## 1. WRITING A TCL SCRIPT

As shown in Figure 1, we desire to simulate a simple network in which the source node (stationary source wired to the AP) is generating data with a Mobile WiMAX subscriber node as the destination.

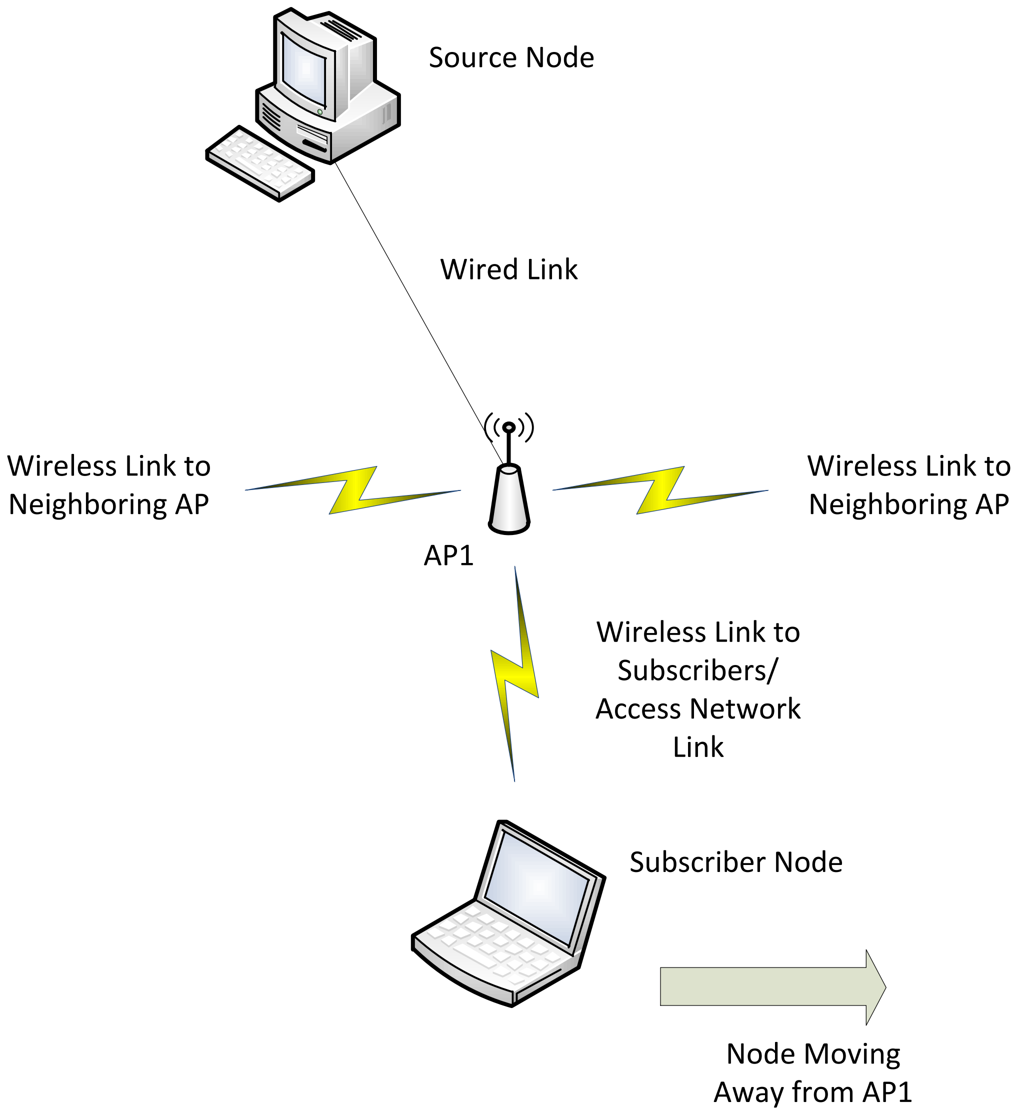

Source Node
Wired Link
Wireless Link to Neighboring AP
Wireless Link to Neighboring AP
AP1
Wireless Link to Subscribers/ Access Network Link
Subscriber Node
Node Moving Away from AP1

**Figure 1: The simulation scenario we will build**

Writing the tcl script for this scenario include the following steps

i) Define simulator.
Creating a simulator is essential for any NS-2 simulation. It is done using the following simple codes.

```
set simulator_name [new Simulator]
```

ii) Define topology.
In this step, define the area (in terms of size) under which the simulation occurs. WE must define it keeping in view the possible locations of our nodes.

```
set topography_name [new Topography]
$topography_name load_flatgrid size_x size_y
```

iii) Define output trace files.
Trace files record events that occur during any simulation like node creation and data transfer, among others. There are several trace formats and choosing one among them would depend on the results we are looking for and what the files log. A trace file is defined as follows.

```
set handler [open filename w]
$simulator_name trace_type $handler
```

Please see http://nsnam.isi.edu/nsnam/index.php/NS-2_Trace_Formats for information on information logged in trace files.

iv) Define the General Operations Director (“GOD”).
Quoted from CMU document, "GOD (General Operations Director) is the object that is used to store global information about the state of the environment, network or nodes that an omniscient observer would have, but that should not be made known to any participant in the simulation." In simple words, we need to define the GOD to manage the details of operations we supply such as the movement patterns in our simulations.

```
create-god number_of_nodes
```

v) Define access point/base station and configure the AP/BS.

To create and configure a base station or an access point in NS-2, it is preferable to define the options we want to configure the AP/BS with. The configuration is handled as follows.

```
$simulator_name ap-config   –option_1 value_1 \
                            –option_2 value_2 \
                            ……………………........
```

```
–option_n value_n
```

Please refer to documents provided for options and their possible values.
To create an AP/BS with the set configuration, we can now define it as follows.

```
set ap_name [$simulator_name ap]
```

Then we need to disable random motion for the AP and give it a location.

```
$ap_name random-motion 0
$ap_name set X_ x_location
$ap_name set Y_ y_location
$ap_name set Z_ 0
```

After you are done with creating the AP, switch off the configuration mode.

```
$simulator_name apConfig OFF
```

vi) Define wireless node, configure it and attach to AP/BS.

Creating and configuring a wireless node is very similar to what we needed to do with the AP. To set the configurable options, we use the following code.

```
$simulator_name node-config    –option_1 value_1 \
                               –option_2 value_2 \
                               ……………………........
                               –option_n value_n
```

To create a wireless node (in fact even a wired node):

```
set node_name [$simulator_name node]
```

Disable random motion and set location as we did earlier.
Finally we “attach” the wireless node to the AP.

```
$node_name base-station [AddrParams addr2id [$ap_name node-addr]]
```

vii) Define the wired source, configure it and connect it to the AP/BS.

The creation and configuration of the wired source is similar to that of the wireless node, except we do not “attach” it but rather “connect” it with a wired link.

```
$simulator_name duplex-link $node_name $ap_name bandwidth delay queue_type
```

Please refer to the accompanying documents for possible values for the options.

viii) Define traffic generator agent and attach it to the source and destination node.

To define the traffic flow, we need to designate the nodes as source or destination and decide the type of application/traffic.

The first step is to define the source agent.

```
set src_agent_name [new Agent/src_agent_type]
```

Then designate which node assumes the agent we defined.

```
$simulator_name attach-agent $src_node_name $src_agent_name
```

Define the traffic/application type and attach it to source agent.

```
set $traffic_name [new Application/Traffic/traffic_type]
$traffic_name attach-agent $src_agent_name
```

It should be noted that each traffic type comes with default values of traffic parameters like rate and packet size which can be easily overridden with the **set** command.

```
set traffic_parameter value
```

Now define the destination agent and attach it.

```
set dest_agent_name [new Agent/dest_agent_type].
$simulator_name attach-agent $dest_node_name $dest_agent_name
```

It must be noted that source and destination agents form a pair, and though many options are available for a type of agent, not all of them all compatible with each other.

Finally, we can connect the agents.

```
$simulator_name connect $src_agent_name $dest_agent_name
```

It is necessary to tell the simulator when to start and stop the traffic flow. To do so we schedule events that control the agent’s activities.

```
$simulator_name at start_time "$traffic_name start"
$simulator_name at stop_time "$traffic_name stop"
```

ix) Run simulator.

To order the simulator to execute, we simply need to provide the following statement in the script.

```
$simulator_name run
```

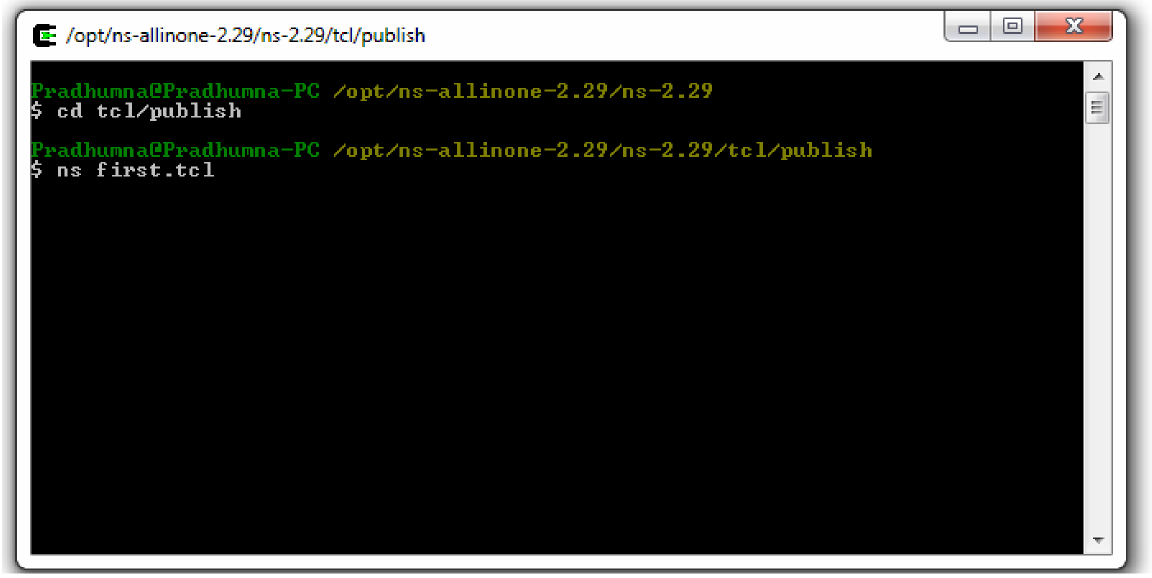
It is necessary to clear variables and close files after the simulation ends. It is customary to create a procedure for that.

```
$simulator_name at simulation_stop_time "proc_name"
```

```
proc proc_name { } {
                    global vaiable_list file_list
                    $simulator_name flush-trace
                    close $file_name
                    exit 0
          }
```

## 2. HOW TO RUN IT?

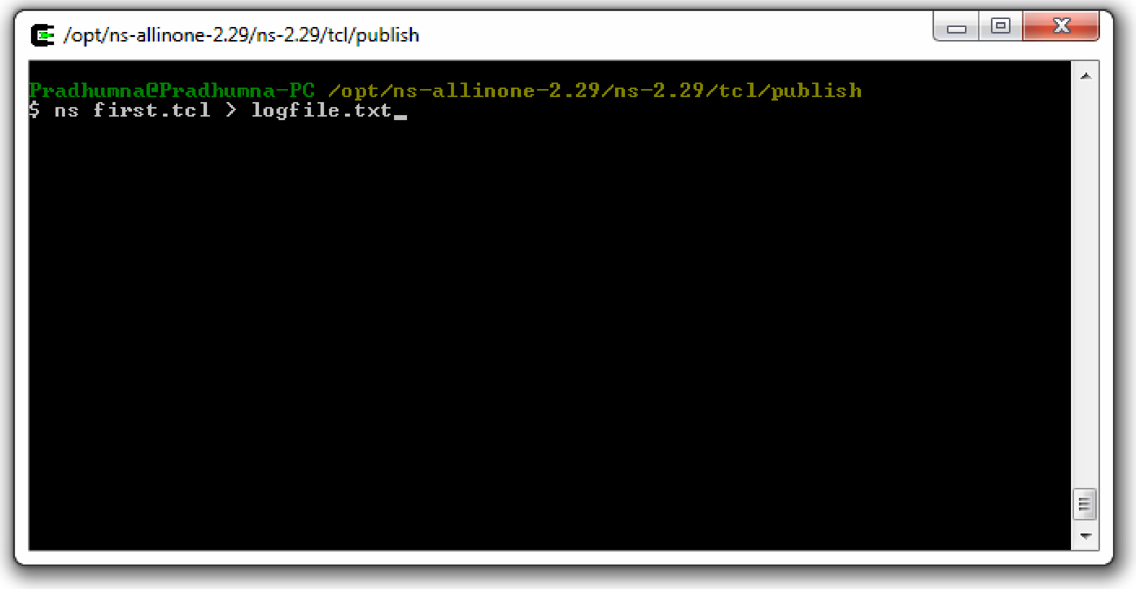
Running a TCL script is very simple. Because of the path we set during installation, NS2 application can be accessed from any subfolder on the cygwin bash shell. We first need to change our prompt to the current working subdirectory where the script resides and then run the script as shown in Figure 2.



```
Pradhumna@Pradhumna-PC /opt/ns-allinone-2.29/ns-2.29
$ cd tcl/publish

Pradhumna@Pradhumna-PC /opt/ns-allinone-2.29/ns-2.29/tcl/publish
$ ns first.tcl
```

**Figure 1: Running a TCL script**

Often we may desire to save messages produced by running the script. For that we simply have to redirect the messages to desired file as shown in Figure 3.



```
Pradhumna@Pradhumna-PC /opt/ns-allinone-2.29/ns-2.29/tcl/publish
$ ns first.tcl > logfile.txt_
```

**Figure 2: Storing the simulation log**

## 3. Using The Trace Files

As we mentioned earlier, NS2 simulation allows us to create trace files to log the results of our simulation. Depending on the size of simulation, these trace files can be very large, which makes it impossible to extract any meaningful information out of them manually. Hence we resort to using text processing tools. Since we are using a cygwin based system to simulate a UNIX-like environment on WINDOWS, we recommend the use of UNIX-based text processing tools like AWK.

## 4. AWK Programming

AWK programming is like any other high level programming. Since we have to only deal with reading text files and extracting relevant results, we can limit ourselves with learning simple features of the language like defining variables, reading files and displaying results.

Since there are different trace formats, the same AWK code will not work for all trace files, however the basic concept is the same. AWK identifies the strings separated by tabs and spaces on a single line in the text as a single unit and accordingly designates those numbers.
For example, the following is one line from one of our trace files.

```
r -t 11.160143120 -s 0 -d -1 -p cbr -e 1400 -c 2 -a 0 -i 550 -k MAC
```

Please refer to http://nsnam.isi.edu/nsnam/index.php/NS-2_Trace_Formats for information on the meaning of the entities.
Since AWK reads the text one line at time, we are only concerned with this single line and not with what follows or precedes it. There are 19 individual entities in this line, separated by space. Therefore AWK designates location values 1 through 19 for them. Syntactically, we have to use a '$' sign before the numbers. Hence, AWK identifies them as **$1=”r”, $2=”-t”, $3=11.160143120**, and so on. Please note how AWK is able to differentiate between strings and numbers. If we desire to count the number of **“r”** in our text file, then all we have to do is set up a counter which increments every time AWK finds an **“r”** in the first position of every line.

### *How to write AWK code?*

The syntax of writing AWK code is straightforward. The general way of writing the code is as follows.

```
BEGIN {         #starting block to define the variables
……………………
……………………
}

{                    #central processing block where all the calculations occur
……………………
……………………
}

END {           #finishing block to terminate the code and display results
……………………
……………………
}
```

Sometimes it becomes necessary to monitor an event after every line or simply keep track of an AWK variable. In such cases, it is possible to use the central block to display results.

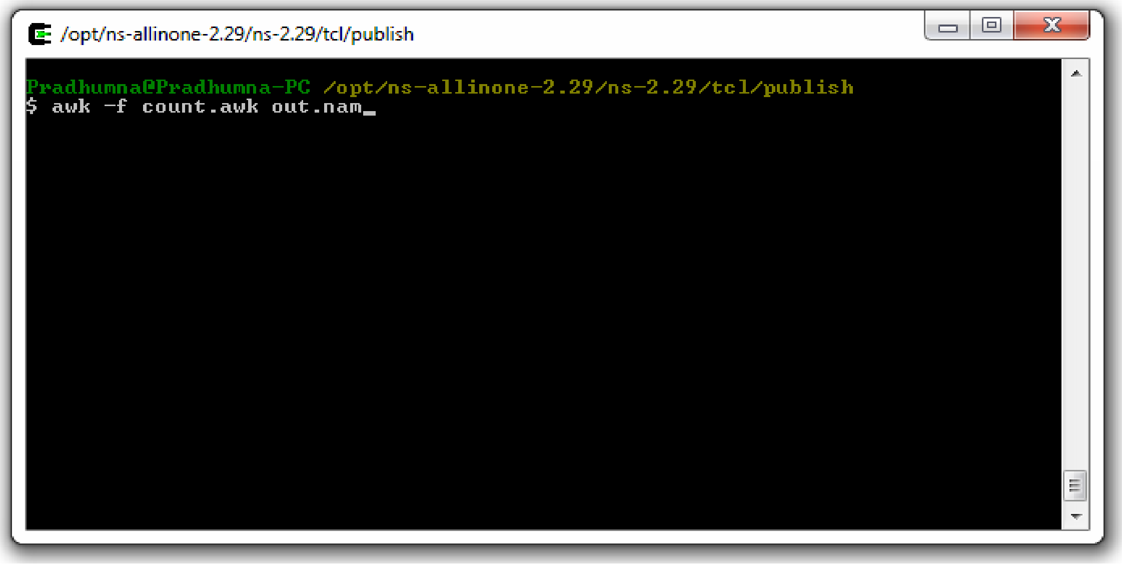
Let’s look at a simple example to illustrate the coding.

```
BEGIN {
        number_of_r=0
}

{
        if ($1==”r”) {
                number_of_r= number_of_r+1
        }
}

END {
        printf (“Number of r = %d \n”,number_of_r)
}
```

Please remember to save the script file with an “.awk” extension.

## *Executing an AWK script*

The AWK application can be accessed from any location in cygwin. However it is better to have the script and the trace files in the same subfolder. The script can be run as shown in Figure 4.

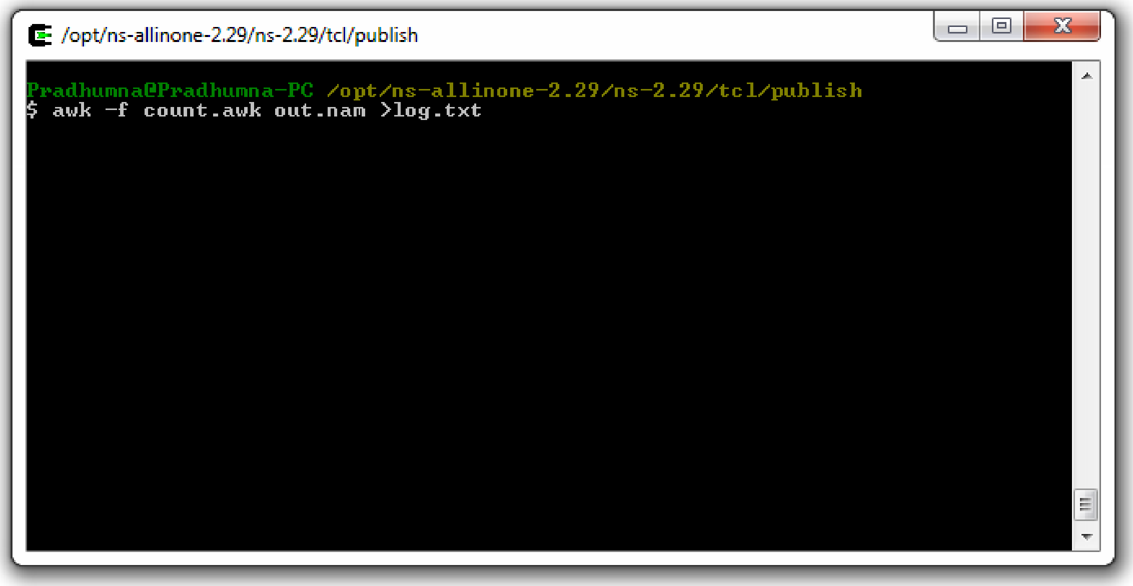


/opt/ns-allinone-2.29/ns-2.29/tcl/publish

```
Pradhumna@Pradhumna-PC /opt/ns-allinone-2.29/ns-2.29/tcl/publish
$ awk -f count.awk out.nam_
```

**Figure 3: Running an AWK script**

As with the case of TCL script, if we want the displayed results to be logged in a file for reference or easy access by graph plotting tools like EXCEL or MATLAB, we can redirect it to a text file.



/opt/ns-allinone-2.29/ns-2.29/tcl/publish

```
Pradhumna@Pradhumna-PC /opt/ns-allinone-2.29/ns-2.29/tcl/publish
$ awk -f count.awk out.nam >log.txt
```

**Figure 4: Storing text processing results**

### *Results*

Figure 6 shows the end-to-end throughput extracted from the output of simulation of 100 seconds. The users are encouraged to extract similar graphs using the accompanying scripts.

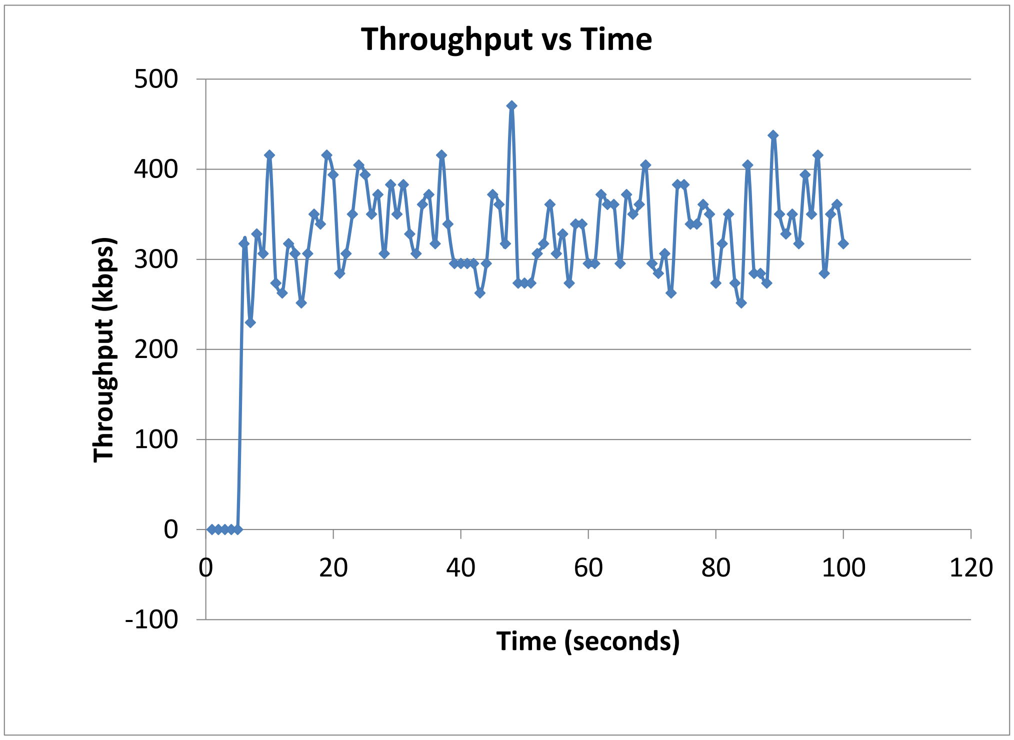


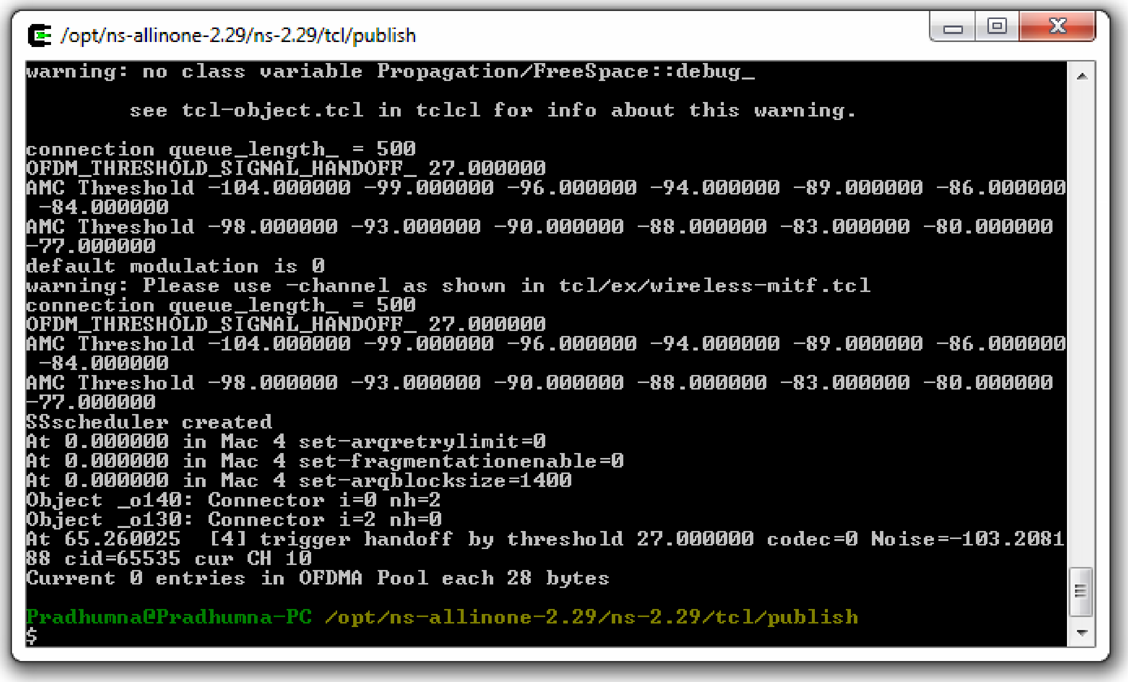
**Figure 5: End-to-end link throughput**

## 5. INTRODUCING MOBILITY

To be of any practical use, unlike in our previous example we want the mobile nodes to actually move. Fortunately, NS-2 makes it really simple to move mobile nodes. We just have to add one line in the script after creating the node.

**$***simulator_name* **at** *time* **"$***node_name* **setdest** *X_destination Y_destination speed***"**

In the accompanying sample script the node's destination is set farther away from the AP. Since the node is moving away, it exits the coverage area of the AP as soon as the SNR falls below a specified level. In fact it triggers a handoff, but in our case there is no other AP that the node can be handed over to and hence the connection is lost. This effect is also logged by the simulator as shown in Figure 7.



```
warning: no class variable Propagation/FreeSpace::debug_

        see tcl-object.tcl in tclcl for info about this warning.

connection queue_length_ = 500
OFDM_THRESHOLD_SIGNAL_HANDOFF_ 27.000000
AMC Threshold -104.000000 -99.000000 -96.000000 -94.000000 -89.000000 -86.000000
 -84.000000
AMC Threshold -98.000000 -93.000000 -90.000000 -88.000000 -83.000000 -80.000000
-77.000000
default modulation is 0
warning: Please use -channel as shown in tcl/ex/wireless-mitf.tcl
connection queue_length_ = 500
OFDM_THRESHOLD_SIGNAL_HANDOFF_ 27.000000
AMC Threshold -104.000000 -99.000000 -96.000000 -94.000000 -89.000000 -86.000000
 -84.000000
AMC Threshold -98.000000 -93.000000 -90.000000 -88.000000 -83.000000 -80.000000
-77.000000
SSscheduler created
At 0.000000 in Mac 4 set-arqretrylimit=0
At 0.000000 in Mac 4 set-fragmentationenable=0
At 0.000000 in Mac 4 set-arqblocksize=1400
Object _o140: Connector i=0 nh=2
Object _o130: Connector i=2 nh=0
At 65.260025  [4] trigger handoff by threshold 27.000000 codec=0 Noise=-103.2081
88 cid=65535 cur CH 10
Current 0 entries in OFDMA Pool each 28 bytes

Pradhumna@Pradhumna-PC /opt/ns-allinone-2.29/ns-2.29/tcl/publish
$
```

**Figure 7: Handoff being triggered due to node movement**

The effect of introducing mobility in the simulation on throughput is shown in Figure 8. It is clear that as soon as the connection drops, no throughput is available.

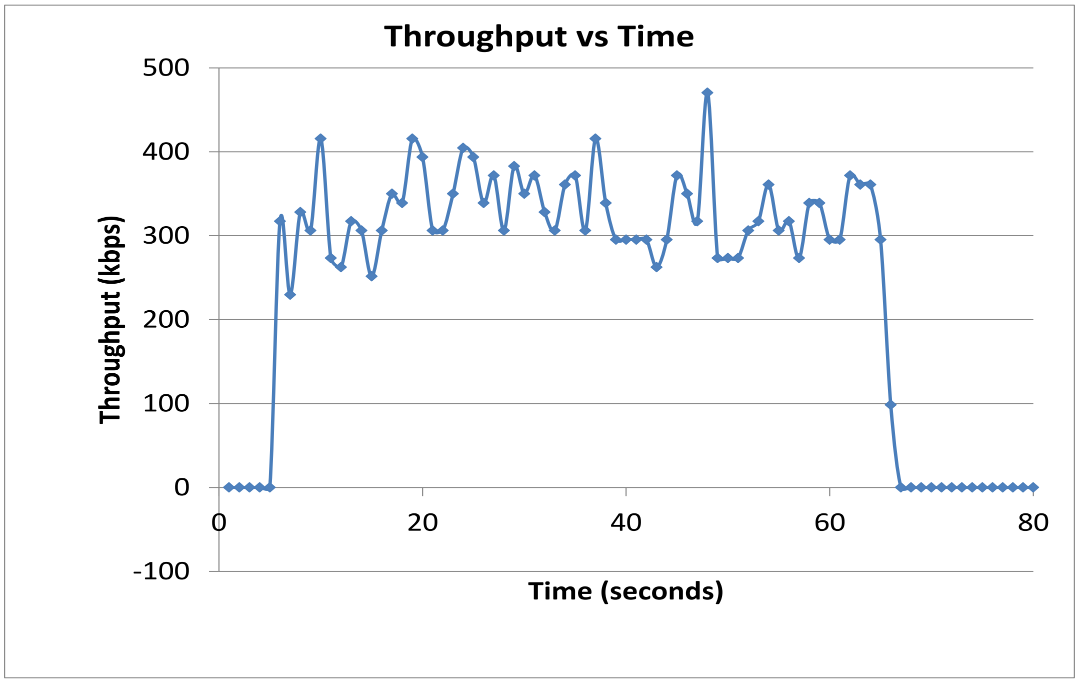


**Figure 8: End-to-end throughput**

## 6. PARAMETERIZING A TCL SCRIPT

Sometimes it becomes necessary to perform similar simulations under different scenarios and traffic conditions. The user may not be familiar with changing those parameters in the script, or the process may be too tedious, or even the script may not be available in editable format for end users. In such cases, it is possible to script TCL code with provisions of supplying the varying parameter values from the command line.

The format of adding this feature to a TCL script is straightforward and mostly independent from the rest of the code. There are three TCL keywords to be familiar with.

**argc**: number of arguments entered from prompt
**argv**: vector of arguments entered from prompt
**lindex**: integer pointing to the elements of **argv**

The code is written as follows.

```
if {$argc=1} {
        set parameter_1 [lindex  $argv 0]
        set parameter_2 default_value_2
        ……………………………………….
        ……………………………………….
        set parameter_n default_value_n

} elseif {$argc=2} {
        set parameter_1 [lindex  $argv 0]
        set parameter_2 [lindex  $argv 1]
        ……………………………………….
        ……………………………………….
        set parameter_n default_value_n
…………………………………………………
…………………………………………………
} elseif {$argc=n} {
        set parameter_1 [lindex  $argv 0]
        set parameter_2 [lindex  $argv 1]
        …………………………………….
        …………………………………….
        set parameter_n [lindex  $argv n]

} else {
        set parameter_1 default_value_1
        set parameter_2 default_value_2
        ……………………………………….
        ……………………………………….
        set parameter_n default_value_n
}
```

## 7. Running The Script

To provide the parameters from the prompt, we just have to enter the desired values.

*prompt>***ns** *script.tcl value_1 value_2 ………………value_n*

It should be taken care that the order in which the values are supplied should match the order as defined by the lindex pointer in the script and default values must be defined for each case, in case no arguments are provided from the prompt for those parameters. It should also be noted that these logical statements can be utilized to inform users about providing the parameter values. For example,

```
…………………………………………………………
…………………………………………………………
} elseif {$argc=1} {
puts “Wrong number of arguments. Default values will be used.”
…………………………………………………………
…………………………………………………………
```

This becomes useful when certain parameters have to be used in pairs.