Modeling Interface for ns-2

Version 2.1

[1] Pascal Anelli [1] Cedrik Larose
[1] University of La Reunion
Laboratory of Computer Science and Mathemathics(LIM)
CS 92003. 15 Avenue René Cassin
97744 Saint Denis Cedex 9 - France
Pascal.Anelli at univ-reunion dot fr

Contents

1 Motivations **3**

I Users Guide **3**

2 Modeling **3**
2.1 Principles . . . 3
2.2 Resource file structure . . . 4
2.3 Resource file syntax . . . 5

3 Components definition **6**
3.1 Network topology . . . 6
3.2 Endpoints . . . 7

4 Error model **10**
4.1 Principles . . . 10
4.2 Error declaration . . . 10
4.3 Error call . . . 11

5 Trace analysis **11**
5.1 Analysis processing syntax . . . 12
5.2 pkt resource . . . 12
5.2.1 Flow Scope . . . 13
5.2.2 Queue scope . . . 14
5.2.3 Metrics of metrics . . . 15
5.3 param resource . . . 15
5.3.1 TCP scope . . . 15
5.3.2 RED scope . . . 15
5.4 Other resources . . . 15

6 Execution control **16**
6.1 Command line . . . 16
6.2 Execution . . . 17
6.3 Nam viewer . . . 17

II Modeling Guide **17**

7 Model Structure **17**
7.1 Test Suites . . . 17
7.1.1 Root class functions . . . 19
7.1.2 Topology . . . 20
7.1.3 Sources . . . 20
7.1.4 Post-process . . . 21
7.1.5 Tracing . . . 21

8 Model Extension **21**
8.1 Command line . . . 24
8.2 Model development . . . 24

III Appendixes **24**

A Settings **24**
A.1 Agent TCP: Source . . . 24
A.2 Agent TCP: Destination . . . 25
A.3 RED . . . 26

B Utilities files **26**

1 Motivations

Designing a simulation model is a long process requiring a very good knowledge of the OTcl interface of ns-2. With a partial documentation, this script writing process is very tedious. When the parameters change, the simulation model may evolve into a number of files difficult to handle. Given these difficulties with the use of ns-2, the University of Reunion Island has developed a library of scripts which can be used to easily elaborate a simulation model for ns-2 without a good knowledge of Tcl. The basic idea is to separate the data from the instructions. The data are the parameters describing the simulation model and the analytic processing to be performed on the simulation traces. The values of these parameters are defined in a file known as the resource file. A such file is interpreted by the functions of the library in order to carry out a study. This document contains the *User's guide* (Part I), the *Modeling guide* (Part II) and the *Appendixes* (Part III).
The User's Guide presents the following aspects: use principles of the library (Section 2), the specifications of the different components (Section 3), the description of the error model (Section 4),the requests for the trace analysis (Section 5) and the execution control a simulation (Section 6).
The Modeling Guide details the structure of the model (Section 7) and how to extend the model (Section 8), and the Appendixes presents the main parameters of the endpoints (Section A) and the utilities files (Section B).

Part I
Users Guide

2 Modeling

The library of scripts provides fast simulation models without the need of manipulating Tcl code. It also provides analysis functions for traces produced by the simulation. These post-process functions calculate the metrics to assess the service perceived by a flow in terms of loss rate, rate of flow, and so on. The simulation model as well as the post-processes is specified in the resource file. The resource file contains the parameters and values specific to a simulation model description. The library contains a set of Tcl files located in the `0-Lib` directory. The elements contained in the resource file describe:

- routers and communication links
- host and transmission sources as well as their access link,
- analysis made at the end of the simulation,
- calculation parameters of analysis also known as post-process.

When many studies are to be carried out, a resource file must be written for each study. Thus, it is possible to run a simulation again or to read over the parameters of the simulation model during the analysis of its results. Furthermore, the simulation results can be easily retraced. Thus a recording process concerns only the resource files.

2.1 Principles

The simulation model is composed of two types of elements: the endpoints and the network (communication infrastructures). The defined structures may be of *classical* or *diff-serv* type. The diff-serv type distinguishes the role of the router according to its location from the packet flow. The input and output points of the network are formed by an edge router (edge). Central routers (core) are used between the two points.

In a classical infrastructure, this distinction between the roles of the routers does not exist. The communication infrastructure represents the part of the network shared by a set of flow. This is where contentions between flows occur.

The endpoint(host) is a node where the sources and the traffic sinks are situated. The hosts are set in pairs: a source host is associated to a destination host, a pair of hosts is associated to one or several packets flow. The arrangement of the hosts and flows is the concept of traffic pattern. Specifically, a traffic pattern is a set of homogeneous flows entering and exiting through the same routers. A traffic pattern can be composed of one or n pairs of hosts. A source host is composed of at least one traffic source, i.e. it emits at least one packet flow in the network.

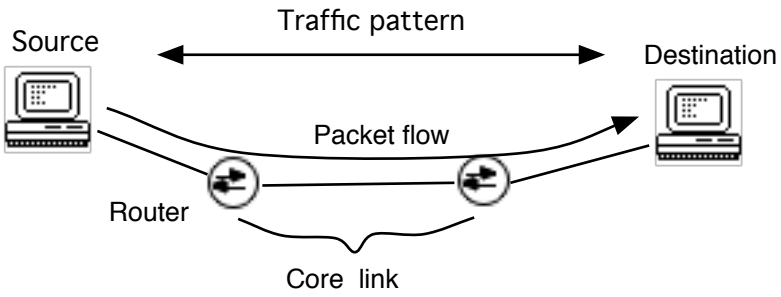

Figure 1: Model structure

The simulation models which can be created using this library are limited to the study of contention with one or several bottlenecks. A simulation model representing a simple bottleneck is shown in figure 1.

2.2 Resource file structure

The simulation model is defined in a resource file. The resource file is made of a set of description parameters for the study: the simulation model and the analysis. A resource is a parameter defined by the *set* command. A resource defines on of the components of the simulation model. The resource file structure into 5 main parts:

- assignation of default values for the class parameters applying to the model
- assignation of default values to the resource of the arguments
- definition of the communication infrastructures
- definition of the traffic pattern applied to the communication infrastructure
- determination of the measurements to be made and the trace analysis processing

The 5 main parts listed above can be defined as follows:

- Assignation of default values for the class parameters applying to the model:

```
#=====================================================
# BEGIN of resource attributes
#=====================================================
# Default parameters for the model
#     List of parameters, see tcl/lib/ns-default.tcl
```

- Assignation of default values to the resources of the arguments of the command line: The resources which are initialized in the resource file are those which are not given as arguments in the command line.

```
# Parameters initialized in the resource file
#-----------------------------------------------------
set opt(nam)          0; # Write traces in nam format
set opt(namgraph)     0; # session graph window in nam
set opt(stoptime)     0; # Simulation duration
set opt(quiet)        0; #boolean: draw graph (0) or not (1)
set opt(verbose)      0: # print  log file at the end
set opt(replication)  1; #Replication number for randomgenerator
set opt(wd)           tmp; # Working directory
set opt(grapher)      xgraph; # or gnuplot
set opt(cleanup)     1; # boolean, remove tmp file in wd
set opt(pp)          1; # do a post process on the trace
set opt(f)           ""; # Ressources file
set opt(t)           "x" ; # term for gnuplot x = x11 e=eps a=aqua
set opt(prefix)      ""; # filename prefix
```

- Definition of the communication infrastructures:

```
# Network topology
#================
#core network
set opt(corelink)   {}
set opt(routelength) 1
#edge link (for Diff-Serv network)
set opt(edgelink)  {}
```

- Definition of the traffic pattern applied to the communication infrastructures:

```
# Host
#================
# The host section defines the traffic patterns.
set opt(pattern) {}
```

- Determination of the measurements to be made and the trace analysis processing:

```
#  Post Process
#================
# Analysis#
#       Packets trace file
set conf(pkt) {}
#       Variables trace file
set conf(param) {}
set conf(plot) {}
set conf(parameter) {}; # default parameters in test-metric.tcl:default-parameter
#=================================================
# END of resource attributes
#=================================================
```

2.3 Resource file syntax

The syntax of the resource file is written in Tcl. The *opt* table describes the resources for ns-2. The *conf* table contains the necessary resources for processing the traces after the simulation. The values assigned to the elements of the table follow the syntax of the list type of Tcl. The list type description in [1] is as follows:

> Lists are used in Tcl to deal with collections of things. Lists allow you to collect together any number of values in one place, pass around the collection as a single entity, and later get the component values back again. A list is an ordered collection of elements where each element can have any string value, such as a number, a name or a word of a Tcl command. Lists are represented as strings with a particular structure; this means that you can store lists in variables, type them to commands, and nest them as elements of other lists. In its simplest form a list is a string containing any number of elements separated by spaces or tabs. For example, the string: 2 Alice Bob 3 is a list with four elements. There can be any number of elements in a list, and each element can be an arbitrary string. When a list is entered in a Tcl command, the list is usually enclosed in braces. The braces are not part of the list; they are needed on the command line to pass the entire list to the command as a single word. When lists are stored in variables or printed out, there are no braces around him. Braces are often used to nest lists within lists, as in the following example:
> `set variable {a b {c d e} f}`
> In this case element 2 of the list is itself a list with three elements. There is no intrinsic limit on how deeply lists may be nested (an index of 0 corresponds to the first element of the list).

According to the Tcl list type syntax, the grammar used for the resource file is presented as follows:

```
resources    :=  [resource  {";" resource }]
resource     := designator valeur | "#" {string}
designator   := "set arrayname "(" resourcename ")"
resourcename := string
```

```
arrayname    := opt | conf
value        := simplevalue | list
simplevalue  := string | "{}"
list         := "{"  { value | doublevalue } "}"
doublevalue  := "{" pair | parameter "}"
pair         := value value
parameter    := parametername value
```

A resource dedicated to the simulation part consists of an assignation to an element of the *opt* table, for example:
`set opt(routelength) 2`
The resource named *routelength* takes the simple value 2. If it is not a simple value, it should then be declared by a list as follows:
`set conf(parameter) {{bottlebw 0.4Mb}}`
In this example, the *parameter* resource consists of a compound element (bottlebw) which is declared as a list element. The first bracket declares the list of values and the second bracket declares the list for a compound value. According to the grammar, the compound value is described by the *doublevalue* rule. The value of a parameter can also be represented as a list of values such as:
`starttime {10 20 30}`
A list given as value of a parameter is processed orderly. The concept of ordered processing indicates that the position value in the list refers to the sequence number of occurrence of the parameter. In our example, the first value (10 seconds of simulated time) applies to the first use of the *starttime* parameter. The starttime parameter specifies the start time of the traffic sources. For each source creation, the sequence number is incremented. Thus, the first source starts at 10 s, the second at 20 s, and so on.

When there is no exact match on the size of the list with the number of occurrences of a given parameter, the last value is applied to the occurrences of the parameter that exceed the maximum order number of the list. For example, the list value1 value2 indicates that value1 applies to the first occurrence of the parameter and value2 for the second and following occurrences of the parameter. A generalisation of this principle is that a list composed of a simple value indicates that all the occurrences of a given parameter will take the same value.

A parameter can be composed of a list of pairs, for example, for the propagation time:
`tp { {50ms 50ms} }`

In this case, the list of pair of values is in fact a pair of simple value whose first value applies to the upstream (igress) and the second to the downstream (egress). This kind of definition is used in the description of the access links. A compressed form of writing can be used when the upstream link is equal to the downstream link:
`tp { 50ms }` or `tp 50ms`

The value of a parameter can take the form of a list of values as seen previously (starttime example). According to the list rule, the element of this list can also be a doublevalue such as a pair of values as shown in the following case:
`Flow { { {0 1} {throughput} } }`

This statement reads: at the flow level, the flow 0 and 1, determine the throughput.

The names of the parameters never take the "_" character (underscore). The latter is automatically added by the resource file analyser.

3 Components definition

In this section, we present the resources and their parameters. For each resource, the defining rules will be specified.

3.1 Network topology

The core network consists of the following resources:

- routelength: determines the number of hops in terms of number of core routers. The minimum value is 1.
- corelink: defines the features of the links of the core network. This is an unordered list.

A link is characterized by the following parameters:

Figure 2: Core network with 5 hops

- bw: bandwidth for both directions,
- tp: propagation time for both directions,
- queuesize: size of the waiting list. This is actually the limit beyond which the waiting list is overflowed. A limit of 2 means that there can be only one packet in the waiting list. When a packet arrives in a waiting list of size 1, it is dropped. There cannot be a waiting list with a limit of 1, because as all the packets go through the waiting list, they will all be lost. When the packet is being transmitted, it is no longer in the waiting list. The value 2 can then be considered as the maximum number of packets in an output interface.
- queuetype: the discipline of service or queuing delay applied to all the links (the value is thus compulsorily a simple value),
- linkparam: the common parameters to be applied to all instances of queuetype.

At the corelink level, the concepts of upstream and downstream do not exist. Furthermore, the links are symmetrical. The doublevalue rule is thus not applied. The values of the parameters of the corelink resource are lists which are processed orderly. The first value is for the first link, the second value for the second link and so on. This rule applies to all the parameters except *linkparam*. This parameter contains the initialization parameters proper to *queuetype* class which are applied to all the instances in the network.

In order to illustrate the definition of a network, let us study figure 2 in which the waiting list has a length of 20 packets and whose list management is of RED type.

The declaration of this network is as follows:

```
set opt(corelink)  {{bw {2Mb 1Mb 1Mb 2Mb}} \
                            {tp {5ms 5ms 10ms}} \
                            {queuesize 20} \
                            {queuetype RED} \
                            {linkparam { \
                       {setbit false} \
                       {gentle true} \
                                   {mean_pktsize 1000} \
                                   {idle_pktsize 200} \
                            } \
                          } \
                        }
set opt(routelength) 5
```

Note: the "\" character indicates a continuation of the list on the next line. After the continuation character, there can be no other character except a new line. Otherwise, the list is flawed thus producing an interpretation error.

The nodes are identified by a number. The given sequence number is the order in which the node has been created. The sequence number starts at 0. The node creation process starts with those of the network and continues with the endpoints part. For the endpoints, the sequencing is carried out by node pairs (source-destination) starting with the source. Using the example of figure 1, the numbers assigned to the nodes are indicated by figure 3.

3.2 Endpoints

The definition of trace sources is made by the *pattern* resource. This resource is defined as a Tcl list and may include several items related to traffic patterns. A traffic pattern is defined as a set of flows with similar features in the transport layer with the same input-output point at the network level.

Figure 3: Node numbering.

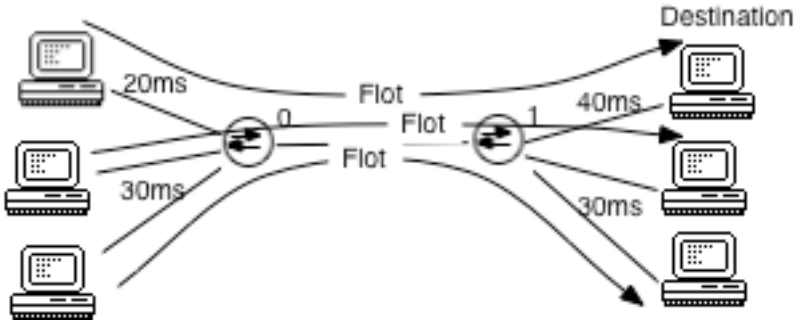

Figure 4: Traffic pattern with 3 node pairs

The description of a traffic pattern has two parts:

1. the contextual part; this part contains specific information to a node or a flow
2. the static part: this part gathers the configuration information of the transport layer and/or application layer common to the whole pattern.

There also exists the *class* concept. A class can be defined as a group of flows, in a pattern, which have common caracteristics. For example, the active flows in a given direction.

The contextual part consists of the following elements:

nbflow The number of traffic flows for each node pair. The value of this parameter is an ordered list. For example, the specification of 2 node pairs where the first node is composed of 3 sources and the second of 5 sources gives:
`nbflow {3 5}`

nbnode The number of pairs, i.e. node pairs source and destination

bw The bandwidth of an access link and the exit link of each node pair. The value of this parameter is described by an ordered list of pairs of simple value. The first value is the access link (igress) and the second value the exit link (egress). For example, the connectivity description for a network as shown in fig.4 is as follows:
`bw {{20Kb 40Kb} {30Kb}}`

tp The propagation time for an access link and the exit link of each node pair. The defining rules for this parameter follow the same rules as the *bw* parameter.

transfersize The size of the file to be transferred. This parameter applies only to TCP flows. The -1 value indicates an endless transfer. The *transfersize* parameter consists of the elements *distribution* and *paramlist* as shown below. The *distribution* parameter defines the distribution function used for the transfer (Exponential,ParetoII,...) and the *paramlist* parameter defines the list of parameters used by the distribution functions.

```
{transfersize { \
                  {distribution Exponential} \
                  {paramlist {{shape 1.3} \
                             {avg  45000 } } \
                  } \
               }  \
} \
```

starttime The start time of the source of a node. This parameter is a double ordered list. The first list consists of items relevant to a node. Thus, for example, the start time of the third flow in the second node can be indicated.The description of this parameter follows the same rules as the transfersize parameter.

finishtime The stop time of the source. This parameter applies only to UDP flows. The -1 value indicates the creation of an endless transfer. The description of this parameter follows the same rules as the starttime parameter.

The static part of this traffic pattern contains information that is independent of the number of nodes or flows of the traffic pattern. It includes:

hook The grip nodes of the source and destination. This parameter applies to a classical network (non diff-serv). This parameter connects the ends with the network. The hook nodes of the network are specified by the sequence number through a pair of single values. For example: to indicate the need to link the input traffic pattern on node 0 and the output traffic pattern on node 1:
`hook {0 1}`

type The type is a basic class of agent. It thus indicates the traffic class of the traffic pattern. This parameter acts as a selector for applying specific processing to a class of agent in the model composition. The value of this parameter is a simple value and takes a TCP or UDP value.

connection The parameter describing the type of connection used which can be either steady(Long) or dynamic(Short) flows.

sourceparam The describing parameters of the characteristics of the flows of the traffic pattern. The value of this parameter is a non-ordered list of parameters. The elements of this list depend on the type of the class of the traffic pattern.

For TCP flows, the sourceparam parameter consists of the following elements.

tcpversion TCP version of the source,

tcpsink TCP version of the destination,

agentparam The configuration parameters of the TCP agent. These parameters are those defined for the TCP model (cf. agent parameters section of this document).

For example, the specification below shows a TCP/NewReno pattern with 1000 byte packets, a flow control window of 60 packets and a file transfer of infinite size.

```
{sourceparam { {tcpversion TCP/Newreno} \
               {tcpsink TCPSink} \
               {agentparam { \
                             {window 60} \
                             {packetSize 1000} \
                           } \
               } \
             } \
} \
```

For UDP flows, the *sourceparam* parameter consists of the following elements:

trafficgenerator The traffic generator type (CBR, Exponential, etc.) The traffic generators of ns-2 are detailed in the ns manual in the "Applications and transport agent API" chapter.

trafficparam The parameter of the traffic generator. The list of parameters specific to each traffic generator is described in the ns manual.

agentparam The UDP agent parameter. The UDP chapter in the ns manual presents the list of parameters.

Let us consider a CBR traffic generator with an emission rate of 100 Kb whose messages are of 1250 bytes and the UDP messages size limited to 1250 bytes.

```
{sourceparam {{trafficgenerator Traffic/CBR} \
              {agentparam { \
                            {packetSize 1250} \
                          } \
               } \
               {trafficparam { \
                               {random 1} \
                                {rate 100kb} \
                               {packetSize 1250}\
                              }\
               } \
             } \
 }\
```

4 Error model

This section describes how to introduce packet losses into a simulation. The error model principles follow the presentation made in chapter 13 in [3].

A huge range of situations can be simulated using the ns-2 error-model. We propose a modeling interface for common uses: the possibility to model a sequencing error or a drop on a particular flow, at the individual flow level or on the set of flows transiting through a link.

4.1 Principles

The ns-2 error model is defined in the modeling by the *Error* class. This class is a wrapper of the ns-2 *ErrorModel* class. It is used to interprete the given arguments for the creation and configuration of an error model. Actually, a set of error models can be defined with the respective actions of each error model chained to each other. The Error class is used to handle this set of error models. In the object-oriented concept, this class defines an aggregation. The Error class is implemented in the test-error.tcl file.

4.2 Error declaration

The declaration of an *Error* is done either at the pattern level or at the network level. This declaration is done by the *error* attribute. The following example represents an *error* object declaration which is the composition of a sequencing error model and a drop model.

```
{error { \
                            {{type List} \
                             {todo delay} \
                             {paramlist {{unit pkt} {delay 0.1}} }\
                             {actionlist {23 24}}\
                            } \
                            {{type Rate} \
                             {todo drop} \
                             {paramlist {{unit pkt} {rate 0.005} }} \
                            } \
                           } \
         } \
```

The *error* attribute is defined by a list in which each element is an error model. The attributes to create an error model are:

type the ns-2 errormodel model type(Rate for the default ns-2 model)

todo the action performed on the erreneous packet

paramlist the parameters and their values in the errormodel

actionlist the sequence number of the erreneous packets(List type errormodel)

4.3 Error call

The interpretation of the declaration of the *error* attribute is not integrated to the modeling interface. The function calls interpretation must be done from a specialized script derived from the Test class(test-suite-template.tcl). This base script is defined by the test-suite.tcl. The derived classes from the Test class offers a degree of flexibility in the description of the models. For example, a model consisting of an error model can be run with or without the interpretation of the error model. The choice is done on the command line. When the *error* attribute is placed in the corelink parameter, the interpretation must be performed by the *add-error* function(test-topologies.tcl). The function declaration is as follows:

```
Core instproc add-error {replication {granularity 0} igress egress}
```

with igress and egress the node numbers where the error must occur, a granularity of value 0 or pass. The first case corresponds to the application of the error model to all the flows and the second case corresponds to the case where only flow 0 will suffer from errors.

Typically, the *add-error* function is called from the *create-topology* function of a specialized class. This will be declared as shown below:

```
Class Test/ErrorCore -superclass Test/BestEffort

Test/ErrorCore instproc create-topology {} {
        $self  instvar corenetwork_ replication_

        $self next
        $corenetwork_ add-error $replication_ 0 0 1
}
```

If the *error* attribute is placed in the pattern parameter, the error model can be applied on an individual flow basis(identified by a flow id) or on all the flows emitted from a source. The choice is made by the function call of the *error* parameter. A distinction can be made between 2 possible calls in the test-source.tcl:

- SourcePattern instproc add-error-flow returnpath 0 front 1 : each flow of the pattern has its own error model.
- SourcePattern instproc add-error-node returnpath 0 front 1 : all the flows of the pattern pass through the same error model.

By default, the error model is placed on the access link of the source on the data path(upload). Similarly, the function to add an error is called from a specialized class. In details, this is done by the following lines:

```
Class Test/ErrorFlow -superclass Test/BestEffort

Test/ErrorFlow instproc create-pattern {pattern replication} {
        $self  instvar pattern_ opt_

        $self next $pattern $replication

        for {set i 0} {$i <[llength $pattern]} {incr i} {
                # Add Error Model on data path of exit link
                $pattern_($i) add-error-flow  0 0
        }
}
```

5 Trace analysis

Once the simulation is completed, the trace files may have been created. These files include:

- packet activities: the file is called packet type and suffixed by .pkt
- state parameters: the file is called parameter type and suffixed by .param

- the packet activities and the state parameters of the agents in the nam network viewer format (Network Animator). The file for the NAM graphical tool are suffixed by .nam

The processing to be made on these files is activated by a second set of functions of the library. The control of these processing is done by the resources of the *conf* table. The *conf* table is interpreted before and after the simulation:

- before to attach the traces and fetch the files to be traced,
- after to execute the requested processing of the trace files.

The nam file case is a particular one (cf. section 6).

5.1 Analysis processing syntax

The description syntax of the analysis processing follows the one shown in the syntax section of the resource file. In the context of the *pkt* resource and the *parameter* resource, the grammar is specialised as follows:

```
list          := "{" {element} "}"
element        := "{" who what "}"
who            := simplevalue | compoundvalue
what           := list | metrics
simplevalue    := string | all | numbers
compoundvalue  := "{" {simplevalue | interval } "}"
interval       := "{" number "-" endnumer "}"
endnumber      := number | "$"
all            := "*"
metrics        := metric | "{" {metric} "}"
```

The "$" character indicates to the last. The "*" character indicates for all. The available metrics are indicated in 5.2.3. As shown by the *element* rule; the analysis request principles consist in indicating "who" then "what". To illustrate the use of this grammar, let us take the following example:

```
set conf(pkt) {  \
                {Flow  { \
                        { * {activity throughput AverageEWMA} } \
                        {{1 3 5} {delay} \
                       } \
                } \
              }
```

In this example, the *pkt* resource is composed of a list of elements (list rule). An element describes who and what. The semantics of "what" concerns the scope of the analysis. The scope means the object class for which the processing is requested. The 2 scopes of the packet resource are:

Flow A flow is a sequence of packet from the same source to the same destination and from a source. A flow is identified by a flow number (noted fid). This scope deals with network service to the user.

Queue This scope grips to the waiting list found in the output interfaces of the routers. This scope studies the network operations from the operator's point of view.

After the scope ("who" rule), the "what" indicates which analysed process must be performed. According to the "what" rule, the latter is described as a list of items in which the concerned flows are identified by the indicated metrics. In the example below, the description of the processing is interpreted as follows: for all the flows; calculate the activity, the instantaneous and mean throughput for each flow and the flows identified by number 1, 3 and 5, calculate the time of transfer. It should be noted that when the "who" changes, another element should be created. We are now going to describe the specificities and descriptions of each resource.

5.2 pkt resource

This resource concerns the processing of the trace files of packets. When a packet makes a hop between 2 nodes, the different steps are recorded in the trace files. Figure 5 presents the order of these events during the delivery of a packet between 2 nodes.

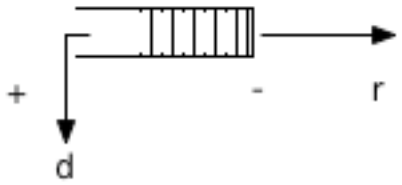

Figure 5: Types of recordings

5.2.1 Flow Scope

The processing on the *flow* scope can be done based on the applicative flow or an aggregation of flows. The calculation method used is the list of metrics with the instructions:

OverAll calculation of metrics on the indicated aggregation of flows,

PerFlow calculation of metrics based on individual flows. By default, it is the selected mode.

The calculation method once indicated cannot be changed and applies to the set of metrics of the element. However, it is possible to combine calculations on the aggregate and on the individual flow basis by constituting different element.For example, the statement to indicate that the throughput for flow 1 and 2 is calculated on their aggregate and individually is as follows:

```
{{1 2} {OverAll throughput} } {{1 2} {PerFlow throughput}}
```

The relevant metrics to the flow scope are:

activity this curve represents the packet exchange from the source viewpoint.

delay the transfer time of packets from end to end. The curve represents the packet transfer time over time.

dropratio the rate loss of packets over time. The rate loss is calculated per size sample of packets received at the destination.

dropput the rate loss of packets over time. The calculation is that of the flow by considering the dropped packets. The calculation is identical to that of *sendrate*.

dropevent the representation of drop events over time.

dropratioT the average packet loss rate over the total time measurement period. For every loss the loss rate is calculated from the beginning of the measurement period.

droppprecedenceT the global loss rate of packets over their loss priority.

rate the packet emission rate over time. The rate is calculated using a time window advancing by hops. The hop size is indicated by the *timeinterval* parameter. The rate is expressed in percentages. For this issue, the normalization rate is indicated by the *bottlebw* parameter

throughput the throughput flow (or reception) over time. The calculation is identical to that of rate.

throughput_TSW the throughput over time by using the Time Sliding Window algorithm (TSW) [2]. The measured time window is given by timeinterval parameter.

throughputT the mean rate over the total time measurement. The calculation is identical to rate but using the accumulated amount of data that have passed through the network. This quantity represents the sustainable flow.

goodput the useful throughput over time. The calculation is identical to rate. The rate calculation uses the acknowledgements. Subsequently, this metric applies only to TCP flows.

goodputT the total useful throughput over the measurement period. The calculation is identical to *sendrate* but uses the accumulation of the received data without considering the retransmissions.

receivepkt the number of received packets over time

sendpkt the number of sent packets over time

droppkt the number of dropped packets over time

5.2.2 Queue scope

The *queue* scope applies to the waiting list of a link. The link on which the metrics are applied should be indicated. The link identification is done by identifying the connected nodes at each end. The nodes are identified by their creation sequence number in the simulation model. The printed description of the network topology helps to retrace the identifiers in the log messages printed on launching the simulation. The topology is described as followed for the case of figure 1:

```
 TOPOLOGY
-------------------x-------------------x--------------------
Core network
                node id: 0 index: 0
                node id: 1 index: 1
Source Pattern: TCP
         source node id: 2 index: 0
         destin node id: 3 index: 0
                connected to network node id 0 1
                agent class: Agent/TCP/Newreno fid: 0 index: 0
-------------------x-------------------x--------------------
```

The node identifiers are printed through the node id label. From a syntax viewpoint, the identification of a link is made through a number pair. For example, to trace the packet activity on the waiting list of the link 0-1, the description is as follows:

```
{Queue { \
          {{0 1} {activity}} \
        } \
} \
```

Some metrics resume the definitions of the metrics of the flow scope. By default, these metrics process the flows as an aggregate. These are:

- activity
- dropratioT
- throughput
- throughput_TSW
- throughputT
- dropratioT

The new metrics defined for the *queue* scope are:

rateinstant the instant rate offered per packet (indicated by the samplesize parameter (default=1))

inputcurve let $R(t)$ be the cumulative function of the emitted data over time.

arrivalcurve let $\alpha(t)$ be the deconvolution (1) of the cumulative function of emitted data.

$$\alpha(t) = sup_u\{R(t+u) - R(u)\} \tag{1}$$

length the instantaneous length and mean length of the waiting list over time. The mean length calculation is done by a pondered mean of the weight indicated by the weightlength parameter. The time estimation separating 2 packets emission is guessed from the bottlebw parameter.

congestion_event (experimental) number of waiting list overflow from the beginning of the measurement period.

5.2.3 Metrics of metrics

After calculating the performances metrics, we can ask to calculate the statistics. These metrics of metrics are:

statistics

sustainable arithmetic mean

AverageFloat float mean

AverageEWMA exponential average

pdFunction probability density function

cdFunction cumulative distribution function

The specifications of the processing are done on the string principle. For example, to calculate the mean throughput of the flow 0, this is written : `Flow { {{0} {throughtput AverageEWMA sendrate}} }`
The AverageEWMA metric applies to the preceding one. The sendrate metric is not a metric of metric, thus the chaining stops and the processing restart with the trace file of packets.

5.3 param resource

This resource indicates the processing to be done on the param file. This file contains the state variables of an object. The name of the state variable of the objects are the metrics. The syntax of the elements of the *param* resource follows the "who what" principles exposed in the syntax section of analysis processing. The scope indicates to which object type, the state variables to be traced belong. The defined scopes are RED and TCP.

5.3.1 TCP scope

The TCP agents for which the state variables are to be traced are identified by their sequence number in the traffic pattern (index field in the printing of the topology description). For example, trace the congestion control window for the first agent of all the traffic patterns of TCP type is described :

`{TCP {0 {cwnd}} }` The agent sequencing in the traffic pattern starts with 0. The state variables of TCP are defined in the parameter section of the agent. To conclude this scope, the example below traces the congestion control window of all TCP sources and the acknowledgement numbers for the TCP agent 0 and 1 of all the traffic pattern of TCP type

```
{TCP { \
             {* {cwnd}} \
     { {0 1} {ack}} \
     } \
 }\
```

5.3.2 RED scope

The RED scope concerns the objects of Queue/RED class. In fact, this scope functions only if there is one instance of the class. The "who" in the metric list is fictitious. However it should be represented, for simplification of the analysis procedure of the metric list. The available metrics for RED is length only. The statement is thus:
`{RED { {{0 1} {length}}}}`

5.4 Other resources

The parameter resource defines the values of the parameter for the metrics calculation. Table 1 lists all these parameters; the default value is the metric(s) which use it. The parameter resource is described as a list of parameters following the following example:

The parameter resource is described as a list of parameters based on the following example:

```
set conf(parameter) { {bottlebw 0.8Mb} {timeinterval 0.4} \
                      {timetransit 10} {sizessampledrop 1} }
```

Parameter	Value	Metric	Label
timeinterval	1	rate and TSW	Time slot
weightlength	1.0/500	length	
weight	1.0/32	AverageEWMA	
timetransit	0	All	time before steady state
sizesampledrop	1	dropratio	number of drops per writing
accuracy	1.0/100	pdFunction	
bottlebw	1Mb	rate, length	to normalize rate
floating	20	AverageFloat	number of samples
modulo	1000	counting and sequence number	
wrap	90	activity	seqnum modulo for activity
scale	0.01	activity	TCP seq num

Table 1: Calculation parameters of the metrics

In this example, the measurement period start at time 10s. Before this time, the model is in a transition state and is not representative. The plot resource gives the instructions for regrouping the curves on the same figure. For example, to print on the same graph the rate and its mean, the description of plot resource is as follows:

```
set conf(plot) { {throughput AverageEWMA} }
```

6 Execution control

Once the modelling and specification phase have been completed, we can move to the execution of the simulation model. This section presents the different activities and the execution results of the simulation model and the results analysis.

6.1 Command line

The execution of the simulation model is launched by a command line whose syntax is indicated below. The launching command is composed of arguments in order to control the execution of the simulation from the command line.

```
SYNOPSIS
ns <script name> [options] <resource file name>
<script name>
                test-suite.tcl : for example to use the sample models provided
<resource file name>
                the resources file
OPTIONS
-wd DIRECTORY
                Directory for the result of the simulation(by default tmp)
-stoptime NUM
                Maximum duration of the simulation expressed in simulated time
-nam 1
                Run nam when simulation is over
-quiet 1
                Do not draw anything.
-verbose 1
                Indicates what is being executed or trying to be executed.
-namgraph 1
                session graph window in nam
-replication NUM
                Replication number for random generator
-grapher GRAPHNAME
                grapher type: xgraph (gnuplot not yet supported)
-testname  TEST
```

`Class name of the TestSuite to execute`

The resource file is a compulsory argument passed by the -f option.

6.2 Execution

The execution of the simulation model generates huge size files. It is thus important to use a local directory of the computer performing the execution, for example: */tmp*. The work directory of the simulation is indicated by *-wd* in the command line. This work diretory is either created, if it does not exist, or overwritten. When the execution of the simulation begins the messages of the actual processes are printed on the terminal. The running of the file processing is also printed. The important information given by these messages is the name of the result file (suffixed by .data) associated to the calculated metrics. The result file is in the directory indicated by the *-wd* option or by default in the *tmp* directory of the current directory. Concerning the metrics which end by the letter "T", the final result is printed on the console. In case of an execution error of the Tcl code, the execution stack is printed (starting from the top), the error description is in the first printed lines.

6.3 Nam viewer

The nam trace file is constituted when the nam or namgraph argument is passed in the command line (-nam 1 or -namgraph 1). The processing according to the simulation is exclusively nam, it eliminates the other traces processing. Consequently, the *conf* table is not interpreted unless the namgraph argument has been indicated. In this case, the param resource is interpreted in order to print the state variables asked by the TCP agents. The namgraph argument prints the session graph in a NAM window (Analysis menu, item Active Session). Thus, the state of certain agents can be visualized during the animation.

Note: Errors with the namgraph have been reported when the state variable sstresh of TCP is retained.

Part II
Modeling Guide

7 Model Structure

7.1 Test Suites

The `0-Lib` directory contains test-suites file. A test-suite is a set of files written to verify the correctness of a module in ns. The ns tcl script is used to run through the tests defined. In our model, there exists a library containing different functions, named the test-suite-template.tcl. This library contains the different functions which can be used to create other test suites and it is located in the `0-Lib` directory.The `0-Lib` directory also contains other small libraries which are loaded by the test-suite-template.tcl which are: global.tcl, test-topologies.tcl, test-source.tcl, test-error.tcl and test-post-template.tcl. The test-suite-template.tcl file contains the TestSuite class. The TestSuite class is a super class in which the functions required for each test case is defined as well as other configurations. These functions can be regrouped into 5 main sets of functions:

- root class functions
- source functions
- post-process functions
- topology functions
- tracing functions

Figure 6 represents these set of functions.

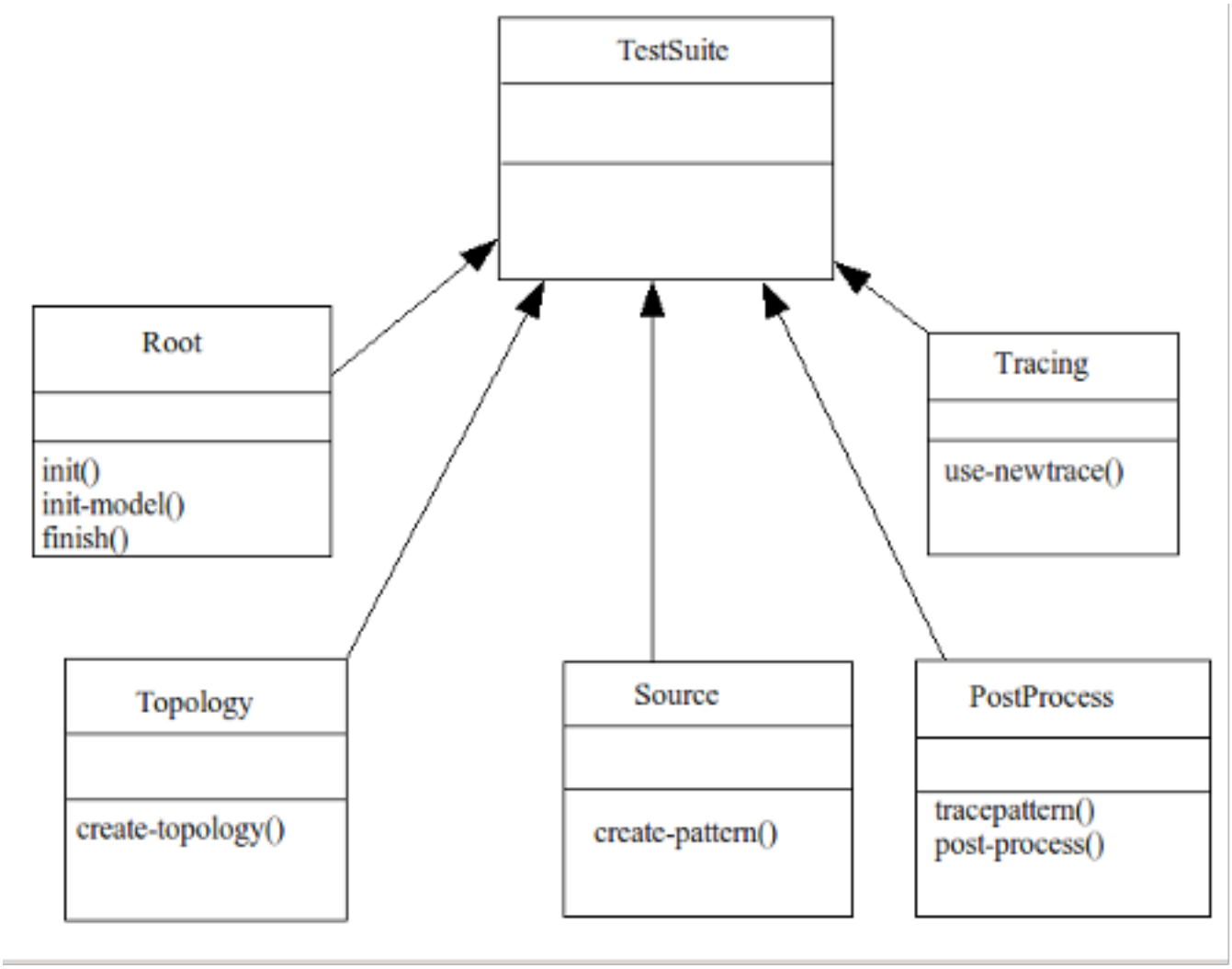

Figure 6: Sets of functions

7.1.1 Root class functions

These functions are essentially used to initialize the model and to define the processes to be performed on launching the model. The main functions in this class include:

init : the function which is used as a constructor for the model and can be used to define operations and set certain parameters for the model. For example, to set the parameter for a *fid* count:

```
set fidcnt_ [new FidCnter]
```

The following lines set the parameters test_ and name if the test_ parameter does not already exist:

```
if {![info exists test_]} {
        set name [$self info class]
        set test_ [lindex [split $name /] 1]
}
```

The syntax to create a simulation instance runnning on multicast is as follows:

```
set ns_ [new Simulator]
    if {[llength $args] > 0} {
        if {[lindex $args 0] == "-multicast"} {
        $ns multicast
             }
}
```

finish : This function defines the different procedures to be applied at the end of the simulation. These procedures include function calls which are made to:

- close all the trace files
- start Network Animator
- call post processes and drawing functions

The following code is an example of a call the to a post-process function:

```
TestSuite instproc finish {} {

. . . .

if {$opt_(pp)} {
    set cmdline "ns $env(LIBSUITE)/post.tcl \
            -f $env(PWD)/$opt_(f) \       # read the resource file
            -prefix $prefix_ \            # definition  of the file prefix
            -wd $env(PWD)/$opt_(wd) "     # location of these files
    foreach i "quiet  verbose cleanup" {
            append cmdline  "-" "$i " "\"$opt_($i)\" "
           }
$self verbose "$cmdline "


}
```

init-model : This function regroups all the essential function calls made on initialization which are common to all the models. The example shown below include calls to:

- create a topology class
- create a traffic source
- set specific post-process measurements

```
TestSuite instproc init-model {} {
        $self instvar  opt_ corenetwork_
        global conf           # global variable

        $self create-topology
        $corenetwork_ configure-corelink     # configure queue size
        $self create-pattern $opt_(pattern) $opt_(replication)
        $self post-process $conf(param)
}
```

7.1.2 Topology

The topology functions is used to define the topology of our simulated network as well as the different processes which can be performed on this topology. The most important function in this set is the *create-topology* function. This set of functions can create the core network by calling the constructor defined in the test-topologies.tcl file. It also defines the behaviour of this core network. The following code can be used for creating a network:

```
TestSuite instproc create-topology {} {
    $self instvar opt_ ns_ corenetwork_ net_

    set corenetwork_ [new Core/$net_ $ns_ $self $opt_(routelength) $opt_(corelink)]
# routelength and corelink initialised in template.rc

    . . . .
}
```

Some specific processing can also be performed depending on the type of the structure. The following example will create a distinction between edge and core routers if the structure is of *diff-serv* type. The constructor to create edge routers is defined in the test-topologies.tcl file.

```
TestSuite instproc create-topology {} {

    . . . .

    if {$net_ == "Diffserv"} {
        for {set i 0} { $i < [llength $opt_(edgelink)] } {incr i} { #edgelink initialised in templa
            set edgelinkparam [value $i $opt_(edgelink)]
            set edge_($i) [new Edge/$net_ $ns_ $self $corenetwork_ $edgelinkparam]   #create edge c
}
     }

}
```

7.1.3 Sources

This set of functions is used to create the patterns and to connect the source and destination nodes of pattern to the edge network or to the corenetwork. These patterns will be either of diff-serv or classic type. The major function is the *create-pattern* function which takes two arguments: *pattern* and *replication*. These two parameters are initialised in the resource file(section 2.2). A new pattern is created by calling its constructor defined in the test-source.tcl file. The connection between the sources and destination to the network is done by calling the *connect-pattern* function which takes as argument the queue management algorithm used. The connect-pattern function is also defined in test-source.tcl file.

```
TestSuite instproc create-pattern {pattern replication} {

$self instvar pattern_ patterntype_
```

```
    for {set i 0} { $i < [llength $pattern] } {incr i} {
        set pat [new SourcePattern/$patterntype_ $self [lindex $pattern $i] $replication]   # creat
        set pattern_($i) $pat
        $pattern_($i) connect-pattern  DropTail          #access queue managed according to DropTai
}
}
```

7.1.4 Post-process

The post-process functions are used to define the processes which are executed at the end of the simulation. The major functions in this set are: *post-process*, *tracequeue* and *tracepattern*. The tracequeue and tracepattern functions process parameters by object types. The tracequeue function is used to perform post-processing on objects in core network such as *Queue* whereas the tracepattern function is for objects in the pattern(agent or conditionner node).

For example, the tracepattern function is as follows:

```
TestSuite instproc tracepattern {} {

    $self instvar param_ pattern_

    . . . .

    foreach item  $param_ {
        set typeobj [lindex $item 0]
        set totrace [lindex $item 1]

        # check if typeobj is an agent type then on= true
        set on [expr [lsearch -regexp [Agent info subclass] $typeobj] != -1]
        $self processtracevar $typeobj $totrace $on
}
}
```

The post-process function calls the *tracepattern* and *tracequeue* functions. This function is called by the *init-model* function at the initialisation of the model.

```
TestSuite instproc post-process {paramlist} {
    $self instvar param_

    set param_ $paramlist

        $self tracepattern
        $self tracequeue
}
```

7.1.5 Tracing

This set of functions is used to perform the tracing of the curves of the different parameters after the processing. It contains the *use-newtrace* function.

8 Model Extension

The model can be extended by writing different test-suites(test-suite-xxx.tcl) and using the functions defined in the library as well as other functions written by the modeler. The writing of a new model can be divided into 5 main tasks which are to:

load the libraries : The libraries are loaded as follows:

```
set currentdir [pwd]
catch "cd $env(LIBSUITE)"

source test-suite-template.tcl
catch "cd $currentdir"
```

define set the default parameters : These parameters include the default parameters which are defined here instead of being passed by the command line.

```
# Parameter given by command line

set opt(stoptime) 20
set opt(verbose)  1 ;# 2 for  print log file
set opt(nam) 0

# set default parameters

Queue set limit_ 50
Queue/RED set mean_pktsize_ 1500
```

write the base class and base functions : The base class is generally a sub-class of the *TestSuite* super-class which has been defined in the library. It contains the init function which is the constructor for the simulation model and proper to the model. The *init* function calls the *init-model* function which is the template used for all the different test-suites. An example of the base class would be:

```
Class Test/BestEffort -superclass TestSuite

Test/BestEffort instproc init {} {
        $self instvar net_ patterntype_

        set net_  "BestEffort"
        set patterntype_ Classic

        $self next
        $self init-model
}
```

Figure 7 represents different function calls.

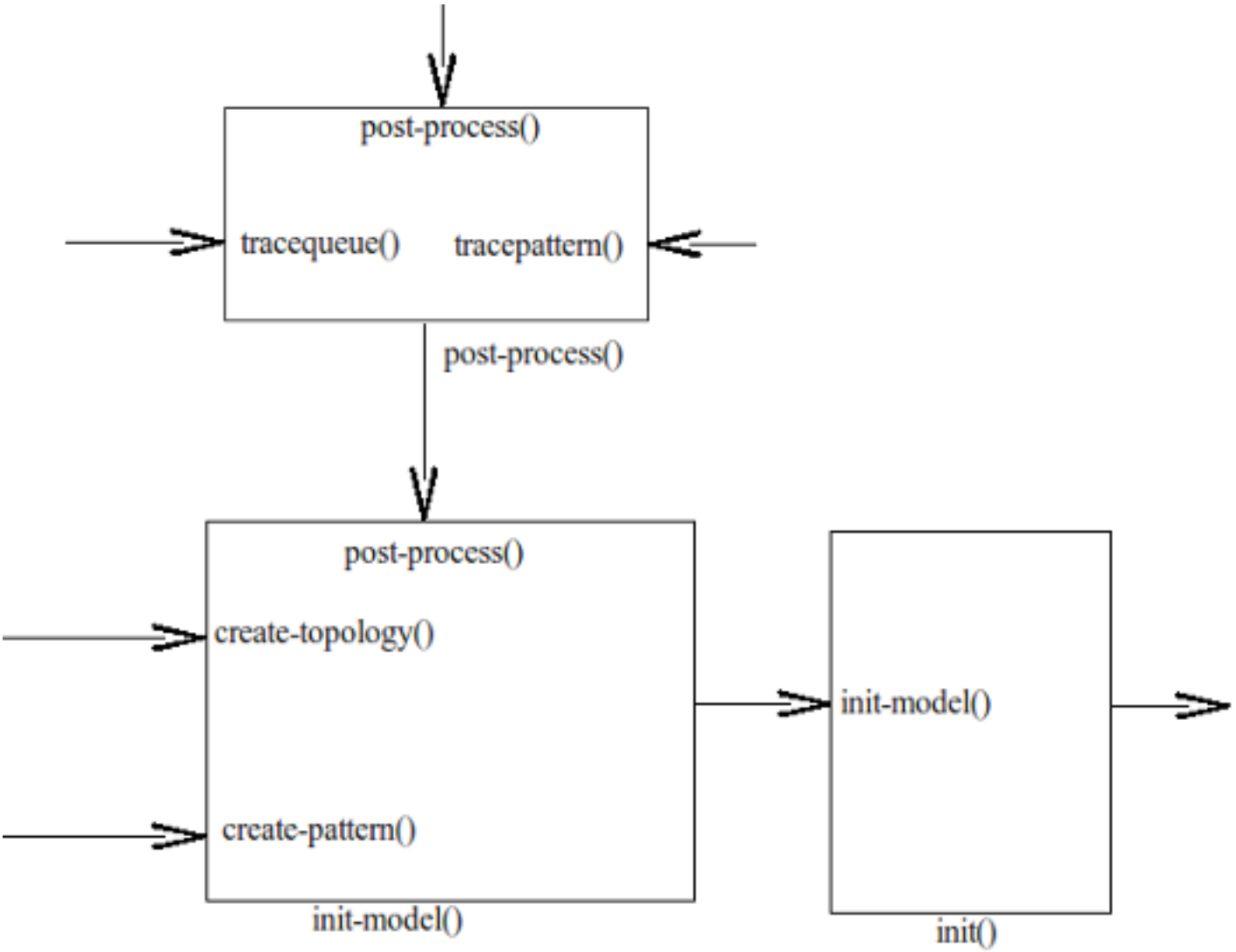

Figure 7: Function calls

write the overloaded functions : The overloaded functions are written based on the functions which have been defined in the libraries but they adapted to the model. These functions inherit the initial caracteristics of the functions defined in the libraries. For instance, the create-pattern function can be overloaded to obtain a particular pattern for a model as shown below:

```
TestSuite instproc create-pattern {pattern replication} {
        $self instvar pattern_ patterntype_ prefix_

        set queuetype [lindex [split $prefix_ _ ] 0]
        for {set i 0} { $i < [llength $pattern] } {incr i} {                                    # c
        if {[regexp short "[lindex $pattern $i]"] && $queuetype == "FavorTail" } {
            set pat [new SourcePattern/seq $self [lindex $pattern $i] $replication]             # t
        } else {
            set pat [new SourcePattern/$patterntype_ $self [lindex $pattern $i] $replication]   # t
            }
        set pattern_($i) $pat
            $pattern_($i) connect-pattern  DropTail
        }
}
\begin{description}
```

write the specialized classes : These specialized functions are new functions which are proper to the new model. For example, these can be specific post-process functions which differ from one model to another.

8.1 Command line

The new simulation model is launched by a command line which is almost similar to that shown in section 6. The arguments to control the execution of the simulation are given by the command line.

8.2 Model development

The existing model can be extended by developing other features to add up to the existing ones. These features may include: agents, traffic generators, queue class managers and other network objects. The ns simulator is written in C++ but uses OTcl as a command and configuration interface. The agents and other features are generally implemented in C++. The procedure for creating an agent is detailed in Chapter 10 in [3]. The other features can also be created by refering to [3]. For any relevant documentation to C++, refer to [4].

Part III
Appendixes

A Settings

This section details the class variables of the agents. These variables are instanced at the same time as the agent. They are used to configure the agent or to represent the states. The changes of state of a state variable can be preserved in a parameter trace file (cf. trace analysis section). The configuration parameters are used with agentparam parameter in the traffic pattern description (cf. component definition section).

A.1 Agent TCP: Source

The emitter version is indicated in the TCP version.

Configuration parameters

Agent/TCP	Tahoe version
Agent/TCP/RFC793edu	educative Reno version
Agent/TCP/Reno	
Agent/TCP/Newreno	
Agent/TCP/Vegas	
Agent/TCP/Sack1	

Table 2: TCP versions

windowInit	cwnd initial value
window	Receiver window
maxcwnd	max # cwnd can ever be; if 0 no bound
windowOption	CA algorithm: 1 standard, 8 High speed TCP
max_ssthresh	max value for ssthresh, for limited slow-start
packetSize	fixed packet size
useHeaders	boolean: Add TCP/IP header sizes
ecn	Explicit Congestion Notification
singledup	Send on a single dup ack (limited transmit)
bug_fix	1 avoid unnecessary multiple-fast-retransmit
maxburst	max # packets can send back-2-back (if 0 no bound)
overhead	add random time between sends
partial_window_deflation	1 to deflate (cwnd + dupwnd) by amount of data acked (1 by default to reflect RFC 3782)
partial_ack	for tcp sack: set to "true" to ensure sending a packet on a partial ACK
newreno_changes1	1 to allows the retransmit timer to expire for a window with many packet drops (the retransmit timer reset only for the first partial new ack) (1 by default to reflect RFC 3782 Impatient variant)
timestamps	boolean; use RFC1323-like timestamps

For modeling SYN and SYN/ACK packets

syn	1 for modeling SYN/ACK exchange
delay_growth	delay opening cwnd until 1st data recv'd

For timer

rfc2988	boolean, to be RFC2988-compliant: clamping minrto before applying t_backoff
maxrto	max value of an RTO
minrto	min value of an RTO
tcpTick	clock granularity
rtt	sample round trip time
srtt	smoothed round-trip time
rttvar	variance in round-trip time
backoff	current multiplier, 1 if not backed off

State variables

dupacks	number of duplicate acks
t_seqno	number of last transmitted packet
seqno	highest seqno "produced by app"
maxseq	max (packet) seq number sent
cwnd	current congestion window (packet)
ack	highest ACK received
recover	highest pkt sent before dup acks, timeout, ECN echo
ssthresh	Slow start threshold (packet)

A.2 Agent TCP: Destination

The version of the destination is indicated in the tcpsink parameter:

Agent/TCPSink	Classical receptor
Agent/TCPSink/DelAck	Delayed ACK Agent parameter of DelAck version interval:number of seconds to wait between ACK
Agent/TCPSink/Sack1	

A.3 RED

bytes boolean whose value 'true' activates the byte-mode functioning (for the case where the packet size affects the probability of packet rejection).

queue-in-bytes boolean whose value 'true' indicates to calculate the mean size of the waiting list in bytes rather than in packets.

thresh the value of the minimum floor (minth) in packets by default. Value for which the mean length of the waiting list must be superior so that loss can occur.

maxthresh the value of the maximum floor (maxth) in packets by default. Value for which the mean length of the waiting list must be superior so that all the packets are dropped.

mean_pktsize approximate estimation of the mean size of packets in bytes. This parameter is used in calculating the mean size of the waiting list after an activity period.

q_weight weighting for calculating the exponential mean of the size of the waiting list.

linterm the maximum drop frequency (maxp-1). Maxp determines the aggressivity of RED when congestion occurs. For example, a value of 10 means a drop for each 10 packets maximum.

setbit boolean where the value true indicates that the packets are marked rather than dropped.

B Utilities files

The `bin` directory contains the utilities files. These files are the scripts which are common to all the users and can be used for specific processing. The `bin` directory is located in the `0-Lib` directory and contains the following scripts:

completedflow.py filters packet trace file in order to keep flow with a beginning and an end. The flows starting in the transition period are not considered. The script takes two arguments:

```
python completedflow.py [options] [tracefile]
```

The options include:

- -h: for help
- -t: transit time
- -f: trace file is by flow
- -r: filename of rejected flow
- -c: cut segment without ack at the end, transform flow in a completed flow

The tracefile argument is the parameter file.

e_blockcut.sh cuts a block from a file organized in blocks. The index begins at 0.

equalfid.sh makes a trace files with the same fid in each trace file.

fid_activity.sh generates the data activity file from a packet trace file and a list of fid.

fidc.sh counts the number of fid in a packet trace file.

fid_dst.sh traces the activity of the flow at the destination.

fid_filter.sh erases lines in a file according to the value of a field given by another file.

fid_flow.sh prints the flow characteristic for a specific flow (fid) list.

fid_grep.sh prints the pkt traces for a specific flow (fid) list.

fid_list.sh counts the number of fid in a packet trace file.

fid_src.sh traces the activity of the source.

node_filter.sh to retain the traffic between 2 node id. It takes two arguments. The casting argument is the source-destination pair which are the two node id to be considered. The second argument is the parameter file.

ns_env.sh defines the environment in which the simulation takes place. It loads the libraries, specifies the directory of modeling interface and the tcl script directory.

ns_load.sh the shell-script launching the ns simulation. It takes the parameter file as argument.

ns_param.tcl reads the parameters for the script file.

oneway.sh keeps the traffic in one way only. The reverse traffic is deleted. It takes two arguments: nodeid argument which is the nodeid of the last node for that way and the tracefile argument which is the parameter file.

prepare_pkt.sh keeps the flow in one way and erase incomplete flow. It is called when the simulation has finished. The function takes 4 arguments:

```
prepare_pkt.sh [options] <tracefile> <onewaynode> <transittime>
```

The options are:

- -h: for help
- -v: verbose
- -c: clean up all temporary files

The tracefile argument is the parameter file, the onewaynode argument is the maximum id of the node to keep and the transittime argument is the simulation time to ignore. The prepare_pkt.sh script calls two other scripts: oneway.sh and completedflow.py

References

[1] Ousterhout, J.K. (1994). Ed.: Addison-Wesley Publishing Company, 458 p. *Tcl and the TK Toolkit.*

[2] D.D. Clark, W. Fang, "*Explicit Allocation of Best Effort Packet Delivery Service*", IEEE/ACM Transactions on Networking, August 1998, Vol 6. No. 4, pp. 362-373.

[3] UC Berkeley, LBL, USC/ISI,Xerox PARC, *The Ns Manual*, The VINT Projec, April 2002.

[4] Frank B. Brokken, Published at the University of Groningen, ISBN 90 367 0470 7. *C++ Annotations Version 8.2.0*, 1994-2010.