Network Simulator “ns”

Chadi BARAKAT

INRIA Sophia Antipolis, France
PLANETE research group

Email: Chadi.Barakat@sophia.inria.fr
WEB: http://www.inria.fr/planete/chadi

Outline

- Introduction to “ ns ”.
- Structure of the simulator: C++ code, OTCL interface, tools.
- Programming in “ ns ”: The OTCL language.
- Exploiting the results of a simulation.
- Extending the capacity of “ ns ”.

Introduction

- A simulator for communication networks, developed at Laurence Berkeley National Laboratory (LBNL) within the VINT project. It fast became a property of the entire research community, where everyone can add its own modules, and contributes to the development of the simulator.
- Download your free copy at:

  http://www.isi.edu/nsnam/ns/

- Can be installed on Unix, Linux, and Windows.
- The first version of " ns " was experimental.
- Now, we are working with the second version called " ns-2 ".

Introduction (ctd)

- “ns” is mainly designed for Internet protocols, particularly the TCP protocol. However, its good hierarchical organization has motivated many researchers to use it for the study of:
  - New protocols they propose for the different layers of the Internet (routing, transport, application).
  - The impact of new transmission media on Internet protocols (ATM, satellites, Wired LANs, Wireless LANs).
  - The performance of new architectures proposed to improve the QoS in the Internet (DiffServ, IntServ, buffer management).

- The different contributions to “ns” can be found on the web site of the simulator, and by searching the mailing list archive.

Structure of the simulator

- “ ns ” is an object-oriented simulator. A simulation is no other than the motion of packets, which are objects, among the different objects representing the elements and protocols of the network.

- The core of “ ns ” is written in C++. This eases the addition of new protocols and mechanisms. The main classes are:
  - Application: The parent class of all applications (e.g., ftp, telnet).
  - Agent: The parent class of all protocols that run at layers 3 and 4 of the Internet, e.g., TCP, UDP, TFRC, RTP, RIP, OSPF, SRM, DVMRP, PIM.
  - Node: Represents the set of nodes in the network. A node can be a host, a switch, a router, or a gateway. Each node contains a Classifier which decides on where to send a packet coming from an interface or from an agent. The packet is to be sent either to an agent attached to the node, or to an outgoing interface.

Structure of the simulator (ctd)

- Queue: The parent class of all buffers, e.g., Drop Tail, Drop From Front, RED.
- LinkDelay: This class simulates the propagation delay and the transmission time over links of the network. Together with the class Queue, this class simulates the layers 1 and 2 of the Internet + the buffer management at the IP layer.
- Packet: The class of packets that propagate through the network. This class points towards two classes, the first one for the Header and the second one for the Payload. To define a new protocol, one has to define a Header class.
- TimerHandler: The parent class for all timers used by network protocols. When the timer of a protocol expires, a particular function of the class representing the protocol is invoked.

Structure of the simulator (ctd)

- Every object of a simulation (except the receiving application) posses the handler of an object to which the packet has to be forwarded on its way to the destination, e.g., the LinkDelay object gives the packet to the input interface of the node at the output of the link.
- To give a packet to an object, we have to call the function recv of this object using the handler we detain. We pass to this function a pointer to the packet. The pointer to a packet passes from one object to another until it reaches its destination.

What is then a simulation?

- Define the objects of the simulation (and set their parameters).
- Connect the objects to each other (topology of the network).
- Start the source applications. Packets are then created and are transmitted through the network.
- Exit the simulator after a certain fixed time, or when a particular event happens (e.g., end of the transfer).

The scheduler

- ❑ The core of the simulator. Its role is the scheduling of events. " ns " is called an event-driven simulator.

  Event = An action to be done after a certain time in the future, e.g., retransmit a packet when a timer expires, give a packet to the node at the output of a link after a time equal to the propagation delay.

- ❑ How the scheduler works ?
  - At the beginning of the simulation, the user schedules a certain number of events to be executed during the simulation lifetime, e.g., start of an application, end of the simulation.
  - The objects of the simulation schedule other events.
  - All the events are placed in one queue by the order of their due time.

The scheduler (ctd)

- The scheduler comes to this queue, dequeues the event at the head of the queue, advances the time of the simulation (by the time elapsed since the last event), executes the event, then dequeues another event, and so on, until the event exit is found. Here, the simulation stops.
- The time in " ns " does not correspond to the real time. There is no relation between the duration of the simulation and the time it takes to end. The time a simulation takes depends on how many events are executed during it.
- A packet in " ns " is itself an event. We ask the scheduler to hand the packet to an object after a certain time. The scheduler ensures that a packet is not queued more than one time in the queue of events.

The OTCL interface

- The configuration of the simulation, and the parameters of the different objects we want to create, are passed to " ns " in a file written in the OTCL language:

```
> ns configuration.tcl
```

  OTCL is an object-oriented programming language as C++.
- " ns " disposes of an OTCL interpreter which translates the commands written in OTCL to their equivalence in C++.
- When " ns " is called, the OTCL interpreter creates an image of the C++ class hierarchy at the OTCL level. The user writes the configuration file as if the simulator is written entirely in OTCL.
- This translating of the C++ class hierarchy allows the user to create instances of the classes of the simulator, and to define new classes and functions, if he wants to extend the simulator.

Accompanying tools

- " nam ": A tool that allows to visualize the motion of packets through the nodes and links of the network.
- " xgraph ": A tool that allows to plot the results of the simulation in the form of curves.
- " gt-item ": A tool that allows to create large graphs and to control their hierarchy.
- " sgb2ns ": A tool that translates the graphs created by " gt-item " to the OTCL language, so that they can be integrated in the configuration file.

Programming in “ ns ”

- To program in “ ns ”, all what we need is:
  - Knowledge of the OTCL language, in order to write the configuration file.
  - Knowledge of the OTCL class hierarchy.
  - Knowledge of the different methods and variables of the OTCL classes.
  - No need to know the C++ class hierarchy. The OTCL interpreter does the job for us !
- When the user creates an instance of an OTCL class, the OTCL interpreter creates an instance of its corresponding C++ class. The same thing applies to methods and variables.
- The function “ main ” of the simulator is very simple:
  - Run the OTCL interpreter.
  - Run the scheduler.

Basic instructions in OTCL

- Assign a value to a variable: `set a 10`
- Read the content of a variable: `set b $a`
- Open a file in mode write: `set f [open file w]`
- Write the content of a variable in a file: `puts $f "$a"`
- Array of variables: `set tab($index) 0 ($index = 1,2,3,… )`
- Arithmetic operations: `set c [expr 2.0 * $a / $b]`
- Control structures:
  - `if {$a == $b}  { … }`
  - `for {set i 1} {$i <= 10} {set i [expr $i + 1] } { … }`
- Pass parameters to the simulator: `set a [lindex $argv 0]`
  - On the command line, write: `> ns Configuration.tcl value-of-a`

Basic instructions in OTCL

- Functions:

```
proc function-name {par1 par2 … } {
global a b …
…
return [expr a + $par1]    }
```

  To call the function: `set c [function-name $par1 $par2 …]`
- Create an instance of a class: `set obj [new Class1/Class2/Class3]`
- Call a method of an object without return: `$obj method-name $par1…`
- And with return: `set a [$obj method-name $par1 …]`
- Assign a value to a variable of an object: `$obj set var-name $a`
- Read the value assigned to a variable of an object:

```
set a [$obj set var-name]
```

- Free an object: `delete $obj`

Create a scenario

- The first thing to do is to create an instance of the class Simulator, which contains a set of methods necessary for the programming:

```
set ns [new Simulator]
```

- Second comes the creation of nodes (use arrays in case of a network topology with large number of nodes):

```
for {set i 1} {$i <= $NodeNb} {set i [expr $i+1]}{
    set n($i) [$ns node]  }
```

- Then comes the creation of (duplex) links between nodes:

```
$ns duplex-link $n(1) $n(2) bandwidth delay type-of-buffer
```

  - `bandwidth = 10Mb, 100Kb, 10KB, 1MB, etc.`
  - `delay = 10ms, 1s, etc.`
  - `type-of-buffer = DropTail, RED, etc.`

- We can set the buffer size at the input of a link:

```
$ns queue-limit $n(1) $n(2) buffer-size-in-packets
```

Create a scenario

- Now, it is time to create transport agents, to attach them to (peer) nodes, and to connect them to each other (equivalent to connection establishment procedure). Here is an example of a TCP connection:

```
set transp1 [new Agent/TCP]              //TCP source
$ns attach-agent $n(1) $transp1
set transp2 [new Agent/TCPSink]          //TCP destination
$ns attach-agent $n(2) $transp2
$ns connect $transp1 $transp2           //establish the connection
```

- For a UDP agent, we still need to connect the source to the destination. This is necessary to set the destination address in UDP packets:

```
set transp1 [new Agent/UDP]              //UDP source
set transp2 [new Agent/Null]             //UDP destination
$ns connect $transp1 $transp2           //connect agents
```

Create a scenario

- We create then the application that generates data. The application needs to be attached to the source agent. No need to attach an application to the destination agent. For an FTP application:

```
set app [new Application/FTP]
$app attach-agent $transp1
```

- Finally, we precise the starting time of the application, the end time of the simulation, and we launch the simulator:

```
$ns at 10.0 "$app start"    //send an infinite amount of data
$ns at 10.0 "$app send 100" //send 100 bytes of data
$ns at simulation-time "exit 0"
$ns run
```

- Save everything in configuration.tcl and call " ns ".

Routing in “ns”

- By default, “ns” routes packets via a Shortest Path Tree. The tree is computed one time at the beginning of the simulation.
- There is the possibility to make the routing protocol dynamic, i.e. it adapts to any change in the status of links during the simulation.
- Routes can also be defined manually:

```
$ns rtproto Manual    //Set the routing to manual
set DestID [$D id]    //The ID of the destination node
set Interface [[$ns link $R1 $R2] head]     //The interface of
//the router R1 to which we want to route all packets sent to D
$R1 add-route $DestID $Interface
```

  - Define all routes in this way.

Some classes and variables

- TCP:

```
set tcp [new Agent/TCP(TCP/Reno)(TCP/Newreno)(TCP/Sack1)]
set sink [new Agent/TCPSink(TCPSink/DelAck)(TCPSink/Sack1)]
$tcp set window_ size-in-packets
$tcp set ssthresh_ size-in-packets
$tcp set packetSize_ size-in-bytes
```

- RED:

```
$ns duplex-link $n(1) $n(2) bandwidth delay RED
set q [[$ns link $n(1) $n(2)] queue]
$q set limit_ size-in-packets
$q set thresh_ min-threshold-in-packets
$q set maxthresh_ maximum-threshold-in-packets
$q set q_weight_ averaging-weight
$q set linterm_ inverse-of-pmax
```

Some classes and variables (ctd)

- CBR source:

```
set cbr [new Application/Traffic/CBR]
$cbr set packet_size_ size-in-bytes
$cbr set rate_ XKB (Kb) (MB) (Mb)
```

- Poisson source:

```
set poisson [new Application/Traffic/Poisson]
$poisson set interval average-time-between-packets-in-s
$poisson set size packet-size-in-bytes
```

- Uniform random variable:

```
set var [new RandomVariable/Uniform]
$var set min_ left-edge
$var set max_ right-edge
puts "$var value"       //print on the screen a random number
```

- The entire list of classes and variables can be obtained from the manual, the C++ code, or the file ns-default.tcl (default values).

Extract results from a simulation

- At any moment in the simulation, we can read the value of a variable of an object, and write it on the screen, or in a file. Here is an example on how to plot the congestion window of a TCP connection versus time:

```
set WindowVsTime [open WindowVsTime w]

proc plotWindow {tcpSource file} {
  global ns
  set time 1              //the window is read every second
  set wnd [$tcp window]
  set now [$ns now]
  puts $file "$now $wnd"
  $ns at [expr $now+$time] "plotWindow $tcpSource $file” }

$ns at 0 "plotWindow $tcp $WindowVsTime"
```

Extract results from a simulation

- In case we want to extract results about packets, the class Trace of the simulator is needed. Objects of this class can be associated to the elements of the network (for example to buffers), and give us the instants of arrivals/departures/drops of packets. This is an example of lines written by an object of type Trace:

```
- 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
r 1.84609 0 2 cbr 210 ------- 0 0.0 3.1 225 610
+ 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
d 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
```

- With a simple function, we can associate Trace objects to all elements of the network, and get information about all packets:

```
set file [open out.tr w]
$ns trace-all $file
```

Extract results from a simulation

- We can also ask the simulator to associate a Trace object to a particular buffer: `$ns trace-queue $n(1) $n(2) $file`
- Queue monitor: Very useful objects to get statistics about the motion of packets through a particular buffer of the simulated topology. Among others, we can at any moment read the number of packets (bytes) that have arrived to the buffer, the number of packets (bytes) that have left, the number of packets that have been dropped, etc.

  ```
  set monitor [$ns queue-monitor $n(1) $n(2) stdout]
  puts $file "[$monitor set pdepartures_ (barrivals_) (pdrops_)]"
  ```
- Flow monitor: Very useful objects to get statistics about the motion of packets of a particular flow through a buffer of the topology. This class uses the colors that we can associate to flows: `$tcp set fid_ 1`

Extract results from a simulation: Flow monitor

- Define first the Flow Monitor that filters packets based on their Flow ID, then associate it to the desired link and get a pointer to its classifier:

```
set flowmon [$ns makeflowmon Fid]
set L [$ns link $R1 $R2]
$ns attach-fmon $L $flowmon
set fcl [$flowmon classifier]
```

- When statistics on a flow X are required, probe the classifier for packets of that flow, then read statistics on that flow as with Queue Monitor:

```
set flowstats [$fcl lookup auto 0 0 X]
puts "[$flowstats set pdepartures_ (barrivals_) (pdrops_) ...]"
```

- `lookup` results in an error if no packets have not been seen from Flow X.

Extension of the simulator

- ❑ Two methods exist:
  - Work with the C++ code, by modifying the existing classes and methods, or by adding new classes (to implement a new protocol for example). This method requires a knowledge of C++, and it is the most efficient since most of the classes of " ns " are defined at the C++ level.
  - Work with the OTCL code, by modifying the existing classes and methods, or by adding new classes. This method requires a knowledge of OTCL. It is less efficient than the first method, since we are not able to access from the OTCL level to many functions and variables defined at the C++ level.
  - The choice of one of the two methods finally depends on the skills of the programmer.
- ❑ A good knowledge of C++ and OTCL is a guarantee for a good understanding and managing of " ns " !!