INTRODUCTION TO THE USAGE OF NETWORK SIMULATOR

(measurement guide)

Department of Telecommunication and Mediainformatics

Compiled by Felicián Németh (nemethf@tmit.bme.hu)
Revisioned by András Császár (andras.csaszar@tmit.bme.hu)
20 October, 2003.

1. Introduction

Network Simulator is a discrete event-controlled, packet-levelled network simulator, which is supported by numerous universities and research institutes; its resource code is freely available[1]. There is a possibility among others for examination of route selector, multicast or a TCP protocol even in cases of wireless networks with the help of this program.

The *ns* was written in C++ and otcl language. Kernel of the simulator, implementation of protocols, and tasks requiring a large calculation were realized in C++ language, but the otcl script language is used for giving the simulation configuration, and for solving simple tasks, which do not require a great efficiency.

Object hierachy of the system is thoroughly deliberated, new functionality can be fitted to the system easily in a C++ environment, but we will not deal with it in this preliminary measurement. So the aim of the measurement is to foreshow the basics of the otcl surface, and to support the learning of some simple network simulation testing methods.

The ns program can actually be considered, as a tcl interpreter with an object-oriented expansion, from which the class hierarchy supporting the network simulation is available.

As a first step of the simulation, we have to prepare an (o)tcl script, which defines the network components, their connection, and timing of the previously given events (e.g. beginning of the transmission of a data source). After that, we have to call the ns program with the script („ns program.tcl”), which generates the recording in the output files, and the simulation results. We can analyse the simulation results during the additional processing with the help of awk, perl, gnuplot, excel, etc. programs. Fort he visual analysis of results we can call the nam (Network Animator) program with a recording file with a special format, which demonstrates graphically the way of the individual packets.

---

[1] See also: http://www.isi.edu/nsnam

2.Tcl basics

Although a debate flared up over time, whether the Tcl language is efficient enough, as a script language expansion of C programs[2], or not, we can say, that it has achieved its purpose in the ns system: it provides a fast development with simple, clear linguistic elements. We can read in this chapter a brief description of the language, with no aim to be comprehensive, which provides nevertheless enough knowledge to the execution of measuring tasks.

Commands of a Tcl script are divided from each other with a new row, or a semicolon. Commands consist of one or several words, where the first word is the name of the command, and words, that may follow it, are the arguments of the command. Words are divided from each other with spaces, or with tabulator signs.

By evaluating a command, the interpreter divides the command, as a first step, into words, and executes the incidental replacements, and then executes the command with the given arguments. The `$variantname` is replaced with the value of the variant, while applying the `[command]`, the return value of the given command will be replaced. Variants have no types, they all stored as a string.

We can read a more detailed description from the language in the Tcl/Tk measurement guide[3] of our Department, which is necessary and expected to know at the beginning of the measurement.

There are some basic Tcl linguistic elements in the script calculating the 10! value:

```
1: set fakt 1
2: for {set i I} {$i <= 10} {incr i} {
3:   set fakt [expr $fakt*$i]
4: }
5: puts $fakt
```

Program 1. Tcl script calculating the 10 factorial

We set the value of fakt variant with the help of the set command to 1 in the first row. In the second row, in the cycle similar to the C language the curly brackets prevent the replacement, while the command is divided into words, thus, for example, by evaluating the kernel of the cycle variants will take always their actual value. The third row demonstrates an example for the command replacement with rectangular brackets, where the expr command calculates a numeric expression, and its value will be the new value of the fakt variant, after the set assignment. Finally, display of the result will be executed by the puts command in the last row.

We can see an example for the declaration of functions from the recoursive modification of calculation of the factorial (see the 2. program).

---

[2] See also: “The TCL war” http://www.vanderburg.org/Tcl/war

[3] See also: http://alpha.ttt.bme.hu/pub/meresek/tcltk4_eng.pdf

```
0: proc faktorialis num {
1:   if {$num > 0} {
2:   return [expr $num* [faktorialis [expr $num-1]]]
3:   } else {
4:   return 1
5:   }
6:   }
7: puts [faktorialis 10]
```

Program 2. A Tcl command calculating factorial

3. Otcl basics

This chapter demonstrates a simple example for the object-oriented expansion of the Tcl object, and for the usage of the otcl language. An average ns user seldom needs to write a new object, but it is necessary to understand the elements of the otcl language, since ns objects can be available from the otcl by preapreing simulation scripts.

```
0: Class mom
1: mom instproc greet {} {
2:   $self instvar age_
3:   puts „$age_year old mom say: How are you doing?”
4: }

5: Class kid –superclass mom
6: kid instproc greet {} {
7:   $self instvar age_
8:   puts „$age_year old kid say: What’s up, dude?”
9: }
10: set a [new mom]
11: $a set age_ 45
12: set b [new kid]
13: $b seg age_ 15
14: $a greet
15: $b greet
```

Program 3. An Otcl example

The classic example at the homepage of ns can be read in the 3. program.

This example program defines in the 0. row class mom with keyword Class, and then it derives class kid in the 5. row with the help of keyword –superclass. After the class definitions the next step will be the definition of methods, using keyword instproc. In the 2. and 7. line role of $self is the same as role of the „this” pointer in C++ language by defining the member functions. Method named instvar declares on one hand the age_attribute (member variable) (if it was not declared in the class, or in one of the parent classes), and on the other hand, it provides a possibility for referring to the attribute only with a variable name. Finally, we can instantiate a class with the new command (10. and 12. line), and methods are called in the 14. and 15. row, which would be as follows:

```
45 year old mom say: How are you doing?
15 year old kid say: What's up, dude?
```

4. Simulation example program

This chapter presents a simple simulation script line by line, demonstrating basic steps necessary for the preparation of a simulation.

A network to be simulated, as it can be seen on Figure 1, consists of four nodes (n0, n1, n2, n3). Capacity of the duplex link between n0 and n2, and between n1 and n2 is 2 mbps, and its delay is 10 milliseconds. Capacity of the third duplex link between n2 and n3 is 1,7 mbps, and its delay is 20 milliseconds. Every node uses a DropTail line with the capacity of 10 packets, which serves the waiting packets according to the FIFO principle, and in case of saturation of the puffer proper for storing the 10 packets it throws the newly arrived packets.

TCP agent connected to the n0 node generates a connection with the TCP sink agent connected to the n3 node. TCP agent – according to the default – is able to generate only packets of one kilobyte. It is the task of the sink to send ACK answering messages.

UDP agent connected to the n1 node communicates with the zero agent in n3. In contrast to the TCP sink, the zero agent simply frees the memory assigned to the packet, since in case of UDP it is not necessary to send a receipt. An ftp traffic generator is connected to the tcp agent, while the cbr traffic generator is connected to the udp agent, and this latter one generates a constant bit rate data flow: it sends 1 kilobyte packets with rate of 1 Mbps. Cbr generates traffic between 0,1s and 4,5 s, and ftp starts transmission in time of 1,0s and finishes the tranmission in time of 4,0 s.

This simulation arrangement mentioned above is realized by the Program 4. Explanations assigned to the more instructive momentums of the program with marking of the proper sequence number can be read below.

- 0: `set ns [new simulator]:` it instantiaties the ns simulator object and stores the object reference in the ns variable. It is the beginning line of all the ns scripts, that initiates the discrete time scheduler among others. The `Simulator` class is able to execute the following tasks through its methods:

  - generation of network objects (nodes, links…),
  - attaching network objects (e.g. `attach-agent`),
  - setting parameters of network elements,
  - defines data connection between agents (e.g. between `tcp` and `sink`)
  - control of displaying parameters of NAM.

- 1-2: `$ns color` *`fid color`*: it sets the color of the packets of the process determined by the `fid` process identifier. This method of the `Simulator` object has an effect only to the NAM display, and not to the real simulation.
- 4: `$ns namtrace-all` *`file-descriptor`*: this method switches on the saving of the simulation tracing data in a NAM input format. The `trace-all` method is similar to it, but it uses a general format (which is easier to be processed).

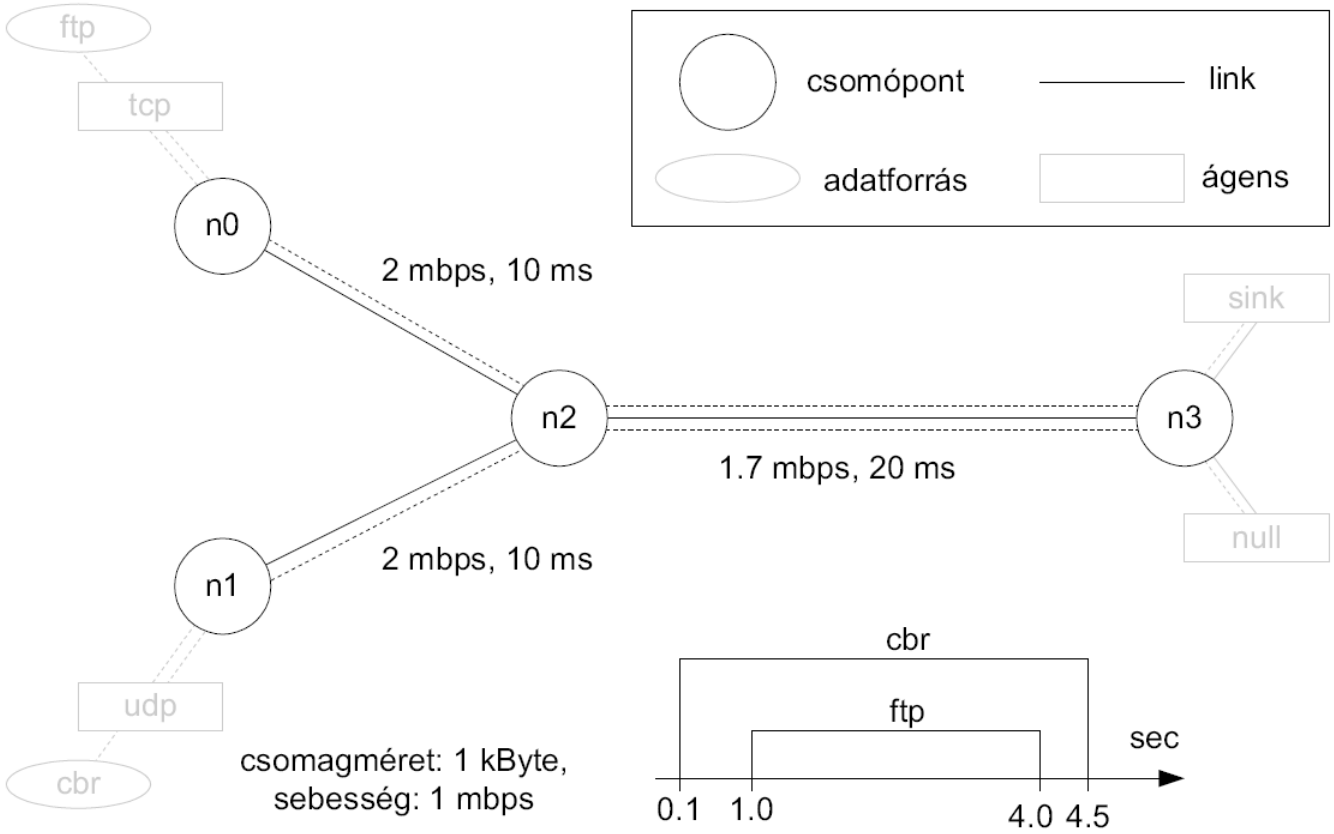

Figure 1. Network arrangement of the simple simulation

- 5: `proc finish {}`: command is called at the end of the simulation by the `$ns at 5.0 „finish"` instruction of the 52. line. After closing the register file, running of the simulation script will be closed with calling the NAM program.
- 12-15: `set n0 [$ns node]`: This method provides a possibility to the generation of a network node.
- 16-18: `$ns duplex-link` *`node1 node2 capacity delaying line type`*: it generates two simplex links with the given bandwidth and delay values, and connets the two given nodes. It is the main feature of `ns`, that the output line of the node is realized as a part of the link, so we should give the type of the ouptput line by generating the link. If we would like to use the `RED` type line using the active line management instead of the `DropTail`, we have to give simple that type, as the last parameter of the method. It is important to know, that there is a possibility for modelling even links with losses. (See further details about the useable line types and links with losses in the documentation of `ns`.)
- 20-23: commands have an important role only by displaying with the NAM program, it is worth to observe changes after commenting out the lines.

This part above defines the network topology, so the next task will be the setting of traffic agents (TCP, UDP) and data resources (FTP, CBR), and connecting the agents to nodes and resources.

- 24, 27, 34, 36: `set tcp [new agent/TCP]`:we can generate a TCP agent this way, and we can generate any of the agents or traffic generators similarly to this method, we only need to know the name of the class. It can be found in the documentation of NS, however, another possibility is to read the „ns-2/tcl/libs/ns-

defaul.tcl" file. This file contains default values of network objects, and configuration parameters, thus we can learn, what kind of simulation parameters exist, and which are the parameters of objects to be set.

- 26, 28, 35, 37: `$ns attach-agent` *`node agent`*: This method of the `Simulator` serves for connecting and agent to a node. *Attach-agent* only calls the `attach` method of the given node, which connects then the given agent to itself. For example, the „`$n0 attach $tcp`" command has the same effect as the 26. line.
- 29, 38: `$ns connect` *`agent 1, agent2`*: after generating the two agents, that want to communicate with each other, we have to build up a logical connection between them; this command serves for it, that sets in agents the network- and port address of the other agent.

After giving the network configuration, we have to define the simulation scenario, i.e. the schedule of events given in advance. `Simulator` class has numerous methods in connection with scheduling, however, the most frequently used method is the following:

- 46-52: `$ns at` *`time command`*: `Simulator` object executes with the help of its scheduler in the given simulation time the given command. For example, it schedules the execution of the „`$cbr start`" command to the 01,s of the 46. line, i.e. it calls the `start` method of the `$cbr` data resource object, which will start the traffic generation because of it.

After giving the network arrangement, schedules and process closing the simulation we only have to start the simulation with `$ns run` command of the 55. line.

5. Process of results

Beyond visual analysis of the simulation results with the NAM displaying program, system provides a possibility to make a register file about all the events of the way of every packet. Every line of the register file generated by the method of Simulator trace-all[4] is related to an event in connection with a packet. Line consists of a field divided by 12 spaces, which are the following:

1. Type and possible values of the event:
   - „r" (receive): packet arrived to the target node
   - „d" (drop): a line dropped the packet away
   - „+" (enqueue): packet got to the queue
   - „-„ (dequeue): packet went away from the queue

2. Time of the occurrence of the event (in seconds)
3. Identifier of the beginning node

[4] Examples used the `namtrace-all` method so far, which, however, generates a file with an output, that is different from the `trace-all` method.

4. identifier of the end node. This one, and the previous value will determine together the link, where the event happened. (Don't mix it with values of fields 9. and 10.)
5. Type of packet
6. Size of packet in bytes
7. Signal bits (we use only the ECN bit presently)
8. In case of a process identifier IPv6, that can be set from tcl (program 4, lines 30. and 39.); even, if the simulator does not use this identifier, we can use it at evaluating the results, or at tracing. (E.g. NAM uses the process identifier for determining colour of packets.)
9. `node.port` shaped resource address
10. `node.port` shaped target address
11. Sequence number of the protocol of the network layer; however, UDP realizations do not use sequence numbers, NS – to support tracing – records also sequence number of packets.
12. Individual identifier of packet.

```
 0:  set ns [new Simulator]

 1:  $ns color 1 Blue
 2:  $ns color 2 Red

 3:  set nf [open out.nam w]
 4:  $ns namtrace-all $nf

 5:  proc finish {} {
 6:      global ns nf
 7:      $ns flush-trace
 8:      close $nf
 9:      exec nam out.nam &
10:      exit 0
11:  }

12:  set n0 [$ns node]
13:  set n1 [$ns node]
14:  set n2 [$ns node]
15:  set n3 [$ns node]

16:  $ns duplex-link $n0 $n2 2Mb 10ms DropTail
17:  $ns duplex-link $n1 $n2 2Mb 10ms DropTail
18:  $ns duplex-link $n2 $n3 1.7Mb 20ms DropTail
19:  $ns queue-limit $n2 $n3 10

20:  $ns duplex-link-op $n0 $n2 orient right-down
21:  $ns duplex-link-op $n1 $n2 orient right-up
22:  $ns duplex-link-op $n2 $n3 orient right
23:  $ns duplex-link-op $n2 $n3 queuePos 0.6

24:  set tcp [new Agent/TCP]
25:  $tcp set class_ 2
26:  $ns attach-agent $n0 $tcp
27:  set sink [new Agent/TCPSink]
28:  $ns attach-agent $n3 $sink
29:  $ns connect $tcp $sink
30:  $tcp set fid_ 1

31:  set ftp [new Application/FTP]
32:  $ftp attach-agent $tcp
33:  $ftp set type_ FTP

34:  set udp [new Agent/UDP]
35:  $ns attach-agent $n1 $udp
36:  set null [new Agent/Null]
37:  $ns attach-agent $n3 $null
38:  $ns connect $udp $null
39:  $udp set fid_ 2

40:  set cbr [new Application/Traffic/CBR]
41:  $cbr attach-agent $udp
42:  $cbr set type_ CBR
43:  $cbr set packet_size_ 1000
44:  $cbr set rate_ 1mb
45:  $cbr set random_ false

46:  $ns at 0.1 "$cbr start"
47:  $ns at 1.0 "$ftp start"
48:  $ns at 4.0 "$ftp stop"
49:  $ns at 4.5 "$cbr stop"

50:  $ns at 4.5 "$ns detach-agent $n0 $tcp ;
51:              $ns detach-agent $n3 $sink"

52:  $ns at 5.0 "finish"

53:  puts "CBR packet size = [$cbr set packet_size_]"
54:  puts "CBR interval = [$cbr set interval_]"

55:  $ns run
```

Program 4. Simple simulation script

In case of running a longer simulation, when, for example, only value of some aggragated features are interesting, it is a redundant waste of space, and a slow solution to store the information of each and every packet. That is why there is a possibility – among

others – for the observation of only one queue – see further details in chapter of „Trace and Monitoring Support" of the NS documentation.

We call the observation by packets „trace" in the NS terminology, and the aggregated state observation, counting we call „monitoring". This latter one is practical to to apply in most of the simulations, because observation of thousands or millions of packets one by one, and transcription of their events could result a very slow simulation, and a tremendous, unmanageable output trace file. We can try a simulation transmitting many packets with switching on NAM tracking, or without it.

So it is practical to apply monitoring for sake of efficiency. Information can be found about these kinds of objects in chapter of NAM documentation mentioned above. We just demonstrate an example here for that how we should complete the previous example to obtain a bandwidth (more precisely, data rate) utilization figure for links 2 and 3, e.g. at a 2 seconds of measurement interval. We can also see the „2-3.bw" output file with the help of *xgraph* or *gnuplot* programs.

```
…
set bwSampleInterval 2.0
…
[$ns link $n2 $n3] attach-monitors [new SnoopQueue/In]
       [new SnoopQueu/Out] [new SnoopQueue/Drop] [new QueueMonitor]
…
proc SampleBW {link} {
       global ns bwSampleInterval
       set qm [$link setg qMonitor_]
       puts „[$ns now] [expr [$qm set bdepartures_] / $bwSampleInterval]"
       $qm set departrures_ 0
       $ns after $bwSampleInterval „SampleB $liknk"
}
…
$ns at 0.0 „SampleBW [$ns link $n2 $n3]"
```

Program 5. Completion of monitoring