Master's Thesis

Simple Software Development Techniques for Wireless Network Simulator

Hyun Woo Choi

Department of Electrical and Computer Engineering

Graduate School of UNIST

2015

Simple Software Development Techniques for Wireless Network Simulator

A thesis/dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Hyun Woo Choi

7. 15. 2015

Approved by

Advisor

Changhee Joo

Simple Software Development Techniques for Wireless Network Simulator

Hyun Woo Choi

This certifies that the thesis/dissertation of Hyun Woo Choi is approved.

7. 15. 2015

Advisor: Changhee Joo

Thesis Committee Member #1: Hyoil Kim

Thesis Committee Member #2: Hyun Jong Yang

Contents

List of Figures

List of Tables

List of Algorithms

ABSTRACT

Network Simulators have been widely used in many uses of academic and educational areas. In advance of applying experimental simulation results to real world networks, it is important to verify their theoretical network performance and some of specific event routines through network simulators (***ns*-2**, ***ns*-3**, ***OPNET***, ***NetSim***, etc.). If the test and verifying procedures are omitted, the system may cause some unexpected problems which can result in unreliability, unexpected maintenance costs and so on. Therefore, it is a common practice to demonstrate the performance of network prior to public release using software simulator. In this context, we focus on problems of the most renowned simulation package, ***ns*-3**. In the point of simulation development performance, the ns-3 uses some of unnecessary modules compared to a customized solution, in particular, for a small and simple network simulation. In this thesis, we define several problems and disadvantages of the ns-3, and present solutions and comparison results to solve high complexity problem of simulations and the accessibility of simulation source code to the beginners who are about to use the simulation for educational use: *1) we present a simple software development techniques which reduce the development times and the number of needless module in C++ objected-oriented programming environment*; As the other way to implement a simple discrete network simulator, *2) we also present the simple network simulator development algorithm(**SNSDA**) that describes programming steps for discrete-event network simulator*; Finally, we compare our simple development algorithm to the ***ns*-3** in case of the most general IEEE 802.11 based wireless network scenario so that the SDA has some advantages in wireless network simulation.

1. INTRODUCTION

In recent years, most researchers have verified and proven their network studies through the various network simulators in response to the various technical requirements and academic achievement [7].

The two popular discrete network simulation packages (including commercial software) are ***ns*-3** and ***OPNET***. The former is the most well-known discrete-event network simulator for the network research communities and the letter provides a powerful and intuitive GUI-based simulation environment with a variety of features [1] [2]. In case of the ***ns-3***, the *network simulator 3* is generally used in the scientific researches because of its function features and network modules are well defined and it is even consisted of open source which is easy to be handled and practically comprehensible.

Recently, it is important that reducing the compilation time and considering putting more resources to the scalability in network simulators and they are the main issues. Nevertheless, in spite of the efforts of simulator designer groups, it still remains as an obstacle toward them [5].

From the view point of network researchers, the scalability and compilation time issues are not much sensitive to their works. Instead, the undiscerning callback function structure of ***ns-3*** can be more important since it lowers the network simulation performance and incurs significant amount of function overhead. Since it is not effective that most network researchers configure and customize the structure of function calls in their simulation, it is on the rise that we should consider the legit simulation structure for small to large-scale networks.

As part of that effort, it is obviously needed to pay attention to the simulation structure and function callback system in existing network simulator, such as ***ns*-3**. As a first defect of existing simulator, even though ***ns*-3** or ***OPNET*** help network researchers who do not familiar with using programming language to make up his own simulation source codes, there are strong demand to use a simple network simulator which does not require in-depth understanding of the structure of network simulator. Secondly, the structure of simulation in ns-3 often requires many modules, attributes and callbacks that turn out to be unnecessary. For example, in case of a basic IEEE 802.11a wireless network simulation, ns-3 imposes insertion of a complete set of network parameters on the users. That is why we should implement a fully customized network simulator and study an approach to get the legit output without useless tracing sink through script language such as ***python***.

To resolve these problems, we proposes a simple approach for development of discrete network simulator and compares it with ns-3 simulator. As the result from consideration of reducing useless modules and function call overhead, we declare and allocate the function as inline function. In case of a function that is frequently called, we just use the normal function callback. After consideration of tracing system and function overhead, we present an approach how to configure discrete network

simulation parameters and network topologies. Finally, In order to reduce programming complexity and unnecessary attributes, we present a programming approach to build a simple network simulator in response to quick testing needs.

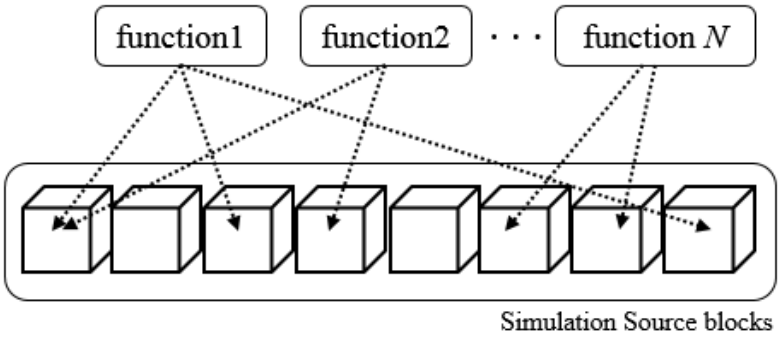

Figure 1. the inline function approach in the source code blocks.

The rest of this thesis is organized as follows. In section 2 we review the concepts and preliminaries of discrete network simulators and the most popular discrete network simulation package, ***ns*-3**, with focusing on its advantages and disadvantages in the function callback and tracing system. In terms of tracing system, we study and show how to reduce the unnecessary components to print an exact output only. Furthermore, the specific example and system flow assist the user to understand the pros and cons of ns-3 tracing subsystem. In section 3, we describe the advantages of ns-3 operation flow with the same example network first (a single-hop wireless network with 1 service node and 2 device nodes). Then we discuss the actual callback overhead through the example wireless network algorithm of ns-3. After the discussion of ns-3, we present the simple network simulation development algorithm (**SNSDA**) for a discrete network simulator and configure base network parameters for discrete simulator. Furthermore, in section 3.3, we focus on how to design a simple discrete network simulator with using object-oriented programming approach and its exemplary applications. Lastly, chapter 4 concludes and describes some advantages of GUI based network simulators.

2. DISCRETE SIMULATORS

Some of features of discrete-event simulators are distinct from other simulator types. In a discrete-event simulator, variables change instantaneously at each different time slot while the variable of continuous model changes incessantly with respect to time. Since the simplicity of discrete-event simulators, is has been implemented in many simulations uses. The main components of discrete-event simulations are:

a) system state – the set of state variables necessary to describe the system at a particular time;

b) main program – one of subprogram which executes the main simulation routine and determine which event come to the next;

c) clock variable – a variable which offers the current simulation time;

d) event set – a set of subprogram routine that updates that simulation state when a specific condition is satisfied;

e) statistical counter – a set of counter variable to store information that the simulation system requires;

These 5 essential components must be set before creating of discrete simulator. Eventually, the main reason to using network simulator is not only reduce the maintenance cost, but also let the target network be reliable in advance of applying to the real world. In ***ns-3***, the simulation clock is maintained as a 64-bit integer in a unit specified by user through the *Time::SetResolution* function. This means that it is impossible to specify event expiration times with anything better than user-specified time accuracy [2]. In addition, the synchronizer class in ns-3 manages the simulation events to some real time and the ns-3 also provides the synchronization between the simulation clock and real-time clock. Unlike with the ns-3, the proposed simulator development algorithm by this thesis uses the event clock through the specifically declared time variable.

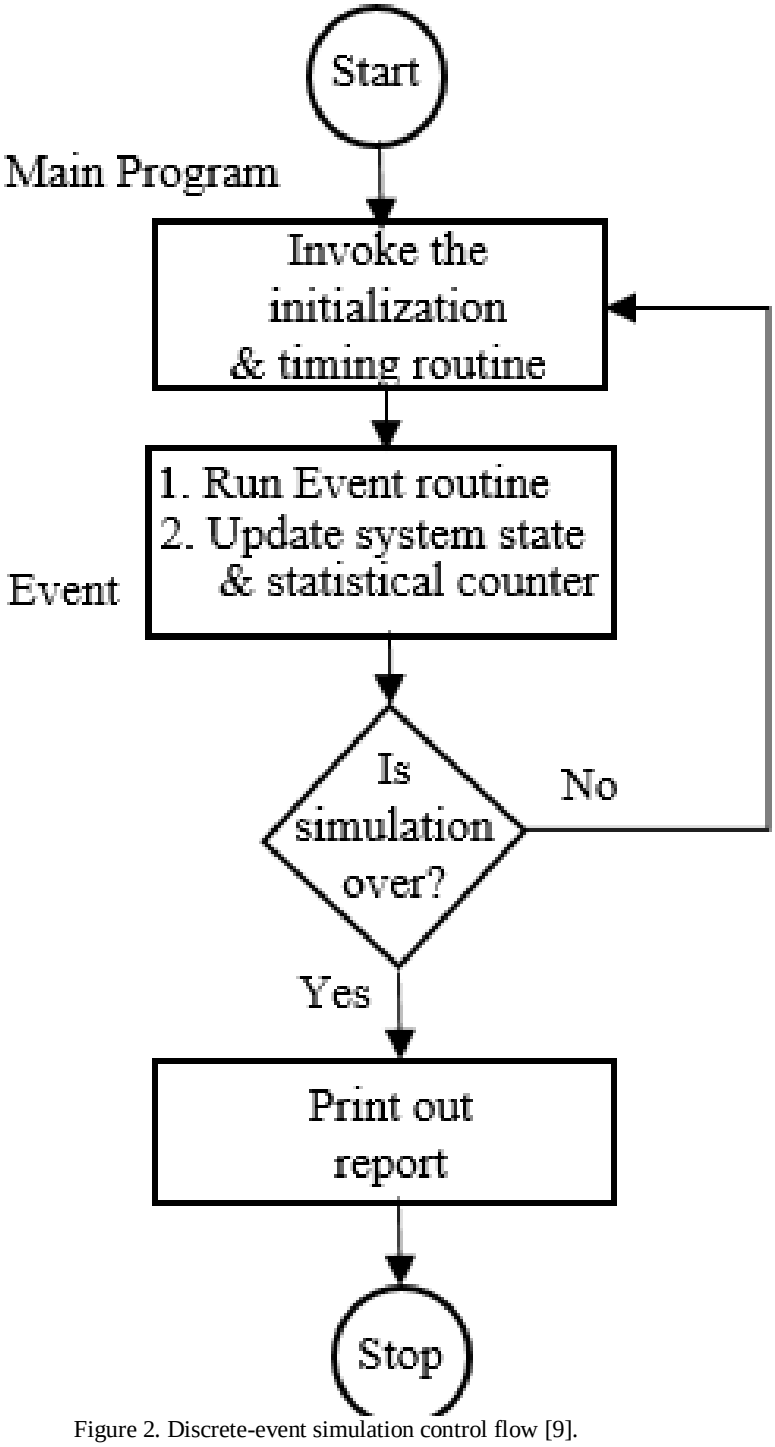

Figure 2. Discrete-event simulation control flow [9].

2.1 Concepts and Design

Literally, network simulation is kind of technique where a program models the behavior of a network either by calculating the interaction between the entities [8]. Unlike with the continuous simulator which has continuously changed variables that described by differential equations, the discrete-event simulator has synchronized simulation clock for each different event case.

The main control flow of discrete-event simulator follows these steps:

1) initialization step– the simulator invokes the initialization routine. When the main program calls initialization functions, it initiates simulation clock, system state and statistical counters. Then, the main program chooses and initiates event list to run a specific simulation event routine.

2) event routine– the event routine does update the system state and statistical counters. If the target network model is stochastic and not deterministic, the event routine let library routines generate random variables.

3) generate report – When the simulation is over, the subprogram which generates report computes estimates of interest and prints out the simulation results.

Furthermore, the discrete-event simulation has an ability to tamper the simulation so that the network researchers can pause the simulation at any steps and restore simulation states easily. Therefore, as remarkable merits of discrete event simulation, conjugating deterministic and stochastic simulation is possible [9].

2.2 The ***ns*-3** Simulator

***NS*-3** is a discrete-event network simulator which is targeted primarily for research and educational use.[4] It is free software and licensed under the **GNU GPLv2** license and publicly available for development and use. Since it is released in 2008, it is now the most renowned and generally used discrete network simulation software package in world.

***NS*-3** is designed and developed to replace ***ns*-2** which was released in mid-90s. The biggest reason for redesigning the ns-2 was to overcome the "limited scalability regarding memory usage and runtime." as mentioned in [2]. Therefore, the ns-3 is distinct from ***ns*-2** which mainly reduces the compilation time by using **C++** source code.

In recent years, many simulators are focused more on scalability and performance, not only on compilation time. The Figure 2 shows that the ***ns*-3** simulation control flow.

The ***ns*-3** topology helper assists creating network topology and makes application. Then, the node container and InternetStackHelper add protocol stacks to the node in order to communicate among of them. After setting up the MAC and IP addresses, users create the application class and fill up all simulation logic.

However, it is not always convenient and it does not have only advantages. Credibility and validation are still remained as obstacle and the scalability too. The limitations of ns-3 are more introduced in [5].

Furthermore, in simulation design processes, we mainly focused on the tracing script sources and its limitations in ns-3 and its performance penalty so that the simple simulator should reduce the programming parts that can be ignored.

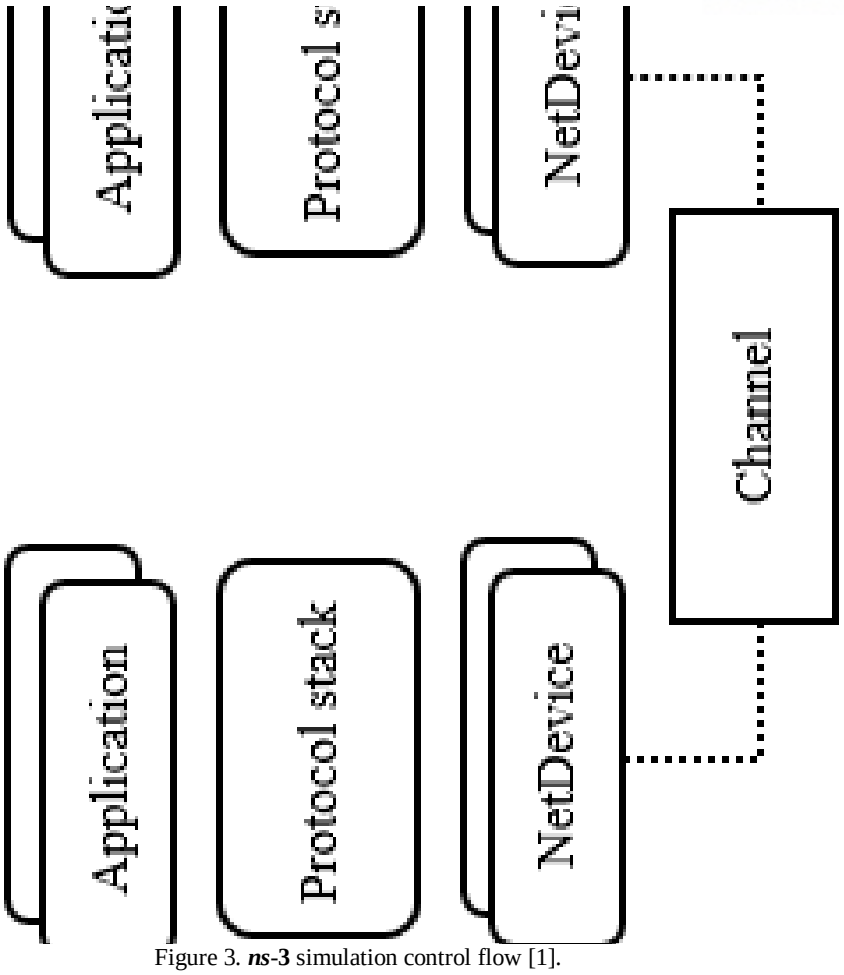

Figure 3. ***ns*-3** simulation control flow [1].

2.3 The Tracing Process

As stated in [3], the main aspects of ***ns*-3** simulation workflow are helper classes. The helper classes assist and contribute to programming the network simulation and its topology with protocols and applications. After the network installation steps, the tracing system come up to generate simulation output for study. The tracing system (called as tracing in previous version of ns-3) is one of the most important subsystem in ns-3 and the whole points of running a simulation is to generated output for the network study as stated in [4]. Using tracing system in ***ns*-3** means that researchers follow the generic pre-define output mechanism and parsing their content to extract interesting information. In this context, the user has an advantages from the supported tracing system. In case of programmers and researchers, it is obviously convenient that using generic pre-define tracing source makes them not requiring change to the ***ns-3***. In spite of the advantage, researchers should write scripts to parse and filter for data of interest [3]. Furthermore the simulation program in ***ns-3*** must be written in the manageable form so that user should consider more than developing own output mechanism. The format of simulation output, *NS_LOG*, is only available in debug builds, so it may be a performance penalty to the network simulation. As illustrated in figure 3, the user designs the tracing sinks that specify what information capture and what to ignore [5]. In this context, we show the efficiency of discarding the useless parts of simulation and how to get rid of them in section 3.2.

The figure 3 shows that the structure of tracing subsystem which is *1:n* relation that a trace source can have *n* trace sinks. Since the trace source cannot independently exist in the simulation, the user must match the trace sinks as a source consumer with the trace source. Thus, because of these inherent characteristics of ns-3, it is practically difficult to fine-grained control of output without tracing subsystem. That is the reason so that we carefully note the complexity of tracing system. In summary, we briefly analyze the drawbacks and the operating flow of the tracing subsystem with a simple example source code in ns-3. In figure 4, since the tracing system is connected with the attributes and the attributes are associated with the objects, the trace source must be inherited the objects [3]. The most important part of the source code in figure 4 is the declaration of *TracedVaule* and call of *AddTraceSource()* function. Since it is obvious that the *AddTraceSource()* function provides a hook to connect the main simulation event routine and TracedValue, it produces chronic problems. We describe the problems of tracing subsystem in the Table 1.

Table 1. The kinds of problems, and the locations, and the degree of risks that may be occur through the tracing subsystem.

Problem	Location (code)	Criticality
Callback function overheads	internal and external	high
Poor fine-grained control	external	extremely high
Code complexity	internal and external	low

* The criticality is the measurement which decides acceptable criterion in the simulation

```
class MyObject : public Object {
public:
  static TypeId GetTypeId (void)  {
    static TypeId tid = TypeId ("MyObject")
      .SetParent (Object::GetTypeId ())
      .AddConstructor<MyObject> ()
      .AddTraceSource ("MyInteger",
                       "An integer value to trace.",
                       MakeTraceSourceAccessor (
    return tid;
  }
  MyObject () {}
  TracedValue<uint32_t> m_myInt;
};
```

Figure 4. An example code of tracing subsystem in ns-3 [3].

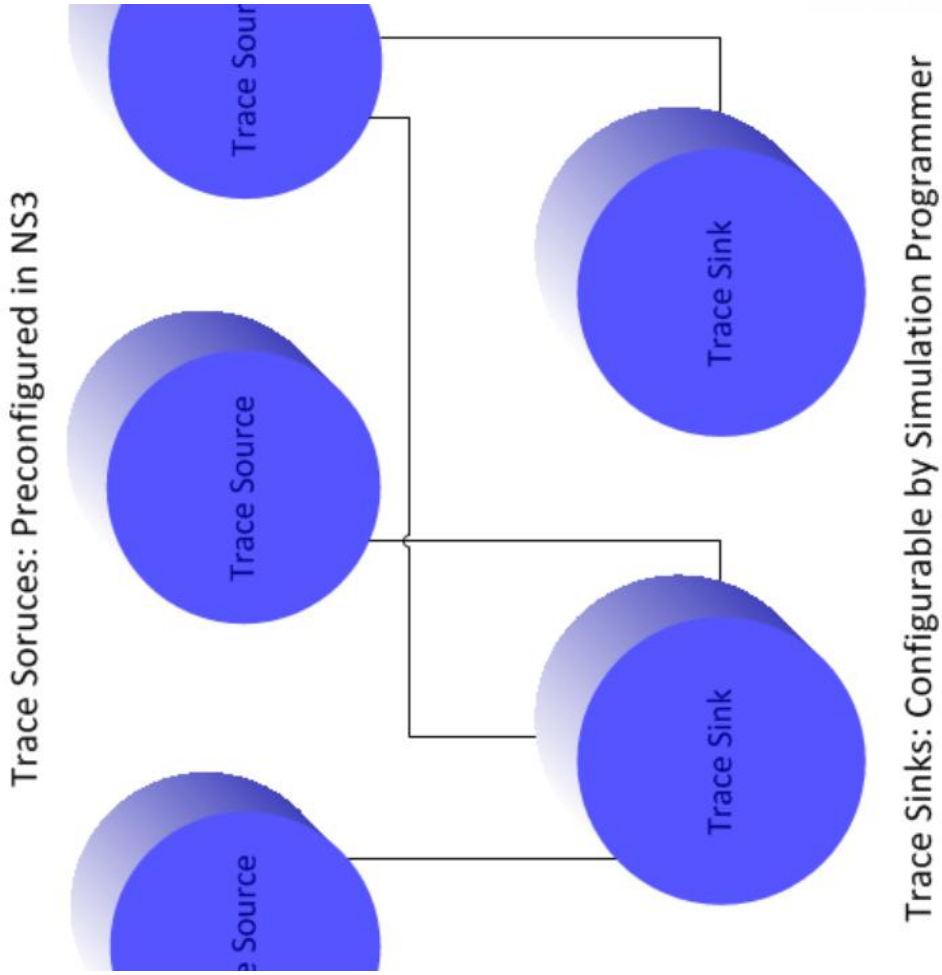

Figure 5. Tracing sources and sinks [5].

3. DEVELOPMENT TECHNIQUES

So far, we described the simulation characteristics and disadvantages so that we elicit the solutions which resolve the 3 problems of ns-3 network simulator. As the aforementioned solutions, we firstly follow the direction of simulation flow of ns-3 with a specific network example. Secondly, after the analyzing ns-3, we propose a simple network simulation development approach and the algorithm of our simulation. Finally, we describe the advantages of the proposed simulation based on the fact that we compare the ns-3 and the proposed network simulation. As stated in Section 2.1, user should design and follow several steps to create discrete simulator.

1) find the best simulation type – the simulation type has to be matched and closed to the research problems. If the set of input data is certain and never changed, the simulator has to be a deterministic. If it is not, users should consider it as stochastic randomness. In case of chaotic model, the simulation type must be the deterministic and there are no predictable elements [9].

2) time aspects – after 1), user should figure out the time model between the static and dynamic model. The former does not need the time scale, and letter changes time variables across the time.

3) discrete and continuous model – when the dynamic model is decided, the time scale of simulation should be decided. The discrete time scale model slices whole time into normally formed slots and they can be numbered in integer field. In other way, the continuous time scale model can also be selected if there is no need to divide time scale into time slots. In that case, the time variable is remained as continuous variables.

In figure 5, the diagram shows the flow chart of a simple network simulator. As mentioned in the last chapter, it is important to decide and define the simulation model, topology and protocol before creation of simulator so that it can be easily implemented. The tracing process can be ignored here so the useless procedure makes the number of program source code line.

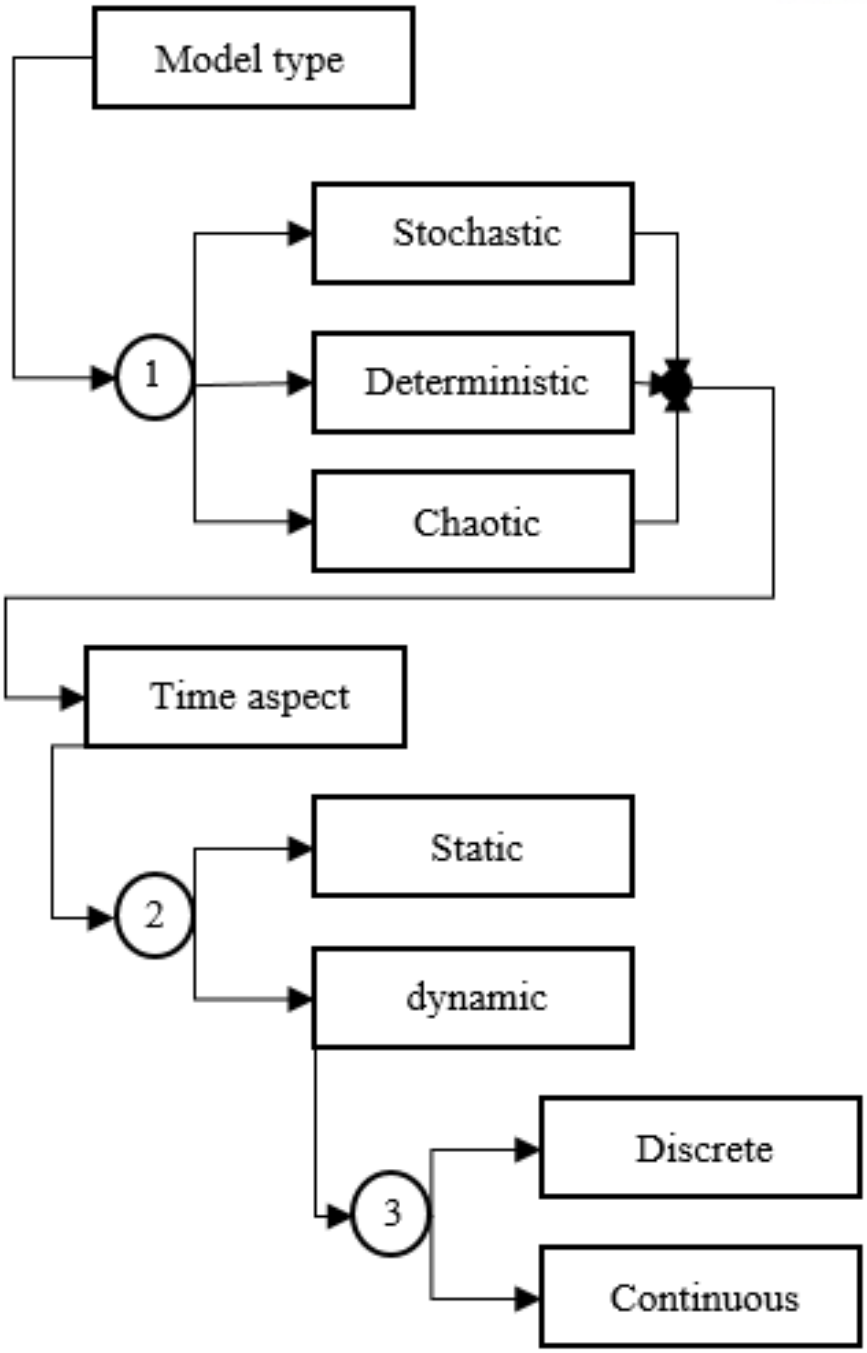

Figure 6. Simulation type selection diagram [9].

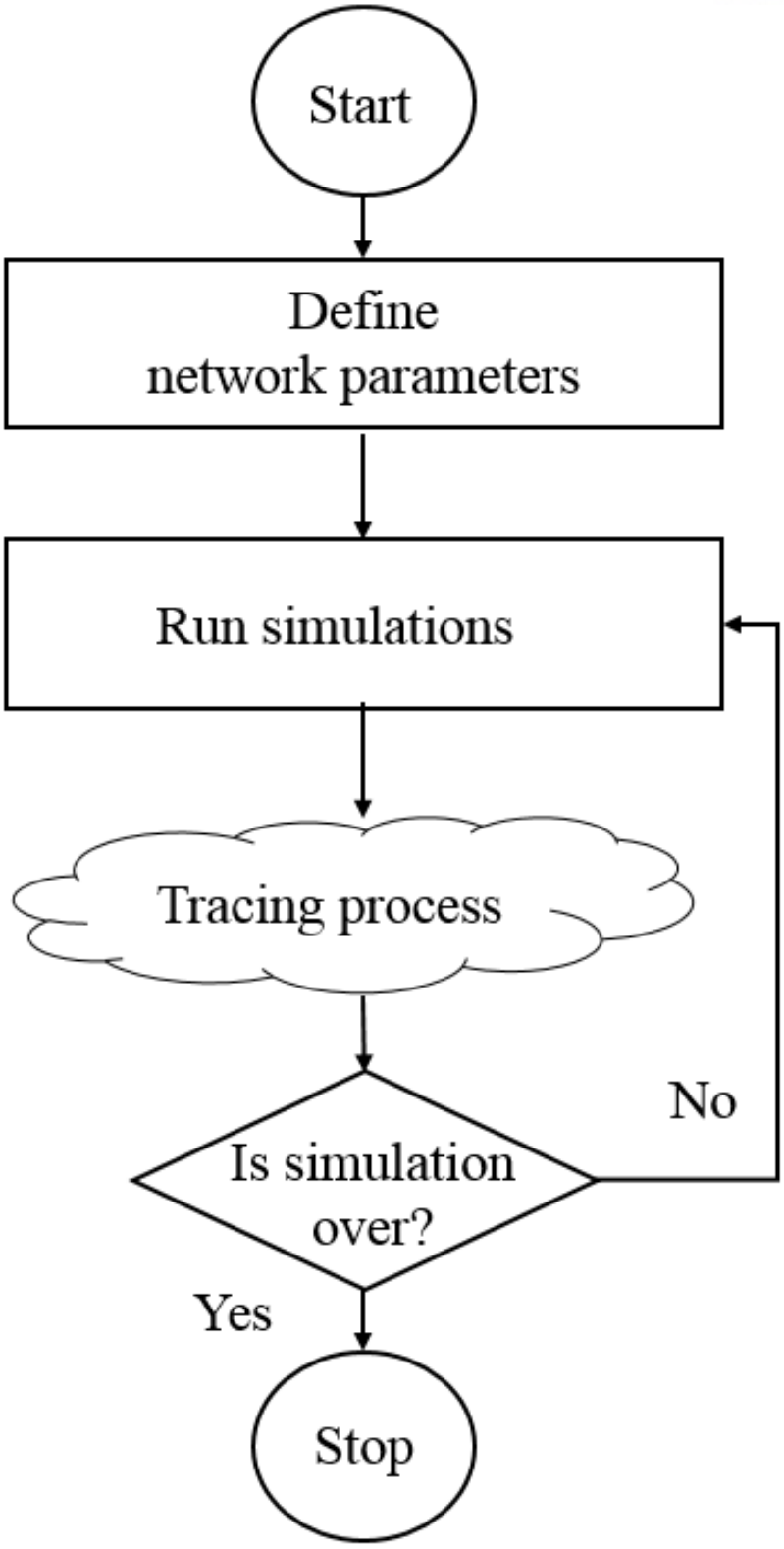

Figure 7. A simple network simulator flow [8].

3.1 Simulator Development Approach

In this chapter, we start to create an example discrete simulator in IEEE 802.11 wireless network. In advance of creating the simulator, we should choose the running and programming environment where is compatible with multi-platform (Linux, Microsoft windows, etc.). For that reason, we adopt the ***C++*** programming language with using standard library to let it be operated and compiled in multiple operating platform, e.g., Linux and Microsoft windows OS.

For reference there are two reasons why we do not use the script language such as ***Python***:

1) nested functions – can often be a drawback which is hard to modify the set of variables in outer scope.

2) hard to catch the syntax errors – if the simulator is a simple and concise source-coded program, it does not matter to use. However, it rises potentially to the debugging surface. Since the main goal on this thesis is creating a simple and general use discrete-event simulator, we avoid that risk in advance. Since the complexity of simulation source code, the installation of network topology and its applications are tend to be overlooked.

Moreover, what we should concern at the next step is design the main simulation case and network parameters. In this context, we present an exemplary test case which is single-hop wireless network that each service device has a single server queue:

· *randomly generated packets* – packets are randomly generated with probability *0.33* and they are stacked into the queue of service node (the probability 0.33 is considered for the convenience of simple calculation in this example network). When the queue of service node has the packet to send, the service node transmits packets to the device nodes. Since the example network has just 2 device node, they are about to receive the packets by Bernoulli random process in each unit time slot. Also, we assume that the queue is unbounded.

· *separated communicate region* – the simulation topology has two transmission flows. *a)* the service node transmits its packet to the device node 2. b*)* since the limited communication environment, the device node2 cannot transmit the packet to the node 3. *c)* when the packet is transmitted from service node 1, it randomly decides whether the receiver is node 2 or 3 by the probability of *1/2*.

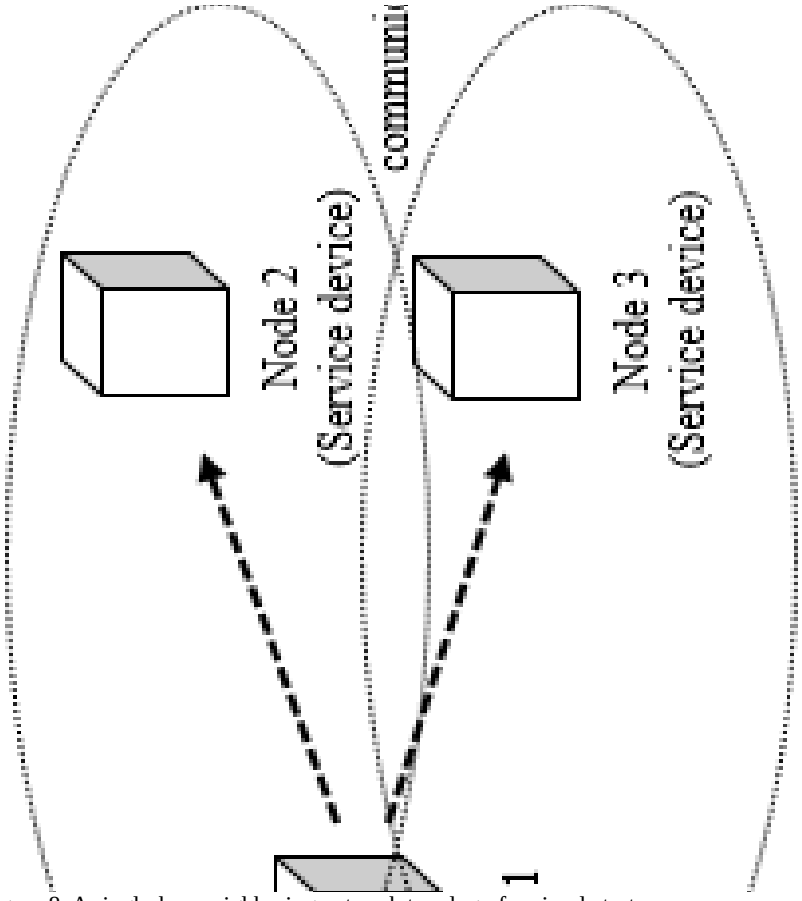

Figure 8. A single-hop neighboring network topology for simple test

3.2 Simulation Algorithm in ***ns-3***

As described in figure 6, we present the whole process according to the example scenarios with using ns-3. The main reason that we perform this simulation through ns-3 is to compare the advantages and disadvantages with the proposed simulation algorithm in this thesis. Before we start to run our network simulation in ns-3, the helper class assists to program the actual network topology and the used protocols. The only thing that researcher should know is understanding ns-3 components and network modules so that the user can simply consider the specific network topology, trace sink and its applications. In figure 6, there is no tracing source code to obtain simulation output so that the users should connect their trace sink to the pre-defined trace sources. Since it is practically difficult to control and add the new network function and even the parameters, the users have to consider the network functions to put on the source codes of main event routine. For example, if the users want to check and modify the above network topology and the conditions, they must use callback function call from the other classes. We attach the example callback function, *CourseChange()* to quickly inquire the structure of callback function of *ns-3*.

```
void CourseChange (std::string context, Ptr<const MobilityModel> model)
{
    Vector position = model→GetPosition ();
    NS_LOG_UNCOND ( "x = " << position.x << ", y= " << position.y);
}
```

Figure 9. A callback function for course change [4]

This above function prototype conducts a rule as trace source in the simulation. As we described in section 2.3, the tracing system, the users must create a trace sink in advance to use this callback function. Moreover, the trace sink for this case is even somewhat complicated because it contains a certain amount of script language and function usage.

In ns-3, all detailed functions are triggered by callback mechanism that the goal is to allow one piece of simulation code to call a function without any specific inter-module dependency. Optionally it can be seemingly simple that the users call the ns-3 functions without any necessary background knowledge, but it also has a drawback that it may take a many steps of lookup during the linking and compilation time. Moreover, if there is no parameter type which cannot be supported by ns-3 callbacks, the user must add and modify the prototype of function in the specific header. Nevertheless, the ns-3 simulator is the easiest simulation tool if the condition that user knows the system mechanism and helper classes.

hm 1 *ns-3* simulation algorithm

ore-module	**call** Mobility Helper;
etwork-module	**create** object Position Allocator;
pplications-module	**set** PositionAllocator (vector (0.0,
ifi-module	**set** PositionAllocator (vector (30.(
sma-module	
ternet-module	**install** mobility;
oint-to-point-module	
	call Internet Stack Helper;
de Container ApNode(1);	**install** internet;
de Container StaNode(nSta);	
	call Ipv4 Address Helper ipv4;
ode;	**set** base;
ode;	**set** device address;
	set packet size;
Helper;	**set** data rate;
agation delay;	
agation loss;	**start** simulation (Seconds)
nel;	**start** application (Seconds)
	stop application (Seconds)
te Station Manager;	**stop** simulator (Seconds)
	destroy simulator (Seconds)
t device Container staDevice;	
fi;	
t device Container ApDevice;	
evice (ApDevice);	
evice (StaDevice);	

Algorithm 1. *ns-3* wireless network simulation algorithm for single-hop wireless network

3.3 Simple Network Simulator Development Algorithm (**SNSDA**)

Considering the 3-node simulation scenario, we present our simple network simulation development algorithm (**SNSDA**). We first consider the classes and the parameters that we have to design, and then descript the development steps of the simulation.

The essential network classes and parameters include:

a) node class – is similar with the ***ns*-3** node container which calculates the transmission costs and keeps the track of a set of node device structure and pointers;

b) event clock timer – is the synchronized timer which is triggered by every event routine. Moreover, it gives us the information of current time value of a simulation event;

c) the number of generated packets – the cumulative amount of generated packet is one of main factor to figure out the throughput;

d) the number of transmitted packets – is determined by the sender's arrival rate and one of important statistical parameter to calculate throughput;

e) the number of received packets – when the receiver node receives the packet from sender successfully, the number of received packets triggers the formulation of throughput;

f) arrival rate – affects to the number of generated packets from sender node;

g) flow decision probability – in this simulation scenario, it is decided by half and half probability;

h) transmission time – is theoretically fixed at 10 micro seconds and one of the key components to calculate throughput;

In figure 8, the event clock timer is set and run by transmission time and waiting time (as a packet length). After the setting of the network topology, node and link object header, the user really does not have to define or declare more steps such as unessential IEEE 802.11a standard parameters. Since the trace process and trace sink source are omitted, the user can easily obtain the appropriate simulation outputs through the simple calculating source code (in this simulation scenario, $throughput = time\ scale * \frac{(Transmission\ time + Waiting\ time)}{Received\ data\ size}$) and not a complex tracing subsystem in ***ns-3*** written in ***python*** script language. Sequentially, the programming steps in algorithm 2 are:

1) design the topology – is often described in the main program source scope with abstraction of the nodes and links. The theoretical locations of service node and device nodes are very important if there are an assumption that the concept of carrier sensing range is applied.

2) node and link class – after design the network topology, the user should code the node and link header that perform the basic routine of node and link. In general, node class has queue operation source code there, since it is natural that each link has its queue so that it manages the queue length and the number of empty slots. Link class takes on the role that connects each links and logical

connection among the nodes. In this context, it is also obvious that link class includes the transmission and receive packet function that triggers to the each node's queue.

3) packet class – the packet class must have two properties that should be contained. The former is the source destination address and the letter is the destination address and sequential number in this simulation model (other properties are declared in the main program scope).

4) main event routine – in this example network mechanism, the AP node only transmits the packet to the station nodes by packet arrival rate and decision probability that chooses a station node. Though the mechanism of simulation is simple, the user must consider and calculate that the event clock timer should be accumulated at every action tick. Furthermore, unlike with ns-3, the tracing part and output generate source codes are mostly located in the main event routine not external scope.

Then we need to discuss to the programming point. Since this simulation code is written in C++ language, it is optionally suggested not to use pointer so that the simulation can avoid memory fault problems (use STL or other library enabling not to use memory arbitrarily [9]). Moreover, the user can use C++ preprocessor to cut out unnecessary part of code in advance of the running time.

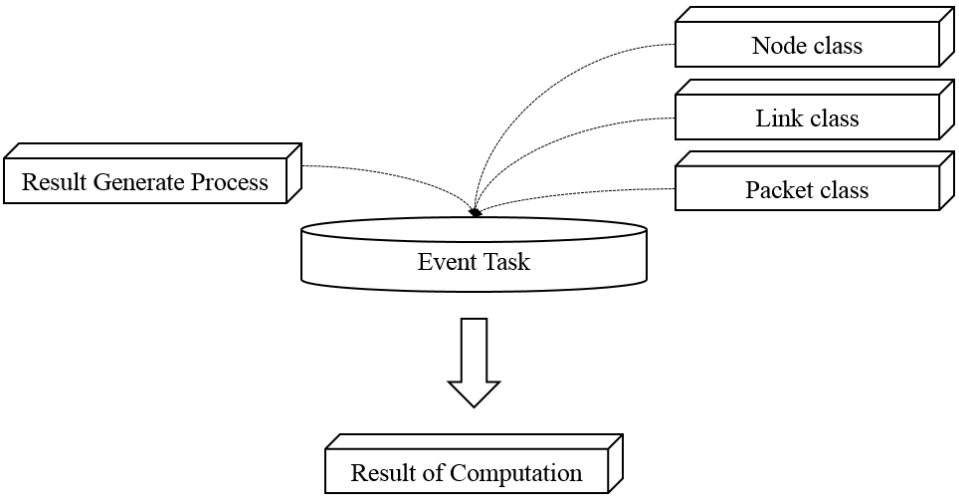

Figure 10. Class structure of **SNSDA**

In addition, we have found some problems that must be overcome while users create the discrete-event simulator. In case of the development of deterministic simulation, we must predict and prevent the unexpected situation that one of nodes processes more than 2 tasks at once. In other words, letting all of nodes be conceded right action priority by legit simulation procedure. In order to address this problem, user must program the anti-exceptional routine processing source code. In [9] and [10], they recommend that users should implement the queue as one of part of the node class. In object-oriented programming environment, it is more reasonable and take the compile time less that one instance handles every task in the particular node. In section 3.4, we discuss the comparative advantages which are the proofs of not using tracing system and callback function inheritance structure.

Algorithm 2 *simple* network simulation algorithm

```
call Node instance
     Link  instance
     Event list

Set ApNode = 1;
    StaNodes = nSta;
    DistanceBetweenNodes;
    PacketSize;
    DataRate;
    TransmissionTime;
    Clock timer = 0;

start Simulation(nPkts);
 for simulation()
     {
      if (nPkts > 0)
        if (the Queue length of ApNode > 0)
          if  (0 < arrival rate < 0.33)
            if (1.0 ≥ decisionProbability ≥ 0.5)
               send data packet to StaNode0;
               clock timer = clock timer + transmission time;
               update Statistical parameter of StaNode0;
               generate packet to ApNode;
               nPkts--;
            end if

            else if (0.0 ≤ decisionProbability ≤ 0.5)
                send data  packet to StaNode1;
                clock timer = clock timer + transmission time;
                update Statistical parameter of StaNode1;
                generate packet to ApNode;
                nPkts--;
            end if
          end if
          else
                generate packet to ApNode;
                clock timer = clock timer + unit packet length;
                nPkts--;
          end if
        end if
      end if
        else
          print throughput of each StaNode0,1;
      }

Stop Simulation();
```

Algorithm 2. simple network simulation algorithm

3.4 Advantages of **SNSDA**

In Sections 3.2 and 3.3, we have clarified the differences between ns-3 simulation and the Simple Network Simulation Development Algorithm (**SNSDA**). In case of the ns-3 which mainly uses the tracing subsystem, the main program (main event routine) is obviously simpler than the **SNSDA** because of the callback trigger and included headers encapsulate and hide the core attributes and objects. However, in spite of the clear advantages of ns-3, we found out some of drawbacks from tracing and callback function so that the network simulator has longer periods of linking and loading processes than the others.

In the latest linking process, the linker decomposes the many of source file to the manageable modules that can be modified and compiled separately [11]. Since the simulation source files are not the one monolithic file, the linker and loader must go through many decomposition processes. The more object files guarantee the slower compilation time so that the reason to cut out of useless modules and attributes appears. For that reason, the user should cut off the indiscreet uses of callbacks and network modules in the network simulators. In particular, the ns-3 does not provide the delicate modification of the trace source, the use should type the source code for the trace sink which calls the callback functions. In the **SNSDA**, we originally allocated the frequently called functions inside of main program scope and called it as inline function to let the linker do not refer to. In addition to these approach, we can freely modify the function prototypes through the direct access to the node, link and packet classes. It makes the proposed simulator tool more intuitive and let the simulation have less function call overhead.

In Table 2, we summarize the pros and cons of each network simulator. As the first criterion of this comparison, the function overhead frequency means that the callback function structure in ns-3 and **SNSDA** let the callback functions be called frequently from the stack so that it consumes the amount of free memory space of the stack. Secondly, as we aforementioned in the previous Section 2.3, the relation between tracing source and sinks makes the fine-grained function control practically impossible. In **SNSDA**, it is easier than the ns-3 in terms of the modification of function prototypes because the **SNSDA** does not connect all the attribute of simulation and objects so the users do not have to consider the trigger effect of function and network parameters. Furthermore, we discussed shortly about the memory fault problems in network simulations. Since the usage of pointer increases the probability that the memory faults and the segmentation faults occur, the **SNSDA** adopts STL which is the standard template library so that a simple memory management approach and even it provides a simple method for many data structural functions. It is definitely risky that the users make many pointers which indicate a same objects. If the one of objects is deleted, the other side has an invalid address value. In ns-3, the pointers are used in every classes and attributes though they use the

smart pointer which is a class declaration of STL. In these context, we realize that a simple network simulation scenario does not have to have unnecessary modules and attributes so that the users satisfy with the performance of network simulation. In terms of extendibility for adding new modules and the platform compatibility, the **SNSDA** is even intuitive than ns-3 because of its light-weighted characteristics.

Table 2. The differences between ***ns-3*** and Simple Network Simulation Development Algorithm (**SNSDA**) and five standards stand for **SNSDA** which does not use the tracing subsystem but inline function style.

type	function overhead frequency	function control level	memory fault	Extendibility of new modules	Platform compatibility
Ns-3	oftentimes	difficult	possible	difficult	bad
SNSDA	infrequently	easy	infrequently	easy	good

* The memory fault is occurred by using pointers in simulation source codes.

3.5 Performance **Evaluation**

So far, we have clearly described the pros and cons of ns-3 and SNSDA. In this section, we numerically compare their performance. We excluded the comparison of the number of the program source code lines, because *a) the programming styles are often different*, and *b) in general, the amount of source code line does not have an effect on the linking and compile processes* [11].

We measure the function overhead frequency and the function control difficulties by the number of function used and relation among the objects, attributes and function prototypes. Along with an example callback structure of ***ns-3*** and **SNSDA** in figure 11, then we attached the results which is records that traced all of function calls in the Table 3.

In figure 11, we describe the results of *CsmaHelper::install()* function which creates an *ns3::CsmaChannel* with the attributes configured by *CsmaHelper::SetChannelAttribute()* with the attributes configured by *CsmaHelper::SetDeviceAttributes()* and then adds the device to the node and attaches the channel to the device. Even though it seems complicated, we omitted the unimportant function calls behind *ns3::Object::ConstructSelf()*. Given these facts, we counted the number of function call in the ns-3 simulation and the **SNSDA**. In case of **SNSDA**, it is obvious that the function callback is not happened since we allocated the function call inside of the main simulation program source. As we can see in the figure 11, the function series of *object class* in ***ns-3*** are redundantly referenced by most of the functions in the example simulation and it may cause the stack overflow if the redundant function call consumes the most of memory space in the stack. If it happens while the ns-3 simulator runs, the users confront the segmentation fault after simulation compile.

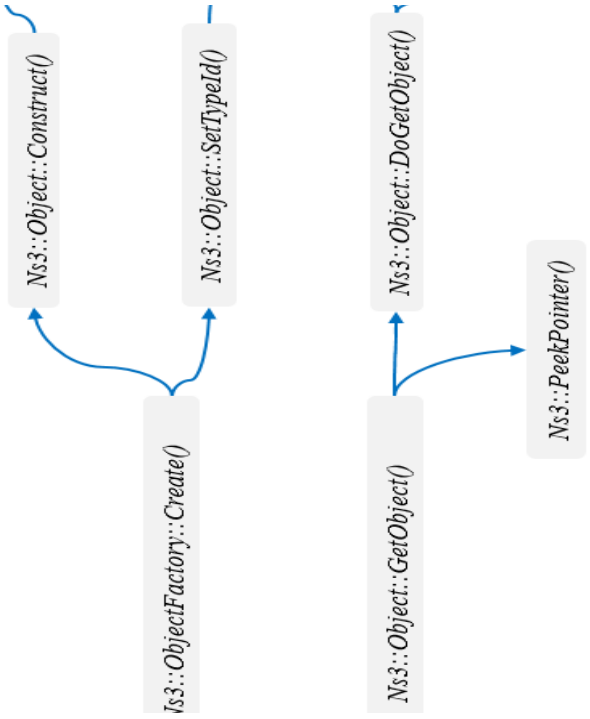

Figure 11. A function call graph for *CsmaHelper::install()* function in ***ns-3*** simulation [4]

Table 3. The number of function calls in ns-3 for example simulation in this thesis.

function	# of calls
ns3::PointToPointHelper ::SetDeviceAttribute	2
ns3::PointToPointHelper ::SetChannelAttribute	2
ns3::PointToPointHelper::install	3
ns3::NodeContainer::Add	6
ns3::NodeContainer::Create	2
ns3::CsmaHelper ::SetChannelAttribute	32
ns3::CsmaHelper::install	36
ns3::YansWifiChannelHelper::Default	19
ns3::YansWifiPhyHelper::Default	18
ns3::YansWifiPhyHelper::SetChannel	30
ns3::WifiHelper::Default	18
ns3::NqosWifiMacHelper::Default	18
ns3::NqosWifiMacHelper::SetType	34
ns3::WifiHelper::install	100
ns3::InternetStackHelper::install	12
ns3::Ipv4AddressHelper::SetBase	12
ns3::Ipv4AddressHelper::Assign	88
ns3::Simulator::Run	50
ns3::Simulator::Destroy	8
The total number of function calls : 490	

* When a function is called, the # of counts includes all reference functions.
** We did not count the functions in C++ standard library.

Unlike with ***ns-3***, the **SNSDA** shows somewhat different results to us since it adopts the different simulation structure. In Table 5, we put the result of function calls from **SNSDA**. Considering the results of Table 3 and 4, we found that the ***ns-3*** is overwhelmingly higher than **SNSDA** in case of the probability of happens of potential segmentation fault through function overhead.

Regarding to the function control difficulty, we introduce a call graph for one of the ns-3's function in figure 11. In fact, the *ns3::CsmaHelper::install()* function provide its information to the object and object factory classes to create a *CsmaNetDevice* and truly updated when the information goes through around 50 single callback processes. If a user makes more than 100,000 device nodes in simulator, the number of function calls can be up to 500,000 times. Since this structural environment of ***ns-3***, it is practically difficult to control a function prototypes or add a new module.

Table 4. The number of function call in **SNSDA** for example simulation in this thesis.

function	# of calls
SNSDA::LINK::default	1
SNSDA::LINK::enqueue	1
SNSDA::LINK::dequeue	1
SNSDA::LINK::setRoute	2
SNSDA::LINK::isIdle	1
SNSDA::LINK::send	2
SNSDA::LINK::receive	2
SNSDA::LINK::retQlen	1
SNSDA::LINK::connect	1
SNSDA::LINK::calThroughput	1
SNSDA::LINK::wait	1
SNSDA::NODE::default	3
SNSDA::NODE::setLocation	1
SNSDA::PACKET::default	2
SNSDA::printThroughput	2
The total number of function calls : 22	

* We include the function in the result generate process of SNSDA

Instead of the functional indicators, the interoperability of ***ns-3*** and **SNSDA** is also one of fundamental issues to users. In official, a version of ns-3 which can be built using a native Microsoft Windows compiler is provided [3]. Despite such efforts to use ns-3 on windows, only the most essential components of the simulator are included. Since the purely Unix-centric source code is still excluded, the more improvement is required. In case of the **SNSDA**, it has an opportunity to extend its module since it only uses the standard library and works on LINUX and WINDOWS as stated in Section 3.1.

4. CONCLUSION

Simulations are a common method to verify performance and reliability of newly designed networks and its applications. In this thesis, we present general-purpose discrete simulators that can deal with both deterministic and stochastic models. Furthermore the proposed simulator can reduce the complexity of useless program components (tracing subsystem, excessive attributes, etc.). In [2] and [3], we already have a brilliant discrete simulation software packages (***ns*-3**, ***OPNET***, ***NetSim***, etc.) so that freely test and verify our research materials. However, network topologies and parameters are not perfectly matched with these network simulator (and hard to get license of commercial software either), we need to design and program the network simulation software by on our own and decide whether the necessary simulation components or not.

Since we only present the methodology that implements a simple discrete-event simulator in this thesis, there are some drawbacks (time scalability and reducing compilation time) in it. a) in this thesis, we intentionally do not consider the compilation time because it is not the main concern of this thesis. However, it is mandatory that the simulator for large-scale network should consider the whole compilation time before running simulator.

Furthermore, for the future work, implementing a free-licensed network simulator with GUI interface is also one of huge challenge so that the end-user can control the simulator easily and get the graphic based report with using the Linux-based graphic library such as ***ncurses***.

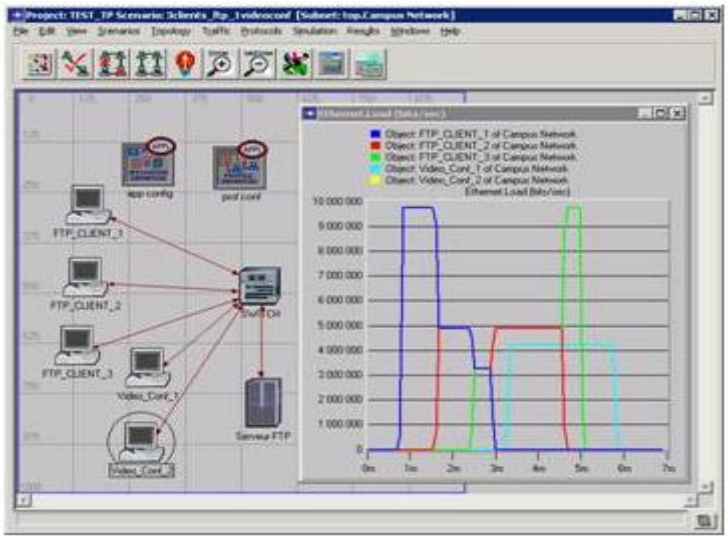

Figure 12. OPNET GUI-based network simulator [2]

REFERENCES

1. Mathieu Lacage, Tomas R Henderson, 2006, 'Yet Another Network Simulator'.

2. Inc. OPNET Technologies, 'Opnet network simulator'
http://www.opnet.com/solutions/network_rd/ modeler.html.

3. NS-3 development team, 'Ns-3 network simulator', https://www.nsnam.org/

4. NS-3 development team, 'Ns-3 network simulator', 'ns-3 tutorial',
http://www.nsnam.org/ docs/release/3.16/tutorial/ns-3-tutorial.pdf.

5. Sebastian Rampfl, Florian Wohlfart, Daniel Raumer, 2013, 'Network Simulation and its Limitations', Seminar Future Internet SS2013.

6. Schwetman, H, 1995, 'Object-oriented Simulation Modeling with C++/CSIM17', Winter Simulation Conference, p529-533.

7. Nurul I. Sarkar, Roger McHaney, 2012, 'Modeling and Simulation of IEEE 802.11 Wireless LANs: A Case Study of Network Simulator'.

8. Wikipedia, 'network simulation', https://en.wikipedia.org/wiki/Network_simulation

9. Grzegorz Chmaj, Dawid Zydek, 2011, 'Software Development Approach For Discrete Simulators'.

10. Law, Kelton, 2000, 'Simulation, Modeling & Analysis 3rd edition'

11. Randal E. Bryant, David R. O'Hallaron, 'Computer Systems: A Programmer's Perspective, 3rd edition', Carnegie Mellon University, p623-637