NS-2 simulator: Part 1
Overview and procedure to add Multi-channel support to the ns2.35 simulator

Content:

Section 1: Overview of NS2

Introducing ns2

NS2 is a discrete event network, packet level simulator. It is an open source and free simulator and the most popular among networking researchers. The network protocol stack is written in C++ and the Tool Command Line (TCL) is used by specifying scenarii and events. With this simulator, we can simulate both wired and wireless networks.

NS2 components and simulated components:

- NS simulator itself. We are working with the latest version 2.35
- NAM: the Network Animator used to:
  - visualize outputs
- Protocols: TCP, UDP, DCCP, SCTP, TFRC, Wi-Fi, etc
- Network models: nodes, links, topology, channel schedulers, buffers management, queues managements, etc.
- Traffic models
  - Applications: MPEG, FTP, CBR (Constant Bit Rate) – an application over UDP
  - Traffic traces

General steps in using NS-2 for a standard use

- Obtaining, installing and configuring NS-2: to be developped
- Writing a TCL script
- Run the simulation
- Post-process the trace generated
- Plot graphs

For a network simulation with ns2, users first write an OTCL script to describe the relevant network topology, to define traffic sources and when to start and stop transmitting packets through an event scheduler. ns-2 has an OTCL interpreter that translates the event scheduler and the network component OTCL objects and member functions into their corresponding C++

counterparts. Such a mechanism provides a linkage between the OTCL and C++ realms, making the control functions and the configurable variables interoperable. Next, ns-2's main program simulates the topology with user-specified parameters. Eventually, the simulation is finished; ns-2 generates an output trace file that contains detailed simulation data. The data can be either post-processed to create simulation graphs or input to a graphical simulation tool called Network Animator (nam) for later viewing. Figure below shows it.

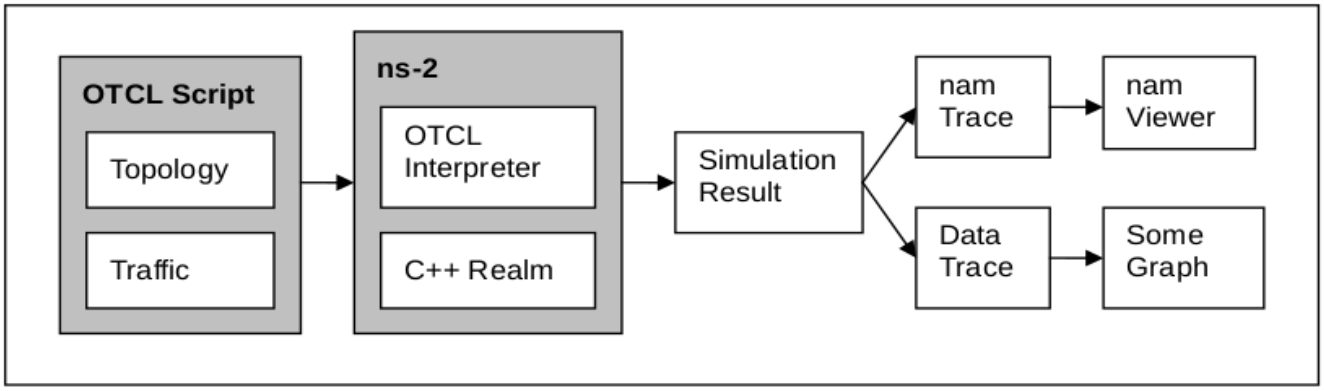
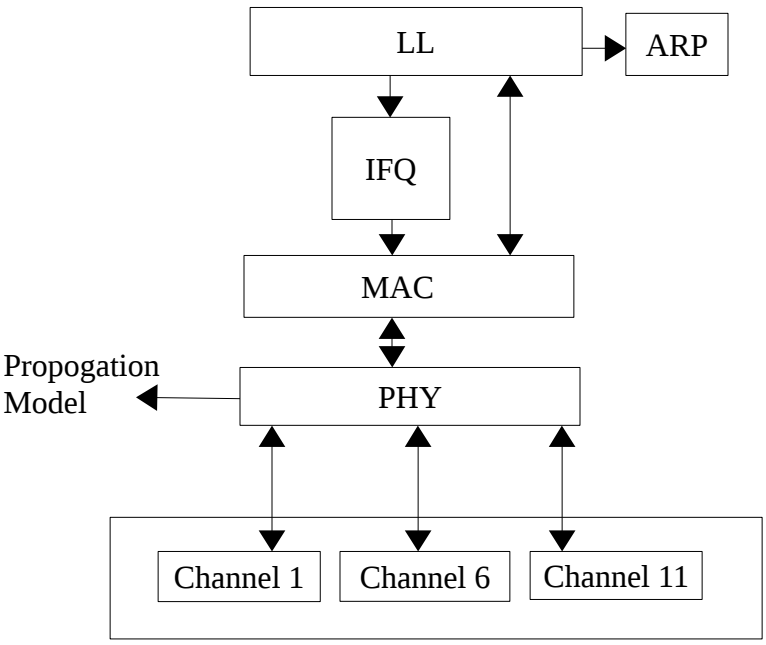


Figure 1: ns2 architecture

General steps in using NS-2 with a costomized use

For many reasons, we can want to add a new module to the existing Ns-2 simulator. For example, the standard ns2 installation does not include to use of multi-interface network interfaces. So to enable the use of it, we can proceed by 2 manners: Either by

- Modifying he existing algorithm: here, the seme header and C++ files are used (mettre le schéma des header et tout)
- Adding new files/protocols: adding new .cc and header files.



→ **LL:** Link Layer object; same as LAN but a ARP module – sends queries to ARP

→ **IFQ:** InterFace Queue – priority to the stored routing protocol packets

→ **MAC object:** Processes data packets received from or to be sent to the link layer

→ **PHY object:** serves as a hardware interface for MobileNode to access the channel

→ **Channels:** Comunication channels

Figure 2: Architecture of a wireless node in ns2 that we want to develop

Section 2: How to modify ns2 in order to enable multi-channel communications among nodes

For this case, we will non completely write a new module to support multi-channel communication, but just modify existing scripts so that the result fit to our needs. We will base our modifications in both OTCL scripts and C++ realms

A- OTCL scripts modifications

The files concerned with this modifications are ns-lib.tcl and ns-mobilenode.tcl.

➔ ns-lib.tcl located in Ns2/ns-allinone-2.35/ns-2.35/tcl/lib/ns-lib.tcl

Firstly, we had to add procedures to handle the multi-channel mechanism. We have to depict that, there is no mean to do it without the use of multiple interfaces. Below are the four procedures

```
# Procedure to change the number of interface
Simulator instproc change-numifs {newnumifs}{
      $ self  instvar numifs_
      set numifs $newnumifs
}
```

This procedure is to be called prior to creating a wireless node in the scenario script. Once called, it will affect all subsequent nodes until another invocation of the procedure is issued.

```
# Procedure to add an interface to a node
Simulator instproc add-channel {indexch ch}{
      $ self instvar chan
      set chan ( $indexch ) $ch
}
```

This procedure takes two arguments: indexch is the index of the channel to be added; ch references to the channel object previously created.

```
# Procedure to get the number of interfaces
Simulator instproc get-numifs{}{
      $ self instvar numifs_
      if [ info exists numifs_ ] {
            return $numifs_
      }
      else
      {
            return ""
      }
}
```

Adds multiple interfaces as an argument to node-config , which is an existing ns-2 command used to configure a MobileNode object, by setting a value for the local variable, numifs_.

```
#Procedure to add multiple interfaces as an argument to node-config label
```

```
Simulator instproc ifNum {val}{$ s e l f set numifs_ $val}
```

Secondly, we had to modify two existing procedures: `node-config` and the `create-wireless-node` procedure in the same ns-lib.tcl script. We need to modify many things and we can not list all these modifications here. But we will just give what we will have to modify in these files:

`node-config args {}` procedure,

- Adds the support for multiple channels.
- For backward compatibility, initializes chan as a single variable if normal operation is used, or as an array if multiple interfaces are defined.
- Adds the numifs_ variable as a new member in the argument list, args , that is passed to the procedure.

`Create-wireless-node {}` procedure:

- Takes in the number of interfaces specified in numifs_ and iteratively calls `add-interface` as many times as the number of interfaces that the node has.
- `add-interface` is an existing ns-2 procedure that adds an interface to a previously created MobileNode object.

➔ ns-mobilenode.tcl is located in the same directory (Ns2/ns-allinone-2.35/ns-2.35/tcl/lib/)

Here also, we have many changes to make in many procedures in this files. So we are just going to list things we will modify to add our adjustements.

`add-target {agent port}`procedure

- This procedure attaches a routing agent to a `MobileNode` object, picks a port and binds the agent to the port number.
- Gets the number of interfaces via calling `get-numifs` ; doing so allows the procedure to determine whether multiple interfaces are present. If the number of interfaces is non-zero (implying that the multiple interface extension is used), the procedure iteratively calls the `if-queue` command as many times as the value returned by `get-numifs`.

`add-target-rtagent {agent port}`procedure

- This procedure, called by `add-target` , adds a target routing agent.
- It first gets the number of interfaces that a node has from `get-numifs` . If the number of interfaces is non-zero, the procedure associates the routing agent with the corresponding link layer target entity as many times as the number of interfaces for both of the sending and receiving targets.

`Add-interface {}` procedure

- This procedure adds an interface to a MobileNode object.
- Originally, it creates one ARP table (for address resolution) per node. We modify it to be one ARP table per interface although having one ARP table per node resembles the real-life case. The reason for such a walk-around is that, if a node is using one interface to communicate with another one, the current design of MobileNode in ns-2 will not allow the node to use another interface since the request to the ARP entity will still be serving the previous interface.

`init args{} and reset{}` procedure

- Due to the above change of assigning the ARP table, these two procedures are modified to initialize and reset the ARP table of a MobileNode object per the number of interfaces defined.

B- C++ codes modifications: includes .cc and .h files

After adding the multiple interface support in the OTCL realm, relevant changes have to be made in the C++ code for the MobileNode object, the channel entity, and the MAC layer model.

➔ The `mobilenode.h` file located in `Ns2/ns-allinone-2.35/ns-2.35/common/`

- Here, it must be a definition of a new declaration of the MobileNode lists to replace the existing ones.
  - ns-2 controls each instance of the MobileNode objects which are associated with a channel by means of a linked-list. Two lists are managed; one references the previous node, `prevX_` , while the other references the next node, `nextX_` .
  - The original format of the list is simply a pointer to a node. In order to support multiple channels, the list is modified to be an array of pointers with the size of the array being the maximum of number of channels:
    - `nextX_[MAX_CHANNELS]`
    - `prevX_[MAX_CHANNELS]`
  - The index for referring to a node is the channel number.
- We remove the inline declaration of the `getLoc()` function. Due to the above changes on the MobileNode lists, the original declaration has been found to always return a zero distance, which leads to wrong packet receptions.

➔ The `mobilenode.cc` file located in `Ns2/ns-allinone-2.35/ns-2.35/common/`
We add the `getLoc()` method definition which retrieves the location of a node.

➔ The `channel.cc` file located in `Ns2/ns-allinone-2.35/ns-2.35/mac/`

- Due to the changes on the MobileNode lists, we modify accessing each node entry when attaching, removing, and updating a new node to a channel to refer to the corresponding channel number. This number can be accessed by `this->index()` , where this is the current instance of the channel class object. In other words, whenever `nextX_` and `prevX_` appear in `channel.cc`, they need to be replaced by:
  - `nextX_[this->index()]`
  - `prevX_[this->index()]`
- In the `affectedNodes()` function, we add a new condition to check which of the interfaces of the destination node is connected to the same channel before transmitting the packet to another interface. The original design of ns-2 does not consider the case with multiple channels; it simply sends the packet to all of the destination interfaces. Accessing the channel of the destination interface is carried out by:
  - `rifp->channel()`, where rifp is a pointer to the receiving interface. The `channel()` member returns a channel object that has the same type as this

➔ The `mac-802_11.cc` file located in `Ns2/ns-allinone-2.35/ns-2.35/mac/`

- For correct handling of multiple interfaces by the routing agent, we modify the `recv()` method in the MAC 802.11 class to register the correct MAC receiving interface in the MAC header. The hardware address of the interface can be access by:
  - `hdr->iface() = addr()`

As mentionned earlier, in this case, we just modify files in the existing ns2.35 package. For this purpose, after modifying files as shown above, we have to “update” the packege in order to enable the use of added functionnalities. For that, we have to follow the 3 steps below:

Step 1: Go to the the main folder of ns2.35

Step2: Execute the following commands:

```
#make clean
#make depend
#make
```

Step 3: For better result, reboot your computer and test the new simulator.

Conclusion:

All these modifications has been done, but we are getting many errors.

Some captures of the simulated netwoks for the before modifying the package

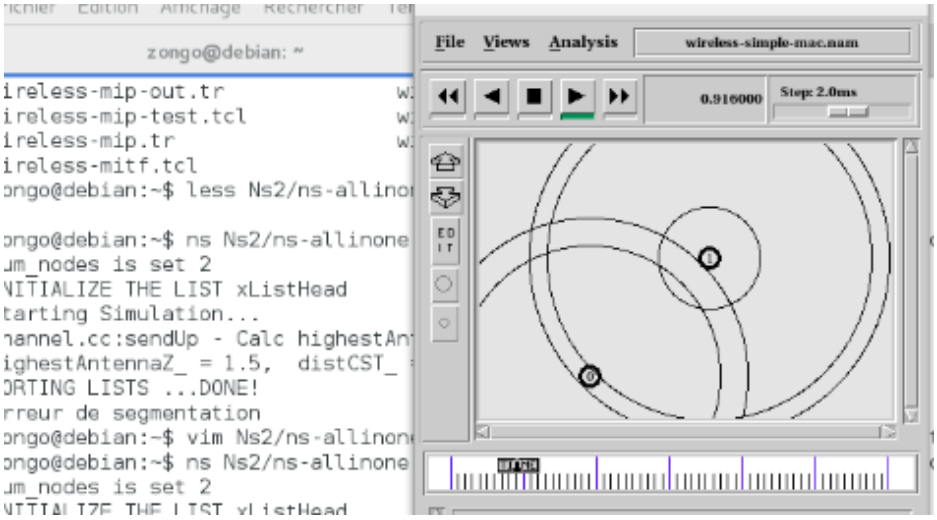


Figure 3: Simulation of 2 wireless nodes sensing their range at 10ms

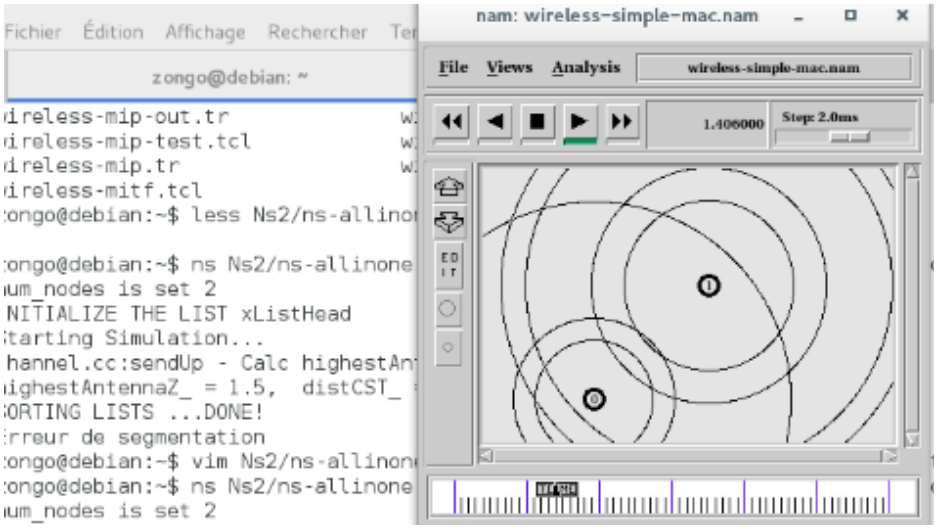


Figure 4: Simulation of 2 wireless nodes sensing their range at 15ms

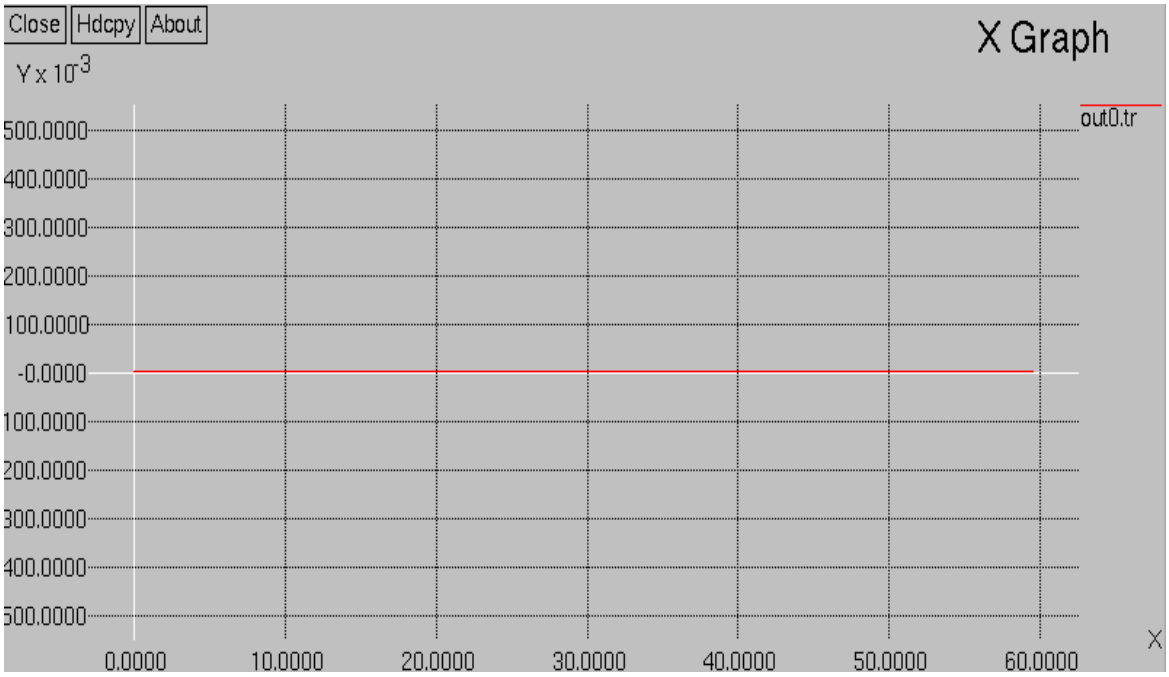


Figure 5: The output of the Trace file with xgraph for a single wireless node

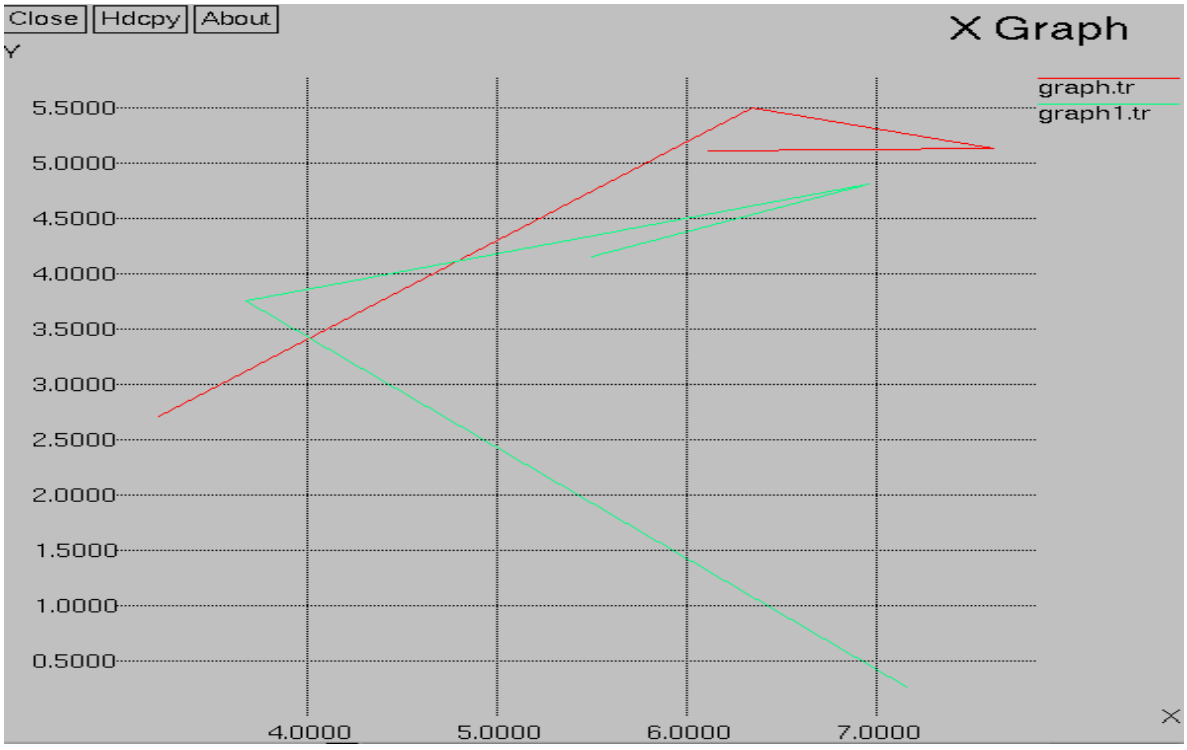


Figure 6: The output of the Trace file with xgraph for two wireless nodes

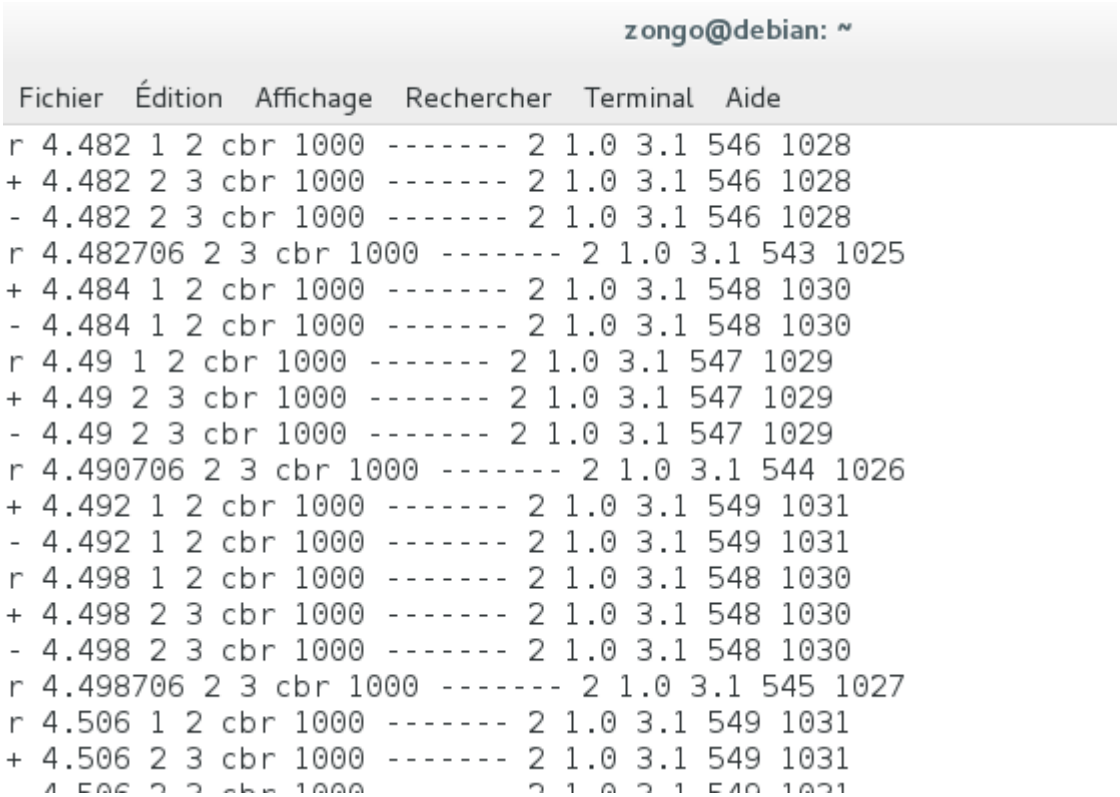


Figure 7: The output of the Trace file within the command line