IT 304 Computer Networks

Lab # 1: Introduction to Network Simulator NS2

Guidelines:

- **This is your first lab with NS2**
- **Majority of the content in this lab handout is based on the book: "Introduction to Network Simulator NS2" by T. Issariyakul and E. Hossain. Publisher: Springer.**
- **For a detailed class hierarchy both in OTcl and C++ refer to the NS2 manual available online at the NS web site.**

1. **Aim:** The purpose of this lab is to get you started with the open source network simulator NS2.

2. **Introduction to NS2:** NS2 is an *event-driven* open source simulation tool that has been extensively used for simulation in computer networks. NS2 is based on the REAL network simulator jointly developed by UC Berkeley and Cornell University. NS2 has benefited strongly from the contributions of various networking researchers and is still the tool of choice for networking simulations, despite the emergence of other proprietary simulation software like OPNET and QUALNET.

   As noted above NS2 is an event-driven simulator which means that the simulation is created and run by a set of events. Unlike in a time-driven simulator (in which the simulation events are executed as per the time of their occurrence), in an event-driven simulator the time gap between two events is not fixed.

3. **Architecture of NS2:** The basic architecture of NS2 is given in Fig.1. NS2 provides end users with an executable command `ns` which takes input argument as the name of a Tcl script file. In most cases the output of the command `ns` is a trace file that can be used to plot graph and/or to create animation.

   Working with NS2 entails usage of two languages namely, C++ and object oriented Tool Command Language (OTcl). C++ is used in the backend to implement the internal mechanism of simulation objects. OTcl on the other hand is used to set up a simulation scenario by configuring objects and also to schedule discrete events. The C++ and OTcl are linked using TclCL. Variables in the OTcl domain known as *handles* are mapped to C++ objects. The handle as such does not have any functionality, indeed the functionality is in the mapped C++ object. Th member procedures and variables in OTcl are known as instance procedures (instprocs) and instance variables (instvars) respectively. A brief tutorial on Tcl and OTcl is given below.

4. **Installation of NS2:** NS2 was developed in unix and therefore installing it on that platform and linux should be straightforward. It may not be installed directly on the native Windows platform, although can be installed on a Cygwin activated Windows platform.

   NS2 source codes are distributed in two forms: the all-in-one suite and the component wise. The beginners are recommended to install the all-in-one suite as it comes with all the required

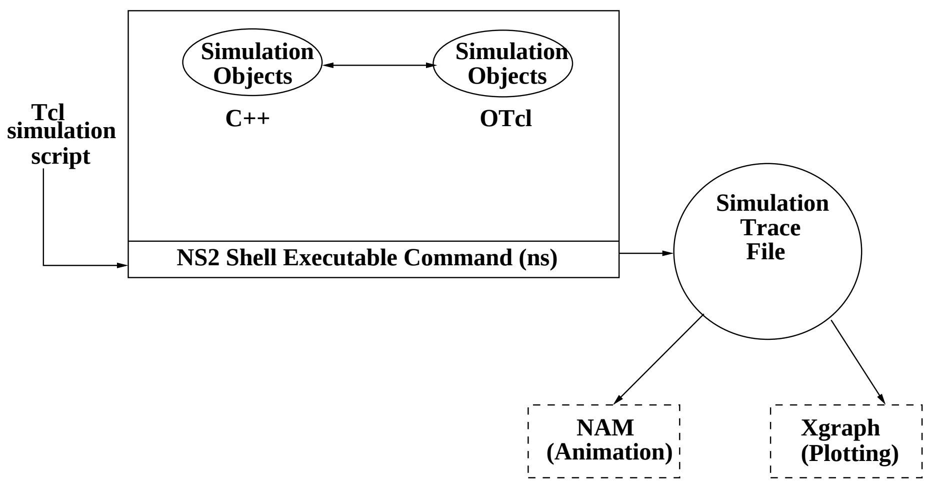

Figure 1: The figure above shows the architecture of the network simulator NS2.

components and some optional components. The current all-in-one suite (which is ns-2.35) comes with the following main components:

- NS release 2.35
- Tcl/Tk release 8.5.10
- OTcl release 1.15
- TclCL release 1.20.

and the following optional components

- NAM release 1.15
- Zlib version 1.2.3 (required for NAM).
- Xgraph version 12.2.

The following steps can be followed to download and install the all-in-one suite of NS on UNIX or Cygwin activated Windows systems:

(a) Download the corresponding .zip file from the NS web page.

(b) Unzip the corresponding .zip file.

(c) Install the all-in-one suite by ruunning the `install` script which can be done by typing the command (`./install`).

(d) Once the installation is successfully done follow the instructions to set the PATH variable and a set of environment variables in the `.bashrc` file.

(e) Once the immediate above step is done validate NS by running the `validate` script which can be done by typing the command (`./validate`).

Suppose that NS2 is installed in the directory `ns-allinone-2.35`. Fig.2 shows the directory structure under directory `ns-allinone-2.35`. Here, directory `ns-allinone-2.35` is on Level 1. On the Level 2 `tclcl-1.20` contains classes in TclCL (e.g., `Tcl`, `TclObject`, `TclClass`). All NS2

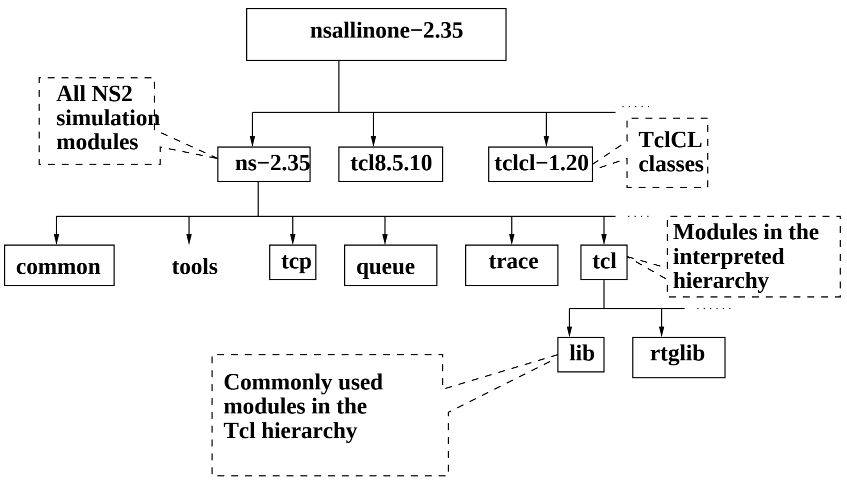
nsallinone−2.35
All NS2 simulation modules
ns−2.35
tcl8.5.10
tclcl−1.20
TclCL classes
common
tools
tcp
queue
trace
tcl
Modules in the interpreted hierarchy
lib
rtglib
Commonly used modules in the Tcl hierarchy

Figure 2: The figure above shows the directory structure of NS2.

simulation modules are in the directory `ns-2.35`. On Level 3 the modules in the OTcl hierarchy are under the directory `tcl`. Among these modules, the frequently used (e.g., `ns-lib.tcl`, `ns-node.tcl`, `ns-link.tcl`) ones are stored under directory `lib` on Level 4. Simulation modules in the C++ hierarchy are classified in directories on Level 2. For example directory `common` contains basic modules related to packet forwarding such as the simulator, the scheduler, connector, packet. Directory `tools` contains various helper classes such as random generators. Directories `queue`, `tcp`, `trace` contain modules for queue, TCP and tracing, respectively.

5. **Tutorial on Tcl:**

   (a) **Tcl script invocation:** Tcl script can be invoked from the shell command prompt with the following: syntax: `tclsh [<filename> <arg0> <arg1>...]`. If NS2 is installed the following invocation would also give the same result (since NS2 is written in Tcl): `ns convert.tcl`.

   (b) **Variable assignment and retrieval:** Tcl stores a variable using the key word "`set`". The value stored in a variable can be retrieved by placing the character "`$`" in front of a variable name. The key word "`puts`" is used to print out onto an output device. The key word "`unset`" is used to clear the value stored in a variable.

   (c) **Procedures:** A procedure is usually used in place of a series of Tcl statements to tidy up the program. The syntax for a procedure is given below:

   ```
   proc <name> {<arg_1> <arg_2> ··· , <arg_n>} {
   <actions>
   [return <returned_value>]
   }
   ```

   (d) **Bracketing:** There are four types of bracketing in Tcl. These are used to group a series of strings. Tcl interprets strings inside different types of brackets differently. Suppose a variable *var* stores a value 10. Tcl interprets a statement "`expr $var + 1` with four different types of bracketing differently as shown below.

   - Curly braces ({`expr $var + 1`}): Tcl interprets this statement as it is.

- Quotation marks (``` ``expr $var + 1'' ```): Tcl interpolates the variable `var` in the string. This statement would be interpreted as ``` ``expr 10 +1'' ```.
- Square brackets (`[expr $var + 1]`): Tcl regards a square bracket as the same way that C/C++ regards a parenthesis. It interprets the string in a square bracket before interpreting the entire line. This statement would be interpreted as "11".
- Parentheses (`(expr $var + 1)`): Tcl uses a parentheses for indexing an array and for invoking built-in mathematical function.

(e) **Data types:** In Tcl/OTcl there is no need to define data type of variables. Instead, it stores everything in string and interprets them based on the context. We demonstrate this by the following sample code (the keyword `expr` informs the Tcl interpreter to interpret the string following the keyword as a mathematical expression).

```
# vars.tcl
set a ``10+1''
set b ``5''
set c $a$b
set d [expr $a$b]
puts $c
puts $d
```

(Once the above script is run the output would be 10+15 and 25 respectively corresponding to the values stored in variables `c` and `d`.)

(f) **Example 1:** The above concepts are covered in the following example (the symbol "#" is used to introduce comments):

```
# temp_scale_conversion.tcl
# The program converts temperature from Fahrenheit to Celsius scale
proc tempconv  {
set lower 0
set upper 140
set step 25
set fahr $lower
while {fahr < $upper} {
set celsius [expr 5*($fahr - 32)/9]
puts ``Fahrenheit /Celsius:  $fahr / $celsius"
set fahr [expr $fahr + $step]
}
```

(g) **Global Variables:** Global variables are common and and used extensively throughout a program. These variables can be called upon by any procedure in the program as demonstrated below:

```
set PI 3.1415926536
proc perimeter {radius} {
global PI
expr 2*$PI*$radius
}
```

(In the above code, the keyword `global` is used here to make "`PI`" global.)

(h) **Array:** An array is a special variable used to store a collection of items. An array stores both the indices and the values as strings. The example below and the explanation following it will demonstrate this.

```
# Numeric indexing
set arr(0) 1
set arr(1) 3
set arr(2) 5
```

(In the above code index "`0`" is not a number, but a numeric string.)

(i) **Lists:** A list is an ordered collection of elements such as numbers, strings or even lists themselves. The key list manipulations are given below:

List creation: A list can be created in three ways as shown below:

```
list mylist 1 2 3
set mylist ``1 2 3''
set mylist {1 2 3}
```

Member retrieval: The following command returns the `n`th element in the list `mylist`:

```
lindex $mylist $n
```

Member setting: The following command sets the value of the `n`th element in the list `mylist` to `<value>`:

```
lset $mylist $n $value
```

Group retrieval: The following command returns a list of whose members are the `n`th member through the `m`th of the list `mylist`:

```
lrange $mylist $n $value
```

Appending the list: The following command attaches a list `alist` to the end of the list `mylist`:

```
lappend $mylist $alist
```

(j) **Input/Output** Tcl employs a so-called *Tcl channel* to receive an input using a command `gets` or to send an output using a command `puts`. A Tcl channel refers to an interface used to interact to the outside world. Two types of Tcl channels include standard reading/writing and file channels. The former are classified into `stdin` for reading, `stdout` for writing, and `stderr` for error reporting. The latter needs to be attached to a file before it is usable. The syntax for attaching a file to a Tcl file channel is shown below:

```
open <filename> [access]
```

(The command returns a Tcl channel attached to a file with the name `<filename>`. The optional input argument `<access>` could be "`r`" for reading, "`w`" for writing and "`a`" for appending an existing file.)

A Tcl channel can be closed by using the command `close` whose syntax is as follows: `close <channel>`.

The commands `gets` and `puts` reads and writes, respectively a message to a specified Tcl channel. The syntax for them is given below:

```
gets <channel> <var>
```

`puts [-nonewline] <channel> <string>` (the nonewline option specifies not to write an end-of-line character to the end of the string. The command "`puts`" does not output immediately onto a Tcl channel. Instead, it puts the input argument (i.e., string) in its buffer and releases the stored string either when the buffer is full or when the channel is full. To force immediate outputting, `flush` is used.)

For an example in file i/o refer to the Tcl code given below to crate a network simulation scenario.

(k) **Mathematical expressions:** Tcl implements mathematical expressions through mathematical operators and mathematical functions. A mathematical expression of either type must be preceded by a keyword "`expr`". Otherwise, Tcl will recognize the operator as a character. Examples are given below:

```
expr log10(10)
expr 1+2
```

(l) **Control structures:** The syntax for `if/else/elseif`, `for` and `while` is given below:

```
if {<condition>} {
<actions_1>
} elseif {<condition2>} {
<actions_2>
}
.
.
.
else {
<actions_n>
}
while {<condition>} {
<actions>
}
for {<init>} {<condition>} {<mod>} {
<actions>
}
```

6. **Tutorial on OTcl:** OTcl is an object oriented version of Tcl, just like C++ is an object oriented version of C. The basic architecture and syntax in OTcl are much the same as in Tcl. However, in Otcl, the concepts of classes and objects are of great importance. The main concepts are summarized below:

   Class and Inheritance: In OTcl, a class cn be declared using the following syntax: `Class <classname> [-superclass <superclassname>]`. If the optional argument in the square bracket is present, OTcl will recognize class `<classname>` as a child class of class `<superclassname>`. Alternatively, if the option is absent, class `<classname>` can also be also declared as a child class of class `<superclassname>` by executing `<classname> superclass <superclassname>`. Note that, class ¡classname¿ inherits the functionalities of class ¡superclassname¿. In Otcl, the top-level class is class `Object`, which provides basic procedures and variables, from which every user-defined class inherits.

   Class Member Procedures and Variables: A class can be associated with procedures and variables. In OTcl, a procedure and a variable associated with a class are referred to as instance procedure (i.e., instproc) and an instance variable (i.e., instvar), respectively. The syntax for instance procedures and instance variables are given below:

   Instance Procedures:

   ```
   <classname> instproc <procname> [{args}] {
   <body>
   ```

}

(where instproc "`instproc`" is defined in the top-level class `Object`. Here, the name of the instproc is `<procname>`. The detail (i.e., `<body>`) of the instproc is embraced within curly braces. The input arguments of the instproc are given in `<args>`. Each input argument is separated by a white space. Once declared, an instproc is usually invoked through an object (of class `<classname>`) using the following syntax: `<object> <procname> [{args}].`)

Instance Variables:

`$self instvar <varname1> [<varname2>,···]`

Unlike instprocs, instvars are not declared with the class name. Instead, they can be declared anywhere in the file as can be seen above. In the above declaration instproc "`instvar`" and an instvar "`$self`" (which represents the object itself) are defined in the top-level class `Object`.

After the declaration, an instvar can be manipulated by using a command `set` with the following syntax: `<object> set <varname> [<value>]`.

Object construction and the Constructor: An object can be created from a declared class by using the following syntax: `<classname> <objectname>`. In the object construction process, instprocs `alloc` and `init` of class `Object` is invoked to initialize the object. Usually, referred as a constructor, instproc `init` defines necessary object initialization. This instproc is usually overridden by the derived classes.

7. **Sample code and Exercises**

   (a) Consider the following simulation code that was used to simulate the network scenario in which there are 3 nodes, say `n0, n1 and n2` respectively. The nodes `n0 and n1` are connected by a duplex link, while `n1 and n2` are connected by two simplex links. All the queues in the nodes adopt droptail queuing mechanism. Further the node `n0` is connected with a UDP source which is in turn connected to a CBR traffic generator. The queue occupancy at various observation instants are written into a trace file, which can then be plotted using the `xgraph` utility.

```
#Create a simulator
set ns [new Simulator]
#Open the Trace file
set file1 [open out.tr w]
#Define a 'finish' procedure
proc finish {} {
global ns file1
close $file1
exit 0
}
#Create three nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
#Create links between the nodes
#Create a duplex link between n0 and n1
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
```

```
#Create 2 simplex links in either direction between n1 and n2
$ns simplex-link $n1 $n2 0.1Mb 10ms DropTail
$ns simplex-link $n2 $n1 0.1Mb 10ms DropTail
#Set Queue Size of link (n1-n2) to 10
$ns queue-limit $n1 $n2 10
#Define queue monitor object
set queue_mon [$ns monitor-queue $n1 $n2 [open queue.tr w]]
set queue_pint [$queue_mon get-pkts-integrator]
#Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n0 $udp
set null [new Agent/Null]
$ns attach-agent $n2 $null
$ns connect $udp $null
$udp set fid_ 1
#Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 1mb
$cbr set random_ false
#Define a procedure which periodically records different parameters to the
file with file handler ‘‘file1’’
proc record { oldqsize olddrops oldarrivals } {
global ns file1 queue_mon queue_pint
#Get the current time
set now [$ns now]
#Set the time after which the procedure should be called again
set time 1
# Queue monitor parameters:  totals
set qsize_ave [$queue_pint set sum_]
set drops_tot [$queue_mon set pdrops_]
set arrivals_tot [$queue_mon set parrivals_]
# Same thing but referring to a record interval this time, for the "instantaneous"
values
set qsize [expr $qsize_ave - $oldqsize]
set drops [expr $drops_tot - $olddrops]
set arrivals [expr $arrivals_tot - $oldarrivals]
#The following line writes the no.  of packets in the queue at a given observation
instant
puts $file1 "$now $qsize"
#Re-schedule the procedure
$ns at [expr $now+$time] "record $qsize_ave $drops_tot $arrivals_tot"
}
# start the CBR traffic at 0.1 sec
$ns at 0.1 "$cbr start"
# stop the CBR traffic at 124.5 sec
$ns at 124.5 "$cbr stop"
# start calling the record function at 0.2 sec
$ns at 0.2 "record 0 0 0"
# stop the simulation at 125 sec
$ns at 125.0 "finish"
$ns run
```

(b) **Exercises:**(a) Try to modify the given code to include a random traffic source instead of a CBR source. (b) Configure a TCP source instead of an UDP source given in the above sample code. Output the TCP congestion window size into the trace file and plot the window size versus simulation time using the `xgraph` utility (you need to figure out as how can TCP window size values be accessed).