhibits better performance than the alternative, especially in a large system of networks. In order to make this implementation possible, there must be two point-to-point devices and `P2PNet` must be declared a friend class. The two p2p devices will be assigned by `P2PNet` when this class installs a point-to-point network on this point-to-point link. Then the `P2PEdge` will later update the properties of these devices appropriately. Nevertheless, there must be a check for exception and wrong usage (e.g. null pointers) of the `UpdateConfiguration()` before attempting to update the attributes.

4.3 Simulation

This class is a partial wrapper for the global simulator of ns-3 with the functionality of constructing and installing physical networks from user input data. However, it only stops at physical topology and does not provide any ways to run a simulation as this is the job of the event classes. Below is the program flow of `SetUpNetworks()`, which is typically invoked when users have added all the edges for their network construction:

1. Flush previous configuration.
2. Parse the edge list and create appropriate networks.
3. Configure those networks and collect their net devices.

As mentioned in 2.7, each time a `Simulation` object is configured, the previous setting must be destroyed by calling `Flush()`. This function begins with resetting the local IP address databases (i.e. `P2PNet::ResetNetCount()`), then removes the attached `ns3::Node` from all the `simul::Node` in the networks, removes old network list, and eventually calls `ns3::Simulator::Destroy()`. Since a `Simulation` object has control over the destruction of the ns-3 simulator configuration (including setting on a `ns3::Node` object), it must take responsibility to flush this node and must be the only one to do this. After this step, a new `ns3::Node` is ready to be created and attached to each `simul::Node` object.

The steps to build networks (in the private method `ConstructNetwork()`) starts by iterating the edge list and use dynamic pointer cast in order to identify the type of an edge. After the type has been found, a network object of suitable type (derived from `Network`) is created and added to the network list. Moreover, any edge that belongs to the same network being built is also added to the network. These steps repeat until the last edge in the list has been considered. Finally, the network list is iterated to configure its net members as well as collect their devices.

`Simulation` also stores net devices from constructed networks and provides a function called `FindDevicesFromNode(NodePtr)`, which can be useful for debugging purpose and later features.

4.4 Network

The `Network` class provides the `AddEdge(EdgePtr)` method for `Simulation` to call when constructing networks. However, contrary to `Simulation`, `Network` must not provide method to add a bunch of edges at the same time as having more than one edge is illegal in some types of networks (i.e. `P2PNet`). More important, aside from the destructor, other methods including constructor are hidden as network objects are meant to be used only by `Simulation`.

All the subclasses that derive from `Network` will utilise ns-3 in order to set up the ns-3 network simulator. The ns-3 documentation should be the best source to understand these implementations.

4.4.1 P2PNet

This class represents a point-to-point network, which is to be installed on two nodes and is used to model the switched Ethernet by combining several of their instances. The overridden `Configure()` works as follows:

1. Retrieve two attached `ns3::Node`, each from a respective `simul::Node`.
2. Use an `ns3::PointToPointHelper` to install a point-to-point network on these two nodes.
3. Set the attributes for the net devices (by assigning two devices to `P2PEdge` and invoke `UpdateConfiguration()`).
4. Assign IP addresses for all the devices.

The IP addresses for this class begin with `"192.168."` and the other two octets are computed using the variable `localNetID`. The static variable `p2pNetCount` keeps track of the number of instatiated `P2PNet` objects and gives `localNetID` a suitable value that starts with `"1"` and grows incrementally as the number of nets increases. The subnet mask is fixed as `"255.255.255.252"` since there are only two devices in a point-to-point network.

The static function `ResetNetCount()`, which is to be called by `Simulation` (section 4.3), resets the `p2pNetCount` to `"0"` in order to prepare for the next configuration.

4.4.2 WifiNet

This model for a wifi network using ns-3 is experimental and can be helpful for later extension. The detailed implementation is generally similar to that of `P2PNet`.

4.5 Event

This abstract class has some static member variables and functions in order to support the launching of simulation globally. The two notable ones out of them are `recommend edStopTime` and `Run()`. When invoked, the latter activates the ns-3 global simulator in order to run all scheduled events. The value of `recommendedStopTime` is the sum of the last scheduled event's time and the `defaultEventLength`.

To avoid false information, `ResetState()` is called at appropriate occasions (in `SetUp ()` and `Schedule()`) when an event prepares to run. This method must be implemented by derived classes and used to change their successful state to `false`.

4.5.1 UdpTestEvent

The implementation for this class makes use of a pair of sockets from two hosts in order to exchange data. The source socket is dynamically allocated while the destination socket

is based on port 7777. These two also have attached receive-callback functions which will be activated everytime they receive a packet.

4.5.2 TcpAppEvent

This class provide the method to invoke a tcp application event between 2 given nodes. The method `TcpAppEvent(NodePtr, NodePtr, uint32_t)` provides the setup interface for the EventInput class to make a setup according to the event file. The virtual method `DoSchedule(double)` must be overwritten. This class is not in use but can be used for later extension of the project

4.5.3 UdpAppEvent

4.6 Ipv4PacketInfoOutput

This implementation utilises the callback system of ns-3 to record the flow of data inside a simulation. Those callback functions are attached to the `Ipv4` object of each `ns3::Node`. In order for this class to work correctly, all event classes must add `TimestampTag` to every packet they send. This tag will help to determine the sent time of a packet, which in turn is used to calculate the end-to-end delay.

5 Unit Test

The unit test for this library lies under `src/test/` and organised in the same way as the library source files. In addition, the test source files are named after their corresponding tested components.

5.1 components

5.1.1 EdgeTest.cpp

`EdgeExceptionTest` checks the exception handling:

- null nodes
- same node

`EdgeMembersTest` checks setters and getters of:

- source node
- dest node

`EdgeAttributeTest` checks all attributes:

- default values
- setters & getters

5.1.2 NodeTest.cpp

`NodeMembersTest` tests all the public setters and getters.

5.1.3 P2PEdgeTest.cpp

`P2PEdgeSimpleTest` tests the returned boolean value of `UpdateConfiguration()`.

`P2PEdgewRunTest` tests `UpdateConfiguration()` in real simulation scenarios:

- delay has effect
- data rate has effect

5.2 input

5.2.1 EventInputTest.cpp

`EventInputTestwRun_n_Output01`, `EventInputTestwRun_n_Output02`, and `EventInputTestwRun_n_Output03` read files and check:

- number of events is correct
- simulation runs without problems
- captured data exists

`EventInputTestwRun_n_Output04` tests reading `UdpTestEvent` from file:

- number of recorded bytes is as expected

`EventInputAttributesTest` tests the reading of an event file by counting the numbers of `udp` and `tcp` protocols.

5.2.2 GraphMlInputTest.cpp

`GraphMLInputwEventTest` tests reading graph from file:

- `UdpTestEvent` runs successfully

`GraphMLInputAttributesTest` tests reading graph from file:

- `Edge`'s attributes are read correctly

5.3 nets

5.3.1 P2PNetTest.cpp

`NetP2PMembersTest` tests `Configure()`:

- 2 devices are point-to-point devices
- number of `Node` and `Edge` is as expected

`NetP2PExceptionTest` checks exception handling of null edges.

`NetP2PAttributesTest` tests `Configure()`:

- edges' attributes are set in devices.

5.3.2 WifiNetTest.cpp

`NetWifiMembersTest` checks `Configure()`:

- devices are all of wifi type
- number of `Node` and `Edge` is as expected

`NetWifiExceptionTest` checks exception handling of null edges.

5.4 output

5.4.1 GraphMLOutputTest.cpp

`GraphMLOutputAttributesTest01` and `GraphMLOutputAttributesTest02` test writing graphml files that have been read:

- numbers of `Node` and `Edge` from 2 files are equal
- sum values of attributes from 2 files are equal

`GraphMLOutputwRunTest01` and `GraphMLOutputwRunTest02` tests writing files and reading them again:

- `UdpTestEvent` runs successfully

5.4.2 Ipv4PacketInfoOutputTest.cpp

`Ipv4PacketInfoOutputNormalTest` checks captured data:

- outgoing list is larger than delivered list
- packet types are correct
- packets' travel time is as expected

`Ipv4PacketInfoOutputExtremeTest` checks captured data:

- total number of bytes is as expected

5.4.3 SimplePacketInfoTest.cpp

`SimplePacketInfoTest` tests the attributes of `SimplePacketInfo`:

- setters and getters
- conversion of attributes' values works correctly

5.5 simulation

5.5.1 EventFixtures.h

Different simulation scenarios for `UdpTestEvent` only:

- `EventFixture_P2P`: fully connected, line topology, disconnected event
- `EventFixture_Wifi`: connect wifi subnets together through a wifi net
- `EventFixture_Mixed`: wifi subnets form a line topology (with p2p connections)

5.5.2 UdpTestEventTest.cpp

`UdpTestEventExceptionTest` checks exception handling:

- null nodes
- same node

`UdpTestEvent...Test...` uses fixtures mentioned above to test running simulations.

`UdpTestEventSimpleTest` tests `ReSchedule()`:

- event runs successfully
- number of recorded bytes is as expected

5.5.3 TcpAppEventTest.cpp

`TcpAppEventTest` tests sending 20 MB through an infinitely fast network.

5.5.4 UdpAppEventTest.cpp

`UdpAppEventTest` tests sending 20 MB through an infinitely fast network.

5.5.5 SimulationTest.cpp

`SimulationMembersTest` tests :

- `GetNode()`
- `FindDevicesFromNode()`

`SimpleFlushTest` tests the private method `Flush()`.

`IPAssignmentTest` tests IP address assignment:

- numbers of IP addresses can be accommodated are as expected
- exception thrown when addresses overflow

`SimulationExceptionTest` checks exception handling and no exception.

`TopologyAttributeTest` tests attributes/properties of networks in general:

- setting attributes via `Edge`
- getting attributes via `FindDevicesFromNode()`