Practical – 1

Aim: Understanding the Sensor Node Hardware. (For Eg. Sensors, Nodes(Sensor mote), Base Station, Graphical User Interface.)

1. Components

A wireless sensor network (WSN) is a hardware and software package that typically consists of four parts (see Figure 1):

a) 'Sensors' connected to each node by a wired connection. In our case, we use sensors that can measure soil moisture, electrical conductivity, soil temperature, water pressure, flow rate, or a range of weather variables (light, air temperature, wind, humidity, etc.).

Figure 2. One of many sensors that can be connected to a node, this EC---5 sensor (Decagon Devices, Inc. Pullman, WA) measures volumetric water content (soil moisture).

b) 'Nodes' collect the data from sensors and transmit that to a 'base station'

computer using a one way (in the case of monitoring) or two---way (in the case of monitoring and control) radio. Nodes can simply monitor environmental and soil conditions or can be used to make control decisions. For example, some nodes have the capability rdo an electric valve, such as an

Relay switch, used to irrigation valve(s), located here.

Figure 3. This nR5 (Decagon Devices, Inc. Pullman, WA) node is powered off of 5---AA batteries and is connected to 5 soil moisture sensors via stereo ports. The nR5 node is also capable of controlling irrigation valve(s), based on user---defined settings.

c) *'Base Station' computer* connects the system to the internet, so that data collected by the nodes, then transmitted to the base station computer, can be viewed anywhere an internet connection is available.

d) *'Graphical User Interface'* is the web---based software package, that allows the data collected by sensors to be viewed. The software is also used to set irrigation parameters.

Figure 4. The graphical user interface above depicts the volumetric water content (soil moisture) as horizontal lines and irrigation events and amounts as bars. Notice the increase in soil moisture after each irrigation event.

<u>Not every WSN will have all four components, but to get optimal functionality the systems developed as part of this project do.</u>

A very simple WSN example that many can relate to is that of the wireless environmental monitoring system used by the National Weather Service (NWS). You have probably seen these at a local airport or school. In this case, sensors measure environmental conditions and send this data to a node that wirelessly transmits the data using a cell signal or wireless signal to a base---station computer where NWS employees (and you) can view the current temperature (or rainfall/dew point, wind, etc.) via a website or application ('app').

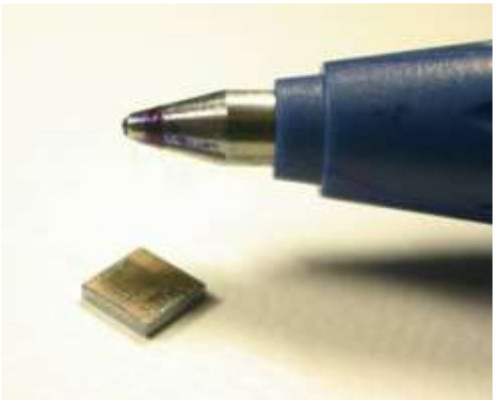
Figure 5. Typical environmental monitoring sensors that you would see at a National Weather Service (NWS) monitoring station. These same components can be used in a wireless sensor network by a specialty crop producer.

Practical-2

Aim: Exploring and understanding TinyOS computational concepts:- Events, Commands and Task.

- **nesC model**
- **nesC Components**

TinyOS and nesC



Outline

- Wireless sensor networks and TinyOS
- Networked embedded system C (nesC)
  - Components
  - Interfaces
  - Concurrency model
  - Tool chain
- Issues / conclusion

Wireless Sensor Networks

- Vision: ubiquitous computing
- Extreme dynamics
- Interact with environment using sensors and radio
- Immense scale
- Limited access
- Small, cheap, low-power systems

Concepts in Sensor Networks

- In-network processing and data aggregation
  - Radio activity 1000 times as expensive as processing
- Duty-cycling: different modes of operation
  - Power down unused hardware
- Systems run a single application
- Applications are deeply tied to hardware

- Require customized and optimized OS

Challenges

- Limited resources: energy consumption dominates
- Concurrency: driven by interaction with environment
- Soft real-time requirements
- Reliability: reduce run-time errors, e.g. races
- High diversity of platforms
- No well-defined software/hardware boundary

TinyOS

- Component-based architecture
  - Reusable system components: ADC, Timer, Radio
- Tasks and event-based concurrency
  - No user-space or context switching supported by hardware
  - Tasks run to completion only preempted by interrupts
- All long-latency operations are *split-phase*
  - Operation request and completion are separate functions

Introducing nesC

- A “holistic” approach to networked embedded systems
- Supports and reflects TinyOS's design
- Extends a subset of C
- A static language
  - All resources known at compile-time
  - Call-graph fully known at compile-time

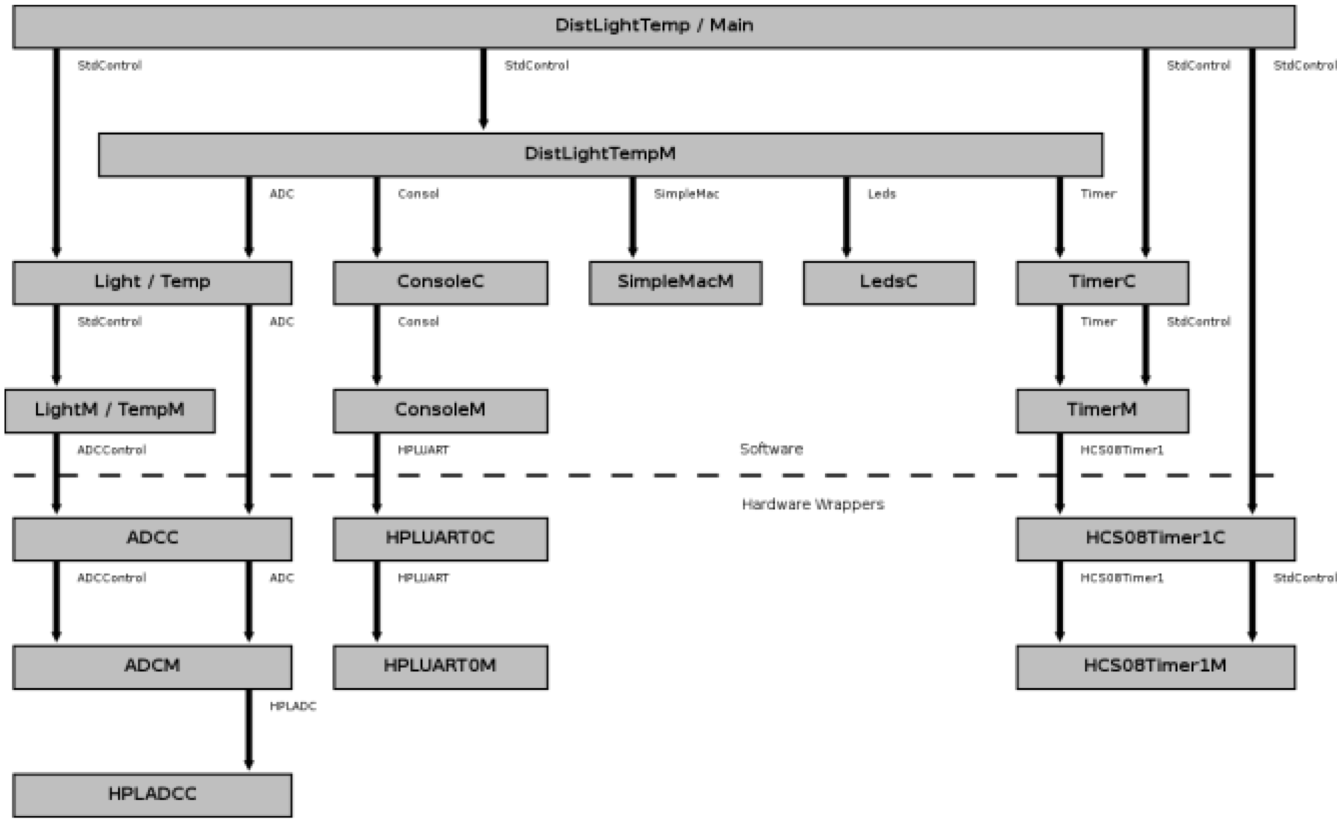
Design Decisions for nesC

- Components
- Bidirectional interfaces
- Simple expressive concurrency model
- Whole-program analysis

Components

- Challenge: platform diversity, flexible SW/HW boundary, applications deeply tied to hardware
- Encourages modular design
- Restrict access to private data
- Allow underlying implementations to be replaced easily
- Can abstract HW using thin wrappers
- Allow specialization of applications to hardware

Example Component Graph



Module Components

- Modules implement application code
- Modules have private state
  - Sharing of data among components is discouraged
- Convention:
  - Module names end with 'M', e.g. BlinkM
  - Module variables start with 'm_', e.g. m_timers

Configuration Components

- Configurations wire other components together
- All applications have a top-level configuration
- A component interface may be wired zero or more times
  - Used for StdControl to implement power management
- Convention:
  - Configuration names end with 'C', e.g. TimerC
    (unless it is the top-level configuration ;-)

Modules and Configurations

```
/* BlinkM.nc */
module BlinkM {
provides interface StdControl as Control;
uses interface Timer;
uses interface Leds;
} implementation {
command result_t Control.init() {
call Leds.init();
return SUCCESS;
}
command result_t Control.start() { /* ... */ }
command result_t Control.stop() { /* ... */ }
event result_t Timer.fired() {
call Leds.redToggle();
return SUCCESS;
}
}
/* Blink.nc */
configuration Blink {
} implementation {
/* Declare used components. */
components Main, BlinkM, SingleTimer, LedsC;
/* Wire components together. */
Main.StdControl -> SingleTimer.StdControl;
Main.StdControl -> BlinkM.StdControl;
BlinkM.Timer -> SingleTimer.Timer;
BlinkM.Leds -> LedsC;
}
```

Bidirectional Interfaces

- Challenge: flexible SW/HW boundary and concurrency
- Support split-phase operations
- Commands: call *down* the component graph
  - Implemented by provider
- Events: call *up* the component graph
  - Implemented by user

Interfaces

```
/* Timer.nc */
includes Timer; /* Include C types from Timer.h */
interface Timer {
	command result_t start(char type, uint32_t interval);
	command result_t stop();
	event result_t fired();
}

/* SyncAlarm.nc */
interface SyncAlarm<Precision_t> {
	command result_t armCountdown(Precision_t timeout);
	command result_t armAlarmClock(Precision_t time);
	command result_t stop();
	event result_t alarm();
}
```

Parameterized Interfaces

```
module TimerM {
	provides interface Timer[uint8_t id];
} implementation {
	/* ... */
	Timer_t m_timers[NUM_TIMERS];
	command result_t Timer.isSet[uint8_t timer]() {
		return m_timers[timer].isset;
	}
	task void timerCheck() {
		uint8_t timer;
		for (timer = 0; timer < NUM_TIMERS; timer++)
			if (m_timers[timer].fired)
				signal Timer.fired[timer]();
	}
	/* ... */
}

configuration MyApp { /* ... */ }
implementation {
	components MyAppM, TimerC, /* ... */;
	MyAppM.SampleTimer -> TimerC.Timer[unique("Timer")];
}
```

Concurrency Model

- Challenge: extreme dynamics and soft real-time requirements
- Cooperative scheduling
- Light-weight tasks
- Split-phase operations: non-blocking requests
- Built-in atomic sections
  - Limited crossing of module boundaries

Sources of Concurrency

- Tasks
  - Deferred computation
  - Run *sequential* and *to completion*
  - Do not preempt
- Events
  - Run to completion, and may preempt tasks and events
  - Origin: hardware interrupts or split-phase completion

Tasks and Events

```
module LightM {
        /* ... */
} implementation {
        uint16_t light_data
        task void processLightdata() {
                uint16_t local_light_data;
                atomic local_light_data = light_data;
                /* Process light data. */
                if (!done)
                        post anotherTask()
        }
        async event result_t Light.dataReady(uint16_t data) {
                atomic lightData = data;
                post processLightData();
                return SUCCESS;
        }
        event result_t SensorTimer.fired() {
                return call Light.getData();
        }
}
```

Whole-Program Analysis

- Compilation can examine complete call-graph
  - Remove dead-code
  - Eliminate costly module boundary crossings
  - Inline small functions
- Back-end C compiler can optimize whole program
  - Perform cross component optimizations
  - Constant propagation, common sub-expression elimination
- Allows detection of race conditions

Synchronous and Asynchronous

- **Asynchronous code (AC)**:
  - Code reachable from at least one interrupt handler
  - Events signaled directly or indirectly by hardware interrupts
- **Synchronous code (SC):**
  - "Everything else ..."
  - Primarily tasks

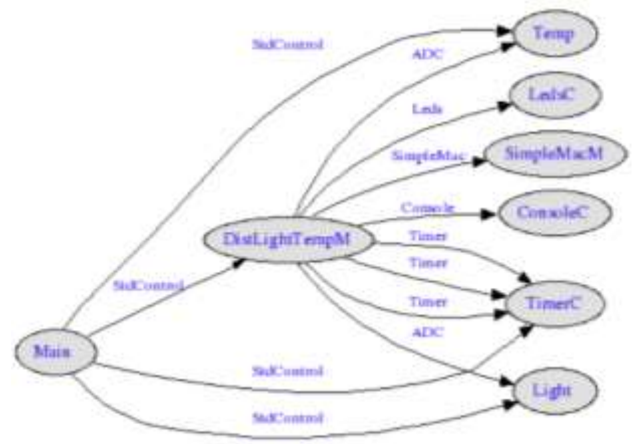
Detecting Race Conditions

- Invariant: *SC is atomic with respect to other SC*
- Two claims about updates for AC/AC and SC/AC:
  - Any update to shared state from AC is a potential race condition
  - Any update to shared state from SC that is also updated from AC is a potential race condition
- Race-free invariant enforced at compile time:
  - Updates to shared state is either SC only or in atomic section

Dealing with Race Conditions

- Use atomic sections to update shared state
  - **atomic** { shared_var = 1; }
- Convert code with updates to shared state to tasks
- Mark false positive with norace qualifier
  - **norace** uint8_t variable;

The nesC Toolchain: nesdoc



- Generate code documentation using simple tags
- Same concept as javadoc
- Can generate a component graph using dot

The nesC Toolchain: nescc

- The nesC compiler for TinyOS
- Implemented as an extension to GCC
- Called via TinyOS wrapper ncc
- Input: path to TinyOS code + nesC files
  - Platform code implements API of macros and functions in C
- Output: C code or object code (if supported by GCC)

The nesC Toolchain: ncg and mig

- Allows integration with Java code
- Typical use: interact with network through base station
- ncg - extract constants from nesC files
  - Generates class that contains constants
- mig - message interface generator for nesC
  - Generates class that encodes and decodes messages

Issues for nesC

- Problem for data shared across components
  - False positives for buffer swapping
- Problem for data shared between split-phase operations
  - Event can potentially fire if other components access HW
- Some TinyOS idioms are not well expressed
  - Parameterized interfaces each with private state

Issues for Applications

- Focus early on modeling it as a state-machine
- Design duty-cycling from the start
  - Affect the state-machine so hard to add later
- Abstracting functionality into components
  - Makes it harder to access shared state:
    encapsulate shared state in a separate module
- Configuring TinyOS for the application needs

– By default there can only be 8 posted tasks

Conclusions for nesC

- Bidirectional interfaces fit the TinyOS model
- Components are a good abstraction
- Concurrency model meets requirements in applications
- The restrictions in nesC introduce practical problems
- Not limited to the domain of embedded systems

Practical – 3

Aim: Understanding TOSSIM for
- Mote-mote radio communication
- Mote-PC serial communication

Steps:

a) TOSSIM for Mote-mote radio communication

Introduction:
TOSSIM is a discrete event simulator for TinyOS sensor networks. Instead of compiling a TinyOS application for a mote, users can compile it into the TOSSIM framework, which runs on a PC. This allows users to debug, test, and analyze algorithms in a controlled and repeatable environment. As TOSSIM runs on a PC, users can examine their TinyOS code using debuggers and other development tools. This document briefly describes the design philosophy of TOSSIM, its capabilities, its structure. It also provides a brief tutorial on how to use TOSSIM for testing or analysis.

TinyOS provides a number of *interfaces* to abstract the underlying communications services and a number of *components*that *provide* (implement) these interfaces. All of these interfaces and components use a common message buffer abstraction, called message_t, which is implemented as a nesC struct (similar to a C struct). The message_t abstraction replaces the TinyOS 1.x TOS_Msg abstraction. Unlike TinyOS 1.x, the members of message_t are opaque, and therefore not accessed directly.

Basic Communications Interfaces

There are a number of interfaces and components that use message_t as the underlying data structure. Let's take a look at some of the interfaces that are in the tos/interfaces directory to familiarize ourselves with the general functionality of the communications system:

- Packet - Provides the basic accessors for the message_t abstract data type. This interface provides commands for clearing a message's contents, getting its payload length, and getting a pointer to its payload area.
- Send - Provides the basic *address-free* message sending interface. This interface provides commands for sending a message and canceling a pending message send. The interface provides an event to indicate whether a message was sent successfully or not. It also provides convenience functions for getting the message's maximum payload as well as a pointer to a message's payload area.
- Receive - Provides the basic message reception interface. This interface provides an event for receiving messages. It also provides, for convenience, commands for getting a message's payload length and getting a pointer to a message's payload area.

- PacketAcknowledgements - Provides a mechanism for requesting acknowledgements on a per-packet basis.
- RadioTimeStamping - Provides time stamping information for radio transmission and reception.

Components

A number of components implement the basic communications and active message interfaces. Let's take a look at some of the components in the /tos/system directory. You should be familiar with these components because your code needs to specify both the *interfaces* your application *uses* as well as the *components* which *provide* (implement) those interfaces:

- AMReceiverC - Provides the following interfaces: Receive, Packet, and AMPacket.
- AMSenderC - Provides AMSend, Packet, AMPacket, and PacketAcknowledgements as Acks.
- AMSnooperC - Provides Receive, Packet, and AMPacket.
- AMSnoopingReceiverC - Provides Receive, Packet, and AMPacket.
- ActiveMessageAddressC - Provides commands to get and set the node's active message address. This interface is not for general use and changing a node's active message address can break the network stack, so avoid using it unless you know what you are doing.

Compiling TOSSIM

TOSSIM is a TinyOS library. Its core code lives in tos/lib/tossim. Every TinyOS source directory has an optional `sim` subdirectory, which may contain simulation implementations of that package. For example, tos/chips/atm128/timer/sim contains TOSSIM implementations of some of the Atmega128 timer abstractions.

To compile TOSSIM, you pass the `sim` option to make:

```
$ cd apps/Blink
$ make micaz sim
```

Running TOSSIM with Python

Go into the `RadioCountToLeds` application and build TOSSIM:

```
$ cd tinyos-2.x/apps/RadioCountToLeds
```

```
$ make micaz sim
```

We'll start with Python in interactive mode. To start the Python interpreter, type:

```
$ python
```

b) TOSSIM for Mote-PC radio communication

Packet sources and TestSerial

The first step is to check that you are able to get your PC to communicate with a mote. Most motes have a serial port or similar interface. For example, the mica family can directly control a serial port: programming boards basically connect the mote's serial port pins to the actual serial port on the board. Telos motes also have a serial interface, but it talks to their USB hardware, which is similar in functionality but very different in terms of cables and connectors.

The basic abstraction for mote-PC communication is a **packet source**. A packet source is exactly that: a communication medium over which an application can receive packets from and send packets to a mote. Examples of packet sources include serial ports, TCP sockets, and the SerialForwarder tool. Most TinyOS communication tools take an optional `-comm` parameter, which allows you to specify the packet source as a string. For example:

```
$ java net.tinyos.tools.Listen -comm serial@COM1:telos
```

tells the Listen tool to use the COM1 serial port (on a Windows machine) at the correct speed for a telos mote, while

```
$ java net.tinyos.tools.Listen -comm serial@/dev/ttyS0:micaz
```

tells Listen to use the serial port `/dev/ttyS0` (on a UNIX machine) at the correct speed for a micaz mote.

The first step to testing your serial port is to install the `apps/tests/TestSerial` application on a mote. This application sends a packet to the serial port every second, and when it receives a packet over the serial port it displays the packet's sequence number on the LEDs.

Once you have installed `TestSerial`, you need to run the corresponding Java application that communicates with it over the serial port. This is built when you build the TinyOS application. From in the application directory, type:

```
$ java TestSerial
```

If you get a message like

```
The java class is not found:  TestSerial
```

it means that you either haven't compiled the Java code (try running `make platform` again) or you don't have `.` (the current directory) in your Java `CLASSPATH`.

Because you haven't specified a packet source, TestSerial will fall back to a default, which is a SerialForwarder. Since you don't have a SerialForwarder running, TestSerial will exit, complaining that it can't connect to one. So let's specify the serial port as the source, using the `-comm` parameter as described above. The syntax for a serial port source is as follows:

```
  serial@<PORT>:<SPEED>
```

PORT depends on your platform and where you have plugged the mote in. For Windows/Cygwin platforms, it is COM*N*, where *N* is the port number. For Linux/UNIX machines, it is `/dev/ttyS`*N* for a built-in serial port, or one of `/dev/ttyUSB`*N* or `/dev/usb/tts/`*N* for a serial-over-USB port. Additionally as we saw in lesson 1, on Linux you will typically need to make this serial port world writeable. As superuser, execute the following command:

```
chmod 666 serialport
```

The SPEED can either be a numeric value, or the name of a platform. Specifying a platform name tells the serial packet source to use the default speed for the platform. Valid platforms are:

Platform	Speed (baud)
telos	115200
telosb	115200

tmote	115200
micaz	57600
mica2	57600
iris	57600
mica2dot	19200
eyes	115200
intelmote2	115200

The Java file `support/sdk/java/net/tinyos/packet/BaudRate.java` determines these mappings. Unlike in TinyOS 1.x, all platforms have a common serial packet format. Following the table, these two serial specfications are identical:

```
serial@COM1:micaz
serial@COM1:57600
```

If you run `TestSerial` with the proper PORT and SPEED settings, you should see output like this:

```
Sending packet 1
Received packet sequence number 4
Sending packet 2
Received packet sequence number 5
Sending packet 3
Received packet sequence number 6
Sending packet 4
Received packet sequence number 7
Received packet sequence number 8
Sending packet 5
Received packet sequence number 9
Sending packet 6
```

and the mote LEDs will blink.

MOTECOM

If you do not pass a `-comm` parameter, then tools will check the `MOTECOM` environment variable for a packet source, and if there is no `MOTECOM`, they default to a SerialForwarder. This means that if you're always communicating with a mote over your serial port, you can just set `MOTECOM` and no longer have to specify the `-comm` parameter. For example:

```
export MOTECOM=serial@COM1:19200 # mica baud rate
export MOTECOM=serial@COM1:mica  # mica baud rate, again
export MOTECOM=serial@COM2:mica2 # the mica2 baud rate, on a different serial port
export MOTECOM=serial@COM3:57600 # explicit mica2 baud rate
```

Try setting your MOTECOM variable and running TestSerial without a `-comm` parameter.

BaseStation and net.tinyos.tools.Listen

`BaseStation` is a basic TinyOS utility application. It acts as a bridge between the serial port and radio network. When it receives a packet from the serial port, it transmits it on the radio; when it receives a packets over the radio, it transmits it to the serial port. Because TinyOS has a toolchain for generating and sending packets to a mote over a serial port, using a BaseStation allows PC tools to communicate directly with mote networks.

Take one of the two nodes that had BlinkToRadio (from lesson 3) installed and install BaseStation on it. If you turn on the node that still has BlinkToRadio installed, you should see LED 1 on the BaseStation blinking. BaseStation toggles LED 0 whenever it sends a packet to the radio and LED 1 whenever it sends a packet to the serial port. It toggles LED 2 whenever it has to drop a packet: this can happen when one of the two receives packets faster than the other can send them (e.g., receiving micaZ radio packets at 256kbps but sending serial packets at 57.6kbps).

BaseStation is receiving your BlinkToRadio packets and sending them to the serial port, so if it is plugged into a PC we can view these packets. The Java tool Listen is a basic packet sniffer: it prints out the binary contents of any packet it hears. Run Listen, using either MOTECOM or a -comm parameter:

```
$ java net.tinyos.tools.Listen
```

Listen creates a packet source and just prints out every packet it sees. Your output should look something like this:

```
00 FF FF 00 00 04 22 06 00 02 00 01
00 FF FF 00 00 04 22 06 00 02 00 02
00 FF FF 00 00 04 22 06 00 02 00 03
```

```
00 FF FF 00 00 04 22 06 00 02 00 04
00 FF FF 00 00 04 22 06 00 02 00 05
00 FF FF 00 00 04 22 06 00 02 00 06
00 FF FF 00 00 04 22 06 00 02 00 07
00 FF FF 00 00 04 22 06 00 02 00 08
00 FF FF 00 00 04 22 06 00 02 00 09
00 FF FF 00 00 04 22 06 00 02 00 0A
00 FF FF 00 00 04 22 06 00 02 00 0B
```

Listen is simply printing out the packets that are coming from the mote. Each data packet that comes out of the mote contains several fields of data. The first byte (00) indicates that this is packet is an AM packet. The next fields are the generic Active Message fields, defined in `tinyos-2.x/tos/lib/serial/Serial.h`. Finally, the remaining fields are the data payload of the message, which was defined in BlinkToRadio.h as:

```
typedef nx_struct BlinkToRadioMsg {
  nx_uint16_t nodeid;
  nx_uint16_t counter;
} BlinkToRadioMsg;
```

The overall message format for the BlinkToRadioC application is therefore (ignoring the first 00 byte):

- **Destination address** (2 bytes)
- **Link source address** (2 bytes)
- **Message length** (1 byte)
- **Group ID** (1 byte)
- **Active Message handler type** (1 byte)
- **Payload** (up to 28 bytes):
  - **source mote ID** (2 bytes)
  - **sample counter** (2 bytes)

So we can interpret the data packet as follows:

dest addr	link source addr	msg len	groupID	handlerID	source addr	counter

02

ff ff	00 00	04	22	06	00 02	00 0B

The link source address and source address field differ in who sets them. The serial stack does not set the link source address; for Blink, it should always be **00 00**. Blink sets the source address to be the node's ID, which depends on what mote ID you installed your BlinkToRadio application with. The default (if you do not specify and ID) is `00 01`. Note that the data is sent by the mote in *big-endian* format; for example, `01 02` means 258 (256*1 + 2). This format is independent of the endian-ness of the processor, because the packet format is an `nx_struct`, which is a network format, that is, big-endian and byte-aligned. Using `nx_struct` (rather than a standard C `struct`) for a message payload ensures that it will work across platforms.

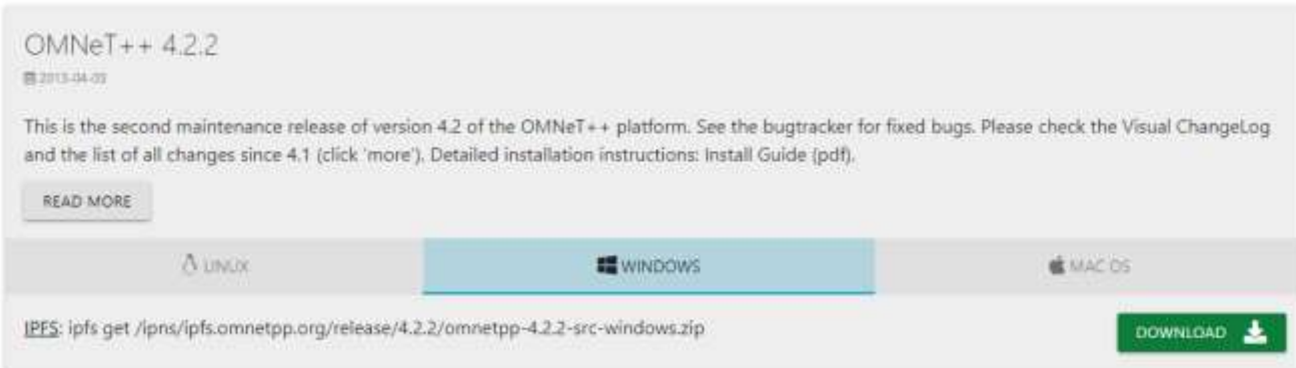
As you watch the packets scroll by, you should see the counter field increase as the BlinkToRadio app increments its counter.

Practical-4

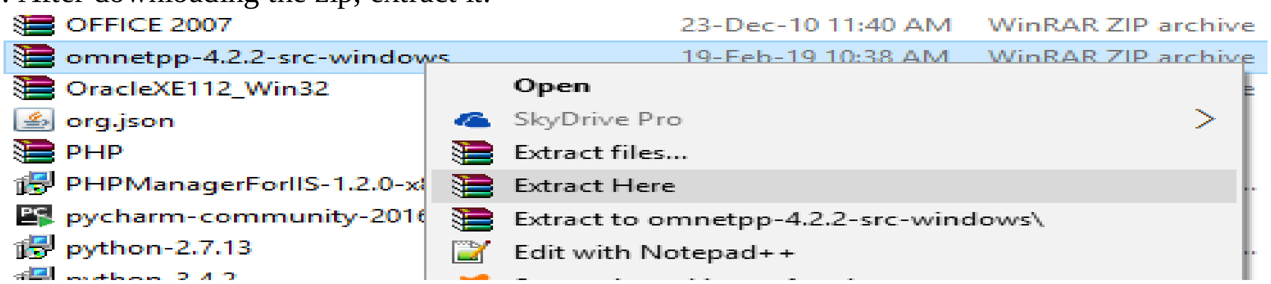
Aim: Create and simulate a simple adhoc network

Steps:

1. Download Omnet++ version 4.2.2 from https://omnetpp.org/omnetpp.



2. After downloading the zip, extract it.



3. Now Omnetpp-4.2.2 folder has been created.

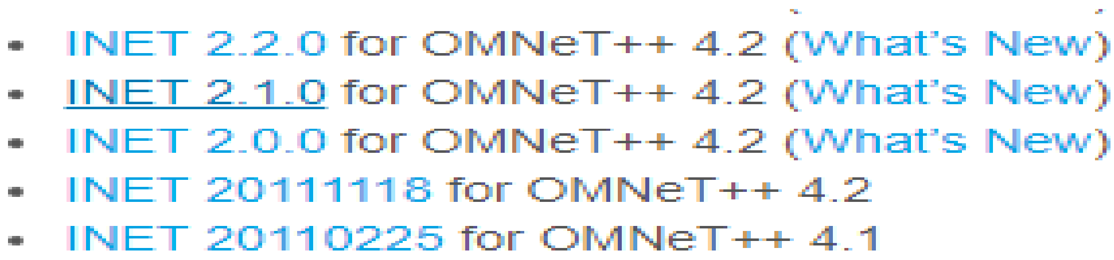
4. Open the folder and Double click on mingwenv.

5. A command prompt will open. Type the command
   I. ./configure
   II. make

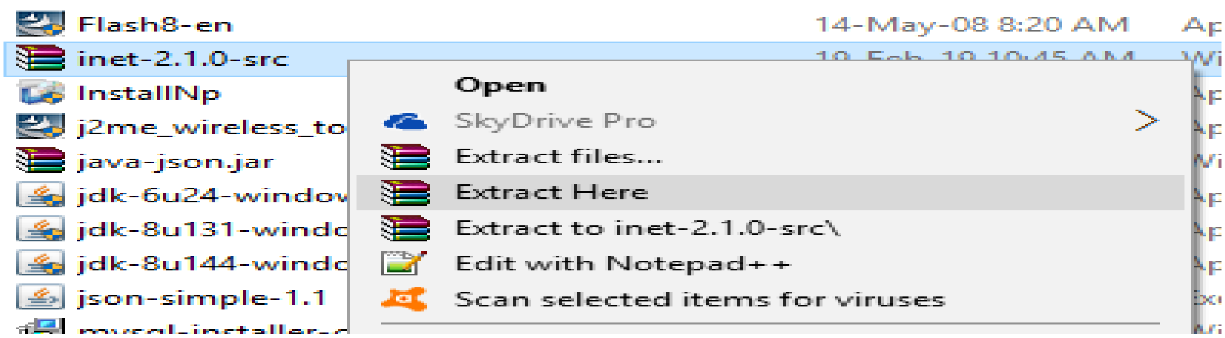
6. When both commands get executed then omnet++ is installed in your system.

7. To check whether it is installed or not type the command omnetpp in the command prompt and omnetpp will get started. If it does not start then try to reinstall.

8. After installing Omnet++, we need to install inet framework version 2.1.0 which is specially designed for wireless simulation. You can download inet framework from below link. https://inet.omnetpp.org/Download.html



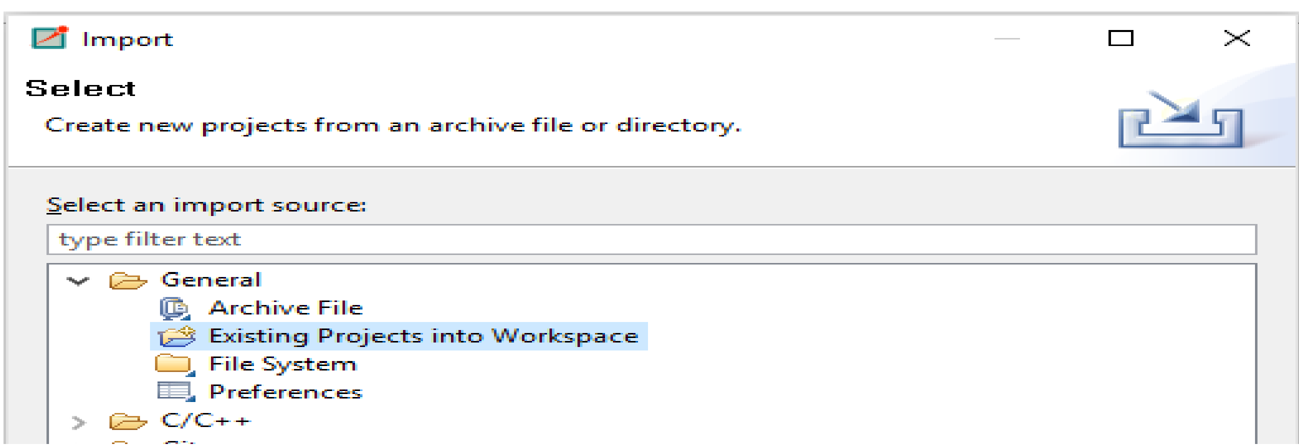
9. After downloading the Inet 2.2.0 unzip the files.



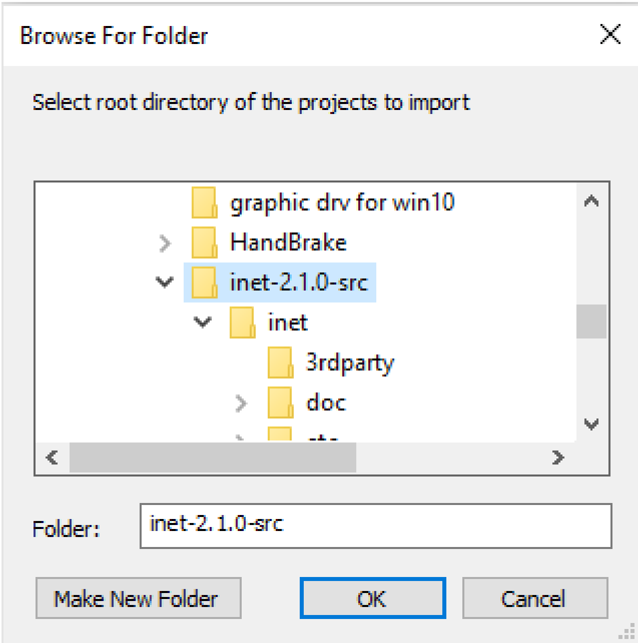
10. After extracting inet-2.1.0-src folder will be created.

11. Open the Omnet++ idle.

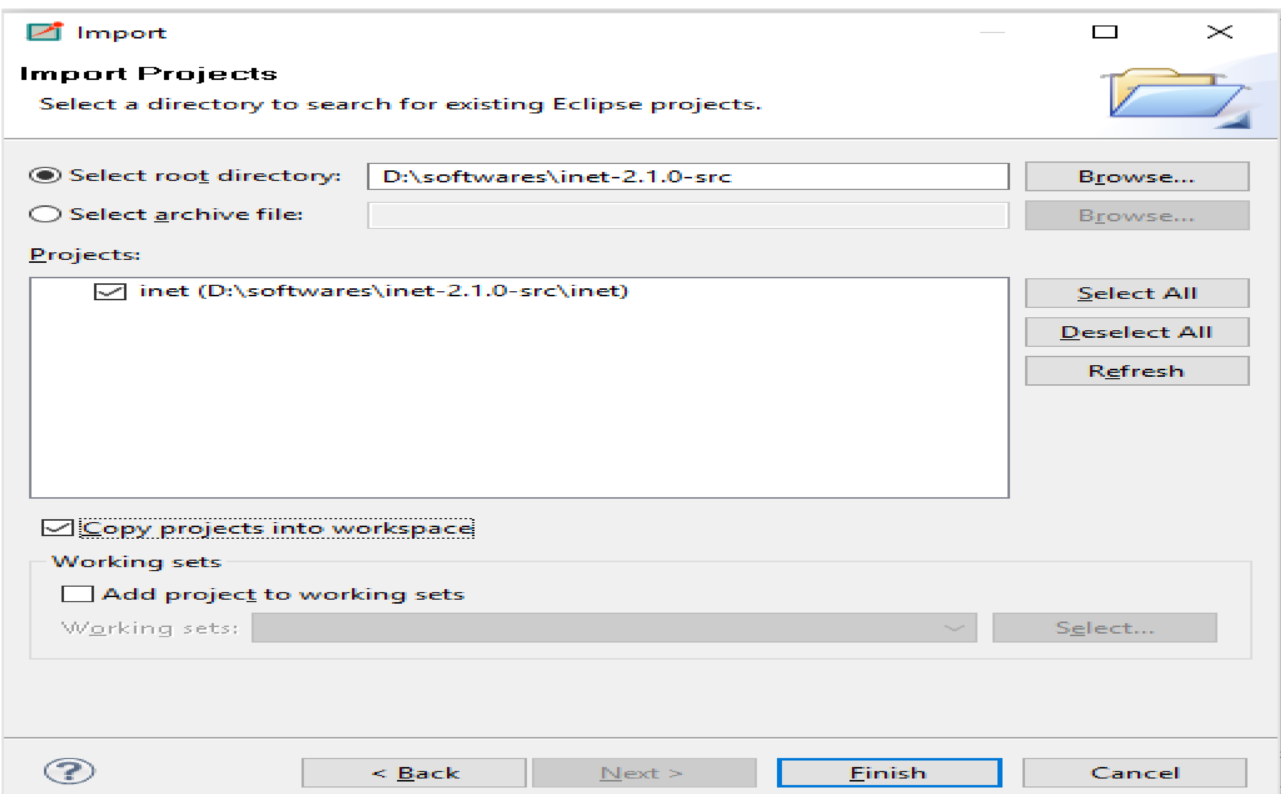
12. Click on File > Import. A window will appear in that click on General > Existing Projects in Workspace.



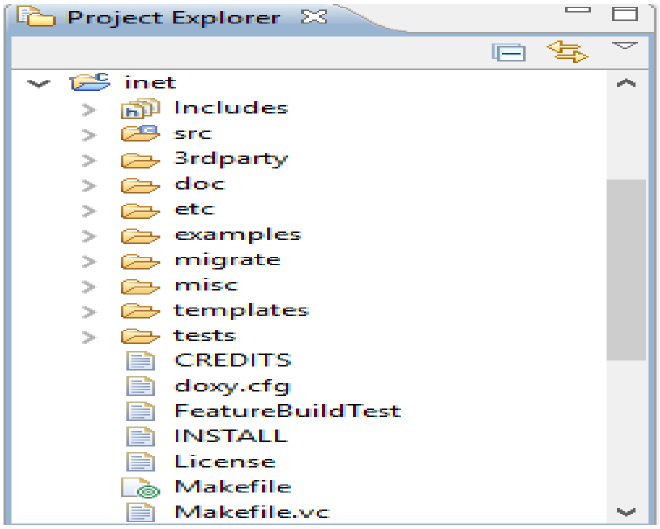
13. Click next and in the root directory browse for the inet-2.1.0-src folder which was created.



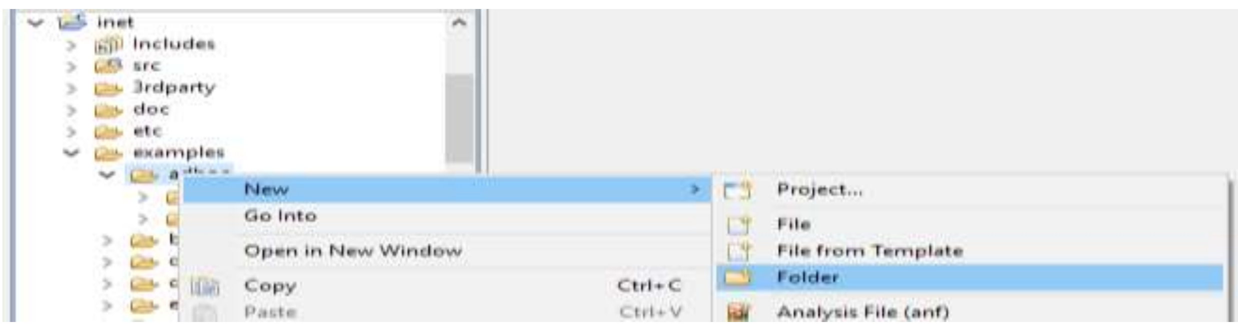
14. Click the checkbox to copy the projects into the workspace and then Finish.

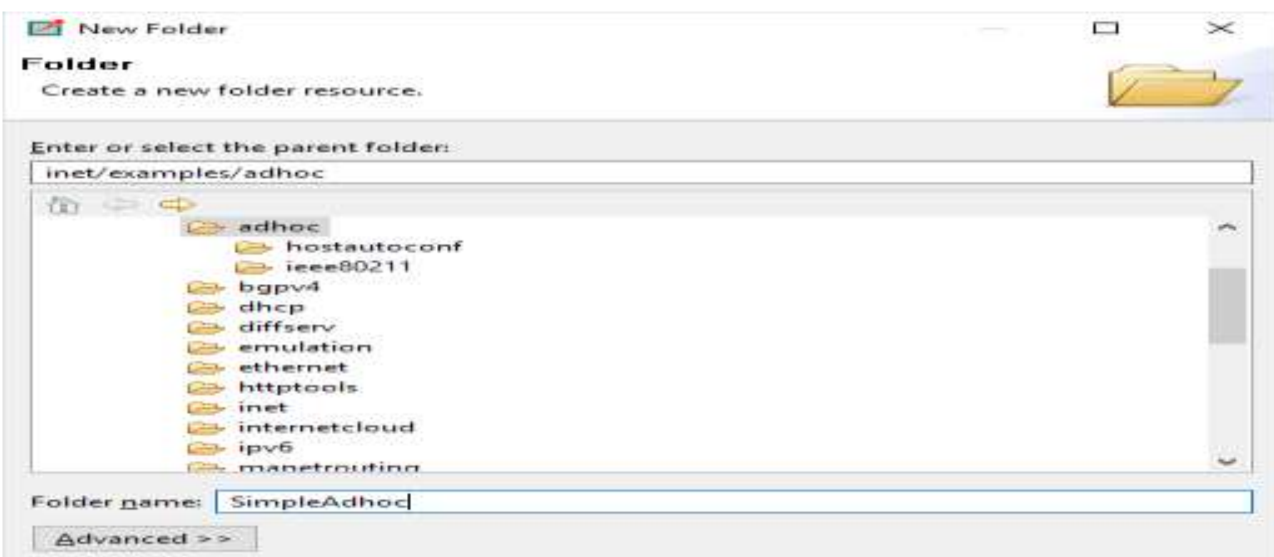


15. An Inet folder will be created in the project explorer.

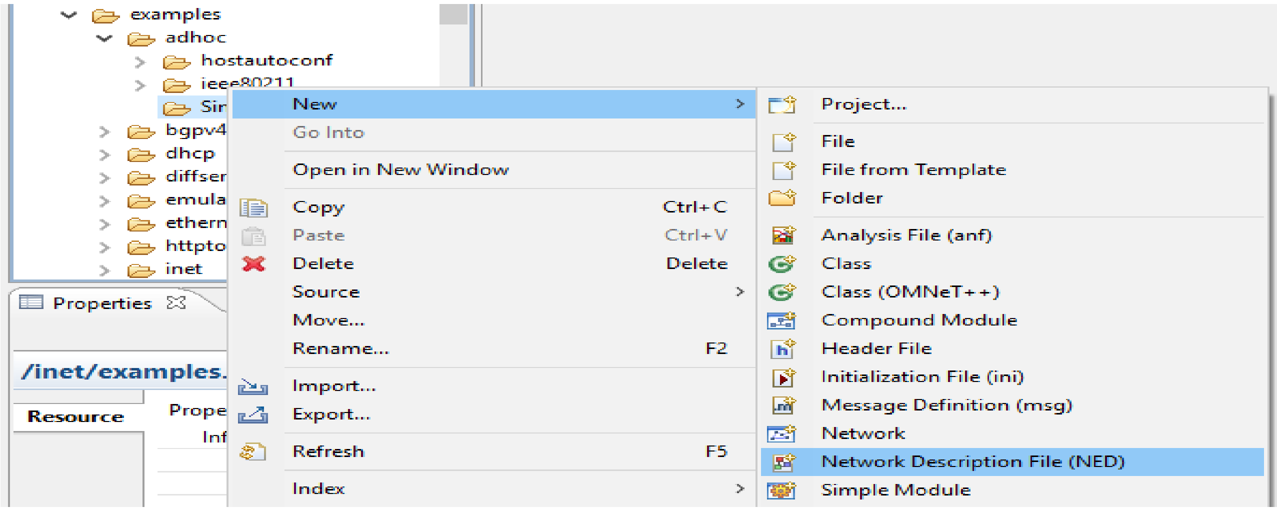


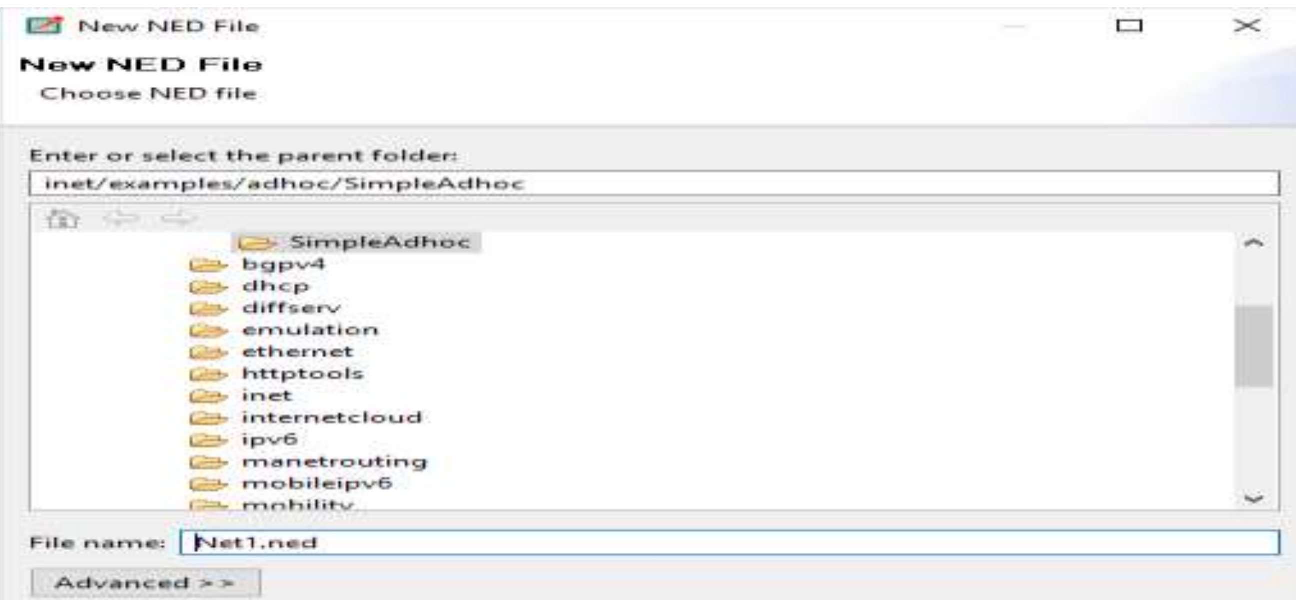
16. Expand the inet folder > examples > adhoc. Right click on adhoc > New > Folder as SimpleAdhoc





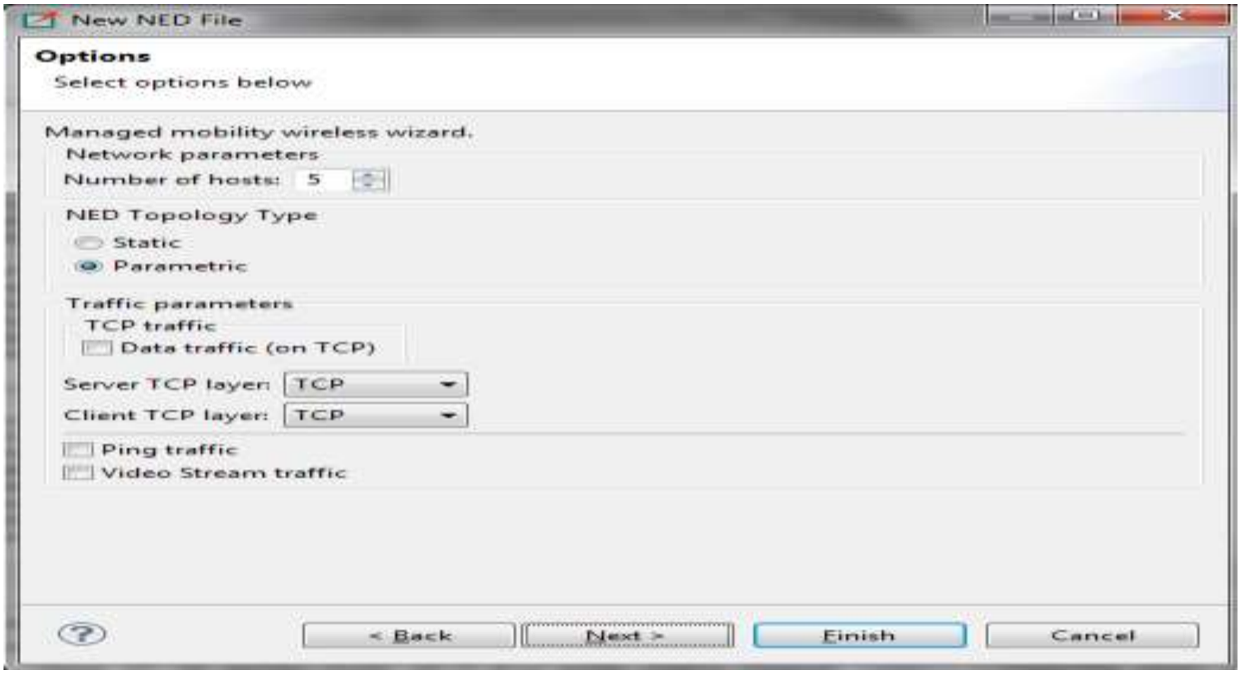
17. Right click on SimpleAdhoc > New > Network Description File (NED). Name the file as Net1.



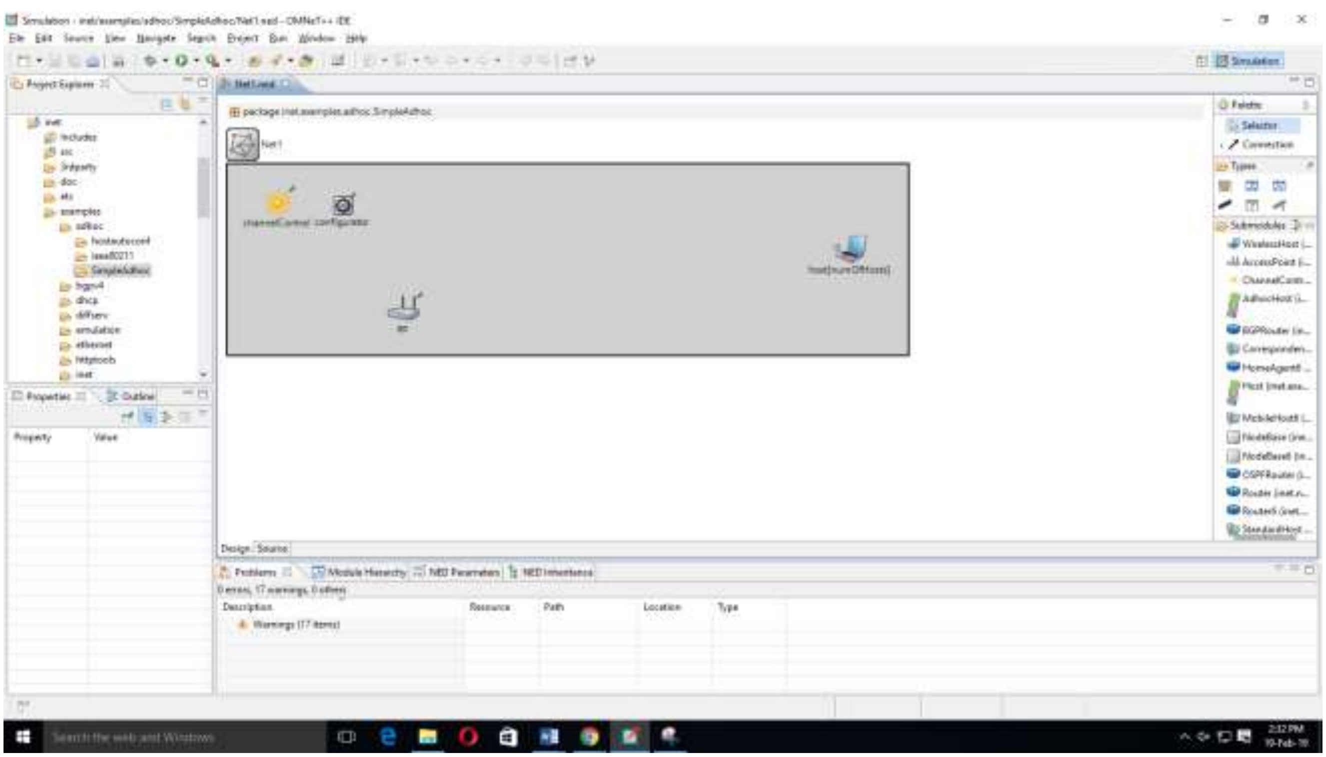


18. Select New Managed Mobility Wireless Network Wizard and Click Next.

19. Configure as follow.



20. Click on Finish.



21. To configure, Right click on inet folder > Run as > Run Configurations. Select Net1 > Browse the directory where the project is > Apply > Run.

22. Below is the code that will be available in source part of net1.ned once configured.

```
package inet.examples.adhoc.SimpleAdhoc;

// numOfHosts: 5

import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
import inet.nodes.inet.WirelessHost;
import inet.nodes.wireless.AccessPoint;
import inet.world.radio.ChannelControl;

network Net
{
   parameters:
      int numOfHosts;

   submodules:
      host[numOfHosts]: WirelessHost
      {
         @display("r=,,#707070");
      }

      ap: AccessPoint
      {
         @display("p=213,174;r=,,#707070");
      }

      channelControl: ChannelControl
      {
         numChannels = 2;
         @display("p=61,46");
      }

      configurator: IPv4NetworkConfigurator
      {
         @display("p=140,50");
      }
}
```

23. Source code for omnetpp.ini

```
[General]
network = Net1

*.numOfHosts = 5

#debug-on-errors = true
tkenv-plugin-path = ../../../etc/plugins

**.constraintAreaMinX = 0m
**.constraintAreaMinY = 0m
**.constraintAreaMinZ = 0m
**.constraintAreaMaxX = 600m
**.constraintAreaMaxY = 400m
**.constraintAreaMaxZ = 0m
**.debug = true
**.coreDebug = false
**.host*.**.channelNumber = 0

# channel physical parameters
*.channelControl.carrierFrequency = 2.4GHz
*.channelControl.pMax = 2.0mW
*.channelControl.sat = -110dBm
*.channelControl.alpha = 2

# mobility
**.host*.mobilityType = "MassMobility"
**.host*.mobility.initFromDisplayString = false
**.host*.mobility.changeInterval = truncnormal(2s, 0.5s)
**.host*.mobility.changeAngleBy = normal(0deg, 30deg)
**.host*.mobility.speed = truncnormal(20mps, 8mps)
**.host*.mobility.updateInterval = 100ms

# ping app (host[0] pinged by others)
*.host[0].numPingApps = 0
*.host[*].numPingApps = 2
*.host[*].pingApp[*].destAddr = "host[0]"
**.pingApp[0].startTime = uniform(1s,5s)
**.pingApp[1].startTime = 5s+uniform(1s,5s)
**.pingApp[*].printPing = true

# nic settings
**.wlan[*].bitrate = 2Mbps

**.wlan[*].mgmt.frameCapacity = 10
```

```
**.wlan[*].mac.address = "auto"
**.wlan[*].mac.maxQueueSize = 14
**.wlan[*].mac.rtsThresholdBytes = 3000B
**.wlan[*].mac.retryLimit = 7
**.wlan[*].mac.cwMinData = 7

**.wlan[*].radio.transmitterPower = 2mW
**.wlan[*].radio.thermalNoise = -110dBm
**.wlan[*].radio.sensitivity = -85dBm
**.wlan[*].radio.pathLossAlpha = 2
**.wlan[*].radio.snirThreshold = 4dB

[Config Ping1]
description = "host1 pinging host0"

[Config Ping2] # __interactive__
description = "n hosts"
# leave numHosts undefined here

**.mobility.constraintAreaMinZ = 0m
**.mobility.constraintAreaMaxZ = 0m
**.mobility.constraintAreaMinX = 0m
**.mobility.constraintAreaMinY = 0m
**.mobility.constraintAreaMaxX = 600m
**.mobility.constraintAreaMaxY = 400m
**.debug = false
**.coreDebug = false

**.channelNumber = 0

# channel physical parameters
*.channelControl.carrierFrequency = 2.4GHz
*.channelControl.pMax = 20.0mW
*.channelControl.sat = -110dBm
*.channelControl.alpha = 2

# mobility

**.host[*].mobilityType = "MassMobility"
**.host[*].mobility.changeInterval = truncnormal(2s, 0.5s)
**.host[*].mobility.changeAngleBy = normal(0deg, 30deg)
**.host[*].mobility.speed = truncnormal(20mps, 8mps)
**.host[*].mobility.updateInterval = 100ms
```

 # nic settings
**.bitrate = 2Mbps

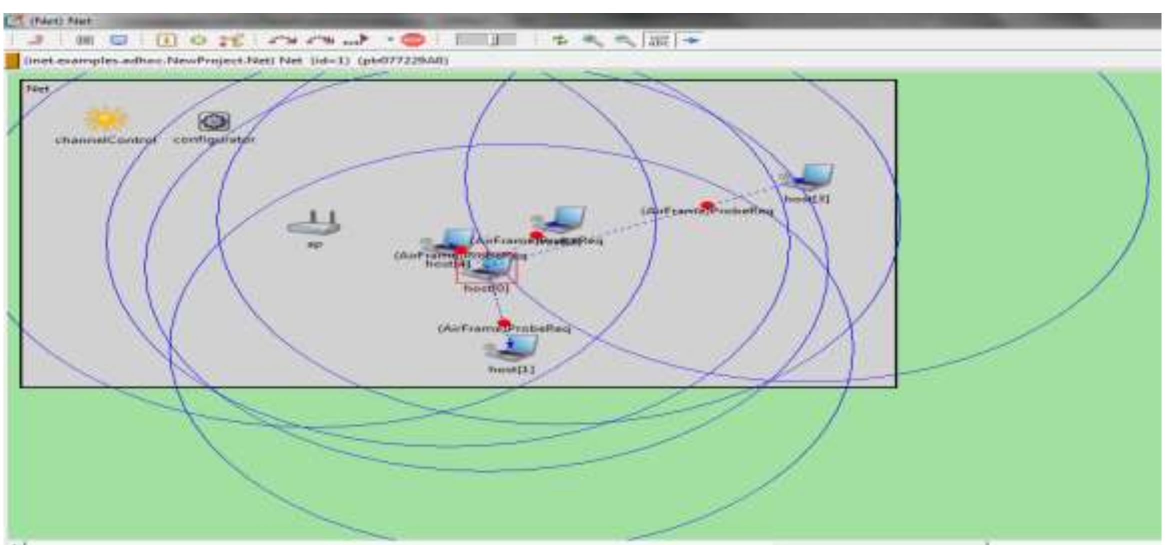
**.mac.address = "auto"
**.mac.maxQueueSize = 14
**.mac.rtsThresholdBytes = 3000B
**.wlan[*].mac.retryLimit = 7
**.wlan[*].mac.cwMinData = 7
**.wlan[*].mac.cwMinMulticast = 31

**.radio.transmitterPower = 20.0mW
**.radio.carrierFrequency = 2.4GHz
**.radio.thermalNoise = -110dBm
**.radio.sensitivity = -85dBm
**.radio.pathLossAlpha = 2
**.radio.snirThreshold = 4dB

 # relay unit configuration
**.relayUnitType = "MACRelayUnitNP"
**.relayUnit.addressTableSize = 100
**.relayUnit.agingTime = 120s
**.relayUnit.bufferSize = 1MiB
**.relayUnit.highWatermark = 512KiB
**.relayUnit.pauseUnits = 300 *# pause for 300*512 bit (19200 byte) time*
**.relayUnit.addressTableFile = ""
**.relayUnit.numCPUs = 2
**.relayUnit.processingTime = 2us

24. Now try to execute by right click on ned file Run as-1-Omnet++ simulation.

Output:

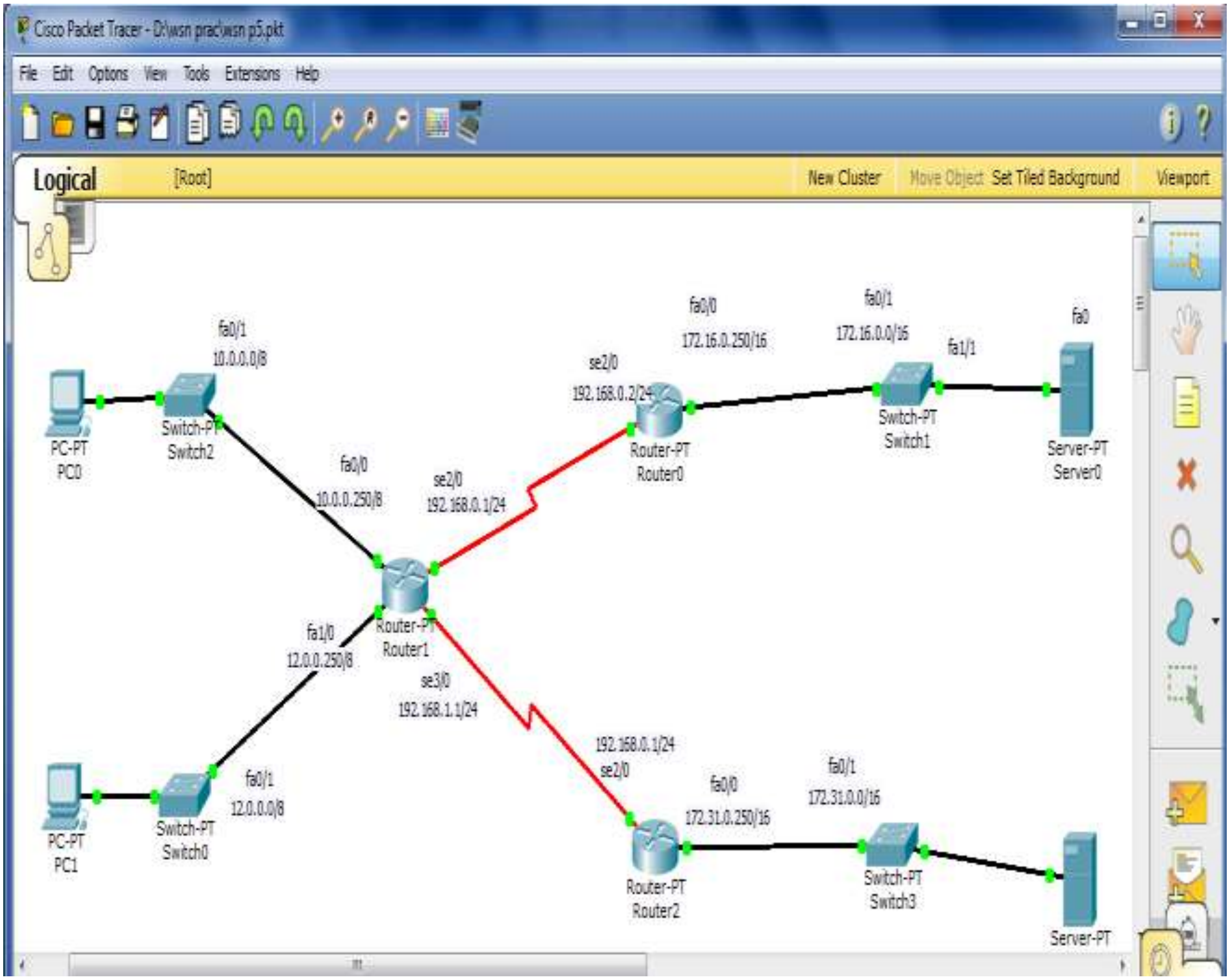


Practical-5

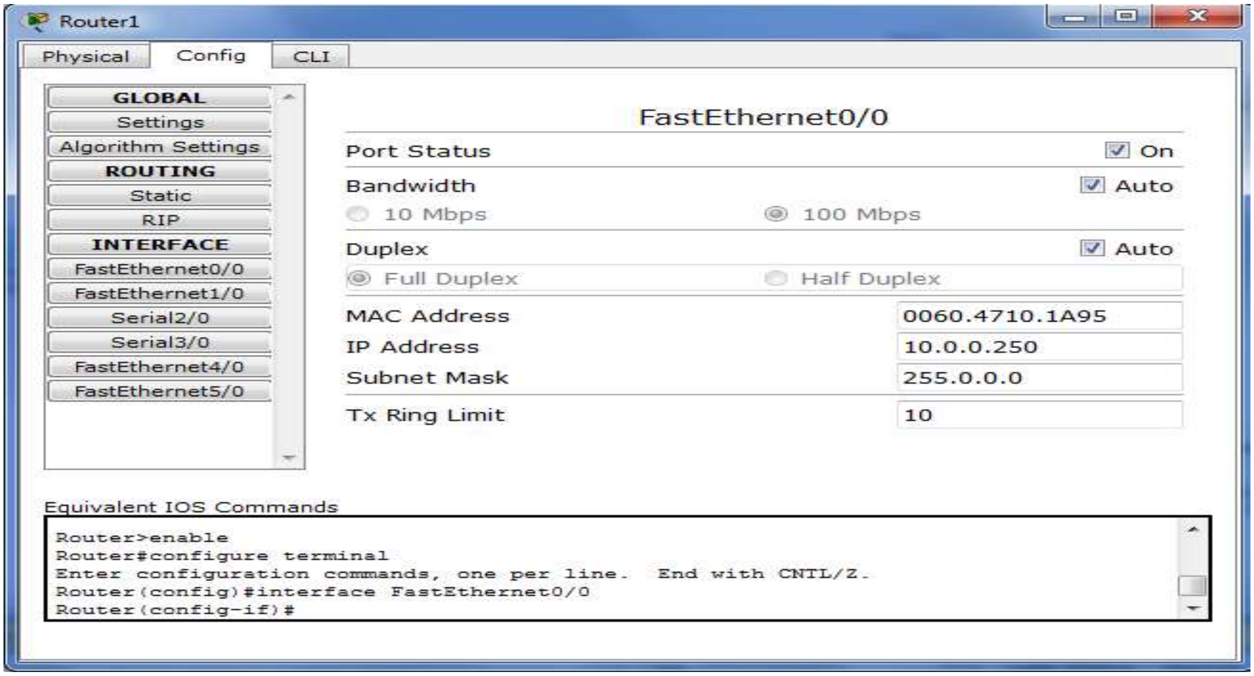
Aim: Understanding, Reading and Analyzing Routing Table of a network.

Steps:

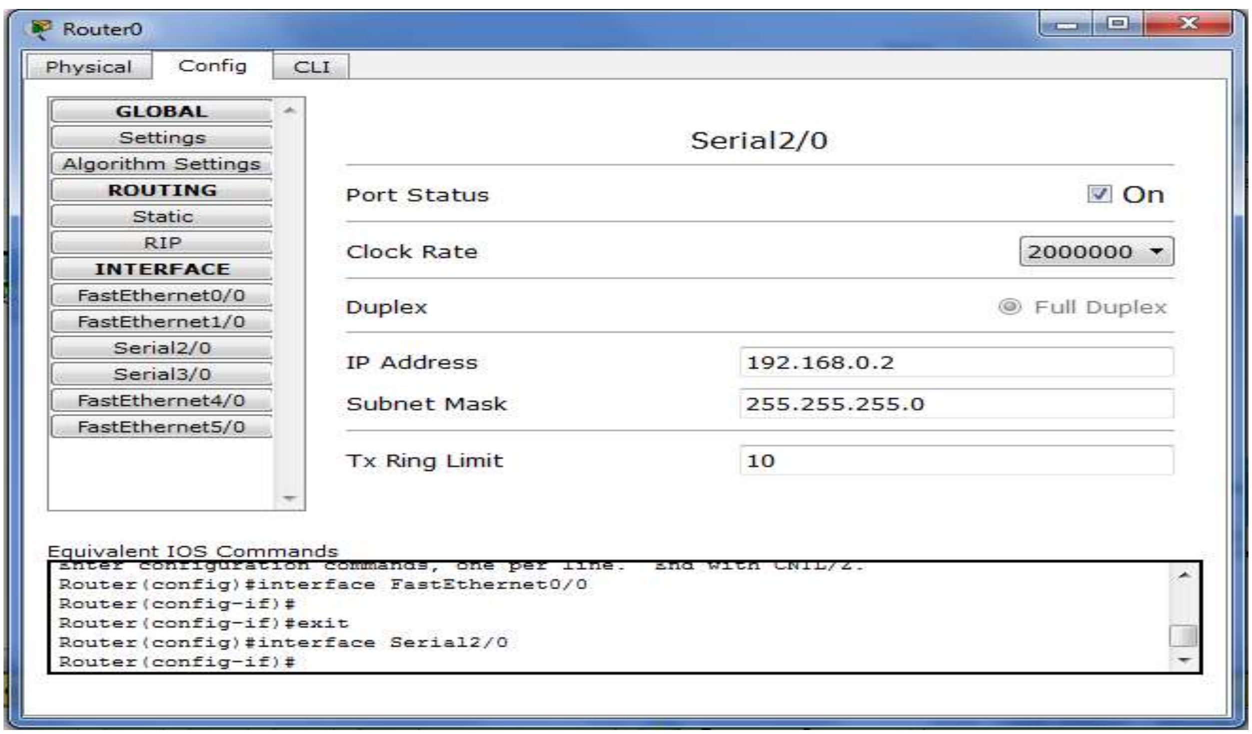
1. Open Cisco Packet Tracer and create the network as shown.



2. Click on Router 1 and make changes in the configuration tab.

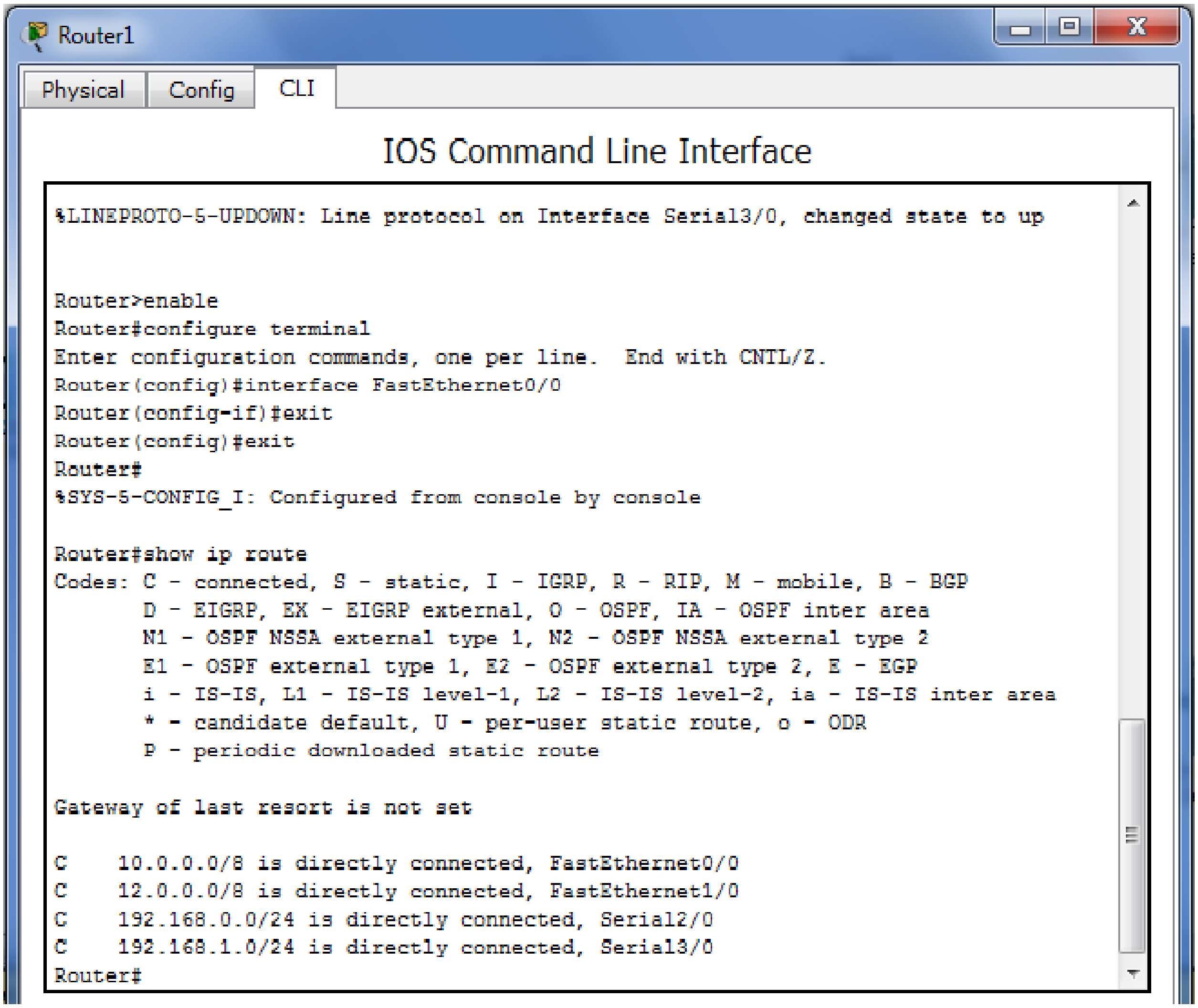


3. Click on Router 0 and make changes in the configuration tab.



4. Click on Router 1. From the following tab click on CLI and analyze the network.

Output:



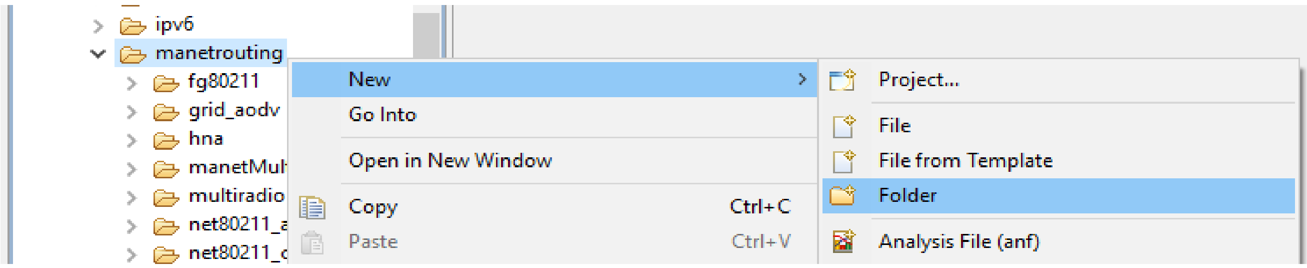
Practical-6

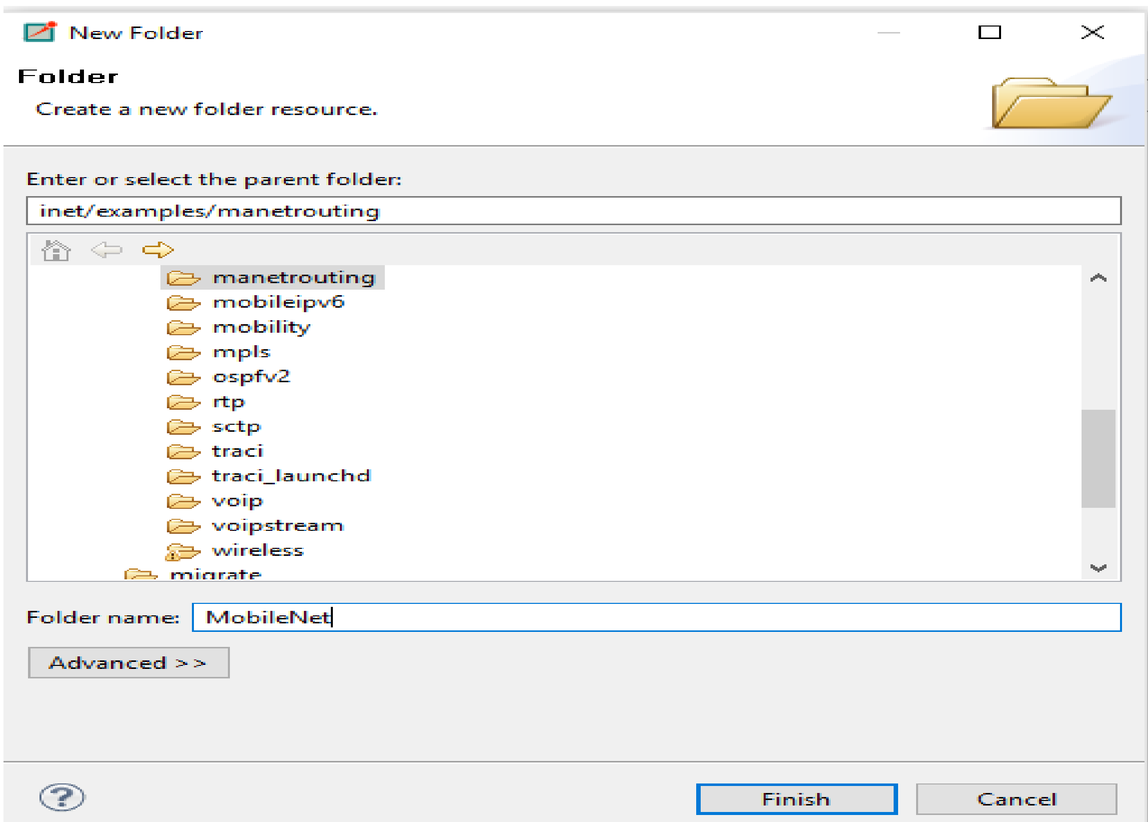
Aim: Create a basic MANET implementation simulation for Packet animation and Packet Trace.

Steps:

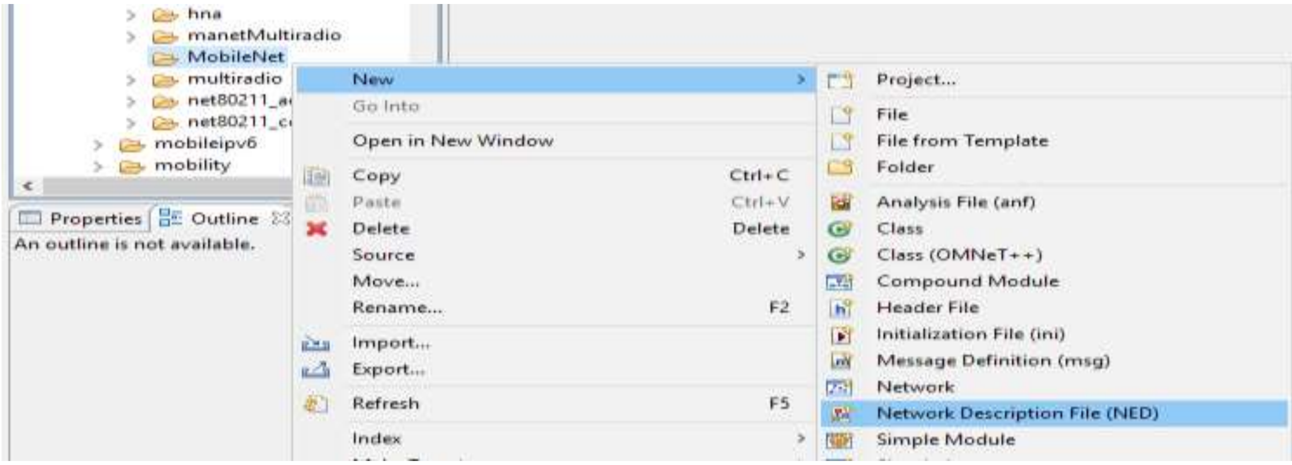
1. Open Omnet++ idle.

2. Expand the inet folder > examples > manetrouting. Right click on manetrouting > New > Folder as MobileNet.

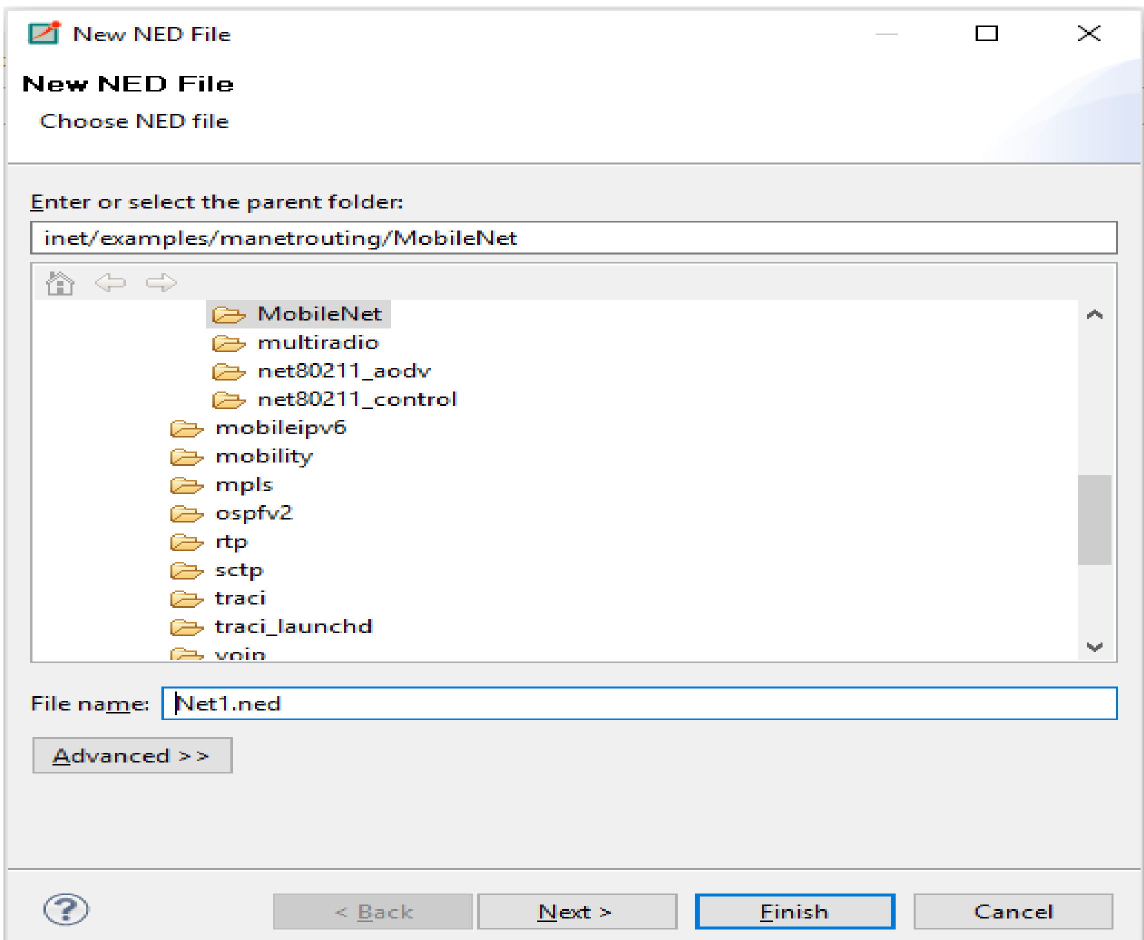




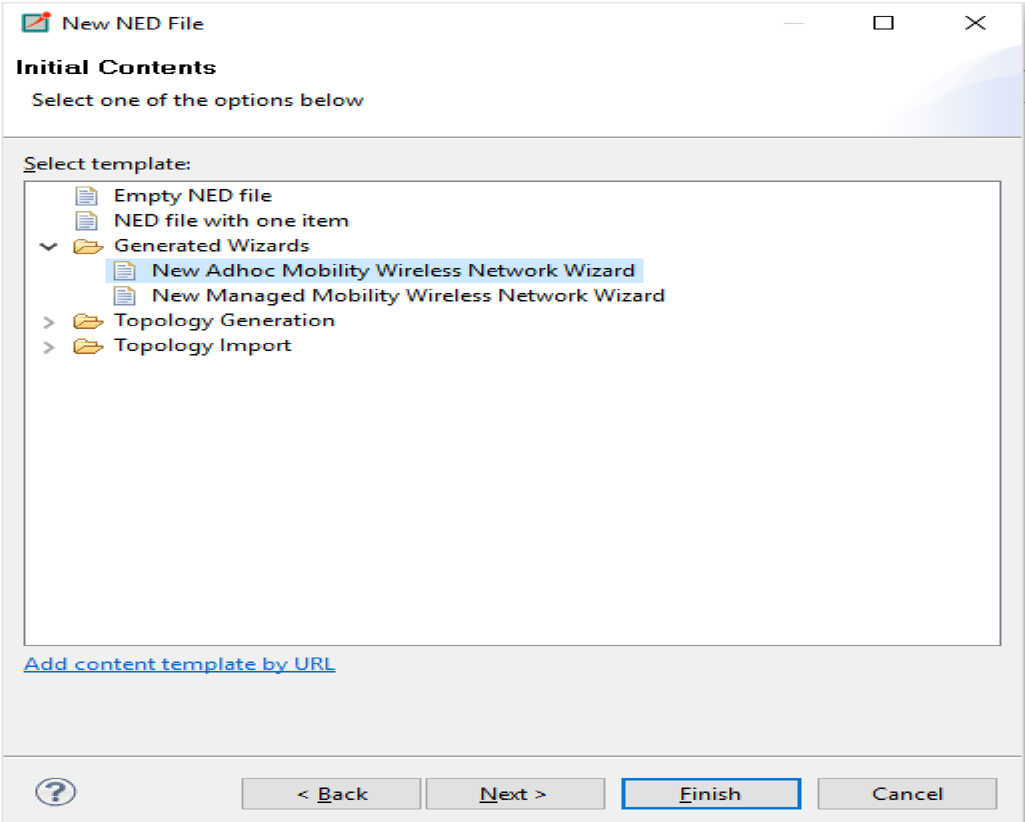
3. Right Click on MobileNet > New > Network Description File (NED).



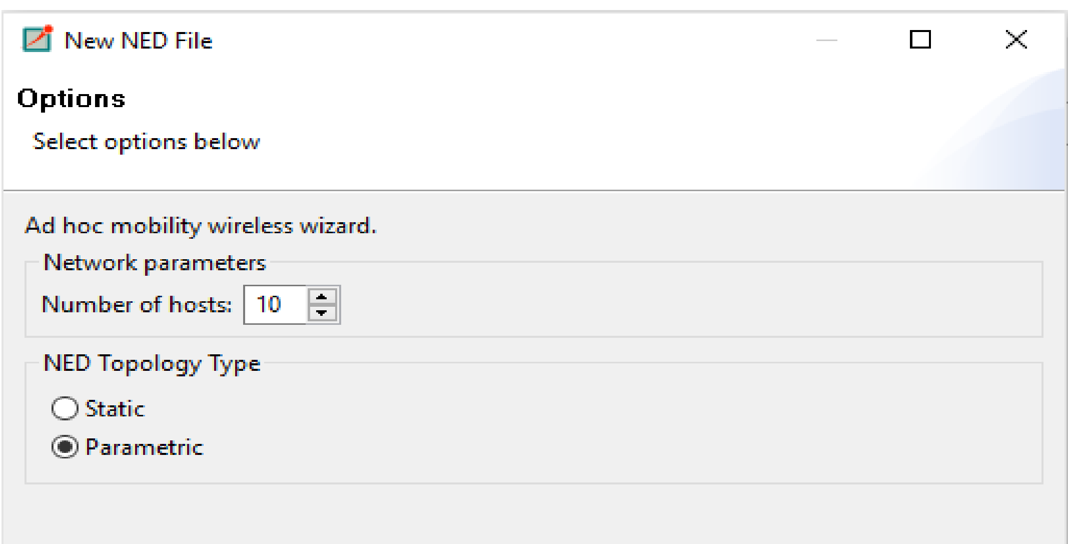
4. Name the file as Net1 and Click Next.



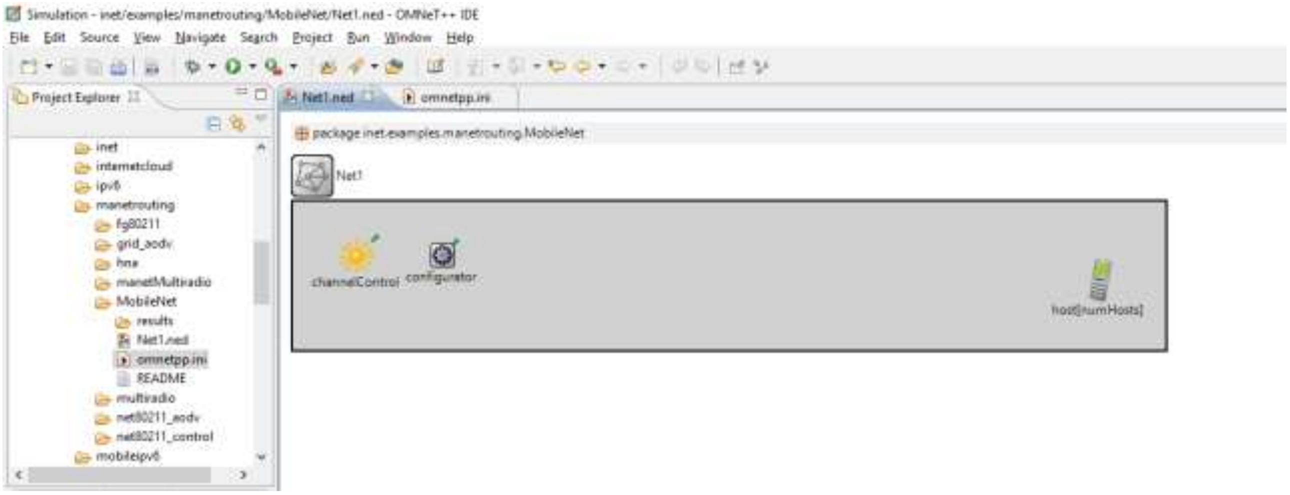
5. Select New Adhoc Mobility Wireless Network Wizard and Click next.



6. Configure as follow.



7. Click on Finish.



8. To configure, Right click on inet folder > Run as > Run Configurations. Select Net1 > Browse the directory where the project is > Apply > Run.

9. Below is the code that will be available in source part of net1.ned once configured.

```
package inet.examples.manetrouting.MobileNet;

// numOfHosts: 10
// parametric: true
// static:     false

import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
import inet.nodes.inet.AdhocHost;
import inet.world.radio.ChannelControl;

network Net1
{
   parameters:
      int numHosts;
   submodules:
      host[numHosts]: AdhocHost
      {
         parameters:
            @display("r=,,#707070");
      }

      channelControl: ChannelControl
      {
         parameters:
```

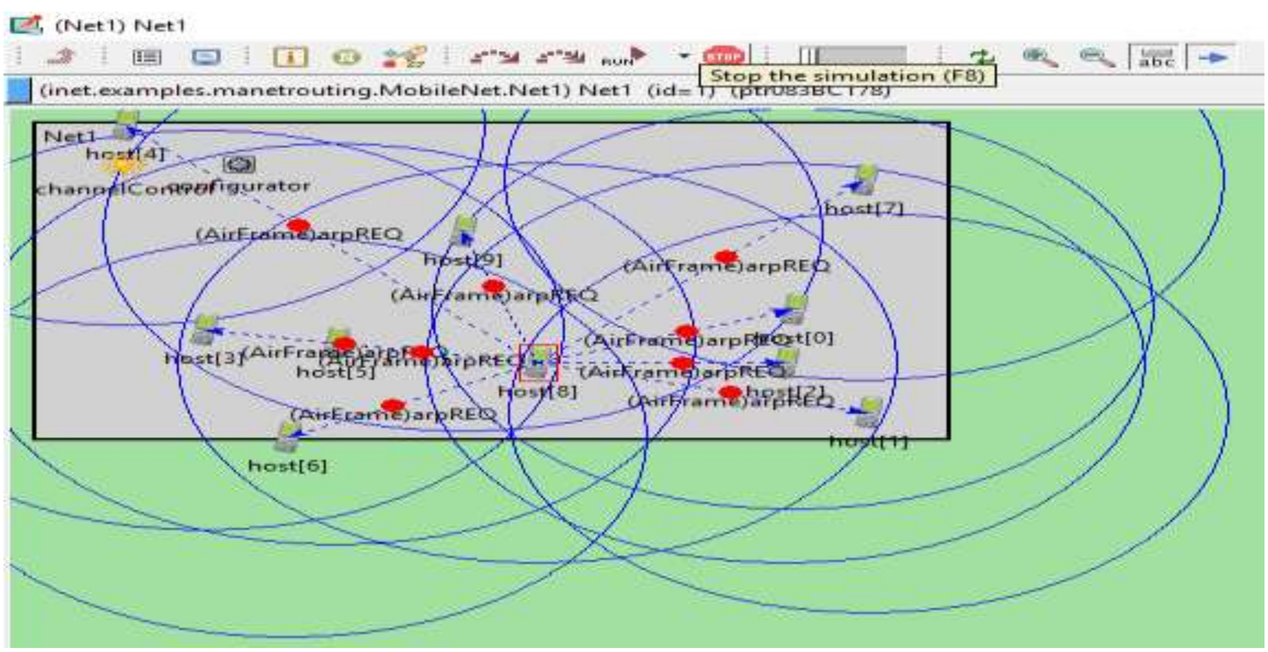
```
        @display("p=60,50");
    }

    configurator: IPv4NetworkConfigurator
    {
        @display("p=140,50");
    }
}
```

10. A file omnetpp.ini will be created with the following source code.

```
[General]
network = Net1
#record-eventlog = true
#eventlog-message-detail-pattern = *:(not declaredOn(cMessage) and not
declaredOn(cNamedObject) and not declaredOn(cObject))

*.numHosts = 10

num-rngs = 3
**.mobility.rng-0 = 1
**.wlan[*].mac.rng-0 = 2
#debug-on-errors = true

tkenv-plugin-path = ../../../etc/plugins

**.channelNumber = 0

# channel physical parameters
*.channelControl.carrierFrequency = 2.4GHz
*.channelControl.pMax = 2.0mW
*.channelControl.sat = -110dBm
*.channelControl.alpha = 2
*.channelControl.numChannels = 1

# mobility
**.host[*].mobilityType = "MassMobility"
**.mobility.constraintAreaMinZ = 0m
**.mobility.constraintAreaMaxZ = 0m
**.mobility.constraintAreaMinX = 0m
**.mobility.constraintAreaMinY = 0m
**.mobility.constraintAreaMaxX = 600m
**.mobility.constraintAreaMaxY = 400m
**.mobility.changeInterval = truncnormal(2s, 0.5s)
**.mobility.changeAngleBy = normal(0deg, 30deg)
```

```
**.mobility.speed = truncnormal(20mps, 8mps)
**.mobility.updateInterval = 100ms

# ping app (host[0] pinged by others)
*.host[0].pingApp[0].destAddr = ""
*.host[*].numPingApps = 1
*.host[*].pingApp[0].destAddr = "host[0]"
*.host[*].pingApp[0].startTime = uniform(1s,5s)
*.host[*].pingApp[0].printPing = true

# nic settings
**.wlan[*].bitrate = 2Mbps

**.wlan[*].mgmt.frameCapacity = 10
**.wlan[*].mac.address = "auto"
**.wlan[*].mac.maxQueueSize = 14
**.wlan[*].mac.rtsThresholdBytes = 3000B
**.wlan[*].mac.retryLimit = 7
**.wlan[*].mac.cwMinData = 7
**.wlan[*].mac.cwMinMulticast = 31

**.wlan[*].radio.transmitterPower = 2mW
**.wlan[*].radio.thermalNoise = -110dBm
**.wlan[*].radio.sensitivity = -85dBm
**.wlan[*].radio.pathLossAlpha = 2
**.wlan[*].radio.snirThreshold = 4dB
```

Output:

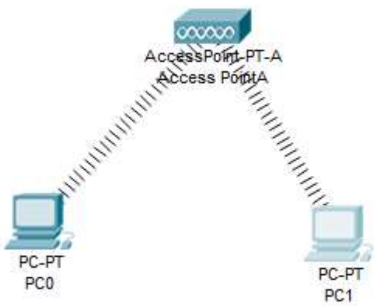


Practical – 7

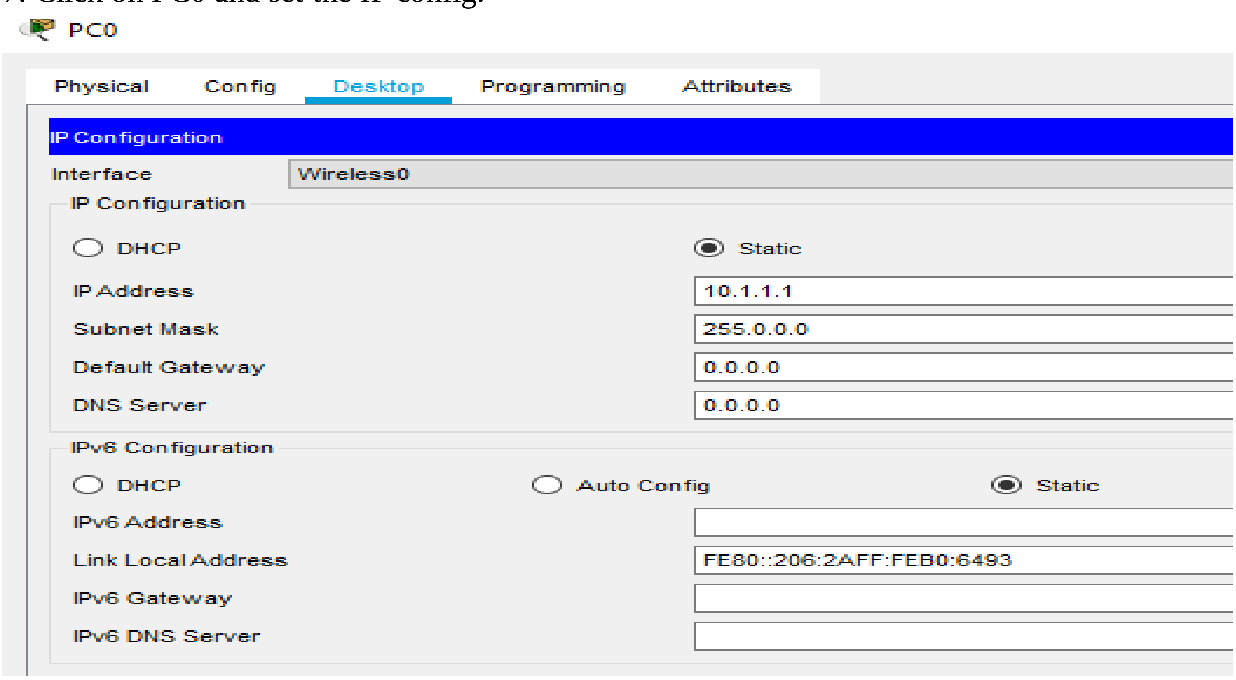
Aim: Implement a Wireless sensor network simulation.

Steps:

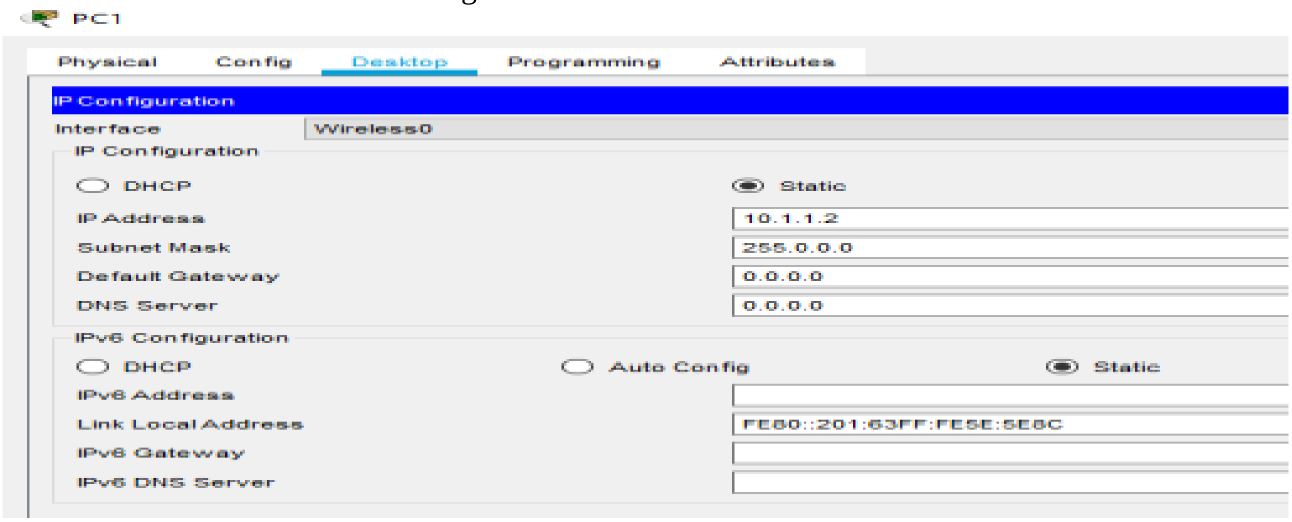
1. Create the following network using AccessPoint-PT-A and PC-PT.



2. Click on PC0 and click on Physical tab.
3. Turn off the CPU and remove the FastEthernet module and install PT-HOST-NM-1W-A and turn On the CPU.
4. A connection will be made between Accesspoint and PC0.
5. Click on PC1 and click on physical tab.
6. Repeat step 3 and see if the connection is done between PC1 and Accesspoint.
7. Click on PC0 and set the IP config.

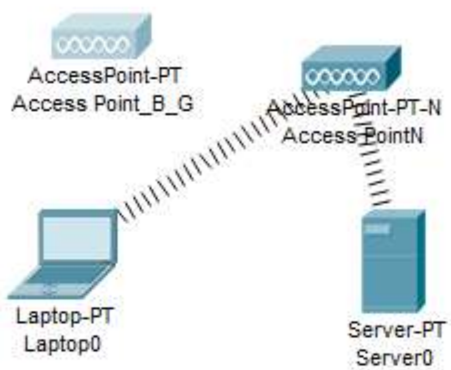


8. Click on PC1 and set the IP config.

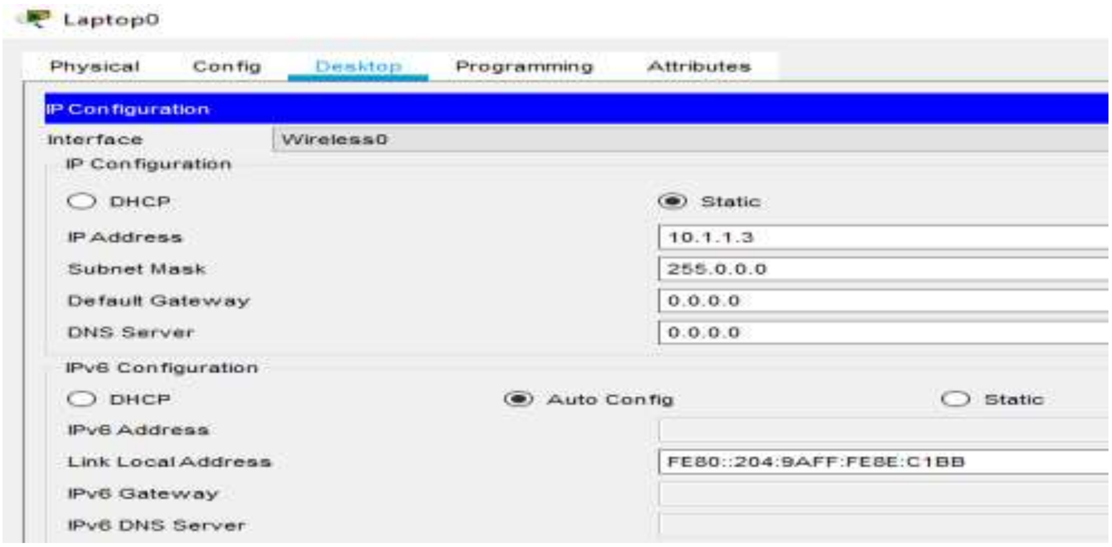


9. Test Access PointA
   a. Ping PC1 (10.1.1.2) from PC0. The ping should succeed.
   b. Ping Laptop0(10.1.1.3) and Server0 (10.1.1.4) from PC0. The pings should fail.

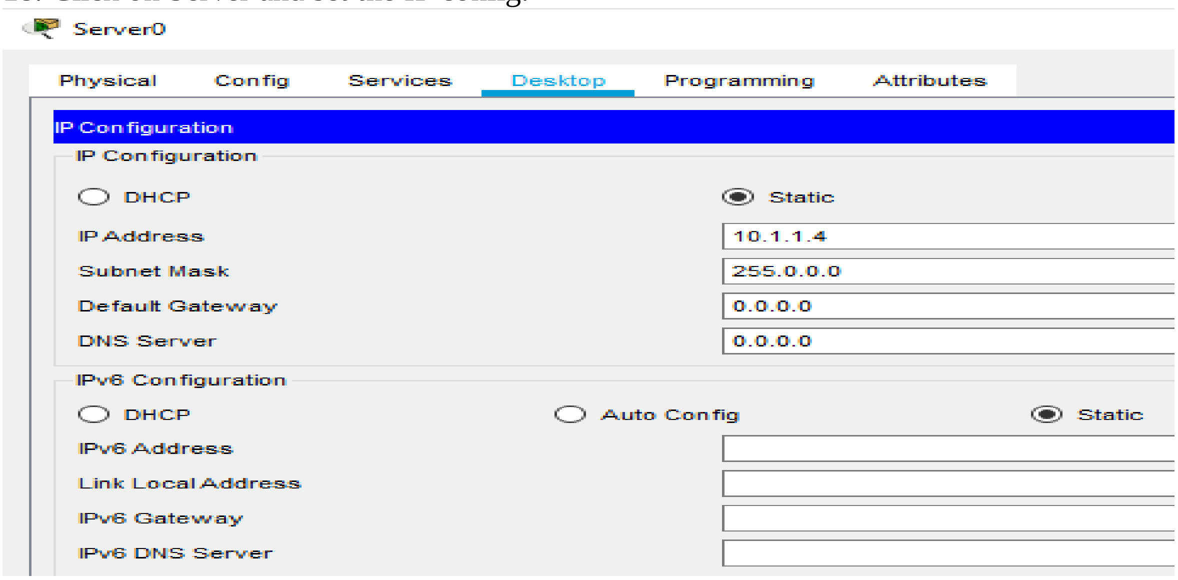
10. Create the following network using Accesspoint-PT, Accesspoint-PT-N, Laptop and Server.



11. Click on Laptop and click on Physical tab.
12. Turn off the Laptop and remove the PT-LAPTOP-NM-1CFE module and install PT-LAPTOP-NM-1W and turn On the Laptop.
13. A connection will be made between Accesspoint-PT-N and Laptop.
14. Click on Laptop and set the IP config.



15. Click on Server and click on Physical tab.
16. Turn off the Server and remove the PT-HOST-NM-1CFE module and install PT-HOST-NM-1W and turn On the Server.
17. A connection will be made between Accesspoint-PT-N and Server.
18. Click on Server and set the IP config.



19. Test Access PointN
    a. Ping Server0 (10.1.1.4) from Laptop0. The ping should succeed.
    b. Ping PC0 (10.1.1.1) and PC1 (10.1.1.2) from Laptop0. The pings should fail.

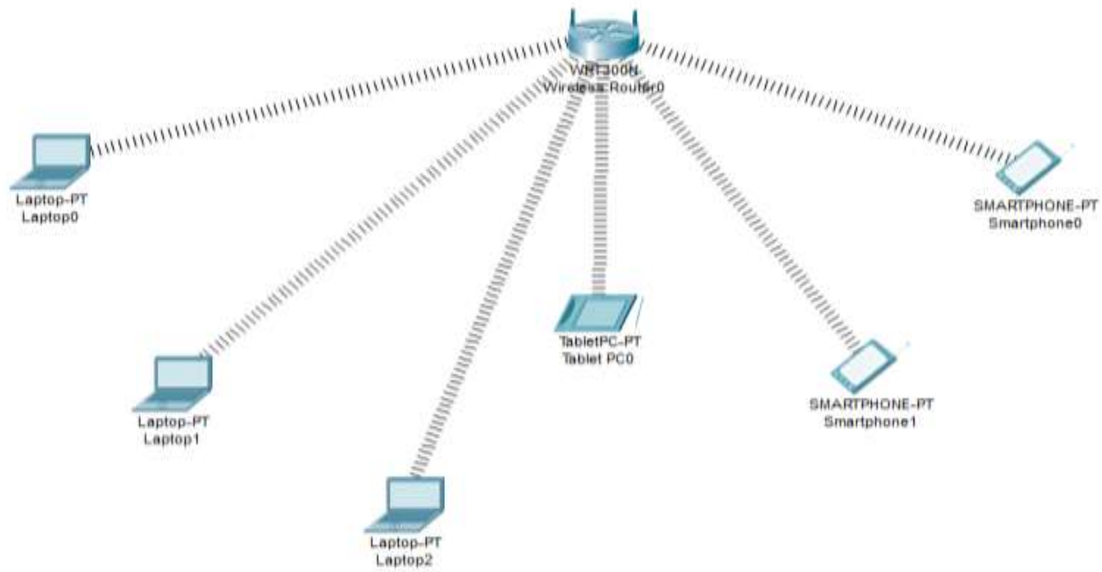
20. Now Turn off the port of AccesspointN and Test Access Point_B_G
    a. Turn on Port1 on Access Point_B_G and turn off Port1 on Access PointN. Laptop0 and Server0 should associate with Access Point_B_G.
    b. Ping Server0 (10.1.1.4) from Laptop0. The ping should succeed.

Practical – 8

Aim: Create MAC protocol simulation implementation for wireless sensor Network.

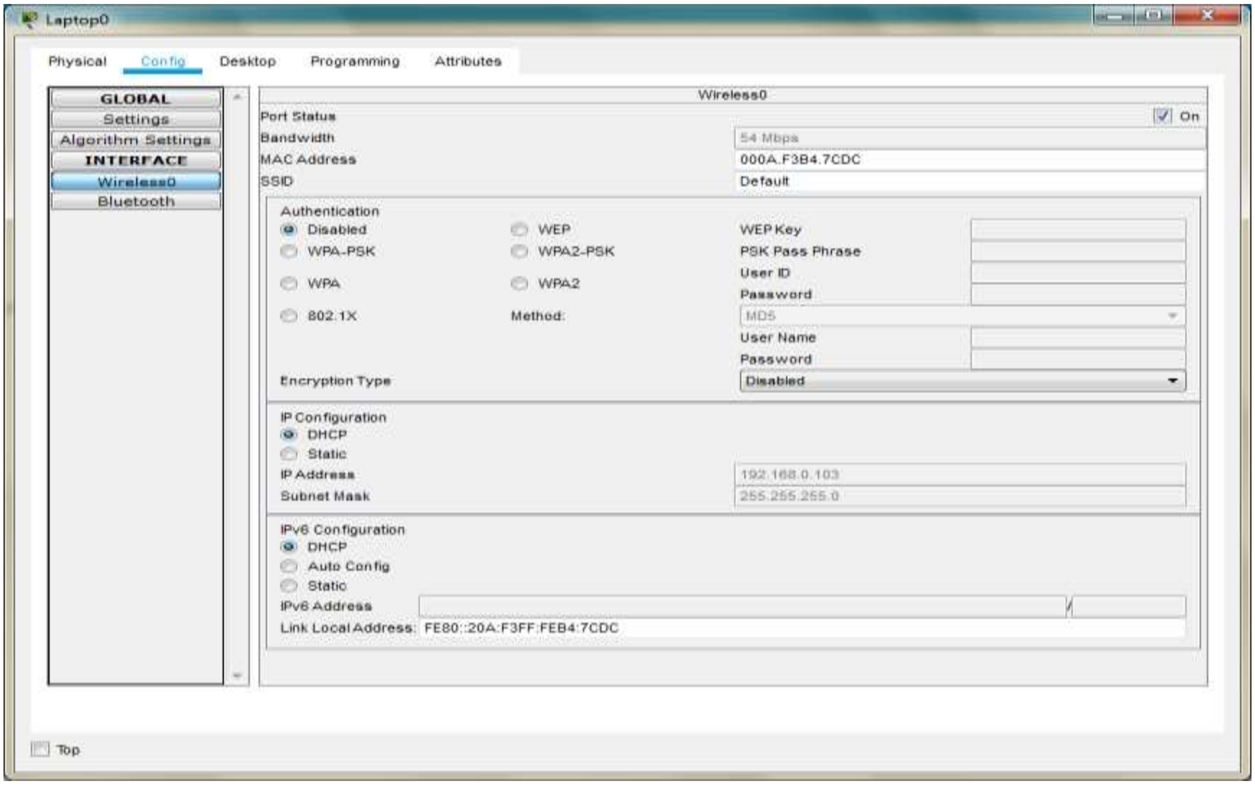
Steps:

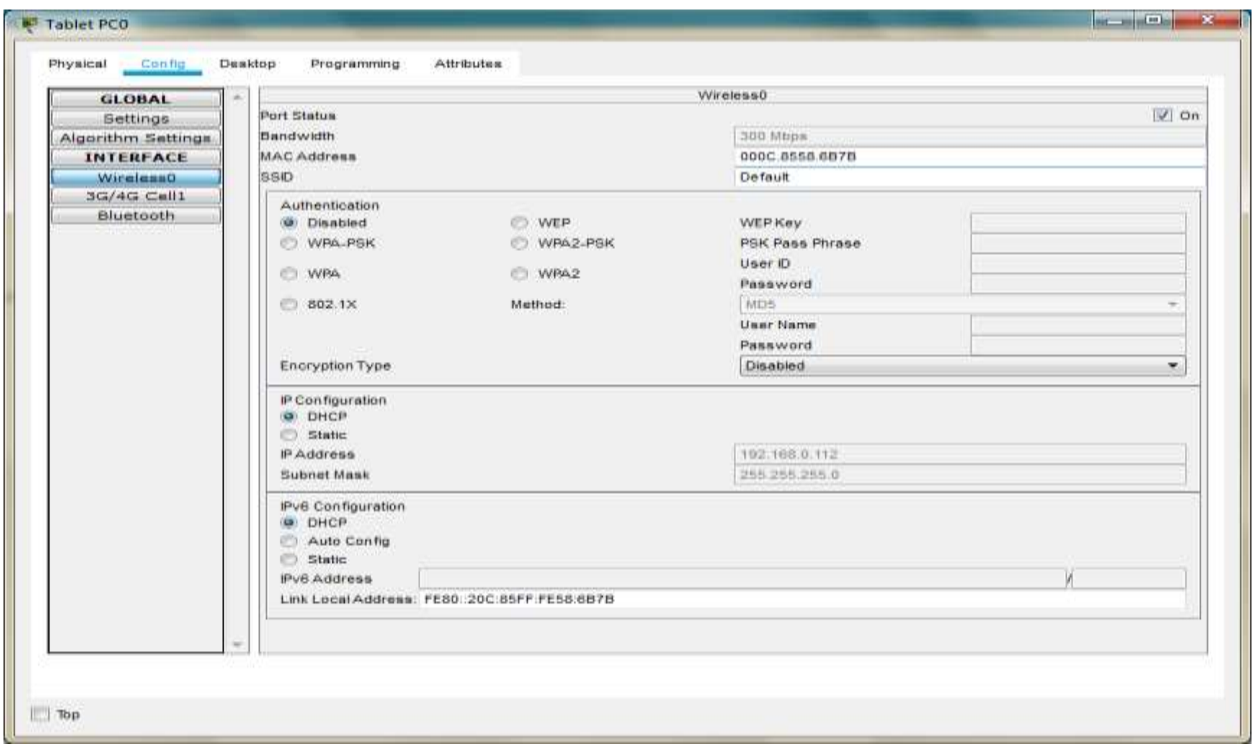
Create the following network.

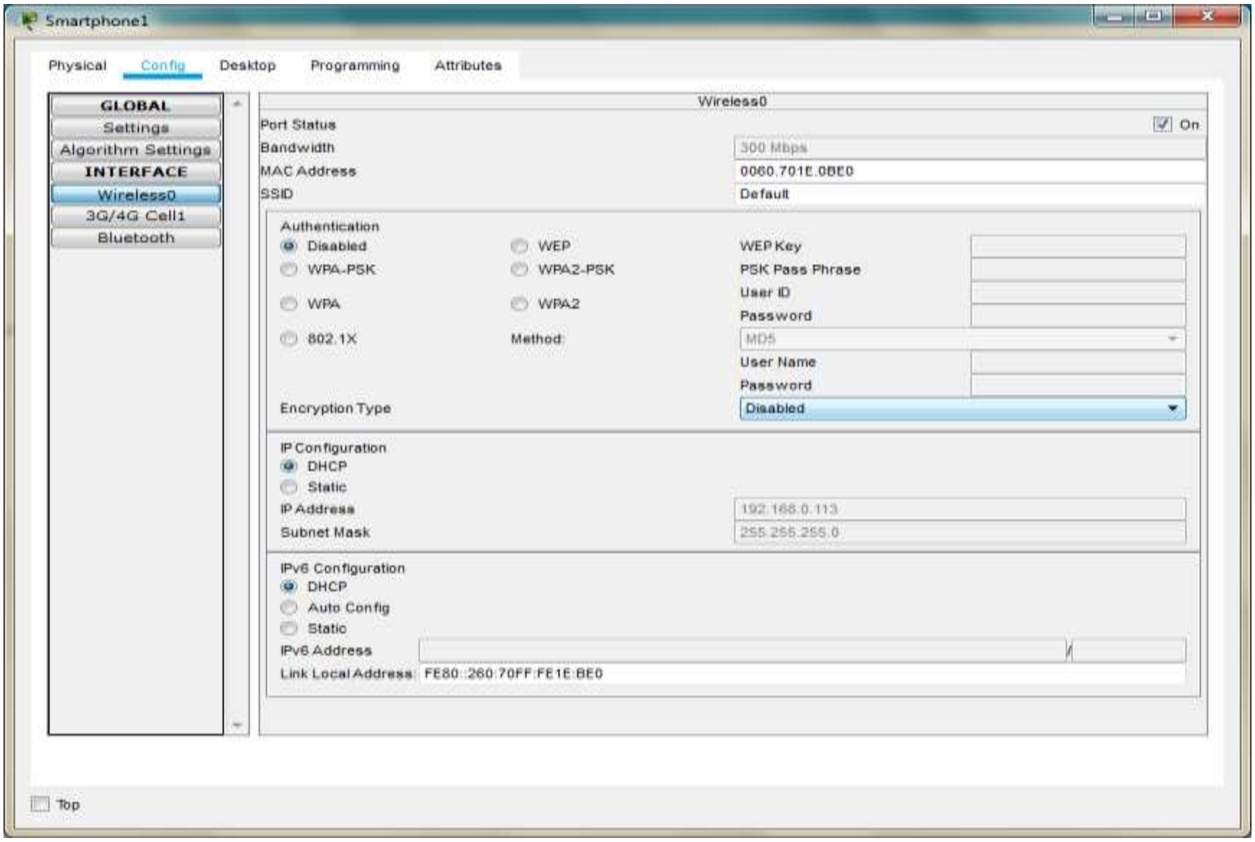


Click on Every laptop and change the interface to PT-LAPTOP-NM-1W.

Copy the MAC address of each component as follows



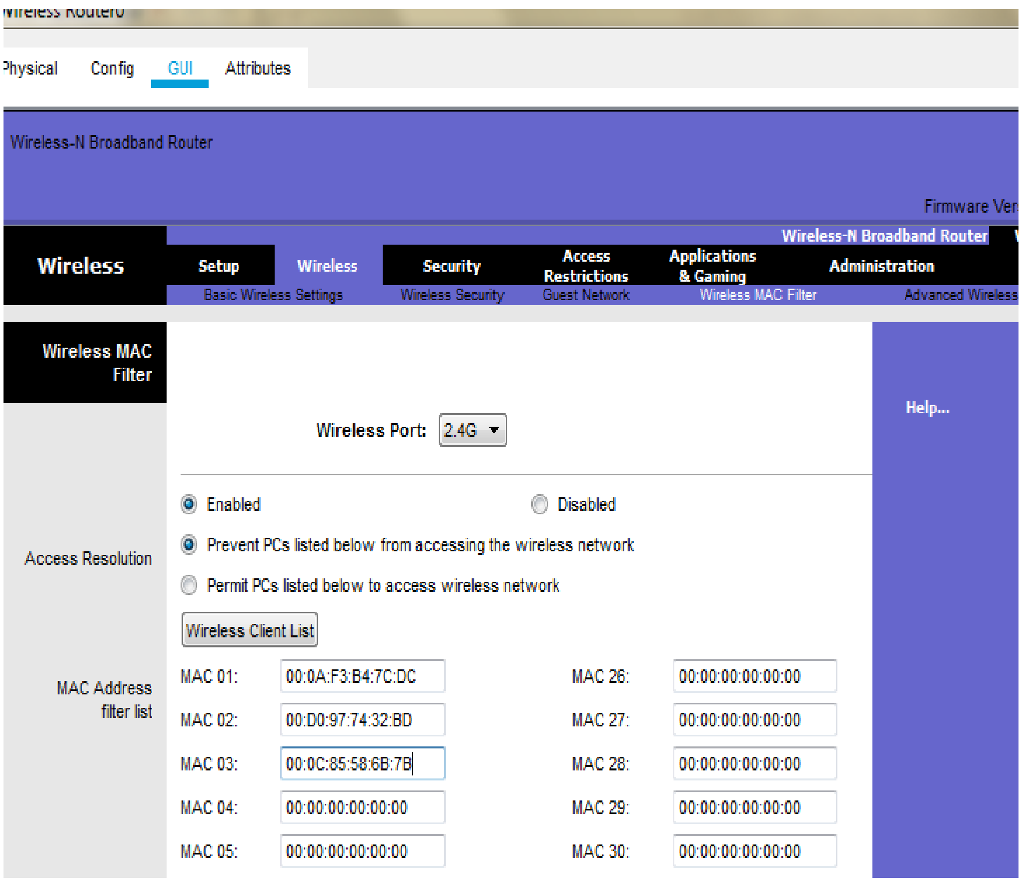




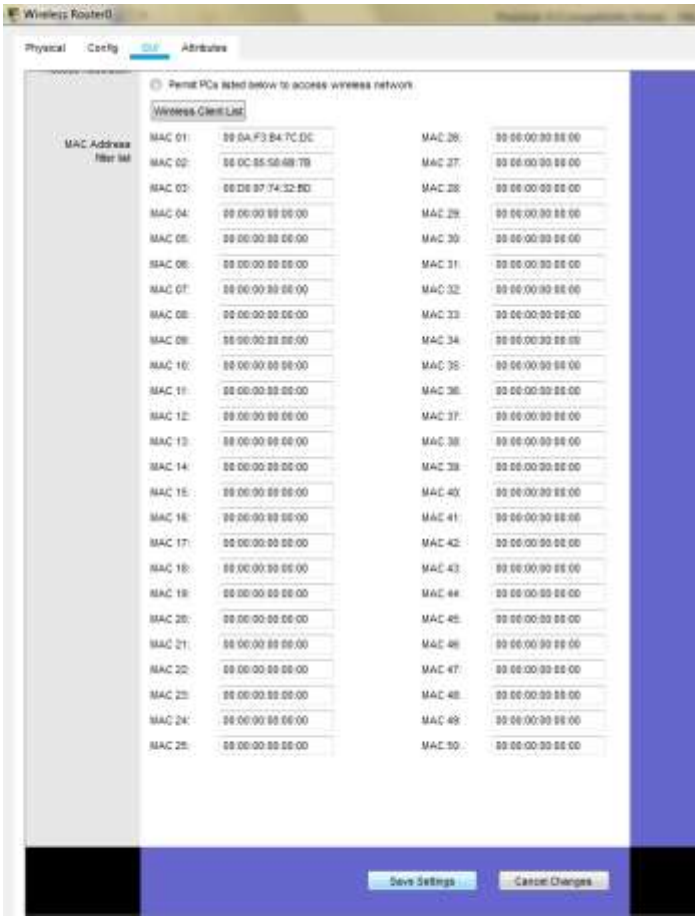
We note the following MAC addresses and convert them to the following form

Component	MAC Address	Converted MAC address
Laptop0	000A.F3B4.7CDC	00:0A:F3:B4:7C:DC
Laptop1	0001.4269.6539	00:01:42:69:65:39
Laptop2	0060.5CB8.B919	00:60:5C:B8:B9:19
TabletPC	000C.8558.6B7B	00:0C:85:58:6B:7B
SmartPhone0	00D0.9774.32BD	00:D0:97:74:32:BD
SmartPhone1	0060.701E.0BE0	00:60.70:1E:0B:E0

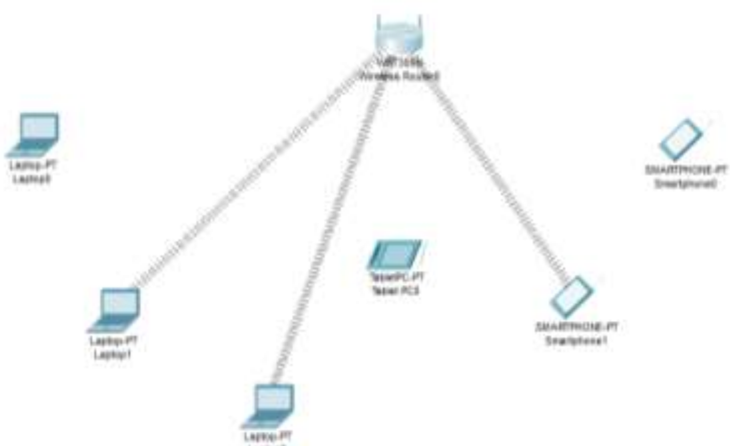
Now we add few addresses in the wireless MAC filter of the Wireless Router and then use the given options for either allow or deny the Wireless access



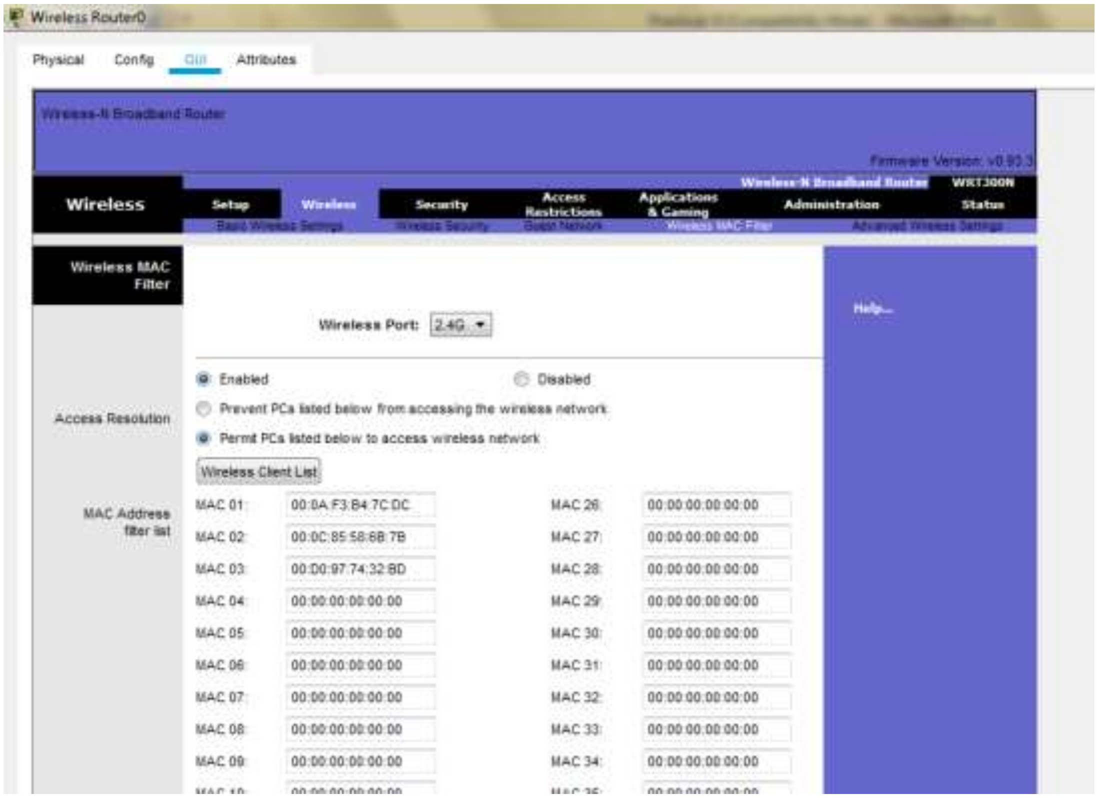
As seen in above screen shot we add the MAC address of Laptop0, TabletPC SmartPhone0 in the list so as to deny them accessing the Wireless network and then save the settings



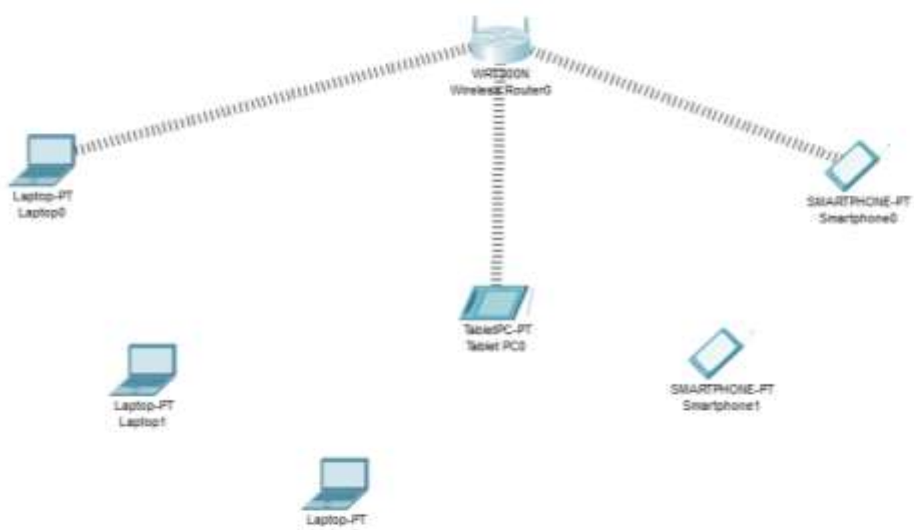
The result so obtained is as shown, the three devices denied any wireless connectivity



Similarly we can change the setting so that the above devices get wireless connectivity and the remaining devices do not get the wireless connectivity



And save the setting and get the following

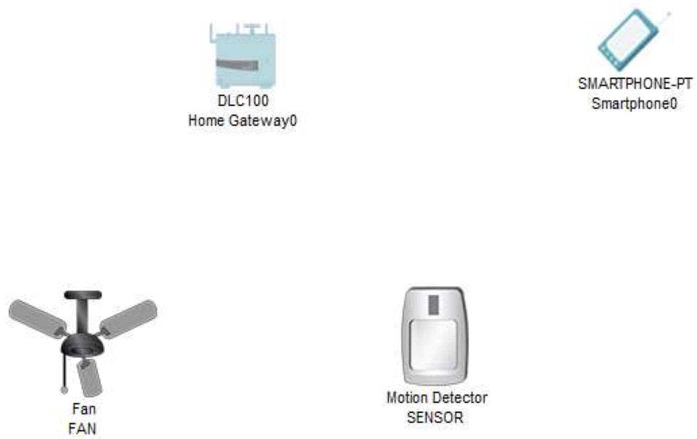


Practical – 9

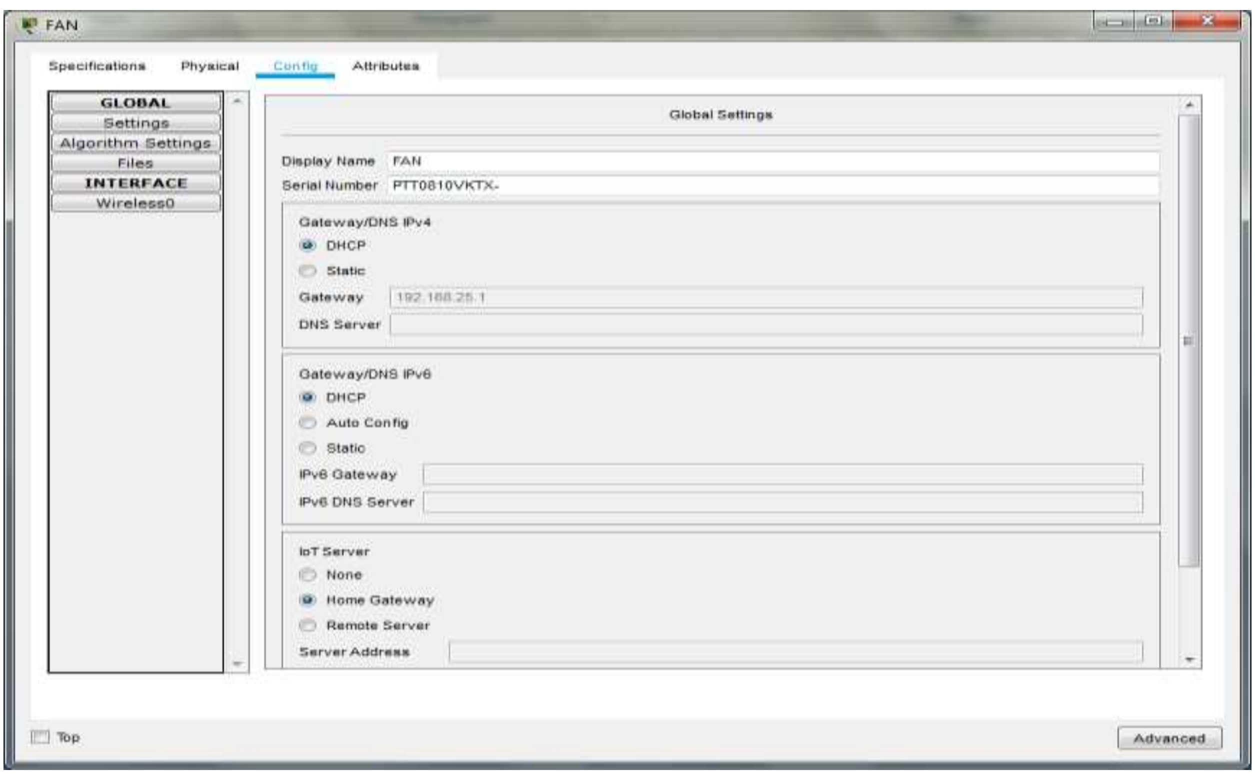
Aim: Simulate Mobile Adhoc Network with Directional Antenna.

Steps:

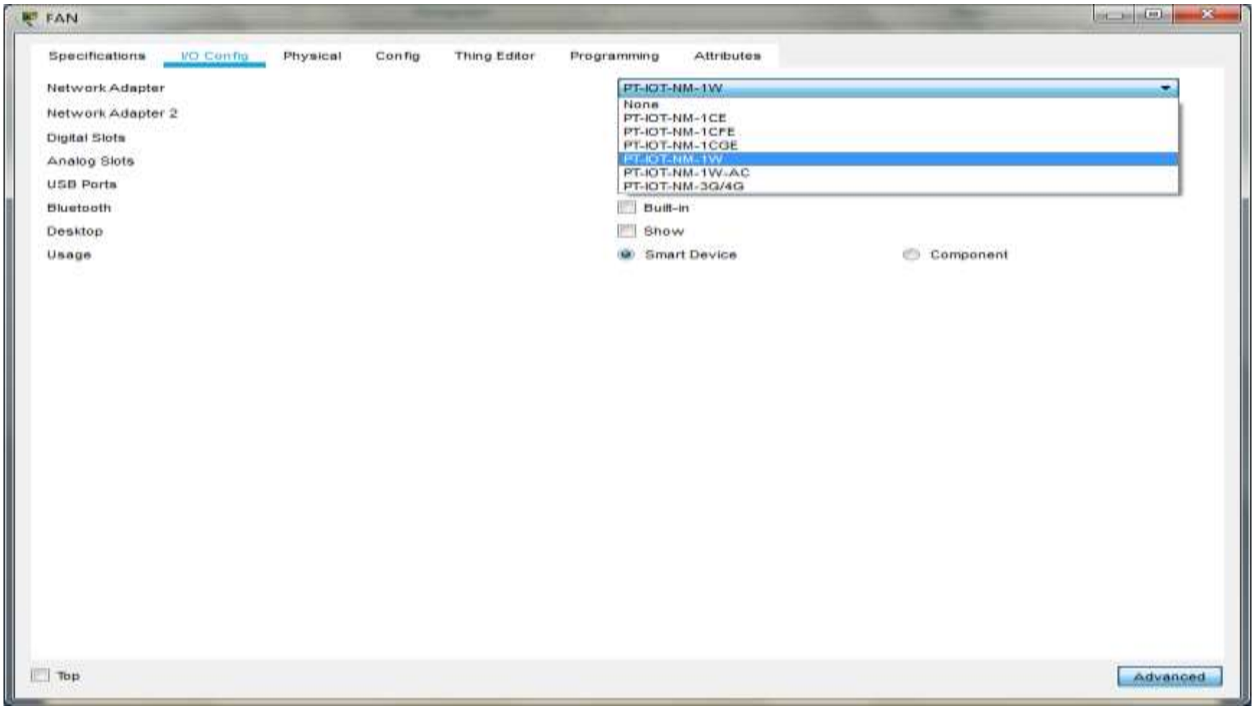
Create the following network.



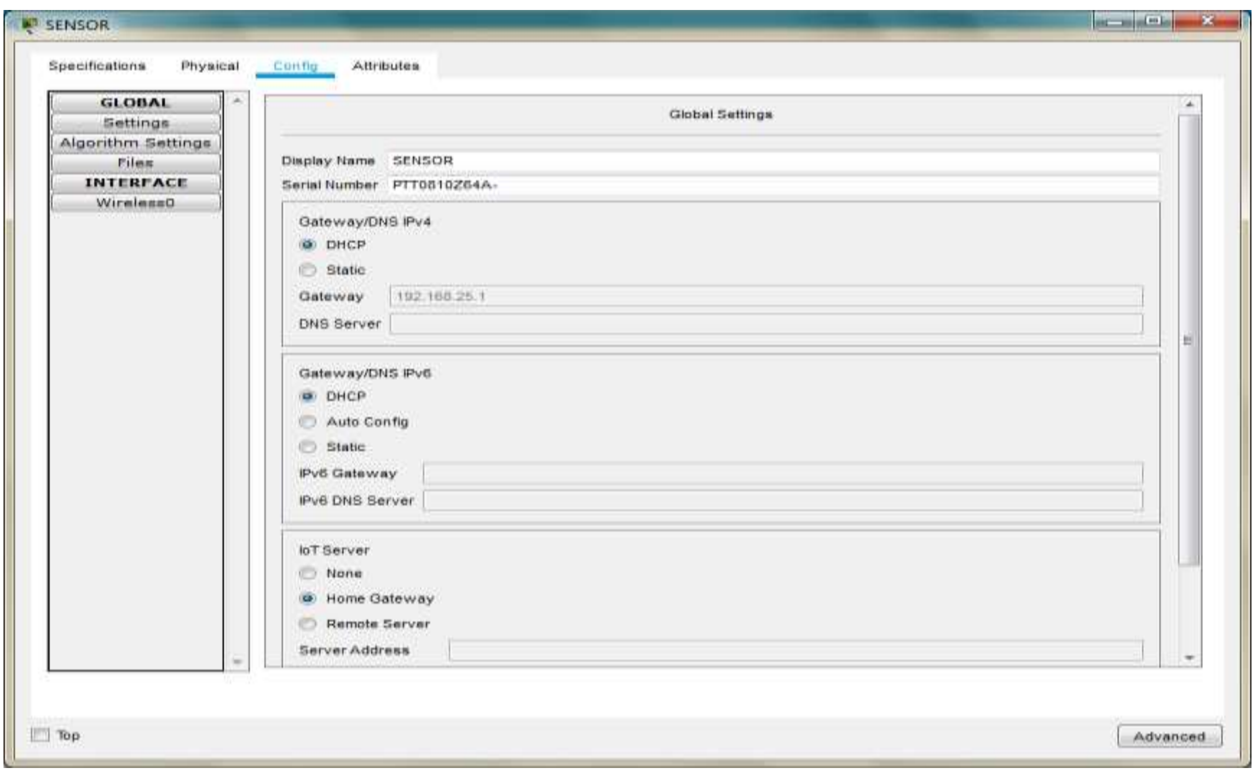
Click on the Fan and do the following



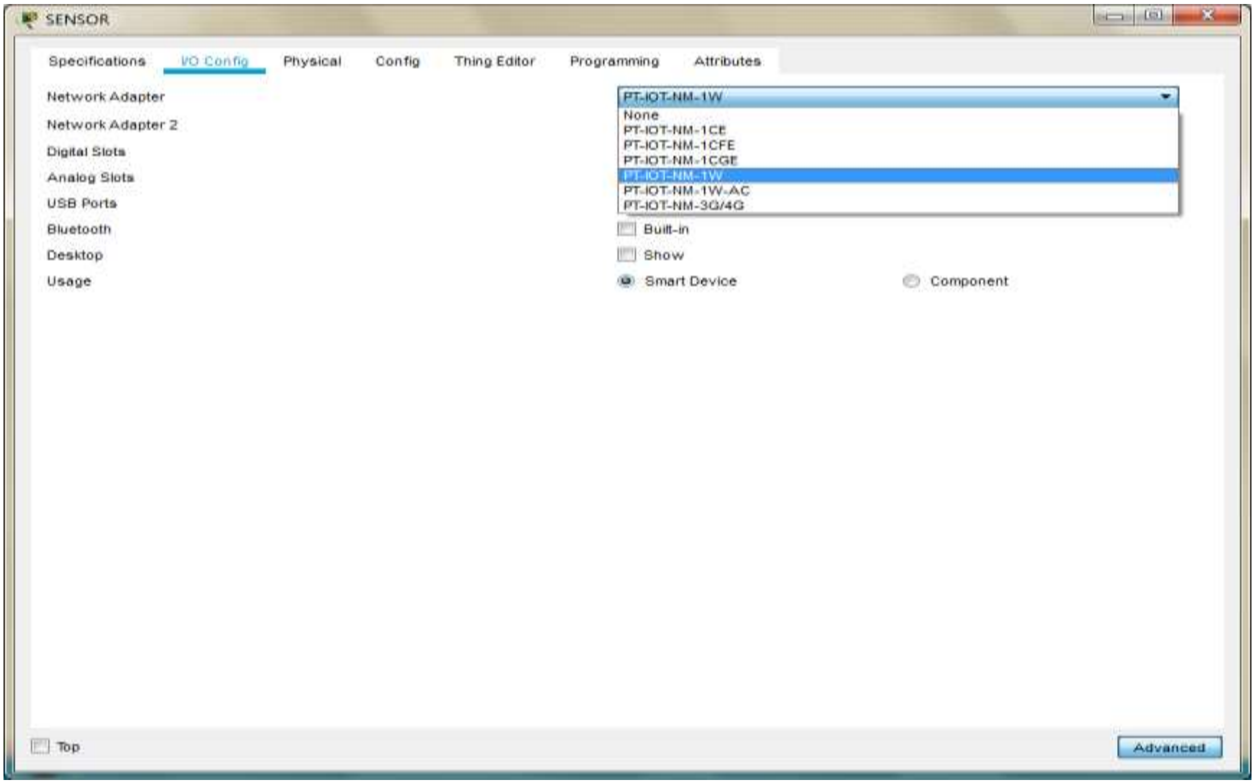
In the Advanced setting do the following for the Network adapter



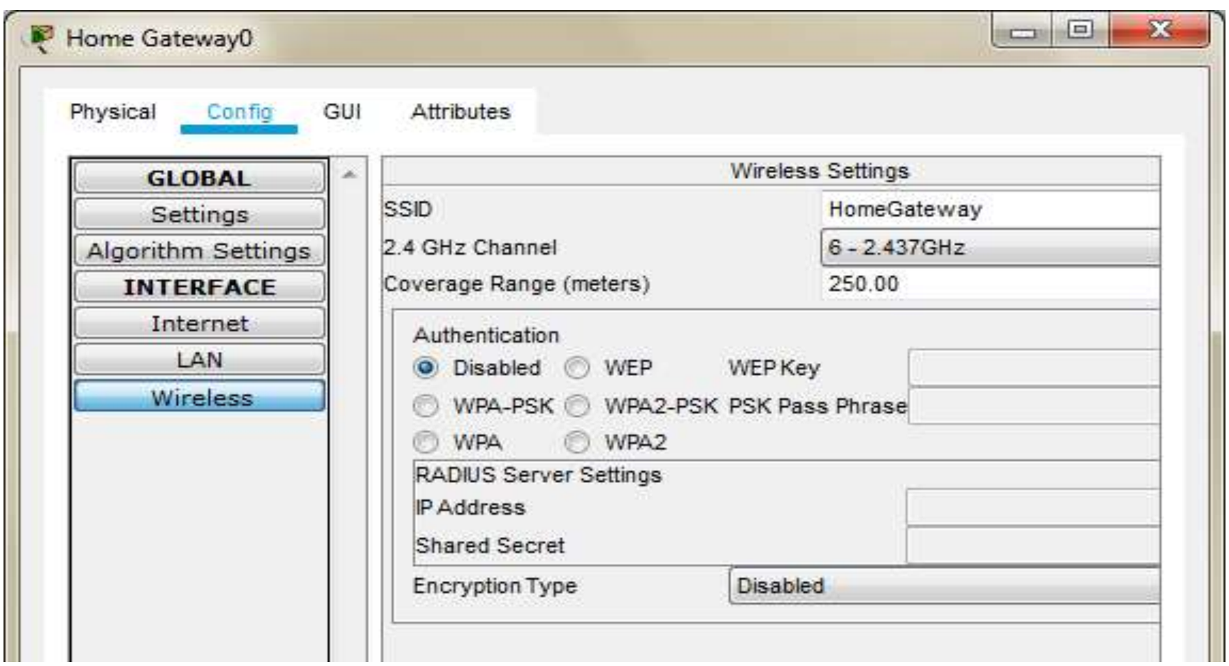
For the motion Detector sensor do the following



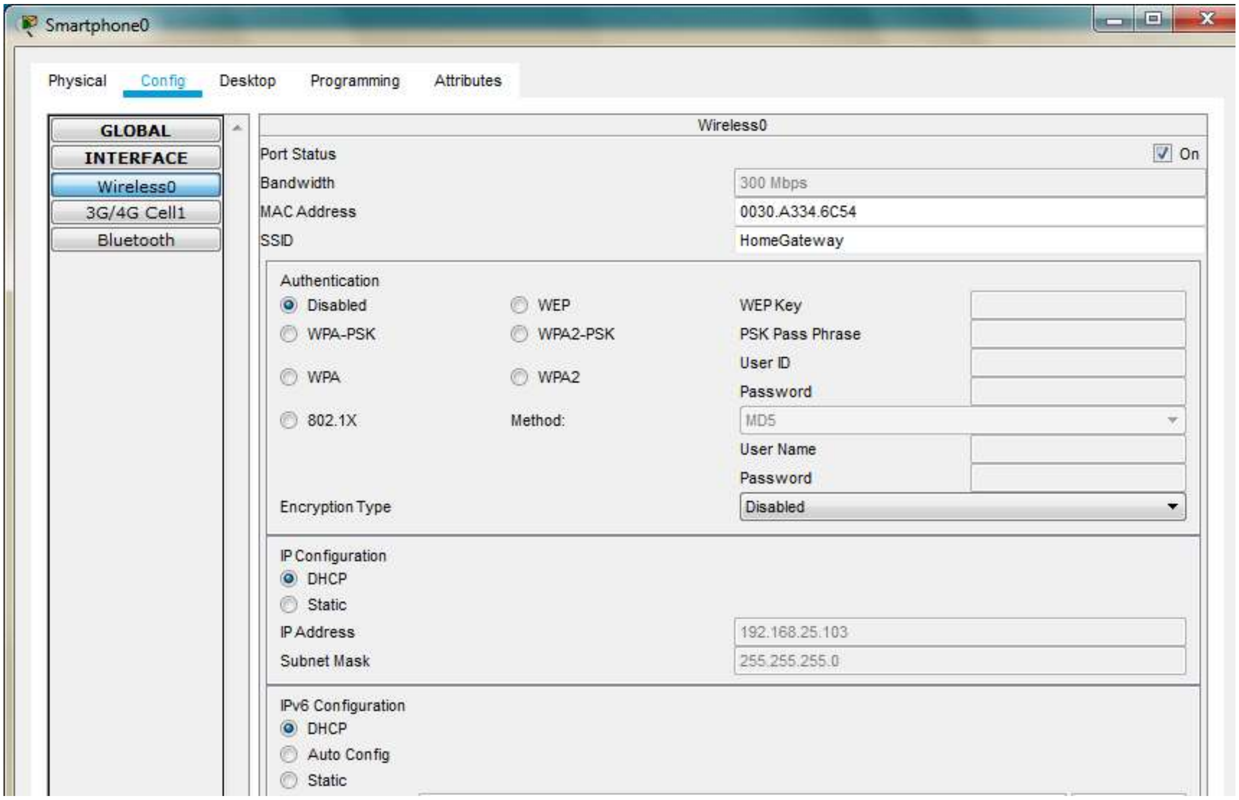
In the Advanced setting do the following for the Network adapter



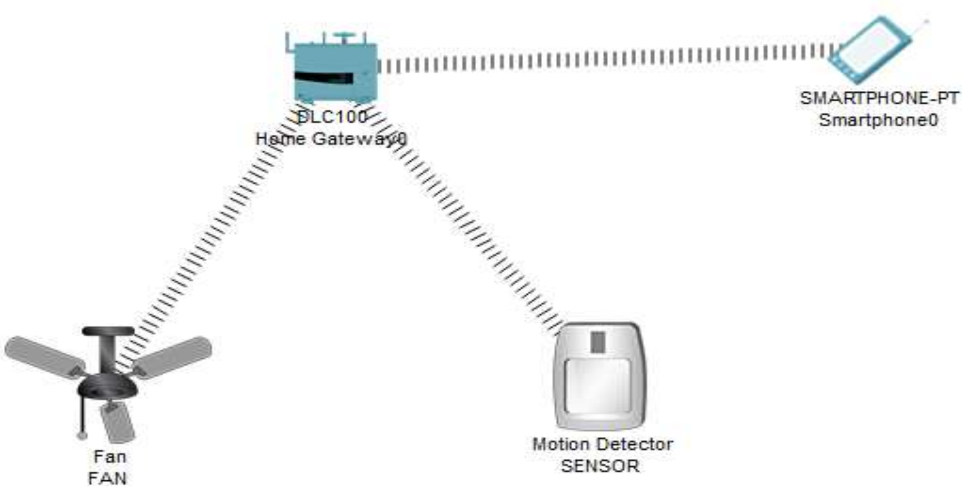
For the smartphone change the SSID to the SSID in the Home Gateway0



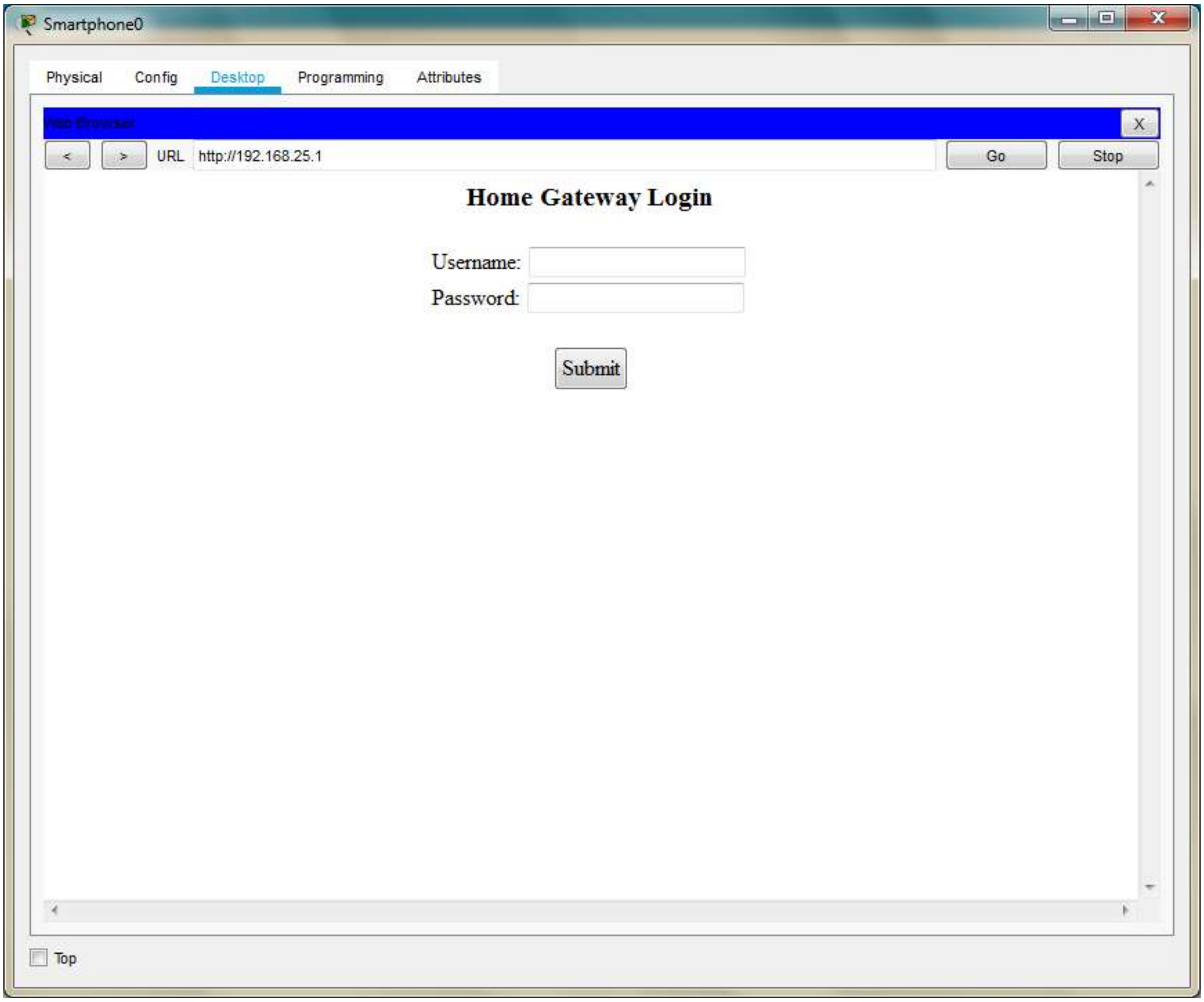
As seen above the SSID is HomeGateway, we use the same and set the SSID in the Smartphone



All the devices are now connected to the Home Gateway



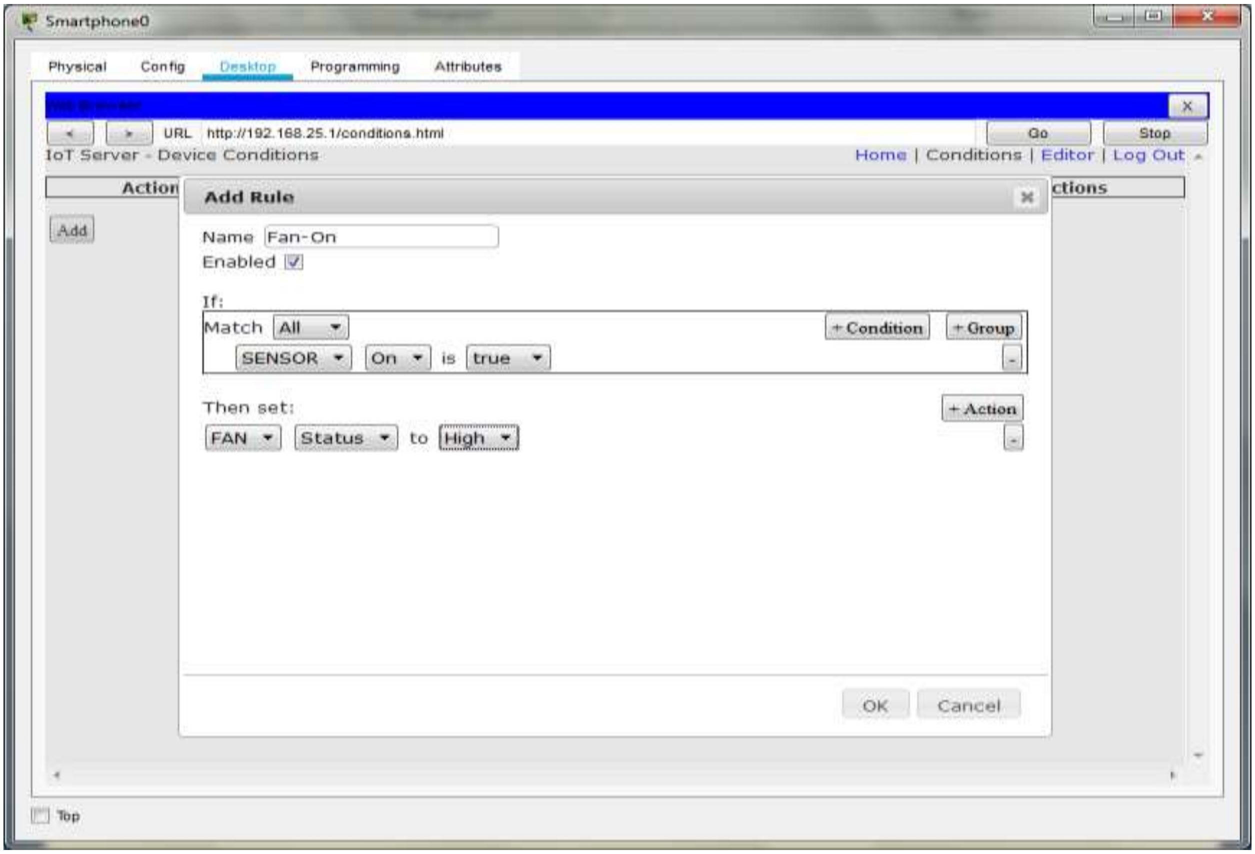
Now open the Web browser of the SmartPhone and type the IP address of the HomeGateway



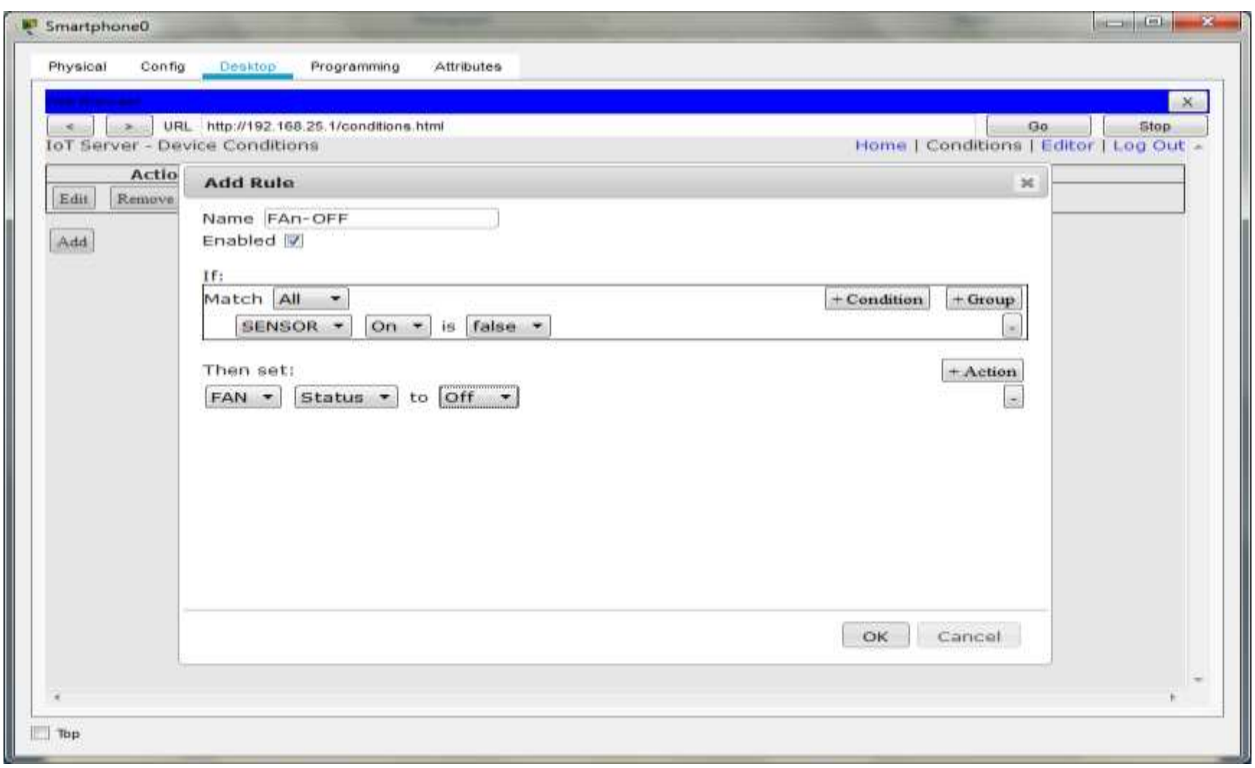
Username : admin

Password : admin

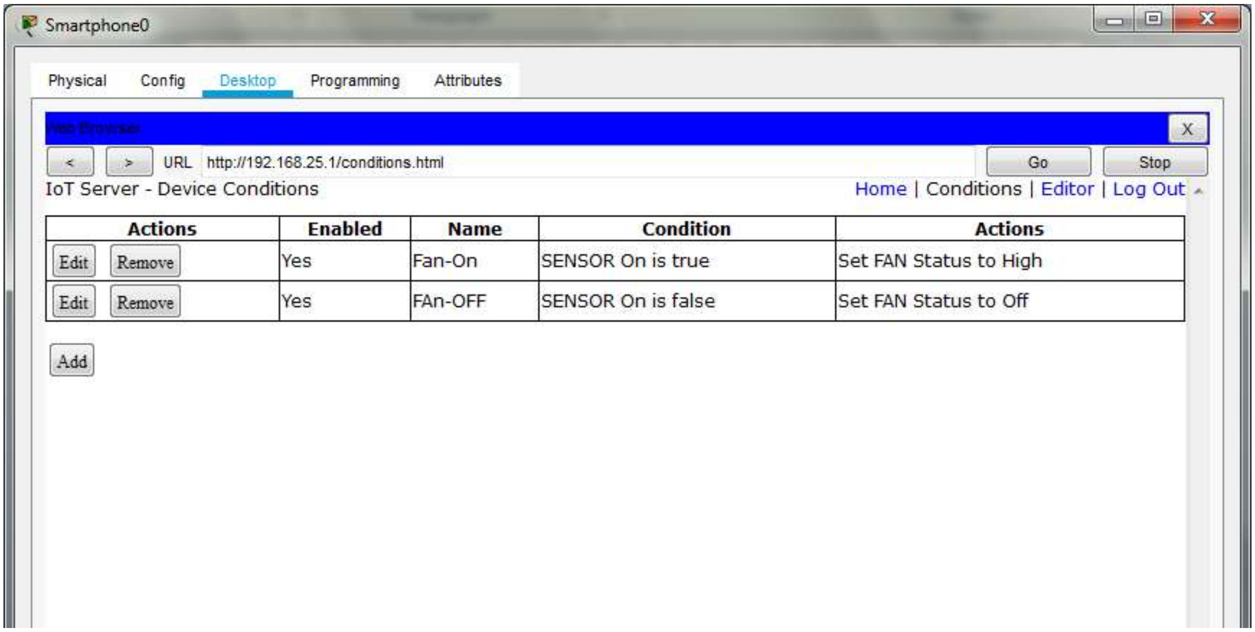
After logging click on conditions and do the following



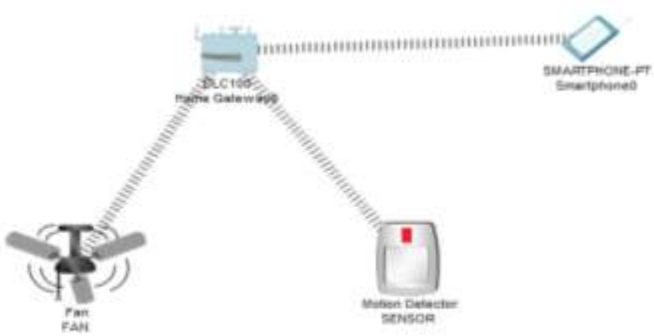
Add another condition as follows



Press the go button after adding the two conditions



In order to turn ON the fan Press the ALT key and left-click the mouse over the Sensor



Practical – 10

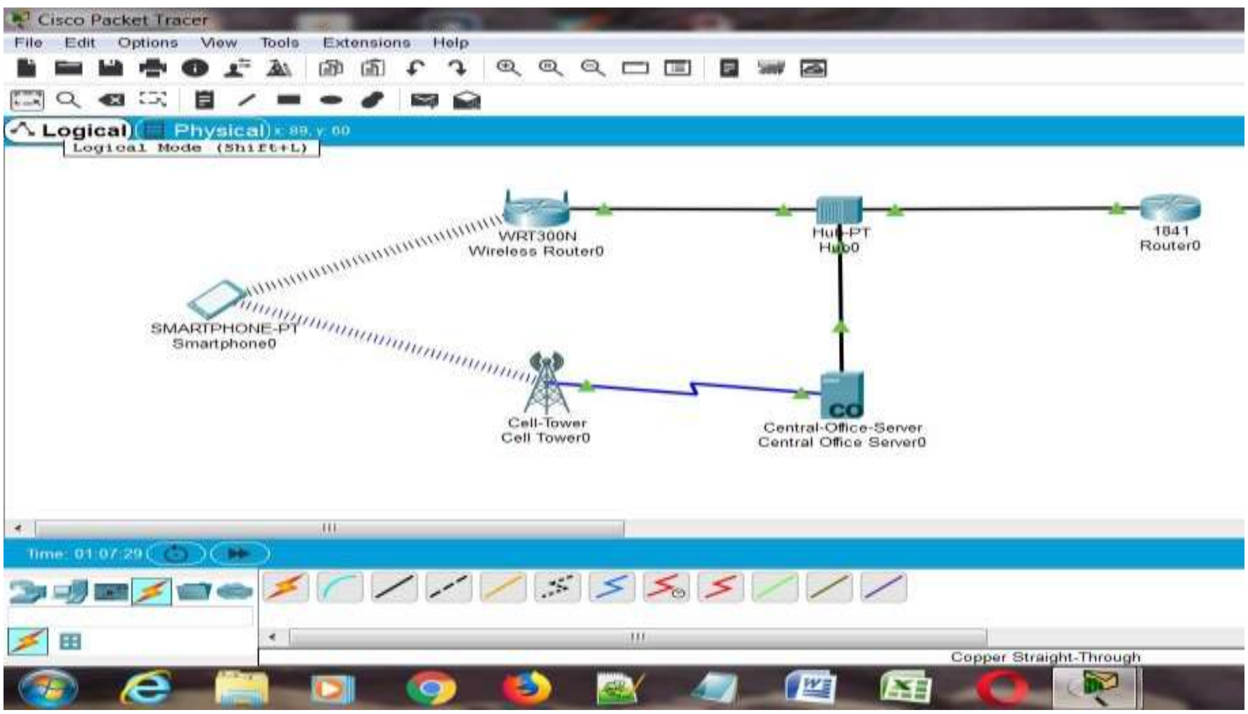
Aim: Create a mobile network using Cell Tower, Central Office Server, Web browser and Web Server. Simulate connection between them.

Steps:

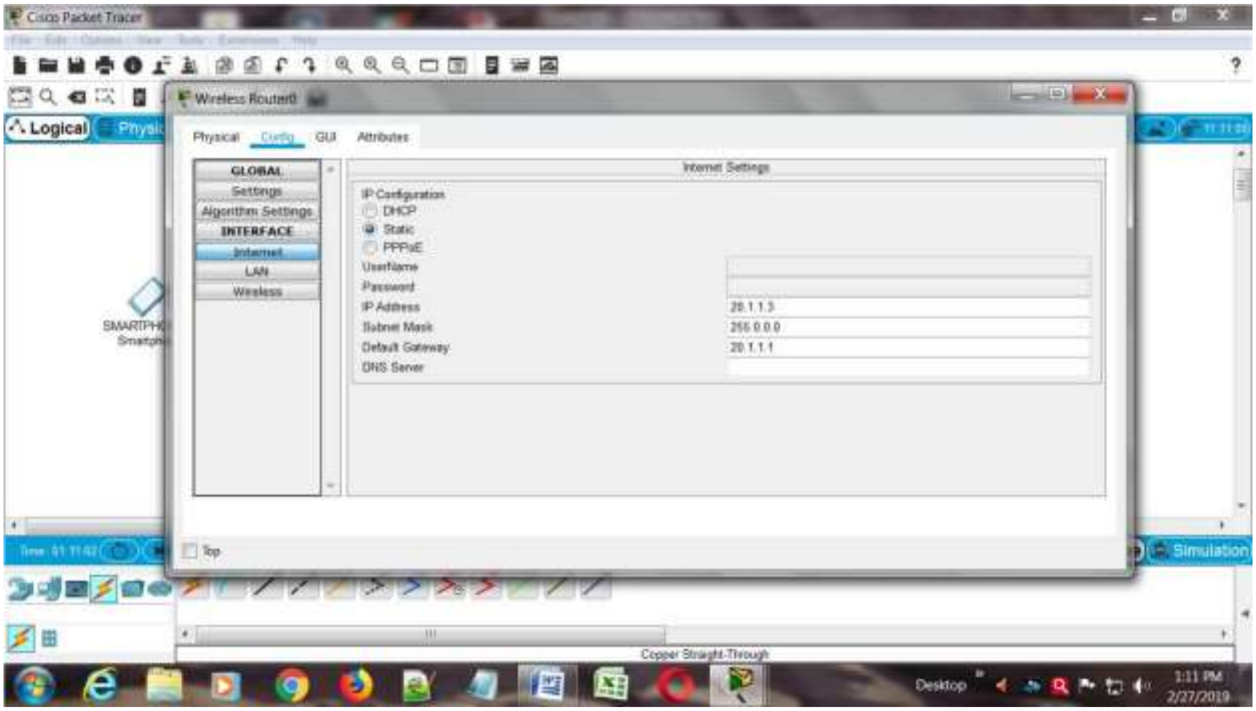
1. Create a network using smartphone, wireless router WRT300N, Hub-pt, 1841 Router, centraloffice-server, Cell-Tower.

2. Connect cell tower and central office server using coaxial cable.

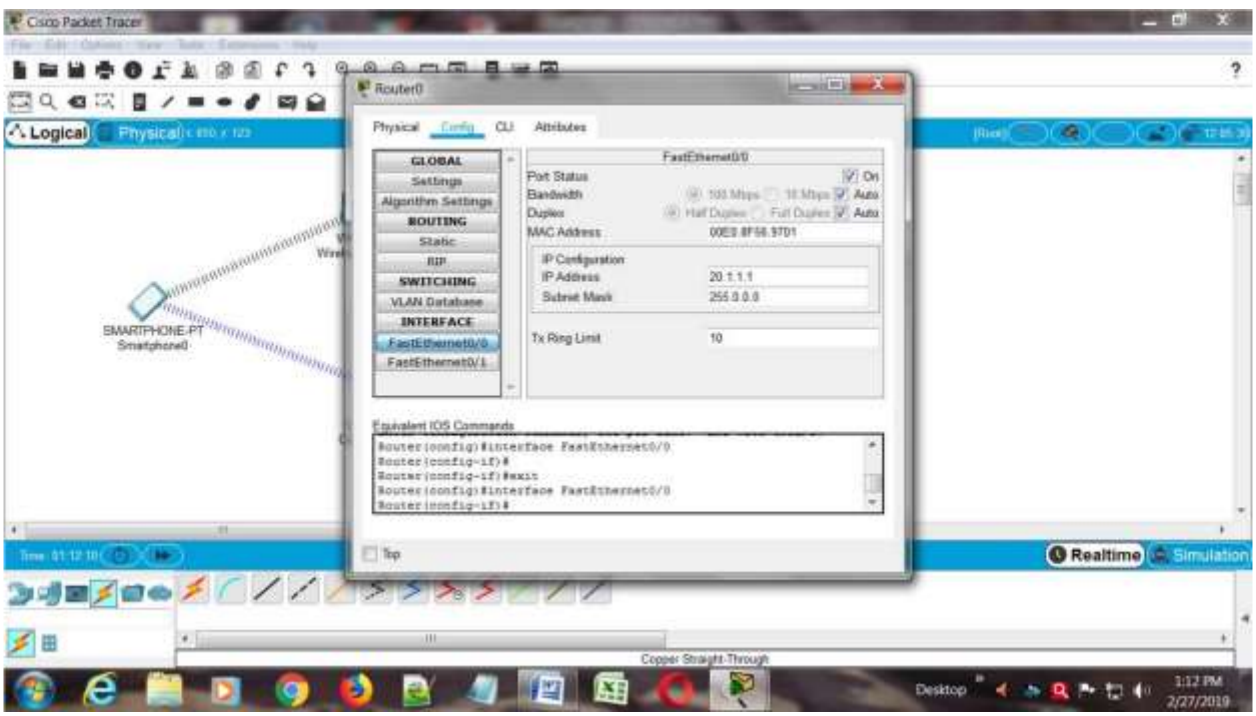
3. Connect wireless router WRT300N, Hub-pt, 1841 Router, central-office-server using copper straight through wire.



4. Click on wireless router.in config tab select internet.in internet choose ip configuration as static and set IP address and default gateway.



5. Click on router 1841. In config tab select interface and give IP address.



6. Click on smartphone and ping router1841