n*script*

Version 1.0a

USER'S MANUAL

Enrique Campos-Náñez [1]
Department of Systems Engineering
University of Virginia
ec3z@virginia.edu

March 13, 2001

[1]This work is supported by Consejo Nacional de Ciencia y Tecnologia through scholarship 68428, and the National Science Foundation through grant ANI-9903001.

Contents

Chapter 1

Introduction

nscript is a simple tool designed to graphically build simulation scripts for the network simulator (*ns* , see [BBE$^+$99]). It allows to edit graphically, using a block metaphore, the elements involved in a simulation, and produces the necessary Tcl script required to run it under *ns* . Specifically, it is designed to aid in the following tasks:

- Build the topology for a given simulation. **nscript** lets you graphically build a topology.
- Create and configure the transport agents and network applications, and connect them to the physical topology.
- Schedule simulation events. Through the use of special graphical objects, **nscript** makes it possible to trigger events at particular moments, or a sequence of moments in the simulation.
- Trace and plot values of interest.

nscript is also extensible, allowing you to add new classes of objects to the graphical environment. By organizing the classes of simulation objects into libraries, new classes can easily be integrated and used in the graphical environment.

1.1 Requirements

The **nscript** application was developed using Java 2, and makes use of the JFC library (Swing) to implement its user interface. To use **nscript** it is necessary to download version 1.3 of the JDK from *http://www.java.sun.com/*.

To run scripts and visualize the corresponding animations, it is necessary to install the *ns*, and the nam application, which can be found at: *http://www.isi.edu/nsnam/ns/*. The **nscript** application runs in all platforms with a JVM that supports Java 2 classes, but in order to run *ns* scripts directly, a Unix platform is recommended. If such platform is not available to use, you can still create a simulation script, and manually run it in *ns* .

To plot some of the file traces, you might consider installing **gnuplot** (http://www.gnuplot.org), **xgraph** (http://www.atl.external.lmco.com/proj/csim/xgraph/xgraph.html) or something similar.

1.2 Installation

The package is distributed as a zip file that unzips to a single directory (nscript-1.0). The unzipped directory should contain:

- **README** Information on the latest changes.

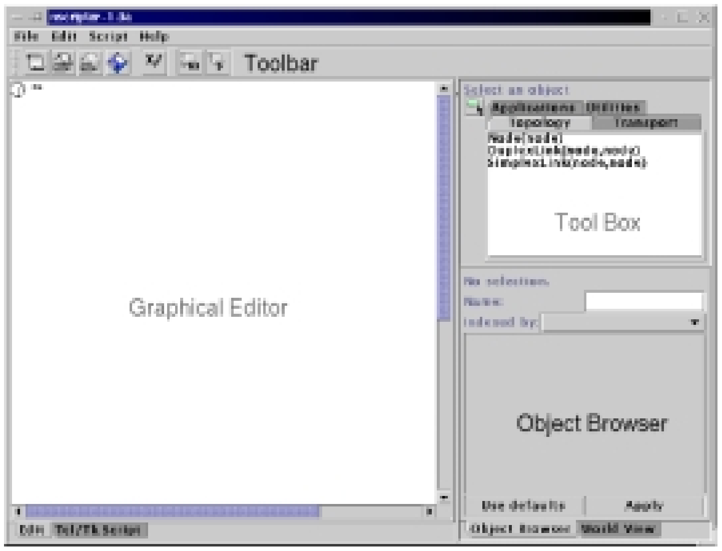


Figure 1.1: The nscript application showing: the graphical editor, the toolbox pane, the object browser pane, and the toolbar.

- **bin** Directory containing the jar files with the java classes that conform the application, as well as the following directories:
  - **settings** Containing the initial settings for the application (libraries loaded by default, and the environment default settings).
  - **lib** Libraries of object templates to use in the simulation.
  - **figs** Icons for the toolbar and other stuff.
- **src** The java source code for the application.
- **examples** Examples from the tutorials in this manual.
- **man** This manual.

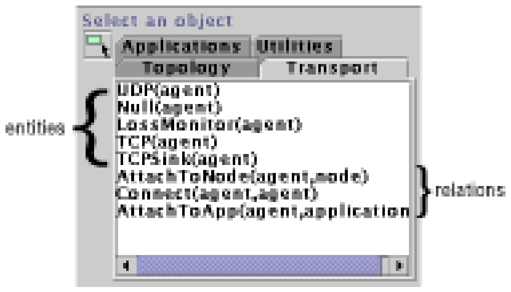
Once the downloaded file is unzipped, change to the bin directory and run the program with:

```
java -jar nscript.jar
```

1.3 A first look at the nscript GUI

Figure 1.1 shows the **nscript** graphical user interface. The **nscript** application consists of four main components, and some auxiliary controls. The main components are:

- **Graphical Editor** The place where the edition of the script takes place. At the beginning of a session, it show a single object labeled **ns** which represents the simulation environment and provides control of the environment settings, such as network dynamics, tracing, nam-tracing (animation), and simulation time. More detailed information on how to configure the environment is provided in the reference to the libraries, in Part II of this manual, but to see the options that are available, select the select tool and



UDP(agent)
Null(agent)
LossMonitor(agent)
TCP(agent)
TCPSink(agent)
AttachToNode(agent,node)
Connect(agent,agent)
AttachToApp(agent,application

Figure 1.2: The ToolBox showing the **Transport** library. Available entity components belong to the base class *agent*, while relations provide ways to relate (transport) agents to other objects. The *AttachToNode* relation that attaches a transport agent to a particular node, the *Connect* relation that connects to agent objects, and the *AttachToApp* that attaches a transport agent to a given application.

click on the **ns** object; a list of its attributes and their current values is shown at the lower-right corner of the application's window, which we will call the **Object Browser**.

- **Object Browser** This pane contains the controls through which is possible to configure any object in a simulation. It shows the name and class of the selected object, as well as other attributes that can be tuned for a simulation. It also contains a **Use Default** button can be used to restore the default values for the parameters of any object, and an **Apply** button to apply the current values to the object being edited.
- **Toolbox** This pane, located at the upper-right corner of the main window, shows the available tools that can be used to create a simulation. The select tool can be used to select and move objects around. On the other hand, a tabbed pane shows the available object *templates*, or classes, that can be used to create objects for the simulation. This objects are classified into libraries, corresponding to their use in a simulation. The default libraries are **Topology, Transport, Applications** and **Utilities**, but additional libraries can be created and added to the environment by using the **Open Library...** option in the **File** menu.
- **Toolbar** Where most of the commands can be launched, for actions like opening / saving a script, exporting the tcl script, or opening a library of simulation objects.

1.3.1 Working with the Editor

The process of creating a simulation involves selecting the desired object in the ToolBox panel, and using the mouse in the **Graphical Editor** to add objects of the selected type into the simulation.

There are two generic kinds of objects, which were model after the way things work in *ns* where a typical simulation script consists of creating and configurating simulation objects, and relating them to do something usefull. For example, to create a topology in *ns* one has to add **node** objects, and then relate them by adding a **link** between them. This process is abstracted as creating *entities* (in this case the nodes), and then a *relation* between them. This abstraction works well for other tasks common in ns, such as creating transport agents (entities), and attaching them to nodes (relation, in this case between a trasnport agent, and a node).

The **ToolBox** lists the available classes, but also gives hints on how to use them. Every *entity* class shows in parenthesis a *base class* to which it belongs. Think of it as a family of classes that share a common behavior. On the other hand *relation* classes will list the base class for the objects it can relate.

For example, as shown in figure 1.2, the UDP, Null, LossMonitor, TCP, and TCPSink classes belong to the same *agent* family or base class. The relation object *AttachToNode* requires an entity of bass class *agent*, and an entinty of class *node*, in that order. The editor checks this, to prevent syntactic errors.

Play for a little while with these tools, and when you feel comfortable, get ready for the first script in the next chapter.

1.3.2 Saving and opening files

You can save your script into a **nscript** file format, that can be read back in a future session. That is accomplished by the normal **Save As...** option in the **File** menu. The usual *Save* and *Open* dialog boxes are used to accomplish this.

Another possibility is to *export* your script to to Tcl, which can be run in *ns* directly.

It is important to remember that the Tcl files can not be read back into the **nscript** environment. Unfortunately at this point we were not able to code that functionality into the package.

1.4 What's in this manual

In this manual, we show how to use **nscript** to build simple simulations. To make the process easier, we decided to base the examples in this manual on the very well known Marc Greiss' tutorial (see this reference and if possible have a printed copy of it [Gre]). This manual contains.

- **Part I: Tutorials** A set of tutorials designed to get you up and running with using the script generator.
- **Part II: Libraries** A detailed reference on how to edit, enhance, and create new libraries for the scripter. Additionally, the appendices contain a description of the file formats used by **nscript** , and other technical information to improve on it.

nscript is a work in progress, and we appreciate your comments on it.

Contact information:

Enrique Campos-Náñez / Stephen D. Patek
Department of Systems Engineering
Olsson Hall / SEAS
University of Virginia
e-mail: ec3z@virginia.edu

Part I

Tutorials

Chapter 2

Tutorial 1: the first tcl script

To start, we will create a very simple simulation connecting only two computers and establishing a constant bit rate flow from one node to the other. The libraries in **nscript** follow a layer approach, so we will follow a layered approach to building the script, first creating the topology, then the transport layer, then finally the application layer, and finally simulation events.

If you have a copy of Marc Greiss tutorial, it is interesting to compare the generated by **nscript** to the one in Greiss' introduction. You can do this by clicking on the **Tcl/Tk Script** tab at the bottom of the **Graphical Editor** at any time during the editing process. This will switch to a window showing the Tcl code that is required to produce produce the same result in *ns* . Try using it after each of the layers of the simulation has been created, to see how things are done using Tcl.

2.1 Building the Topology

First, select the **Topology** library by click at the tab with the same name that can be found on the ToolBox.

1. Click on the list of objects to select the **Node** class. In the rest of this tutorial we'll refer to "Select the *ClassName* class from the *LibraryName* library", as 1) selecting the *LibraryName* tab and 2) Selecting the *ClassName* list from the options.
2. On the **Graphical Editor** click on the place where you want a new node added, an small icon Node 0 appears.
3. Repeat the operation to create a new Node, called *Node1*.
4. Select the **Duplex Link** object from the **Topology** library, and create a link between the node objects created in the previous steps, by clicking in one node and dragging to the other node.

By the end of these steps, the **Graphical Editor** window should look like Figure 2.1.

2.1.1 Configuring the Link

Suppose, that we want this newly created link to have a banwdith of 1Mbps, a propagation delay of 10ms, and a drop tail queue discipline (FIFO). To configure this:

1. Click on the (**Select Tool**) to change to select mode.

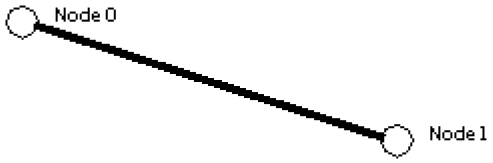
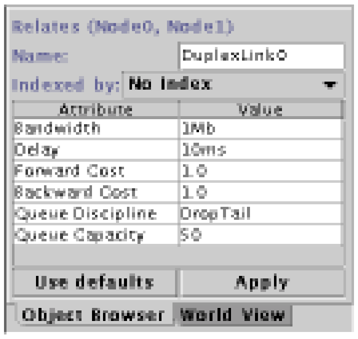


Figure 2.1: Topology for the first example.



Relates (Node0, Node1)

Name: DuplexLink0

Indexed by: No index

Attribute	Value
Bandwidth	1Mb
Delay	10ms
Forward Cost	1.0
Backward Cost	1.0
Queue Discipline	DropTail
Queue Capacity	50

Figure 2.2: Configuration of the link for the simulation.

2. Click on the link. This will have the effect of selecting it which is shown by changing its color to blue. Notice that once the link is selected its properties appear in **Object Browser**, see figure 2.1.
3. Select the *Bandwidth* field, and change the settings from 10Mb to 1Mb. In the same way, select the *Delay* field, and change it to 10ms from 20ms.
4. Click on the *Apply* button to save the settings. Figure 2.2, shows the how the attributes of the link should look like.

2.2 Creating and Configuring Transport Agents

Once the topology has been created, we need to add transport agents to actually send information packets. This agents must be attached to nodes and connected to each other in order for the simulation to succeed. Let's do that:

1. Select the *UDP* class from the *Transport* library.
2. Add a UDP agent on top (north) of Node0.
3. Select the *Null* class from the *Transport* library. This entity class, corresponds to a sink that will receive the packets from the UDP0 session created above.
4. Add a Null agent (Null0) north of Node1.
5. Select the *AttachToNode* class from the same library, and attach agent UDP0 to Node0, by clicking in the center of agent UDP0 and dragging to the center of node Node0, and release the mouse then. This particular order (agent $\rightarrow$ node) is required. To know what is the order required look in the ToolBox, at the

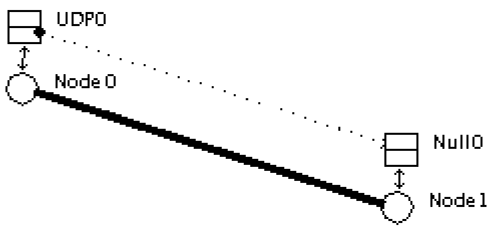


Figure 2.3: Configuration of the link for the simulation.

line for $AttachToNode$ class. It displays **AttachToNode (agent, node)**, meaning that to create a relation of this class, we must first provide an agent, and then a node.

6. Attach agent Null0 to Node1 in the same way.
7. The last step is to connect the two agents, which can be achieved using the *Connect* class from the *Transport* library. Select this tool and connect agent UDP0 to agent Null0. This connection represents the *logical* connection between two agents that don't have a direct physical connection.

By the end of this section, your diagram should look like figure 2.3. This represents two computers directly connected through a physical link, and two transport agents, each attached to one of the computers (routers), and logically connected to each other. The next step is to create an application that sends some data. In this case a CBR flow.

2.3 Creating and Attaching Applications

1. Select the **Application** library, and create a new **CBR** application; place it north to the UDP agent.
2. Select the **Transport** library again, and select the **AttachToApp** class. Attach agent UDP0 to application CBR0, by creating a line between them. Here the order *agent* $\rightarrow$ *application* is required.
3. Use the **Object Browser** to configure the CBR application to send 500 byte packets each .005 seconds. This involves selecting the CBR0 object first, and editing the attributes in the **Object Browser** .

By this point, your diagram should look like figure 2.4.

At this point, everything is set up with the exception that nothing interesting will happen, because no events have been scheduled. For some data to be transmitted we need to tell the CBR application to start transmitting at some point in the simulation time.

2.4 Scheduling Simulation Events

To add a simulation event:

1. Add two **Timer** objects, that can be found in the **Utilities** library.
2. Set the names of the first of them to .5, and the second 4.5. These are the times when we want something to happen. To change the name of an object, select it using the select tool, and the write the new name in the corresponding text field (see figure 2.2, and press the *Apply* button.

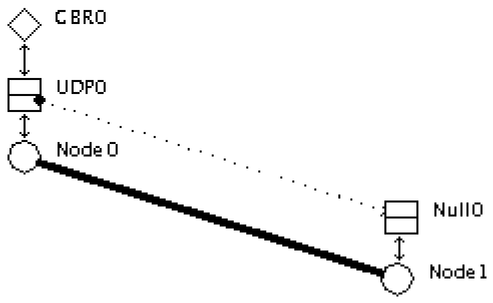


Figure 2.4: Configuration of the link for the simulation.

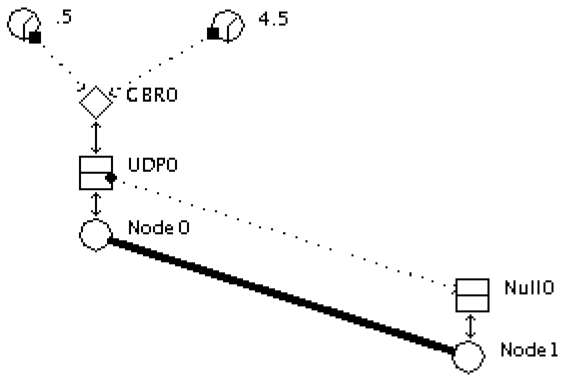


Figure 2.5: Diagram after the events are added.

3. Connect each of these objects (.5 and 4.5), to the CBR application object using an *ApplicationEvent* class. This can be found in the **Utilities** library as well. This objects relate a time and an application by "calling" any of the applications available commands. CBR applications support **start**, and **stop** among others. Your diagram should look like 2.5.

4. Select the connection between 4.5 and the CBR application, and set the Event attribute to *stop.* This has the following meaning: at time 4.5, applications CBR will stop sending info.

Save the script, using the **Save** option under the **File** menu. At this point you can run the script using *ns* directly from the **nscript** application by clicking at the run button in the toolbar ( ). You will be asked to give a name to the script, and little after that the nam (animation) window will appear. Note: this option does not work under Windows.

Chapter 3

Tutorial 2: Making it more interesting

In this tutorial, we will create a more interesting example (see “Making it more interesting”, in [Gre]). We will simulate two CBR flows sending information over different paths of a simple topology.

3.1 A Two Flow Model

3.1.1 Build the Topology

Follow these steps to obtain the diagram in 3.1.

The topology. We start by creating three nodes, and linking them together with duplex-links with 1Mbps capacity, and a delay of 10ms.

- Use the **Topology** library to create the topology in figure 3.1.
- Using the **Object Browser**, configure each of the links to have:
  - Bandwidth of 1Mb.
  - Delay of 10ms.
  - DropTail scheduler.

Transport Layer Here we create a two UDP-based flows, and set them to following different paths in the topology.

- Use the **Transport** library to add two UDP agents.
- Attach them to nodes 0 and 1 (see 3.1), using the **AttachToNode** tool.
- Add two Null agents.
- Attach both of them to node 3, again using the **AttachToNode** tool.
- Use the **Connect** tool to make logical connections UDP agent 1 $\rightarrow$Null agent 1, and UDP agent 2 $\rightarrow$Null agent 2.

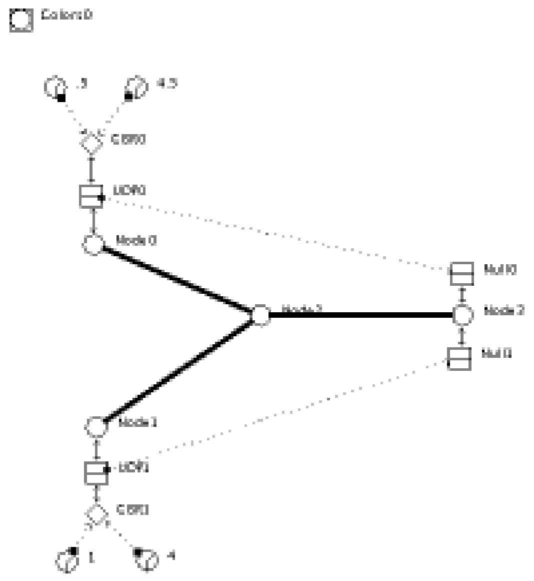


Figure 3.1: Topology for our second, more interesting example.

Application Layer We now create two CBR applications that will use the services of agents UDP1 and UDP2:

- Use the **Application** library to add a couple of CBR applications.
- Use the **AttachToApp** from the Transport library to attach UDP1 →CBR1, and to attach agent UDP2 →CBR2 together (see the finished diagram in figure 3.1).

Packet Colors Now, we will like to be able to distinguish the packets of the two flows since they will share a common link in their way to their destination.

- Use the **Utilities** library to add a **Colors** entity object. This object configures the simulation to use colors to identify the packets according to their flow ID number.
- Configure the Colors0 object in the **Object Browser** to set Color1 to red, and Color2 to green, using the popup list in the **Object Browser** .
- Configure UDP flows, to have a Flow ID of 1, for agent UDP0, and 2 UDP1, using the **Object Browser** . This will set the ID for each of the packets produced by these agents. Nam will map that ID to a color according to the settings of object Colors0.

Events The last step is, again, to tell the simulator when things start and stop happening.

- Use the **Utilities** library to add four Timer events, and set the names of them to: .5, 4.5, 1, and 4.
- Use the **ApplicationEvent** object to link each of the timers .5, and 4.5 to application CBR0, and timers 1, and 4 to application CBR1 (see diagram 3.1).
- Configure the links from 4.5 →CBR0, and 4 →CBR1, by settings their events to **stop**. This will tell CBR0 to start at .5, and stop sending packets at time 4.5, while CBR1 will start at time 1, and end at time 4.

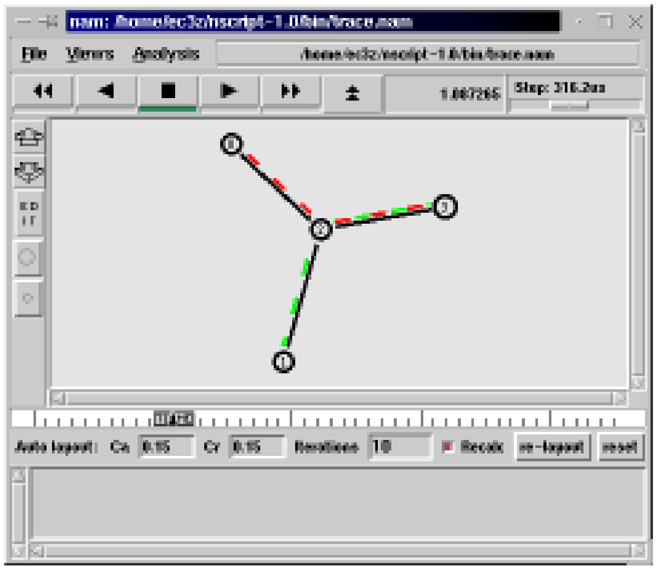


Figure 3.2: Animation of the simulation using the network animator (nam).

Run the simulation in *ns* and at the end you will see the corresponding nam animation window, shown in figure 3.2. The packets for the flow from UDP0 to sink Null0 should be colored red, while the flow in the other path should show a green color. Using the **Object Browser** try reconfiguring object **Colors0** and rerun the simulation.

3.2 Generalizing to an n-flow model

You may have noticed that the task of creating and configuring each of the flows was practically the same. In this section, we will develop a more generic simulation, where the number of flows can be easily changed with the help of arrays.

A note on arrays In mathematics, to simplify the notation, one typically creates an index, such as in $\{x_i\}_{i=0}^{N}$. In **nscript** you can do the same thing, to create . To avoid repeating the code over and over, you can define an index that can be shared by several objects in the simulation. Index sets, called arrays -as they are called in the *ithink* simulation environment, from where the idea was borrowed-, can only be designated for *entity* objects. *Relation* objects use the definitions of the entities it connects, according to the rules shown in figure 3.8. Object arrays are represented in the diagram as a stack of objects, instead of a flat one, to clue the user about this indexing. The effect of arrays on entities and relations is shown in figure 3.2. Also refer to figure 3.8.

3.2.1 First, a base model

First, start a new simulation script, and based on the previous tutorials, build the diagram shown in figure 3.2.2. You can even use the model from the previous example, but taking away node1 and objects UDP1, CBR1, Null1, and timer objects 1, and 4. In the new diagram a single flow is defined. Now, the idea is to use index sets to build a model that contains multiple flows, without having to repeat the same drawings.

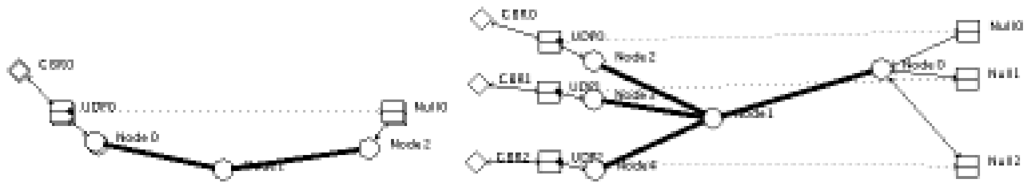


Figure 3.3: Equivalent definitions of the model, using arrays (left), and withoutarrays (right).

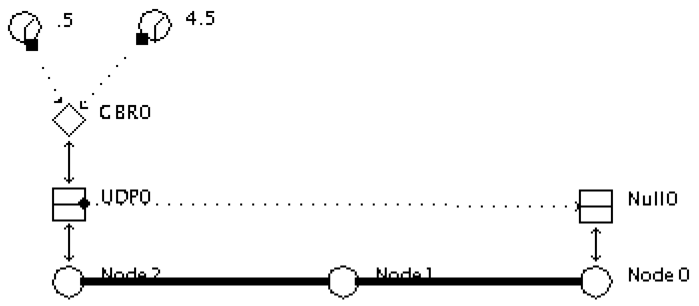


Figure 3.4: Base for the more general simulation model. Notice there is only one flow there.

3.2.2 Creating an array

An array is specified by a name, which has to be a valid variable name for Tcl (anything without spaces or special characters), and the number of copies it will contain. To create a new array called *users*,

1. Open the array editor, by selecting the **Edit Arrays...** option from the **Edit** menu. That will bring up the window shown in sreenshot 3.2.2.

2. Click on the **Add** button to add a new array definition.

3. Edit the array name, and change it to *users*. The array editor window should look like 3.2.2.

4. Close the array editor window, by clicking at the **Close** button.

Once this is completed, the array will be stored as part of the script.

Indexing the Existing Objects Select the Null0 agent using the **Select Tool**, and click on the popup labeled **Indexed by**. Select the entry showing the *users* array, and then click the **Apply** button.

This tells the editor to consider the Null0 agent not as a single object but as an array of objects indexed by the set *users*. Once this is done, you should notice that the array is represented as a small stack of agents instead (see figure 3.7).

Repeate the same indexing process for the following object: UDP0, CBR0, and Node0. Once this is done, the simulation will be set-up in a way that simple objects will be replaced by arrays of them, and whenever a relation exists between a pair, the following rule is obeyed:

- If both objects are indexed by the same array, then a relation between corresponding objects will be created for each element in the array. This corresponds to a one-to-one relation between the elements of the two arrays of objects (see figure 3.8).

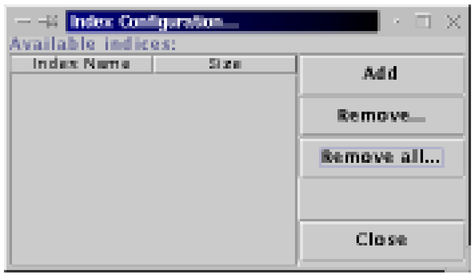


Figure 3.5: The array editor window with no arrays defined.

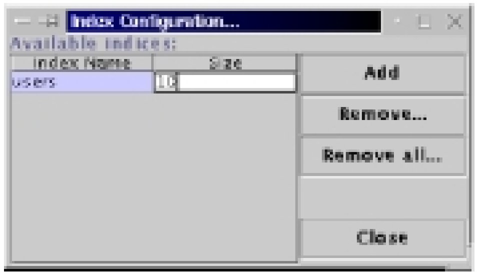


Figure 3.6: The array editor window after adding a new array.

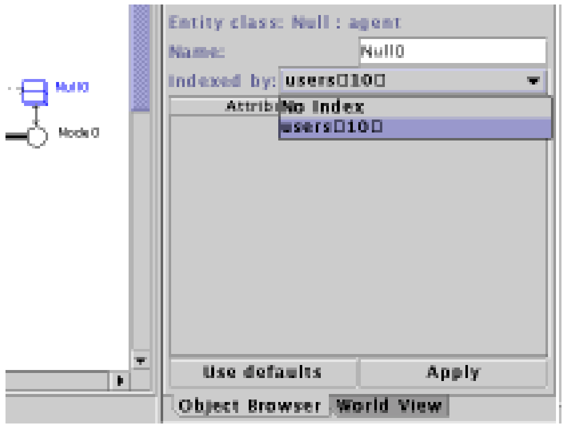


Figure 3.7: The **Object Browser** window used to make an object indexed by an array. Notice the change in the representation of the object from a plane object to a stack.

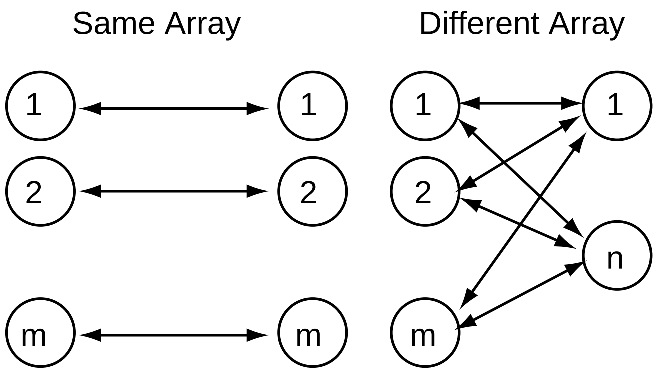
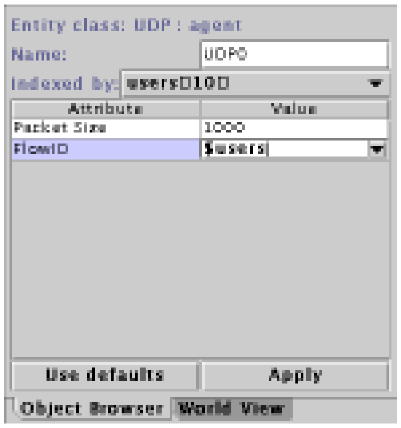


Figure 3.8: How a relation between two indexed entities is created. If the two entities share the same index set (left), a relation is created for each "copy" of the entities. On the other hand (right), if the entities use different arrays, a relation is created for each possible pair of entities between the two arrays.


Entity class: UDP : agent

Name: UDP0

Indexed by: usersD10D

Attribute	Value
Packet Size	1000
FlowID	$users

Figure 3.9: Setting one of the attributes to the value of the index.

- If objects are use different arrays, a all-to-all approach is used, and hence a relation between each possible pair of objects will be created.

For more details on the use of arrays, please read Part II of these manual.

Using the array index. If you look at the Tcl code generated by our diagram, the array name is actually used as an index, which can be exploited. To illustrate how we can use of the arrays index created above, suppose that we want to color each of the 10 flows differently. One way to do this is to use the value of the index (*users*) as the value of the parameter Flow ID of the agent UDP0.

Select the UDP0 object, and change the value of the **Flow ID** field to **$users** ($ is the Tcl syntax equivalent to saying the value of variable *users*), see figure 3.9.

Add a **Colors** object from the **Utilities** library. Then, save and run the script. After the simulation is over, you should see the nam window showing the animation, something like figure 3.10.

In summary, what we have done is to generalize a two-flow simulation, to a n-flow (in this case n=10). To see how the simulation works with more users, edit the array using the **Edit Arrays...** option, and change 10

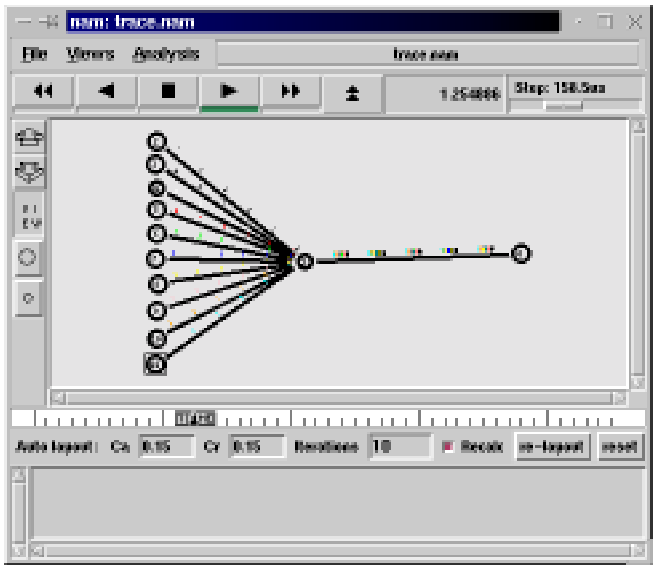


Figure 3.10: Animation window for our more complex simulation. Notice that instead of the three nodes originally added, 12 nodes are created. This is the result of using an indexed object. In this case the index is of size 10.

for a different number and rerun the experiment. technique is usefull to generate multiplexed traffic sources.

Chapter 4

Tutorial 3: Network Dynamics

In this chapter, we develop a simulation with the purpose of understanding the dynamics of network routing under events such as a link going down and coming back up. Please refer also to Marc Greiss tutorial, to fully understand the Tcl part of this simulation script.

4.1 The Base Topology and Model

As always, we start by creating the topology for our model, which in this case will be a cycle of 8 nodes, and duplex links with 1Mb of bandwidth and a delay of 10ms. This is shown in figure 4.1. Only one CBR flow will be present, connecting UDP and Null agents attached to nodes 0 and 3. We will also use two events to start and stop the flow at time .5 and 4.5, respectively.

4.2 Simulating a link failure

We want to simulate the network dynamics during the failure of a link. To make it interesting, let this link be the link between nodes 1 and 2 (Node1 and Node2, in the diagram), which is in the path of the CBR flow. In order to do this, we must:

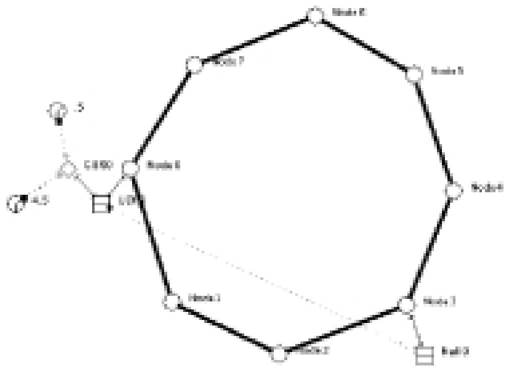
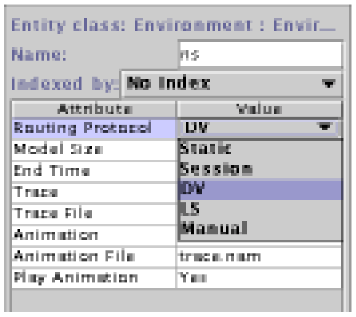


Figure 4.1: Layout for the network dynamics simulation.



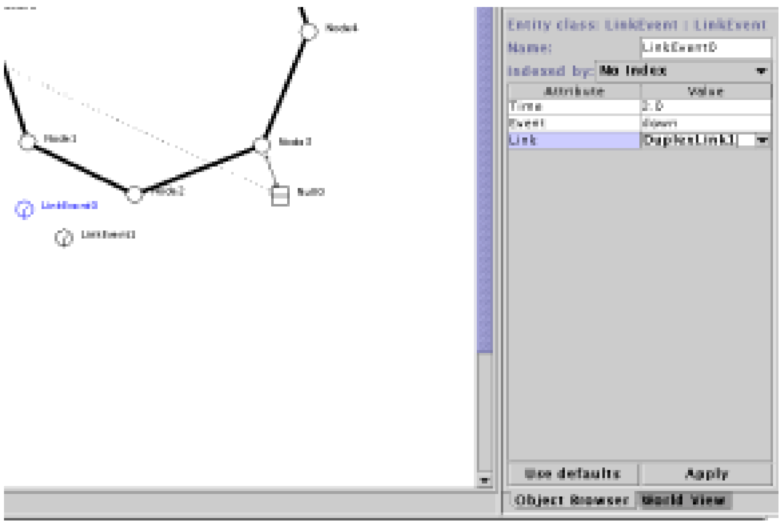
Entity class: Environment : Envir...

Name: ns

Indexed by: No Index

Attribute	Value
Routing Protocol	DV
Model Size	
End Time	
Trace	
Trace File	
Animation	
Animation File	trace.nam
Play Animation	Yes

Figure 4.2: Configuration of the ns (environment), to use a distributed distance vector (DV) routing protocol.



Entity class: LinkEvent : LinkEvent

Name: LinkEvent0

Indexed by: No Index

Attribute	Value
Time	2.0
Event	down
Link	DuplexLink1

Figure 4.3: Configuration of the down link event. Note that the actual name of the link has to be looked up using the object browser.

- Change the routing protocols used by the simulator, to a Distance Vector (dynamic) routing protocol. In this protocol, routing tables are updated dynamically. To do this in **nscript** we can configure the environment object, called **ns**, and located in the upper left corner of the editing window. We must set the **Routing Protocol** to DV (see figure 4.2).
- Add a couple of **LinkEvent** objects, to specify the times at which we want a particular link to go down, and up again. These objects can be found in the **Utilities** library.
- Configure the down event to take DuplexLink1 down at time 2.5. Here DuplexLink1 is assumed to be the name of the link connecting nodes 1 and 2, but the actual name of the link depends on the order in which the links were added. To find out the name of the desired link, select it using the **Select Tool**, and get the name from the **Object Browser** . Once the event has been configured, it should look like figure 4.3.
- Configure the up event to bring the link up at time 3.0 by configuring the second link event object.

Run the simulation, and observe the network dynamick in the time between the link failure and recover. The DV routing protocol is repairing itself, and is able to reroute the packets, as you can see in figure 4.4. It is

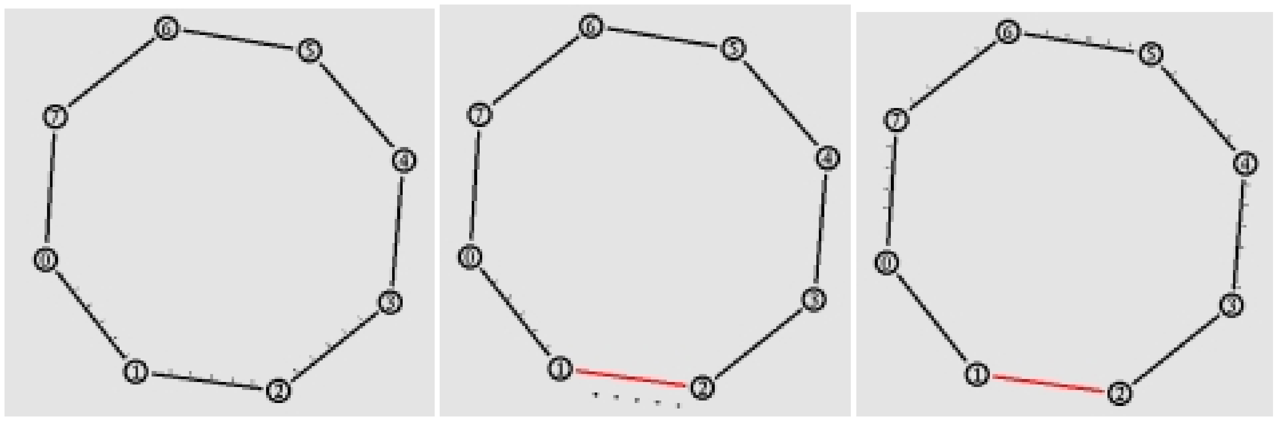


Figure 4.4: Result of the network dynamics simulation experiment. Using the distance vector (DV) routing protocol, the network is able to recover from loosing a link; the routing protocol also allows the network to detect when the link is back up.

also able to know that the link has been restablished, and reroute flow to take advantage of that.

Chapter 5

Tutorial 4: The Ping Agent

In Marc Greiss' tutorial a dummy "Ping" agent was created to get an insight of the process of adding new behavior to *ns* . This tutorial completes this part of the manual, showing how to add a new agent created for *ns* into the graphical environment.

To add the *Ping* agent to the graphical environment, we have to solve to problems:

- Create the *recv* procedure required by the ping object (see [Gre]), which is called when a ping packet is received.
- Create the *ping* class, and include it in the ToolBox, so the object can be used inside **nscript** .

To solve this problems, we will create two entity objects. One will represents the *recv* procedure that we want to add to the simulation, and the other the *ping* object that has to be created.

5.1 Creating a new procedure

Open a new text file using a text editor, and type the following definition. This will be discussed later. Save the file as **ping.lib** under the *nscript-1.0/bin/lib* directory.

```
ping
Ping
1.0a

!entity class PingRecv procedure 4:
begin
  Agent/Ping instproc recv {from rtt} {;
        $self instvar node_;
        puts "node [$node_ id] received ping answer from $from with round-trip-time $rtt ms.";
  };
end
```

This is a library file. The first two lines indicate the library name, and the name under which the library will be listed in the toolbox. The third line identifies the version of the library, for the purpose of version control. After those definitions, classes can be added according to the formats provided in Part II of this manual, in section 8.2. In this case we are adding a single class called *PingRecv* which represents the *recv* procedure required by the ping class.

In a few words, class declarations for entity classes, follow the following format,

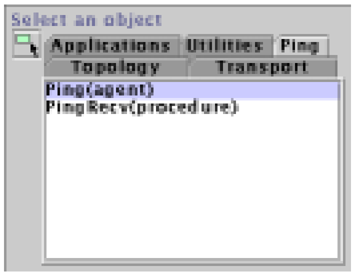


Figure 5.1: ToolBox after the ping library has been opened. Notice that the two classes defined show in the corresponding list box.

```
   [!]entity class <className> <drawing parameters>:
```

where is optional and has the mathematical meaning of unique, meaning that only one object of this class is allowed in a simulation. This unique feature is used in the *environment* object that is present in all simulations, labeled *ns*. Since this object defines the simulation environment, only one of those is allowed in a simulation.

In this case, since the receive procedure has to be declared only once in the code, we declare this class unique. By using 4 as the drawing parameter, we select a generic squared shape. Available shapes are described in figure 8.1, in the last part of this manual. To understand better the rest of the format of this file, take a look at Part II of this manual, specifically to the chapter about libraries.

5.2 The Ping Agent

Add the following lines to the previous text file:

```
entity class Ping agent 1:
begin
  set #name# [new Agent/Ping];
end
```

This adds a new entity class to the library, with name *Ping*, and with base class *agent*. As we mentioned before, the base class is used by **nscript** to constrain the user from building syntactically incorrect relations in a simulation. Save this file, and use the **Open Library...** option in the **File** menu, to open this file, and check that the syntax is correct. If it is not, **nscript** will simply not open the file, without further explanation. If it is correct, a new tab is added to the **ToolBox** window, containing the classes defined above (see figure 5.1).

5.3 Simulating Ping Messages

- Use the libraries to build a simple topology of two nodes connected by a Duplex-Link.
- After that select the **Open Library...** option from the **File** menu, or alternatively, click the open library button, in the **ToolBar** (), to open the file created in the previous steps, *ping.lib*.
- Use the **PingRecv** to create a single object, and place it close to the environment object, to remind us of its nature.

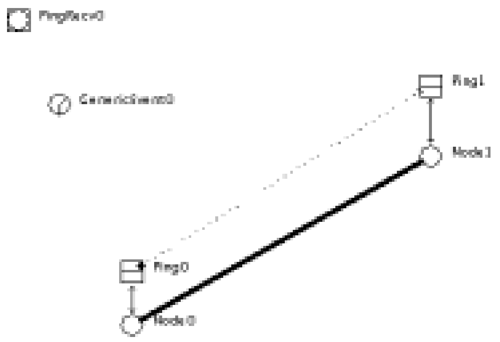
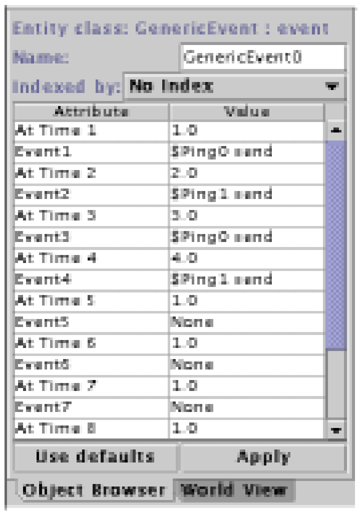


Figure 5.2: Completed diagram for the ping exercise.



Entity class: GenericEvent : event

Name: GenericEvent0

Indexed by: No Index

Attribute	Value
At Time 1	1.0
Event1	$Ping0 send
At Time 2	2.0
Event2	$Ping1 send
At Time 3	3.0
Event3	$Ping0 send
At Time 4	4.0
Event4	$Ping1 send
At Time 5	1.0
Event5	None
At Time 6	1.0
Event6	None
At Time 7	1.0
Event7	None
At Time 8	1.0

Figure 5.3: Configuration of the GenericEvent object to schedule multiple messages.

Once this is done, we can create the Ping agents in the same way that UDP, and Null classes were created. Also, since *Ping* has base class *agent* it is possible to attach it to nodes and to connect two of them, using the connect object:

- Create two ping objects.
- Attach the ping agents to nodes 0 and 1 using the AttachToNode object, and
- Connect them using the **Connect** tool from the **Transport** library.
- Use a **GenericEvent** object from the **Utilities** library. This object allows the schedule of multiple generic events (this events need not to be related to an object). Add one of this objects, so that your diagram looks like figure 5.2.

The only necessary step is to configure the GenericEvent0 object to schedule a sequence of ping messages. Select this object with the **Select Tool**, and configure it so that it shows like figure 5.3.

Run the simulation, and watch the animation at the times where the ping messages are scheduled to be sent.

An important NOTE. As a final point, try saving your files in the **nscript** format (nss). If you come back in a later session, and try to open this file, you will be prompted by a second **Open File** dialog box. In this case **nscript** is asking you to locate the file containing the libraries required by the file you are trying to open. Locate the *ping.lib* file and open it.

Chapter 6

Tutorial 6: Tracing A Variable of Interest

In this example, we will trace the value of the attributes of one of the objects in the simulation, as the simulation progresses over time. We will use the *Tracer* object in the *Utilities* library to collect the information about the congestion window size for a TCP session, during the transmission of a 80k file. The final diagram is shown in figure 6.1

The topology Build a simple topology: a couple of nodes and a duplex-link connecting them.

The transport layer . Create a TCP object close to one of the nodes, and a TCPSink object, close to the other node. Attach them to their respective nodes, and build a logical connection between them (**Connect** tool).

Events Add a timer, and set its name to *.15*. And connect the timer to the TCP Agent, using an *AgentEvent* object. Configure this object according to figure 6.2.

Tracing the Window Size Add a *Tracer* object from the *Utilities* library. This class allows a object member values to be written to a text file. It allows you to specify when to start recording, and the sampling interval. Create a Tracer object and configure it according to figure 6.3.

6.0.1 Plotting the trace file

When the script is run, a file with the same name as this object, and a ***.trc** extension (for trace) will be created. This file is a text file that stores the times at which the observations in one column, and the value of *cwnd_* in the other column. See figure 6.4 to observe the results.

Note: the trace file will be written in the directory in which you started running the **nscript** program, not in the file where your simulation script is saved.

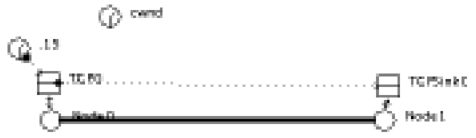
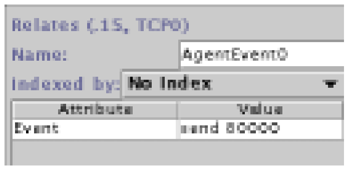


Figure 6.1: Finished simulation script for the tracing example.

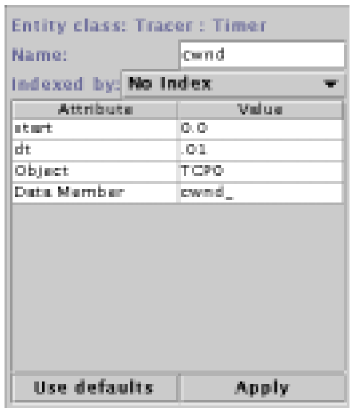

Relates (.15, TCP0)

Name: AgentEvent0

Indexed by: No Index

Attribute	Value
Event	send 80000

Figure 6.2: Configuration of the *AgentEvent* object to schedule the delivery of a 80k file.


Entity class: Tracer : Timer

Name: cwnd

Indexed by: No Index

Attribute	Value
start	0.0
dt	.01
Object	TCP0
Data Member	cwnd_

Figure 6.3: Configuration of the tracer object to record the cwnd_ field of the TCP session. This variable represents the congestion window size of the session. The tracing will start with the simulation, and will be sampled every hundreth of a second.

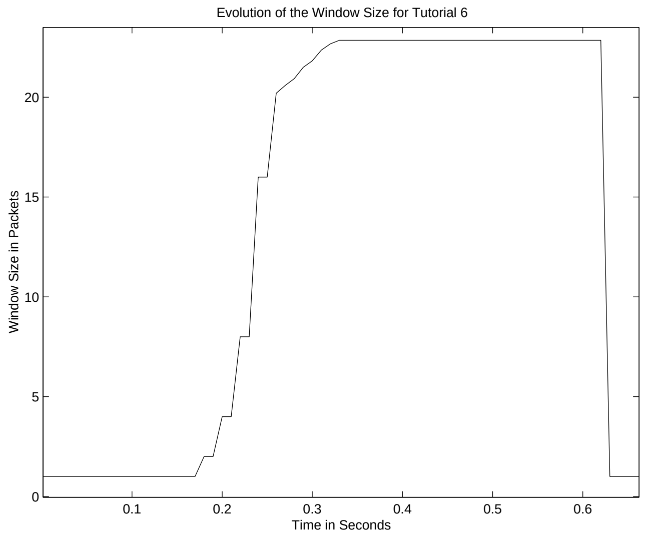


Figure 6.4: Evolution of the congestion window size, recorded using the Tracer object. This plot was obtained loading the file into Matlab, and plotting one column against the other.

Part II

Libraries

Chapter 7

About Libraries

There are some practical issues related to the libraries. First, at start up **nscript** will always proceed as follows:

1. Open the *environment* file under *nscript-1.0/bin/settings* directory This file contains the description of the environment object *ns*, that is always present in the simulations. **Do not remove this file**. You can and will want to modify it to customize the environment to your usual settings, and preferences.

2. Open the *deflibs* file located in the same directory. This directory contains the list of libraries to be opened by default. **nscript** tries to locate the libraries in the *bin/lib* subdirectory. If you want a library, created by you self or others to be opened automatically, add its name to the list in this file.

3. Open each of the libraries specified by the deflibs file.

7.1 Changing the default values

To change the default values for the objects, edit the correponding library file, that can be found under the directory: *nscript-1.0/bin/lib*. Each class definition has a list of attributes and their default values, changing this values will do.

For example, suppose that we want to modify the default values for a duplex-link, to be 1Mb, a delay of 10ms, and RED queueing discipline. Looking at file *topology.lib* and looking for the DuplexLink object definition, we find:

```
relation class DuplexLink node node 0 0 4 0:
  Bandwidth=10Mb;
  Delay=20ms;
  Forward Cost=1.0;
  Backward Cost=1.0;
  Queue Discipline=DropTail:DropTail RED CBQ FQ SFQ DRR;
  Queue Capacity=50;
begin
  $#env.name# duplex-link $#from# $#to# #Bandwidth# #Delay# #Queue Discipline#;
  set #name# [$#env.name# link $#from# $#to#];
  ?Queue Capacity=50::$#env.name# queue-limit $#from# $#to# #Queue Capacity#;
  ?Forward Cost=1.0::$#env.name# cost $#from# $#to# #Forward Cost#;
  ?Backward Cost=1.0::$#env.name# cost $#to# $#from# #Backward Cost#;
end
```

Change this definition to the following:

```
relation class DuplexLink node node 0 0 4 0:
  Bandwidth=1Mb;
  Delay=10ms;
  Forward Cost=1.0;
  Backward Cost=1.0;
  Queue Discipline=RED:DropTail RED CBQ FQ SFQ DRR;
  Queue Capacity=50;
begin
  $#env.name# duplex-link $#from# $#to# #Bandwidth# #Delay# #Queue Discipline#;
  set #name# [$#env.name# link $#from# $#to#];
  ?Queue Capacity=50::$#env.name# queue-limit $#from# $#to# #Queue Capacity#;
  ?Forward Cost=1.0::$#env.name# cost $#from# $#to# #Forward Cost#;
  ?Backward Cost=1.0::$#env.name# cost $#to# $#from# #Backward Cost#;
end
```

Notice the changes in lines 2, 3, and 5 of the definition. We tried to some extent to include the correct default values, but if we failed, it is always possible to correct this.

7.2 Creating New Libraries

In order to create new libraries, you need to check that either the library names nor the class names have been used before (i.e. contained in another library). If this was the case, **nscript** will simply ignore the class or library definitions.

Think of the classes stored in the libraries simply as templates that can be filled by the user. The ping tutorial for example shows the workings of a class, which can be separated into tree important parts:

- **Declaration.** What kind of class (entity or relation), its base class, drawing information for the Graphical Editor.
- **Attribute Declaration.** What information will be provided by the user, and what their default values are.
- **Translation to Tcl.** A Tcl "template" that will be filled with the information provided by the user.

The information on how to specify each of this parts is carefully described in the last chapter of the book. Look at the library definition files in the *bin/lib* subdirectory. These provide a good example of how to achieve results. Note that this will require you to understand how *ns* works.

Modifying a class definition from the default libraries, will make impossible to open previous simulation scripts. Therefore, we suggest the use of separate library files that can be created for your own purposes. If you have library definitions that implement new functionality, or that implement better what is provided, please let us know!

7.3 The default libraries

These are the libraries that are provided with the version **1.0a** of the environment. They are classified under the following four libraries:

- **Topology:** Nodes, duplex-links, and simplex-links.
- **Transport:** UDP, TCP, TCPSink, Null, LossMonitor, Attachment.
- **Applications:** CBR, ExponentialOn/Off, Pareto, FTP, and Telnet.

- **Utilities:** Tracer, Timer, ApplicationEvent, AgentEvent, LinkEvent, and GenericEvent.

The **Topology**, **Transport**, and **Applications**, are equivalent to *ns* objects, and for documentation of how they work refer to the *ns* documentation [FV00].

7.4 The Utilities Library

This library contains objects specifically created for the **nscript** environment, in order to achieve additional functionality. The objects and their functionality is explained in some of the tutorials, but here is a small explanation of their workings.

Tracer Collects information from a member variable of a Otcl object in the simulation. You can sepecify when to start collecting information, the interval of sampling after the recording process has started, the object that is being traced, and the member variable of interest. The parameters for this class can be seen in Tutorial 6, where the TCP congestion window is traced.

Timer Represents a time constant. It is used together with *ApplicationEvent* and *AgentEvent*, to schedule agent or application events.

AgentEvent A relation object that relates a time constant (Timer), and a transport agent, with the purpose of scheduling an action. It has a Event parameter, that represents the action to be taken at the specified time. This object was used in almost all tutorials to schedule the start and stop events for CBR applications.

ApplicationEvent Exactly the same as the AgentEvent, but works with application objects.

LinkEvent A down or up event in a link, which is used to simulate link failures, and was shown in Tutorial 4, regarding network dynamics. In order for these events to have some effect in the simulation, the Distance Vector routing protocol must be selected in the simulation environment (see tutorial 4).

GenericEvent A generic event that can schedule up to 10 events, without any requirements on the events or classes being scheduled. This event was used to schedule several **ping** to be sent in the course of a single simulation (see tutorial 5.)

Chapter 8

The nscript file formats

8.1 The nscript Document file format

The file format for a nsscript environment is defined as:

```
Number of arrays
Array Name 1
Array Size 1
Array Name 2
Array Size 2
.
.
.
Array Name n
Array Size n
Number of Objects
<SnippetName_1> <ObjectName_1>
[Entity Specific Attributes] | [Relation Specific Attributes]
[Object Attributes]
...
<SnippetName_nObjs> <ObjectName_nObjs>
[Entity Specific Attributes] | [Relation Specific Attributes]
[Object Attributes]
```

First, the libraries used in the program are specified, and the each object is also specified by describing its class (snippet), and the particular attributes for it.

8.2 The nscript Library file format

Libraries are stored in ASCII text following the format:

```
LibraryName
LibraryToolBarName
LibraryVersion

Class Description 1
Class Description 2
...
Class Description n
```

Here is a short description of the required fields:

- **LibraryName** A string that provides the name for the library. This name is for internal use, and no two libraries should have the same name. This name will also be used in the toolbar to display the a tab that shows a pane containing all objects that can be created within the library.
- **Library ToolBox Name**. The name for the tab that will be used to access the objects in this class.
- **Library Version**. The version of the library, for the purpose of version control.
- **Class Descriptions** This is the definition of the classes conforming a library. the details on this conform the following section.

8.2.1 Class file format

There are two families of classes supported by the **nscript** : a generic *entity* class which can be used to create objects represented by icons, and a *relation* class which can be used to relate two iconic objects. Both share the same format, with the exception of the declaration (first line) string. The general format for both classes is:

```
<class declaration>:
  <attribute_1>[=defaultValue][:Value1 Value2 ... ValueN];
  <attribute_2>[=defaultValue][:Value1 Value2 ... ValueN];
  ...
begin
  tclPattern | ?<attribute>=<target_value>:<truePattern>:<falsePattern>;
  tclPattern | ?<attribute>=<target_value>:<truePattern>:<falsePattern>;
  ...
end
```

Where the **truePattern** will be used whenever the user sets the **attribute** equals to **target_value**.
As mentioned above, the *class definition* differs from an *entity* and a *relation* object. So the syntax for a class declaration can be any of the following:

```
[!]entity class <class_name> <base_class_name> <icon>:
[!]relation class <class_name> [!]<from_base> [!]<to_base> <base_fig> <style> <width> <end_fig>:
```

The values of these fields are illustrated in the figure 8.1, and correspond to how the object are displayed in the editor.

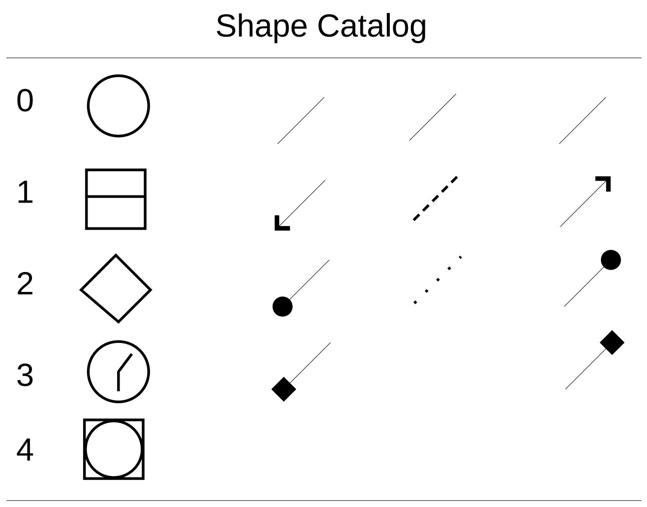


Figure 8.1: Icons and line styles supported by the scripter.

Chapter 9

TclPattern and value substitution

The Tcl patterns used in the previous definitions are normal Tcl expression that contain replacement patterns enclosed in ##. So a Tcl expression containting the replace pattern **#attribute_name#** will be replaced with the value of the attribute, which is provided by the user through the graphical interface.

The following examples illustrate the use of the format, and are taken from the actual libraries used in the nscript.

Example 1: a Node object

```
element class Node node 0:
begin
  set #name# [$#env.name# node];
end
```

Example 2: a duplex link object

```
relation class DuplexLink node node 0 0 4 0:
  Bandwidth=10Mb;
  Delay=20ms;
  Forward Cost=1.0;
  Backward Cost=1.0;
  Queue Discipline=DropTail:DropTail RED CBQ FQ SFQ DRR;
  Queue Capacity=50;
begin
  $#env.name# duplex-link $#from# $#to# #Bandwidth# #Delay# #Queue Discipline#;
  set #name# [$#env.name# link $#from# $#to#];
  ?Queue Capacity=50::$#env.name# queue-limit $#from# $#to# #Queue Capacity#;
  ?Forward Cost=1.0::$#env.name# cost $#from# $#to# #Forward Cost#;
  ?Backward Cost=1.0::$#env.name# cost $#to# $#from# #Backward Cost#;
end
```

Example 3: A TCP Agent

```
entity class TCP agent 1:
  Max Window Size=20;
  Initial Window=1;
  Packet Size=1000;
  FlowID=None:None 1 2 3 4 5 6 7 8 9 10;
begin
  set #name# [new Agent/TCP];
  ?Max Window Size=20::$#name# set window_ #Max Window Size#;
  ?Initial Window=1::$#name# set windowInit_ #Initial Window#;
  ?Packet Size=1000::$#name# set packetSize_ #Packet Size#;
  ?FlowID=None::$#name# set fid_ #FlowID#;
end
```

Bibliography

[BBE$^+$99] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, Steven McCanne, Reza Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu, Haobo Yu, and Daniel Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999. revised September 1999, to appear in IEEE Computer.

[FV00] K. Fall and J. Varadhan. The ns manual. *The VINT Project / http://www.isi.edu/nsnam/ns/ns-documentation*, December 2000.

[Gre] Marc Greiss. ns tutorial. *http://www.isi.edu/nsnam/ns/tutorial/nsbasic.html#inst.*