REAL Manuals

Srinivasan Keshav

September 1989

Abstract

We present five manuals for REAL, a network simulator described in UCB/CSD/TR 88/472. REAL allows the user to set up realistic and large simulations of packet switched data networks with little effort. Several transport protocols and congestion control schemes can be studied.

A table of contents is presented in the manual manual. The installation manual describes the installation process. The user manual guides users through the user interface and provides a tutorial for beginners. The NetLanguage manual describes a language used to describe simulation scenarios. The programmer's manual is intended for advanced users who wish to modify REAL for their own purposes.

This research has been supported in part by the University of California with a MICRO program grant, by Xerox Corporation and by AT&T Bell Laboratories. The views and conclusions contained in this document are those of the author and should not be interpreted as the official policies, either express or implied, of any of the sponsoring organizations.

REAL Manual Manual

REAL is a public domain network simulator that tries to simulate Large networks REAListically. This manual describes the other REAL and Nest manuals.

The REAL package is based on the Nest simulation testbed from Columbia University[1].There are 5 Nest documents described below.

1. Nest manuals

Usenix paper
: This paper presented at Usenix Winter Conference 88 provides an overview of Nest and examples of where it can be used.

Nest System Overview
: Describes the overall system organization of Nest. It also includes descriptions of the client-server protocol and hints for writing node functions.

Nest User's Guide
: Briefly describes Nest source files and how to write node and monitor functions.

Nest Library Reference Manual
: Describes facilities in Nest that are used when writing node functions.

Nest User Interface Manual
: Describes the control window, how to use it and how to customize it.

2. REAL manuals and reports

REAL: A Network Simulator
: UC Berkeley, Computer Science Department Technical Report 88/472. An overview of REAL and some examples of network analysis using REAL. For copies mail jean@ernie.berkeley.edu.

Manual manual
: Describes other Nest and REAL manuals. This manual.

Installation Manual
: How to install, care and feed your REAL simulator.

User Manual
: A step-by-step tutorial on using REAL for naive users.

NetLanguage Manual
: Tutorial introduction to using NetLanguage.

Programmers Manual
: Tour of the files in REAL, and a cookbook for modifying REAL and Nest.

NEST, Copyright 1988 Columbia University, a Network Simulation Testbed, written by Alex Dupuy. Direct technical questions regarding NEST to dupuy@cs.columbia.edu and questions regarding NEST distribution to jed@cs.columbia.edu.

3. Licencing

Please complete the licence documents in sim/sim/user.licence.1 and sim/sim/user.licence.2 and send them off to the appropriate places. If you would like to use code in REAL or NEST in your own software, you should contact keshav@ucbarpa.berkeley.edu for a redistribution licence.

REAL Installation Manual

1. Introduction

This manual provides guidelines for installing REAL in your local 4.3BSD system (sorry, there is no V9 or SysV compatibility).

This manual should accompany the tar file for the REAL release that you have obtained. I will assume that you have a copy of the tar file on a disk with a reasonable amount of space (at least 3Mb).

Your first step is to untar the file. Move to the directory which will the root for REAL and type

```
tar -xpf [tarfile]
```

where tarfile is the name of the tar file. After a suitable interval of time, the file hierarchy described in the user manual should appear. You are now ready to configure and make the system.

Change directory to sim/src and type

```
Configure
```

This will fire up an interactive configuration manager that will ask you several questions about your system, and will figure out many things by itself.

It is possible the Configure will blow up on you. For example, Configure runs a test that creates archives of length zero and this can cause a system crash on some machines. To avoid this, you should comment out the commands that are causing problems, and determine the defaults manually. To continue with the example, you will need to comment out lines 1143-1144 and 1148-1164 in Configure and rerun it. It helps to have a Unix guru handy.

At one point, Configure will ask you to do a shell escape to edit config.sh. At this point, type in

```
![favorite-editor] config.sh
```

You will find a file that looks like this:

```
: use /bin/sh
# config.sh
# This file was produced by running the Configure script.

d_eunice='undef'
eunicefix=':'
define='define'
expr='/bin/expr'
sed='/bin/sed'
echo='/bin/echo'
cat='/bin/cat'
rm='/bin/rm'
tr='/usr/bin/tr'
sort='/bin/sort'
grep='/usr/ucb/grep'
test='test'
contains='grep'
cpp='/lib/cpp'
cppminus=''
d_charsprf='undef'
d_voidsig='define'
libc='/bsd43/usr/lib/libc.a'
```

```
n=''
c='package='Nest'
spitshell='cat'
shsharp='true'
sharpbang=': use '
startsh=': use /bin/sh'
voidflags='7'
defvoidused='1'
d_getopt='define'
d_itimer='define'
d_systime='/bsd43/usr/include/sys/time.h'
d_memset ='define'
d_bstring='define'
d_psignal='define'
d_rename='define'
d_rusage='define'
d_server='define'
d_sigvec='define'
d_sigvectr='undef'
d_socket='define'
d_oldsock='undef'
socketlib=''
sockethdr=''
mkdep='/bsd43/bin/cc -M'
orderlib='false'
ranlib=':'
stackdir='-1'
CONFIG=true
```

You have to edit this file to make sure that it describes your system accurately. This is not easy, and unfortunately there is not much I can do to help you with it. I will describe the variable names in the file, and you will need to determine if their values are appropriate for your system. Nine times out of ten, this will be fine, but on machines that try to support both BSD and SysV (such as the MIPS machine), you can run into annoying problems.

In general, the convention used for variable names is that d_XXX means that a #define for XXX will be created in the configuration file nest.h. Command names stand for the particular flavor of command used by your system.

The file starts off with 'd_eunice', which is defined if you have a eunice system. This is followed by definitions for the location of useful commands. If these are not correct, you should modify that. 'contains' should be your flavor of grep. 'cpp' should be set to your C preprocessor. 'cppminus' is the default flag that you need to give to cpp. 'd_charsprf' is set if sprintf() is of type char. 'd_voidsig' is set if your system defines signal() to be of type (void).

The next definition (for 'libc') is critical, and defines where your C library is. If you have more than one library, then this might be wrong. For example, on a MIPS, it will be set to the SysV library /usr/lib/libc.a, and you actually should have /bsd43/usr/lib/libc.a. 'spitshell' is the command to print data into a file. 'shsharp' is true if '#' can comment out things in your version of shell. 'voidflags' is explained when you do Configure.

Now there are some defines that explore the limits of your operating system. The meanings of these defines are in the file sim/src/nest.h.SH. Take a minute to make sure that they are correct.

'socketlib' is the library that has socket code, if libc.a doesn't. 'sockethdr' is the place where include files for sockets are kept if they are non-standard. 'mkdep' is the command that will generate makedepends for you. Usually this is /lib/cpp -M, but if that doesn't work, you may have to use /bsd43/bin/cc -M. 'orderlib' is true if the archive command creates unordered libraries. 'ranlib' is the command you need to use to create random libraries (set to ':' if it isn't needed, then the command becomes a comment). 'stackdir' is the direction the stack grows in, 1 is up, -1 is down. Finally, CONFIG is set to indicate that you have run config.sh.

After you have edited config.sh, you should return to the Configure session. Configure will now extract some files. Files that end with a .SH suffix are shell scripts that contain shell variables set by config.sh. Configure runs config.sh to set these variables, then runs the .SH file. This creates a file where the appropriate variables have been substituted. For example, if Configure determines that the C preprocessor, cpp, should be called as /usr/lib/cpp -, then the shell variable 'cpp' will be set to '/usr/lib/cpp', and the variable 'cppminus' to '-'. Suppose Makefile needs to invoke the preprocessor in one of its targets. Then Makefile.SH will have an entry of the form '$cpp $cppminus filename'. When Makefile is extracted, the correct invocation is automatically generated.

Finally, Configure will offer to do a makedepend for you. This step consists of editing Makefile to automatically add include file dependencies. The 'depend' target in Makefile spawns a small ed script that edits Makefile to delete the end of the file, and to append dependencies generated by $mkdep.

Configure will now exit. If you are using a Vax, you have two additional steps. First, uncompress the file sim/src/vax/graphs.c.Z by typing 'uncompress graphs.c'. Then, move graphs.c and graph.h to sim/src, overwriting the existing files. Now run a make to create nest.a by typing 'make nest.a'. Ignore any warnings about non-portablity of the code.

You might run into a problem with extracting state.s from state.S. state.S contains assembly code that differs for each machine. In some cases, you may not be able to extract the assembly code correctly. If so, you should extract the state.s file manually, and rerun make. Make will not process state.S if state.s or state.o already exist.

Now change directory to sim/sim to generate REAL. First, extract the makefile by executing the makefile.SH shellscript. (You can do this by typing . makefile.SH or sh makefile.SH). Then type in 'make final' to generate the simulation binary 'simulate'.

You now will need to compile the files for the control window. Change directory to sim/display and type 'make sunvclient'. The last step is to create the NetLanguage handling files in sim/lang. Change directory to sim/lang. 'make lang.c' will create the 'lang' preprocessor and lang.c which is the result of processing the default scenario described in lang.inp.

You are now ready to begin simulations. The REAL User manual and Programmers manual should help you get started. Good luck !

REAL Users Manual

This manual provides a tutorial introduction to setting up a simulation in REAL. If you want to modify features of REAL you should refer to the REAL Programmers Manual.

1. General Orientation

REAL consists of two parts: a simulator and a control window. The two communicate with each other over a pair of BSD sockets. The simulator is informed about what it has to simulate using the control window or a special purpose language, NetLanguage. (NetLanguage is described in the NetLanguage manual.) The simulator then starts off the simulation, ticking off virtual time. Simulations in progress can be controlled by the control window.

The simulator is told to simulate a *scenario*, which is a description of topology, protocols, workload and control parameters. The network is modeled as a graph, where nodes (vertices) represent either sources or sinks of data, or gateways[1]. The interconnection between the nodes is the topology. You also have to specify the protocol and the workload at each source. Finally, you have to specify control parameters such as the latency and bandwidth of each communication line, the size of trunk board buffers, packet sizes etc.

1.1. File organization

The simulator files are in a directory called sim. This has six [2] subdirectories - bin, sim, src, display, lang and results. sim/sim has source code for REAL. sim/src has source code for NEST, the simulation package underlying REAL. sim/display has code for the control window. sim/lang has code for the Net-Language preprocessor. Finally, sim/bin has symbolic links to the binaries for the simulator, the control window and the preprocessor.

2. Setting up a simulation

You can set up a simulation either using a mouse-based interactive control window or using Net-Language. I will describe the control window approach first.

2.1. Using the control window

The first step is to start up the control window. You can do that by running sim/bin/sunvclient by typing:

```
sunvclient &
```

This will put it in the background. In a few seconds, a window that looks like the control window in Figure 1 will pop up. (For the moment, ignore everything except the window labelled 'REAL control panel' in the figure.) The window is divided into 5 layers of subwindows. Let me call them layers 1 through 5 from top to bottom. They are organized as follows:

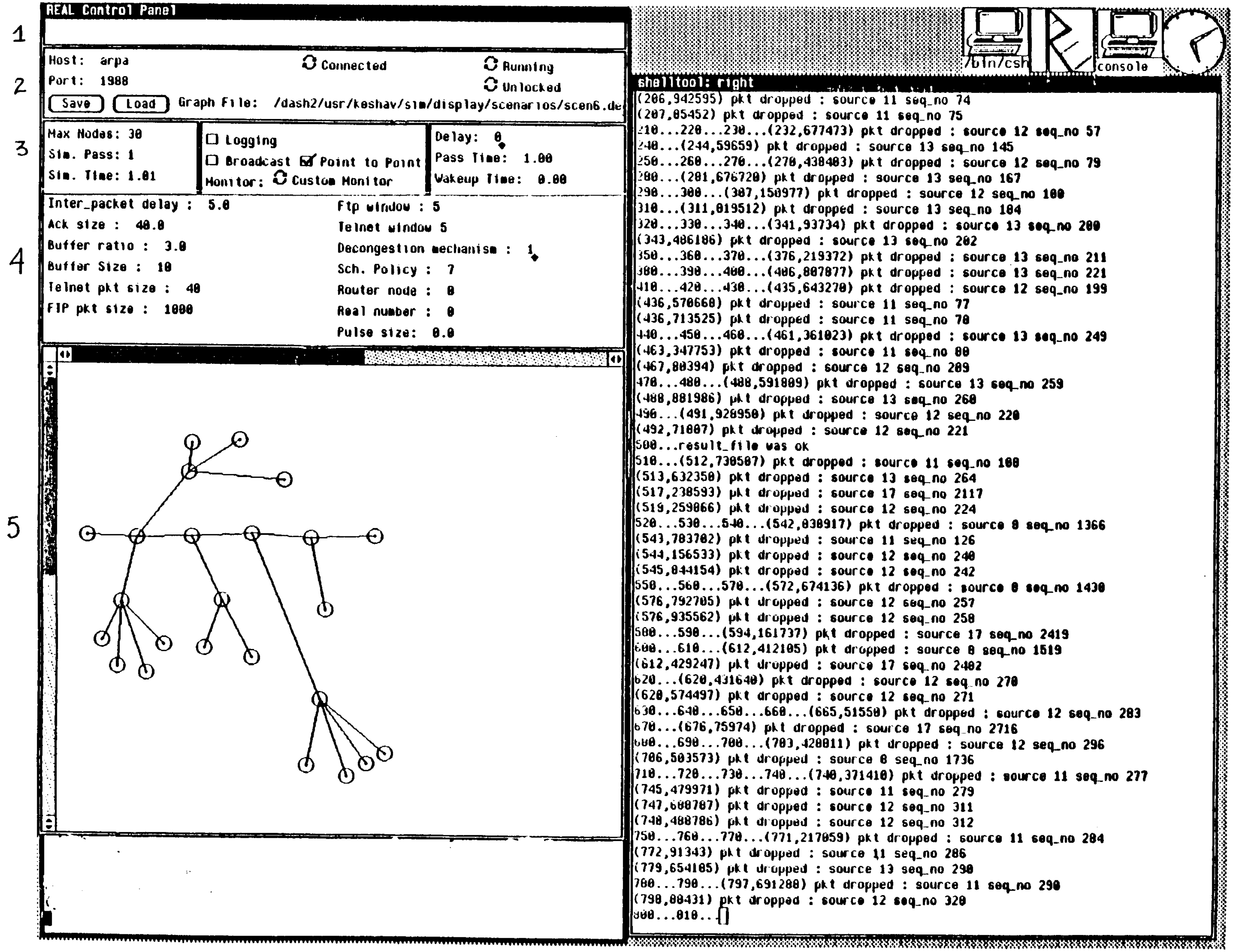
Layer 1	: Read-only, displays error and status messages.
Layer 2	: Connection status and graph file saving and loading.
Layer 3	: NEST parameters.
Layer 4	: REAL parameters.
Layer 5	: Network

I will discuss the details of these windows later. At the moment, you should concentrate on layer 2. This has a host name and a port number. Move the mouse to the host name part and click the mouse left button so that the dark flashing triangle is at the end of the default host name. Use the 'Delete' key on the

[1] Gateways are used synonymously with routers, bridges and switches to mean elements that route, buffer and schedule packets.

[2] If this machine does not run suntools, the display directory will be unusable.

YER



```
Host:  arpa                    Connected           Running
Port:  1988                                        Unlocked
Graph File:  /dash2/usr/keshav/sim/display/scenarios/scen6.de

Max Nodes: 30      [ ] Logging                       Delay:  0
Sim. Pass: 1       [ ] Broadcast [✓] Point to Point  Pass Time:  1.00
Sim. Time: 1.01    Monitor:  Custom Monitor          Wakeup Time:  0.00

Inter_packet delay :  5.0          Ftp window : 5
Ack size :  40.0                   Telnet window 5
Buffer ratio :  3.0                Decongestion mechanism :  1
Buffer Size :  10                  Sch. Policy :  7
Telnet pkt size :  40              Router node :  0
FTP pkt size :  1000               Real number :  0
                                   Pulse size:  0.0
```

```
(206,942595) pkt dropped : source 11 seq_no 74
(207,05452) pkt dropped : source 11 seq_no 75
210...220...230...(232,677473) pkt dropped : source 12 seq_no 57
240...(244,59659) pkt dropped : source 13 seq_no 145
250...260...270...(270,438403) pkt dropped : source 12 seq_no 79
280...(281,676720) pkt dropped : source 13 seq_no 167
290...300...(307,150977) pkt dropped : source 12 seq_no 100
310...(311,019512) pkt dropped : source 13 seq_no 104
320...330...340...(341,93734) pkt dropped : source 13 seq_no 200
(343,406106) pkt dropped : source 13 seq_no 202
350...360...370...(376,219372) pkt dropped : source 13 seq_no 211
380...390...400...(406,807077) pkt dropped : source 13 seq_no 221
410...420...430...(435,643270) pkt dropped : source 12 seq_no 199
(436,570660) pkt dropped : source 11 seq_no 77
(436,713525) pkt dropped : source 11 seq_no 78
440...450...460...(461,361023) pkt dropped : source 13 seq_no 249
(463,347753) pkt dropped : source 11 seq_no 80
(467,80394) pkt dropped : source 12 seq_no 209
470...480...(488,591009) pkt dropped : source 13 seq_no 259
(488,881986) pkt dropped : source 13 seq_no 260
490...(491,928950) pkt dropped : source 12 seq_no 220
(492,71007) pkt dropped : source 12 seq_no 221
500...result_file was ok
510...(512,730507) pkt dropped : source 11 seq_no 100
(513,632350) pkt dropped : source 13 seq_no 264
(517,238593) pkt dropped : source 17 seq_no 2117
(519,259066) pkt dropped : source 12 seq_no 224
520...530...540...(542,030917) pkt dropped : source 8 seq_no 1366
(543,783702) pkt dropped : source 11 seq_no 126
(544,156533) pkt dropped : source 12 seq_no 240
(545,044154) pkt dropped : source 12 seq_no 242
550...560...570...(572,674136) pkt dropped : source 8 seq_no 1430
(576,792705) pkt dropped : source 12 seq_no 257
(576,935562) pkt dropped : source 12 seq_no 258
580...590...(594,161737) pkt dropped : source 17 seq_no 2419
600...610...(612,412105) pkt dropped : source 8 seq_no 1519
(612,429247) pkt dropped : source 17 seq_no 2402
620...(620,431640) pkt dropped : source 12 seq_no 270
(620,574497) pkt dropped : source 12 seq_no 271
630...640...650...660...(665,51550) pkt dropped : source 12 seq_no 283
670...(676,75974) pkt dropped : source 17 seq_no 2716
680...690...700...(703,420811) pkt dropped : source 12 seq_no 296
(706,503573) pkt dropped : source 8 seq_no 1736
710...720...730...740...(740,371410) pkt dropped : source 11 seq_no 277
(745,479971) pkt dropped : source 11 seq_no 279
(747,600707) pkt dropped : source 12 seq_no 311
(748,488786) pkt dropped : source 12 seq_no 312
750...760...770...(771,217059) pkt dropped : source 11 seq_no 284
(772,91343) pkt dropped : source 11 seq_no 286
(779,654105) pkt dropped : source 13 seq_no 290
780...790...(797,691288) pkt dropped : source 11 seq_no 290
(798,80431) pkt dropped : source 12 seq_no 320
800...810..
```

FIGURE 1

keyboard to delete the name, and type in the name of the machine on which you want to do the simulation. Remember not to leave spaces at the end of the entry. (In general, NEVER leave spaces at the end of any entry in the control window).

The control window connects to the simulation client using two BSD sockets. In the socket abstraction, every machine supports a number of 'ports' which are the endpoints of communication sockets. A port has a port number which the other end should know about. REAL uses two consecutively numbered port numbers and we need to ensure that both the control window and the simulator know about the port numbers that they will use. A note about what values to use: Usually the operating system does not allow user applications to use port numbers lower than 1024. So, you should pick two numbers larger than that. The default port numbers are 1988 and 1989 and you must check if these numbers are unavailable by looking at the file /etc/services on both the client and the simulation machine. If they are, you have to change the port number in the 'Port Number' field in Layer 2. Use the same technique described above to change the port number.

It is now time to start the simulator. Log in to the appropriate machine and start off the simulator in sim/bin by typing in

```
simulate -p [port number]
```

where port number is the smaller of the two port numbers you have decided to use. The simulator will sit there waiting for the control window to give it a network to simulate.

We will now connect the control window to the simulator. In layer 2, you will find a field marked 'Detached' with a circle next to it. The circle means that it is a control field that can change state. It is now in the detached state, since the simulator is not connected. To start the connection, move the mouse arrow to the circle and press the left button. The state will change to 'Connected' and connection will be initiated. If the connection does not go through for some reason, the reason will be printed in the Layer 1 subwindow. Possible reasons for non-connection are:

Unknown host: the host cannot be reached by a socket. Either the network is down, or your system's name tables are not ok. In any case, you will need to contact your system guru and ask for enlightenment. (If you are the local guru, you will need to check why gethostbyname() is failing, could be faulty BIND tables, broken yp server...).

Connection in progress...: You typed in the machine name incorrectly, the network is slow, or your simulator has not put out a connection request yet. Try restarting the simulator.

Once the connection is up, you can now send data to the simulator. However, it is first necessary to initialize the control window with a default network that is programmed into REAL. When you connected to the simulator, a new button labelled 'Paused' should have appeared in the far right of Layer 2. This indicates that the simulator is now in a paused state. Move the mouse to this control button and click the left mouse button to start off the simulator. The state will briefly change to 'Running' and come back to 'Paused'. Further, the default graph will appear on the screen in Layer 5. You now have three alternatives: 1) run a simulation on the default topology 2) run a simulation on a stored topology 3) create a new topology to simulate. These are described in sections 2.2 through 2.4. Read all three sections below anyway.

2.2. Using the default topology

Even if you are using the default topology, you may want to change node functions and trunk (line) characteristics. If you press the right mouse button (RMB) when the mouse cursor is placed over a node (in Layer 5), a pop up menu will appear. Move the cursor to the walking menu labelled 'Node Functions'. Another pop up menu will appear with the list of node functions that are available. (If you don't understand what a node function is, you should go read the tech report at this point.) Move the cursor to the function that you want to assign to that node and the function will be set automatically for that node. (Note that if you are using the 'Show Node Data' function, changes made in the data pop up menu are NOT set unless 'Set Node Data' is also selected.)

The other thing you may want to change is edge characteristics. Move the mouse to an edge and click the RMB. A pop up menu will appear, and here you should select the entry that says 'Show Edge Data'. A data window will now appear, in which you can make changes (followed by a 'Set Edge Data' to fix the changes). The 'Edge Weight' entry specifies the delay on the line, and it should be specified in microseconds. The 'Linespeed' field is the bandwidth of the line in bytes per second, and you should enter a floating point number into it (i.e. terminating in .0). The channel function stack is set using the control button in this window. Click the circle icon till the correct channel function appears (usually 'tx_delay'). When all the entries in the data window are correct, select the line for which you wanted to set the characteristics, click the RMB over it, and choose the 'Set Edge Data' option in the pop up menu. You may want to set multiple lines with the same characteristics. In that case, simply do a 'Set' for each of the lines.

You are now ready to set the REAL and NEST parameters.

2.2.1. REAL Parameters

All REAL parameters appear in the Layer 4 subwindow. You will find that the default values for these parameters are already entered. Remember that you should replace floating point values with floating point values, and integer values with integer values. Here is a brief explanation of each field.

Inter_packet delay: is the mean of the exponential distribution of the inter-packet delay for the Poisson workload (telnet workload). Specified in seconds.

Ack Size: is the size of the acknowledgment packet in bytes.

Buffer Ratio: This is the threshold at which a Fair Queueing gateway will start setting bits on a conversation. To be precise, if the parameter is B, then the gateway will set bits on all conversations that use more than 1/B of their fair share of buffers.

Buffer Size: This is the number of buffers that are available per output line (trunk board). One buffer can store a packet of arbitrary size (this is the Xerox XNS convention).

Telnet Pkt. Size: is the size of a telnet packet in bytes.

FTP Pkt. Size: is the size of a FTP packet in bytes.

FTP, Telnet Window: are the maximum sizes of ftp and telnet windows. (These are not the same as the congestion windows, but are their upper bounds.) The window size should not exceed the MAX_WINDOW_SIZE parameter in sim/sim/config.h.

Decongestion Mechanism: This selects which packet to drop when a conversation's queue is full. 0 selects the packet at the head of the queue and 1 selects the tail. When this parameter is set to 2, we select a packet at random amongst all the packets queued at the trunk and drop it.

Scheduling Policy: This is used to select the scheduling policy at the gateways. For historical reasons 3 is FCFS, 7 is Fair Queueing, 8 is FCFS + bit setting (DEC bit), and 9 is Fair Queueing + bit setting.

Router Node: Real Number: Used in distributed simulation.

Pulse Size: An entry of the form X.(Y+1) will generate a pulse of X packets every Y seconds. Y has to be 3 digits long. So, for example, an entry of 3.008 will generate a pulse of 3 packets every 7 seconds. (Do not use 3.8 - that will be interpreted as 3 packets every 799 seconds.) This format is ugly and needs to be changed. Soon, perhaps.

2.2.2. NEST parameters

NEST parameters are found in Layer 3 of the control window. In general you will not need to change them. Here is a description of each field:

Max Nodes: This is the largest number of number of nodes in the simulation. You can set this to the number of nodes in the simulation. Note that memory is allocated in proportion to Max Nodes, so you should not choose something too large. Note also that REAL does not support simulations with more than ABSOLUTE_MAX_NODES nodes. ABSOLUTE_MAX_NODES is defined in config.h.

Sim. Pass: This is a read only entry that records simulation passes as they happen. To save on communication bandwidth, passes are updated only when you send a control message to the simulator - usually a 'Send' (to be discussed).

Sim. Time: This displays the simulation time

Logging: This is supposed to turn on Nest debugging, but this doesn't seem to work. I prefer to use dbx (the 4.3BSD symbolic debugger).

Broadcast/Point to Point: Describes if the communication mode is point to point or broadcast. REAL supports only the point to point mode.

Monitor: Selects the monitor to use. You should use 'Custom Monitor' unless you write your own.

Delay: This is the default delay on a communication line (if the edge weight is 0). Use a 0 value.

Wakeup Time: Use 0.00.

Pass time: How often (in virtual time) the monitor is called. I'd recommend 1 second, but you can make it larger if you want a faster simulator.

Now that you have set up the parameters and the scenario, you should let the simulator know about this. To do that, move the mouse cursor so that it is not positioned near an edge or a node in Layer 5. The right mouse button will pop up a menu that has an entry marked 'Send'. Move the selection bar to this entry and release the button to send the graph and Nest parameters to the simulator. To send REAL parameters, select 'Send Parameters'. Then force initialization by selecting 'Reset'. Each of these commands will be acknowledged by messages in Layer 2. Note that the commands should be sent IN THIS ORDER. If, for example, you do a 'Reset' before a 'Send', the routing function will create routing tables for an incorrect network (and will probably crash).

Now, go back to the 'Paused' button in Layer 1 and click the left mouse button. This will start the simulator, and you are off. A subsequent section of the manual discusses the result files and messages printed out by the simulator.

2.3. Using a Stored Topology

REAL allows you to store topologies in one of two formats : in NetLanguage (Section 5) or in a data file. To use a stored topology, when the simulator returns with the default graph, you must delete it. Move the mouse to a node, and click the right mouse button. The first entry in the pop up menu is 'Delete'. If you select it, the node will disappear, as will the edges connecting it to the rest of the graph. Repeat this for all the nodes, so that the window is empty. Now, press the right mouse button, and select the 'Send' entry. This will inform the simulator that the default network has been deleted. Now, go to Layer 2 and select the field marked 'Graph File'. Type in the full path of the file that has the stored network and then click the 'Load' oval with the left mouse button. The stored network will appear in Layer 5. You may modify node functions, edge characteristics or simulation parameters as above. Then go through the sequence of 'Send', 'Send Parameters' and 'Reset' to inform the simulator about the simulation scenario. Click on the 'Paused' flag in Layer 2 to restart the simulation. The simulator will echo the graph back to you and pause again. Click the 'Paused' flag again to start off the simulation.

You can save any changes you made by typing in the name of a file into the 'Graph File' field in Layer 2, and selecting 'Save' in Layer 2. Note that this will save the topology, node characteristics and edge characteristics only. This will NOT save REAL parameters. (This used to be a feature, I'm not so sure now that it isn't a bug.)

2.4. Creating a new topology

Follow the steps in the section above till the Layer 5 window has been emptied, and the simulator has been informed of the deletions by the 'Send' operation. You are now ready to create your own scenario. Clicking the left mouse button will create a node. You should set the node function as described earlier. (Ignore the start, halt, repeat and clear flags entries.) To create edges, click the middle mouse button when the cursor is on a node. You can now drag an edge to any other node. Set the edge characteristics and simulation parameters as described above. Now, do the three step 'Send', 'Send Parameters', 'Reset' sequence. Change the state of the simulation from 'Paused' to 'Running' by selecting 'Paused' in the Layer 2 window. The simulator will reply with the version of the graph that it received, and go back to the paused state.

You now have to set the sinks for the source nodes. For each source node, decide which sink it should send data to. (Currently, this is decided statically. I may introduce a facility to allow users to dynamically change the destination later.) Find the sink and click the right mouse button on the node. When the pop up menu appears, select 'Show Node Data'. This will create a data window in which the node number of the sink appears. Move to the source node for which the sink has to be set and select 'Show Node Data' again. The data window will be updated with the information for the source node. Type in the sink number in the slot marked 'Sink'. Select 'Set Node Data' to set the sink identity. (If you do not do this, the change will not be incorporated.) The data window also has a field marked 'Plot'. If you want to plot data for that node, select this field and set the data as before. Plotting is discussed in detail in Section 6 of the manual.

When you have finished setting all the nodes, send the graph. You do not need to send parameters or reset again. The simulator will confirm the graph, and you should keep hitting the Paused field till the simulator starts running.

3. Simulation output

The simulator produces four kinds of outputs a) acknowledgments of operations b) announcements of special events c) time ticks and d) a simulation report.

The simulator will acknowledge operations such as resets and dump commands ('dump' is explained later in this section). When you send new parameters, it will echo them back to you. Also, when it receives a new graph, it will print out the nodes, their types, the sink they send to, and the characteristics of all the lines. If you are in stand-alone mode, then the topology being simulated, and the current control parameters will be printed out.

Special events refers to errors in simulation, or events that you have coded into the node functions. Some error messages are not quite intuitive. In particular, if you get 'Result file was empty', this just means that the simulator was unable to create the file "../results/dump". If this happens, you have probably forgotten to create a directory called sim/results. If you ever get a error '... got an invalid packet', you should probably send me mail. There is something seriously wrong in the simulator or in your set up. Error messages are printed out in the stand-alone mode as well.

Time ticks are printed out every ten seconds of simulated time. They are just to show you that something is going on. In Figure 1, the right hand window shows an example of how time ticks and errors are printed out. Time ticks are printed out in the stand-alone mode.

Reports are generated in one of two ways: By default, they are generated every 500 seconds (you can change this by changing the parameter DUMP_INTERVAL in sim/config.h and remaking the binary - type 'make final'). Or, you can force a dump at any time by selecting the dump option in the pop up menu when you click the right mouse button away from any node or edge in Layer 5.

In either case, a file called ../results/dump will be appended to. The report is in the form of table and will look like this:

```
-------------------------------------------------------------------------
          Report #0 : simulation statistics
                    Policy # 7
             Time is now (500, 10000)
-------------------------------------------------------------------------
Source  Type    GatewayWait in        Num     Num     Tao
  #                     queue         dropped retx.
-------------------------------------------------------------------------
1       VJ Tel   3      111,0.0
                                      0       0       0.080
2       VJ Tel   3      98,0.070
                                      0       0       0.082
20      VJ FTP   3      427,2.310
                                      26      30      2.194
21      VJ FTP   3      714,2.151
                                      34      43      1.771
```

The header displays the report number and the scheduling policy used in the simulation. It also has the time at which the report was generated. For each source, the following data is printed:

Source Number: This is the node ID that is assigned to the node by the simulator or by NetLanguage.

Type: This is the name of the node function running on the node.

Gateway: This is gateway number for which the delays are being reported. If the source sends packets through multiple gateways, then the report will print out the delays at each of the intermediate gateways.

Wait in queue: This is a pair of numbers. The first is the number of packets that were actually sent out by the gateway onto the output trunk. It does NOT include a count of dropped packets. The second number is the average number of seconds that a packet from that source had to wait in the queue from the time that its last bit arrived at the gateway to the time that the first bit in that packet was placed on the output trunk.

Num dropped: This is the number of packets from that source that were dropped by intermediate gateways.

Num retx.: This is number of packets that the source retransmitted.

Tao: This is the average round trip time that packets from that source took from the time that the first bit of the packet was placed on the line to the time that the last bit of the acknowledgment arrived. In case of VJ sources, it is computed only on packets that have never been retransmitted.

In order to avoid corrupting simulation results with data from transients that happen at start up, simulation statistics are flushed after each dump. In other words, the report presents statistics for the last 500 seconds of simulation only.

Note that to compute effective throughput you must subtract the number of retransmissions from the number of packets sent by the gateway. The effective load is the number of packets sent by the gateway plus the number of packets dropped by the network.

4. Control of the running simulation

When you click the right mouse button in Layer 5 on a region that is not near any node or edge, you will get a pop up menu that has the following items:

Redisplay: This will redisplay the graph in the window.

Send: Sends the graph to the simulator. Nest parameters will also be sent. (But not REAL parameters.)

Clear: Clears the graph from the screen. Doesn't work properly. It is best to delete nodes one by one. (I'll have to fix this sometime.)

Undo: Undoes a clear.

Kill: Stops the simulation.

Reset Re-initializes the simulation and creates the static routing tables.

Send Parameters: Sends REAL parameters to the simulator.

Write Graph: Writes out a NetLanguage description of the graph and the simulation parameters into a file called LANG_FILE in sim/sim. An existing LANG_FILE will be overwritten.

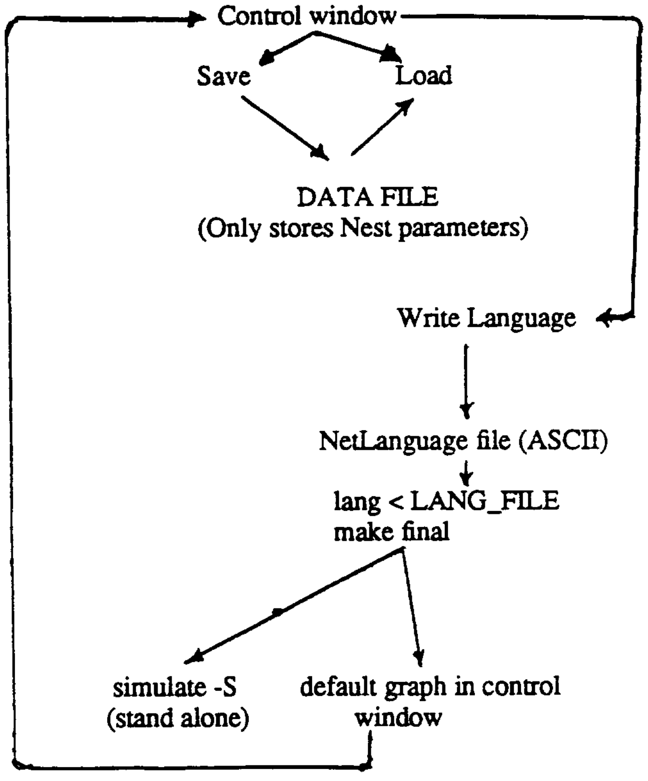
5. NetLanguage

This is a language used to describe simulation scenarios. The language is simple to understand, and I will not describe it in any great detail here. You should read sim/lang/lang.inp which is a sample description, and just pattern match. This file is reproduced in the Appendix. NetLanguage is described in greater detail in the REAL NetLanguage Manual.

I intend three uses for the language. First, it allows you to save experimental scenarios to be reloaded later. Second, you can draw a simple scenario using the mouse, then use your favorite text editor to create much larger graphs that are too complicated to draw. Finally, you can use it to run simulations without using a control window. (This is useful if you don't have access to a Sun.)

The NetLanguage file that you write or generate must be processed as follows. First, the language should be passed through a preprocessor called lang in sim/bin. This will create two files, lang.c and extern.lang.h that will have to be moved to sim/sim (alternately, you can set up a soft link from sim/sim to this file). Then, just type 'make final' to rebuild the simulator with the default scenario set to the one described by your NetLanguage file. You can simulate this scenario without a control window by typing 'simulate -S'.

In case you are confused, here is a picture of how things work.



6. Plotting simulation variables

The node functions in REAL have been instrumented to produce traces of certain variables as the simulation proceeds. I'll describe the nature of these traces, and then how you can plot them out on the screen or on a printer. Finally, I'll describe how you can trace additional variables.

The plotting function is implemented very simply. Node functions are asked to print out the simulation variable whenever it could change, along with the current simulation time. For example, TCP sources print out the size of the congestion control window whenever the congestion window could change. These (time, value) tuples can then be post processed to derive other statistics, such as average values, or can be plotted out.

To plot the time vs. value graphs for these variables, you must first start up a graphical display tool called tektool. You can do this by typing:

```
tektool &
```

Now, select the tektool window, and type in 'tty'. This will tell you what the tty for the tektool window is - it will be of the form ttypX, where X is an integer. To display a graph on this tty, type

```
P filename X
```

where filename is a plot file. The semantics of the plot files are described below.

To get a hard copy of the graph, you will need to talk to your local laser printer. Something of the form

```
uniq filename | graph | lpr
```

will do the job, you may need to consult your local guru for details.

You can select which nodes to plot by bringing up the node data window, selecting the plot option and setting the node data. This should be done when setting up the simulation, and is described in Section 2.4.

The files that plot creates are of the form CCCxx, where CCC is a three character identification tag and xx is the node ID of the node which is producing the plot. The currently available files are

rto: Contains the sequence of retransmission timeout values selected by the node.

rtt: Contains the estimated round trip times.

tao: Contains the *actual* round trip times.

win: Contains the congestion control window size.

seq: The sequence number of the packet transmitted at that time.

In each case, the X axis is time in seconds, and the Y axis is the data value.

To add your own instrumentation, you have to edit the node functions in sim/sim. Choose the variable that you want to plot, and at the appropriate point(s), insert the function call 'make_plot(filename, variable);'. For example, to plot the window size, the command I use is 'make_plot("win", cur_window);'. To plot real numbers, use 'make_flt_plot()'. Recompile the simulator ('make final'), and the plot files will be produced automatically for each node that has the plot option set.

7. A Note on REAL Configuration

Note that some hard limits on simulation size are defined in the file config.h. If you plan to do large simulations, you should change this file and recompile the binary. In particular, if you want to simulate more than 99 nodes, you should reset ABSOLUTE_MAX_NODES to 1000 (or some higher power of 10).

If you would like to do distributed simulation, something that is explained in the technical report and in the programmer's manual, you should #define DISTRIBUTE. (The distributed simulation facility is currently quite unpolished, but you are welcome to try it at your own risk.)

APPENDIX : NetLanguage sample

This is a sample description in NetLanguage that is in sim/lang/lang.inp. The format is explained in the NetLanguage manual.

```
header
{
network:test;
version:1;
}

nest_params {
        passtime = 1,0;
        wakeups = 0,0;
        delay = 0;
        logging = false;
        broadcast = false;
        point2point = true;
        timenow = 0,0;
        passes = 0;
        maxnodes = 30;
        monitor = custom_monitor;
}
real_params {
        inter_pkt_delay = 5.0;
        ack_size = 40.0 ;
        buffer_ratio = 3.0;
        buffer_size = 15 ;
        telnet_pkt_size = 40 ;
        ftp_pkt_size = 1000;
        ftp_window = 5;
        telnet_window = 5;
        decongestion_mechanism = 1;
        sch_policy = 3;
        router_node = 0;
        real_number  = 0;
        pulse_size = 0.0;
}

node_functions {
        {
        name = VJ_FTP;
        function = ftp_source;
        }

        {
        name = VJ_Telnet;
        function = telnet_source;
        }
        {
        name =  VJ_pulse;
        function =  vj_pulse;
        }
        {
        name = DEC_FTP_source;
        function = dec_ftp_source;
        }
        {
        name = DEC_telnet_source;
        function = dec_telnet_source ;
        }
        {
        name = DEC_pulse;
        function = dec_pulse;
        }
        {
        name = Vanilla_FTP_source;
        function = vanilla_telnet_source;
        }
        {
        name = Vanilla_telnet_source;
        function = vanilla_telnet_source;
        }
        {
        name = Vanilla_pulse;
        function = vanilla_pulse;
        }
        {
        name = Malicious_FTP_source;
        function = malicious_ftp_source;
        }
        {
        name = Malicious_pulse;
        function = malicious_pulse;
        }
        {
        name = Gateway;
        function = router;
        }
        {
        name = Sink;
        function = sink;
        }
        {
        name = CC_Router;
        function = real_router;
        }
}

channel_functions {
        {
         name = Tx_Delay;
         function = tx_chan;
        }

        {
        name = Line_Delay;
        function = delay_chan;
        }
}

monitor_functions {
        {
        name = Custom_Monitor;
```

```
		function = custom_monitor;
		}

		{
		name = Default_Monitor;
		function = nest_monitor;
		}
}

nodes {
		default {
				function = ftp_source;
				location = 50,90;
				start = true;
				repeat = true;
				halt = false;
				dest  = 4;
				plot = false;
				}

		node 1 {
				function = ftp_source;
				location = 50,90;
				}

		node 2 {
				function = telnet_source;
				location = 50,110;
				}

		node 3 {
				function = router;
				location = 100,100;
				}

		node 4 {
				function = sink;
				location = 150,100;
				plot = true;
				}
}
edges{
		default {
				bandwidth = 100000;
				latency = 0;
				channel_stack = tx_chan;
				}
		{
		from 1 to 3;
		}

		{
		from 2 to 3;
		}

		{
		from 3 to 4;
```

```
          bandwidth = 1000;
          }
}
```

REAL NetLanguage Manual

This document describes NetLanguage, a language used to describe simulation scenarios. NetLanguage is purely declarative, and is processed to create C text that is linked to the simulator. This manual describes the syntax of NetLanguage and describes how it fits into the general scheme of REAL.

1. NetLanguage Syntax

The basic syntactic entities in NetLanguage are keywords, integers, reals and strings. Standard lexical conventions are used to define these tokens. A NetLanguage description consists of a number of blocks. Each block declares a logically distinct part of the network. The blocks declare, in order, header, nest parameters, real parameters, node functions, channel functions, monitor functions, nodes and edges. Note that each block is preceded by a keyword that describes it.

1.1. Header

The header is used to provide comments to identify the simulation. It has two fields, network and version. The network field allows a user to name the network. The version field provides additional documentation. A sample header is

```
header
{
network : test;
version : 1;
}
```

These are translated into comments in the C file.

1.2. Nest Parameters

Nest Parameters are described in Nest documentation and in the REAL Users Manual. Here is a sample fragment:

```
nest_params {
    passtime = 1,0;
    wakeups = 0,0;
    delay = 0;
    logging = false;
    broadcast = false;
    point2point = true;
    timenow = 0,0;
    passes = 0;
    maxnodes = 30;
    monitor = custom_monitor;
}
```

An entry of the form a,b is a time value that represents 'a' seconds and 'b' microseconds. The monitor declared should be one of the monitor functions that REAL implements (usually custom_monitor). For most purposes, this header can be copied without any changes (except perhaps for the maxnodes entry, and this entry should not be more that 99).

1.3. REAL Parameters

REAL paramters are described in the REAL Users Manual. Here is a sample fragment.

```
real_params {
        inter_pkt_delay = 5.0;
        ack_size = 40.0 ;
        buffer_ratio = 3.0;
        buffer_size = 15 ;
```

```
                telnet_pkt_size = 40 ;
                ftp_pkt_size = 1000;
                ftp_window = 5;
                telnet_window = 5;
                decongestion_mechanism = 1;
                sch_policy = 3;
                router_node = 0;
                real_number  = 0;
                pulse_size = 0.0;
        }
```

1.4. Node functions

This block declares all the node functions that the simulator will use. The syntax is straightforward. The only stylistic note is that bracketed subblocks should not be terminated by semicolons. Also, function names should be a single word. You may use as many underscores as you please, but if you use more than one word, the language preprocessor will assume a syntax error (this needs to be fixed). Here is a sample set of node function definitions:

```
        node_functions {
            {
            name = VJ_FTP;
            function = ftp_source;
            }

            {
            name = VJ_Telnet;
            function = telnet_source;
            }

            {
            name = Gateway;
            function = router;
            }

            {
                name = Sink;
                function = sink;
                }
        }
```

1.5. Channel and Monitor functions

These are almost identical to the node function declarations. Here is a sample declaration:

```
        channel_functions {
            {
            name = Tx_Delay;
            function = tx_chan;
            }

            {
            name = Line_Delay;
            function = delay_chan;
            }
        }
```

```
monitor_functions {
        {
        name = Custom_Monitor;
        function = custom_monitor;
        }

        {
        name = Default_Monitor;
        function = nest_monitor;
        }
}
```

1.6. Node declarations

Node declarations describe the nodes in simulation. Here is an example that I will discuss:

```
nodes {
        default {
                function = ftp_source;
                location = 50,90;
                start = true;
                repeat = true;
                halt = false;
                dest  = 4;
                plot = false;
                }

        node 1 {
                function = ftp_source;
                location = 50,90;
                }

        node 2 {
                function = telnet_source;
                location = 50,110;
                }

        node 3 {
                function = router;
                location = 100,100;
                }

        node 4 {
                function = sink;
                location = 150,100;
                plot = true;
                }
}
```

To make it easier to declare the nodes, the first declaration in the block is the default declaration. The default defines the parameters of each node unless they are overridden by an explicit redeclaration inside some node. You are not allowed to skip any entries in the defaults declaration, but you may skip as many entries in the node declaration as you wish.

The number following the keyword 'node' is the node ID of that node. The function field is the name of the C function that should run on that node. The a,b entries in the location field are the Cartesian coordinates of the node in the control window. (The top left corner is 0,0, and 50 units is about an inch.) Note that you need not specify the location unless you want to see the network in the control window. In particular, if you plan to do a stand alone simulation, you can specify the default location to be 0,0. You should usually set start and repeat to true and halt to false. Dest is the node ID of the sink node to which data is sent. Finally, you should set plot to true if you want the node function at that node to generate plots.

1.7. Edge Declaration

Here is a sample edge declaration:

```
edges{
    default {
        bandwidth = 100000;
        latency = 0;
        channel_stack = tx_chan;
        }
    {
    from 1 to 3;
    }

    {
    from 2 to 3;
    channel_stack{
            tx_chan;
            delay_chan;
          }
    }

    {
    from 3 to 4;
    bandwidth = 1000;
    }
}
```

As with the node declarations, there is a default declaration. However, there is an exception here: NetLanguage will allow you to have only a single function on the default channel stack. Thus, if you want to have multiple channel functions on the channel stack of an edge, you must explicitly declare it each time. As with node defaults, you are not allowed to skip any fields.

The from..to.. field in the edge declaration declares the node IDs of the endpoints of the edge. The bandwidth field will accept both integer and real values. The latency in transmission is measured in microseconds and has to be an integer. The channel stack is the stack of functions that are called in turn by sendm() when a message has to be sent on that edge.

1.8. A note on usage

Usually, you will not need to change the nest parameter and function declaration sections of the language file. In most cases, you should just copy these from the file sim/lang/lang.inp.

2. The Place of NetLanguage in REAL

NetLanguage completely describes a simulation scenario, that is, the topology, protocols, workload and control parameters. Thus it saves the state required to repeat a simulation: you can redo a simulation at a later date by saving its NetLanguage description.

Since NetLanguage is a human readable ASCII description it can be conveniently edited. You can take the description of one scenario and process it to create another. This is useful when generating large simulation scenarios that are painful to draw on the control window.

While it is possible to write out a NetLanguage description by hand, it is more convenient to have someone write it for you automatically. Precisely this facility is provided by the 'Write Graph' command in the pop up menu in Layer 5 of the control window. When you select this option, a file called LANG_FILE that describes the current scenario will be created in sim/sim. You may save this file as a record of that simulation or can edit it to create new scenarios.

A NetLanguage file has to be processed before it can be used in the simulator. This processing proceeds in two steps. First, the language description must be translated into C code. The processor is sim/bin/lang, and it will produce two files in sim/lang: lang.c and extern.lang.h. (You can move these files to sim/sim or should set up soft links from sim/sim to these files.) The invocation for this step is

```
lang < filename
```

Next, this file should be linked into the simulator. You can do this by typing

```
make final
```

in sim/sim. The resulting executable, simulate, will now have its default scenario set to the scenario that you had described in the NetLanguage file.

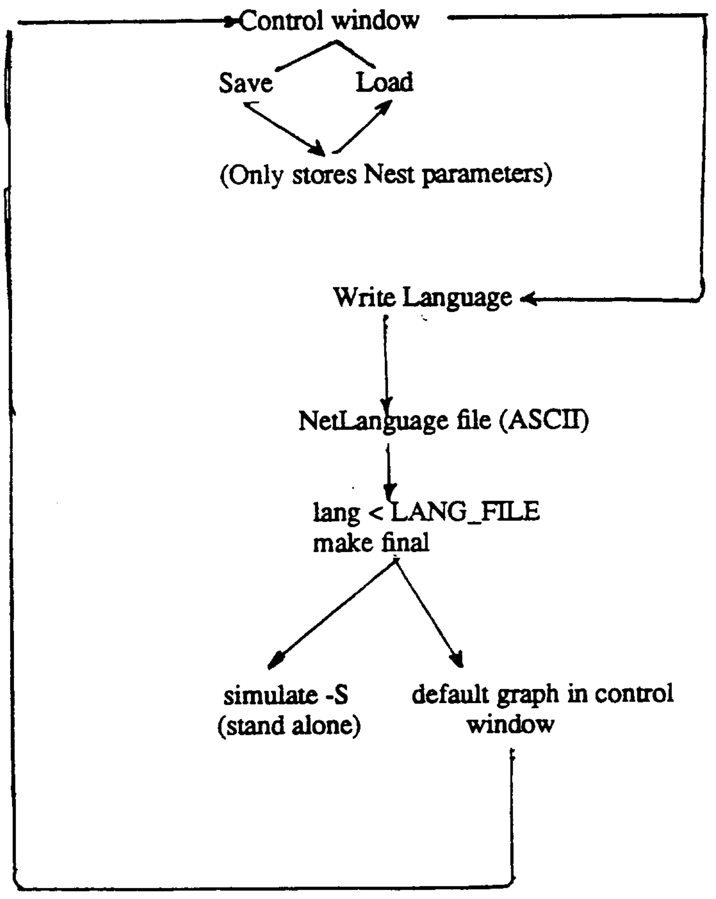
You now have two options. If you would like to the simulation to proceed without any changes, and with no communication with a control window (for example, if you don't have a Sun) then you can run the simulator in stand-alone mode[1]. To do so, type

```
simulate -S
```

Otherwise, proceed as described in the Users Manual, and the default graph that the simulator will come up with will be the one described in the NetLanguage file. There is a bug here, in that the display panel for the REAL parameters will not reflect the values that you placed in the NetLanguage file. What you see is not what you get, but what you get is what you want. In any case, the simulation report will have the correct values. Remember that if you want to retain the REAL parameters from the NetLanguage file, you shouldn't do a 'Send Parameters'.

[1] When you run in stand-alone mode, the simulator will produce exactly the same outputs as it does otherwise. The only difference is that you cannot control the simulation while it is running

The general scheme of things is presented in the figure below.



3. Acknowledgments

The use of defaults in declaring node and edge characteristics was suggested by Ravi Sethi.

REAL Programmers Manual

1. Introduction

This manual provides a detailed description of REAL for users who wish to modify the code or add to it. Since REAL code (ignoring modified Nest code) spans some 7500 lines of C, it is impossible to describe all of it in any detail. Instead, I will give a tour of the files comprising REAL, and then some recipes for things like writing node functions, packet switches and control window code.

You should have read the set of 5 documents on Nest from Columbia. These aren't very complete, but at least they will give you an idea of what is going on. You will also need to read the code to fully understand the system.

2. Tour of the files

The files in REAL can be divided into the following logical classes:

Node functions:	These are the functions that execute protocols in nodes.
Queue management and routing:	These manage buffers in nodes and gateways, and packet switching.
Distributed simulation:	These are functions to implement DisREAL.
NetLanguage generation:	Allow automatic generation of NetLanguage descriptions.
Miscellaneous	

These are discussed in subsequent sections.

2.1. Node Functions

Node functions implement computation at each of the nodes in the network. There are three types of node function: source, router and sink. I will discuss them in this order.

2.1.1. Source

There are eleven source node functions, but almost all of them implement variants of TCP-like transport layer functionality. The source node function from which all others are derived is vanilla_ftp(). ftp_source() adds Jacobson's congestion control modifications. dec_ftp() is vanilla_ftp with the additions required to do window adjustment the DEC way. malicious_ftp() sends out packets as fast as it can, ignoring all flow control signals. The corresponding telnet sources incorporate modifications to set the sleep time, which is the time that the source will sleep before it tries to send another packet. Pulse sources are modified ftp sources that send out a burst of flow controlled packets periodically.

I will now describe *vanilla_ftp.c* as an example of writing a node function with flow control. The file starts with some standard headers. The function then declares some protocol variables (such as round_trip_time, last_ack etc.).

```
#include "../src/nest.h"
#include "../src/graph.h"
#include "../src/defs.h"

#include "config.h"
#include "types.h"
#include "parameters.i"

extern
```

```
enq(), num_in_q();          /* for queueing retransmitted pkts */
extern PKT_PTR  deq();      /* " */
extern TABLE   rt_time[MAX_NODES + 1]; /* round trip time stats */

vanilla_ftp_source()
/* no window adjustment or timeout deviation estimation */
{
    /* Misc. variables */
    PKT_PTR      pkt;
    int          num, node_number, line_busy = 0;
    ident        destn, sender, sink;
    long         key;
    timev        now, gen, diff;
    float        err, timeout;

    /* state variables */
    timev        time_sent[MAX_WINDOW_SIZE];
    float        round_trip_time, mean_deviation;
    int          last_ack, seq_no, last_sent, min_window;
```

The first executable statement is 'stop_time()'. It asks Nest to stop the clock while the node executes. This corresponds to the assumption that simulation nodes are infintely fast. (You should always start your node functions with stop_time().) The subsequent lines set some standard variables such as 'node_number', the simulation node number for that instance of the node function, and 'sink' - the sink to which this node will send data.

```
/* dont let any simulation time pass while the node executes */

  stop_time ();
  node_number = get_node_id ();
  sink = assigned_sink[node_number];
```

All nodes are assigned a node type of -1 when they start (in *init.c*). So, we know that a node is executing for the first time when we see a -1 in source_node_type[get_node_id()]. When vanilla_ftp notes this, it does some initialization of its type and other state variables.

```
    if (source_node_type[get_node_id()] is - 1)
        /* first time this node has been executed */
    {               /* initializations */
        hold();
        printf(" A new ftp source, node is %d  0, get_node_id());
        printf("assigned sink was %d0, sink);
        release(1);
        source_node_type[node_number] = VANILLA_FTP_SOURCE;
        round_trip_time = 10.0;
        mean_deviation = 0.0;
        last_sent = last_ack = -1;
        seq_no = 0;
        alpha_rtt_ftp = (float) (1 / (float) 10.0);
        /* pretend that you got  an INT pkt, go to test */
        goto test;
    }
```

The label 'recv:' is the point at which packets are received by the node. This is done by the 'recvm()' function. Now, the packet is processed according to its type. (This processing is essentially a state machine. It is a good idea to first draw out the state diagram, and then write code around it, which is what I did.)

```
recv:
    sender = recvm (&destn, &key, &pkt);
    switch (pkt->type)
   /* use packet type to decide what to do */
    {
```

When an acknowledgment is seen (type ACK), we first check for duplications (if its sequence number < last_ack). Since acks can be aggregated, all the packets that are acknowledged by the ack have to be ticked off, and their contribution to the round trip time marked. (These RTT estimates are inaccurate, but generic TCP doesn't bother about that.) make_flt_plot(), described in the users manual, generates data for plotting.

```
    case ACK:
      {
        timev       now, diff;
        float       tao;
        int         i, last;

        now = runtime ();
          /*
           * last_ack is the highest seq_no that has
           * been acked
           */

        if (pkt->seq_no > last_ack)
          /* ack not seen before */
        {
          last = last_ack;
          for (i = 1; i <= pkt->seq_no - last; i++)
          {
            diff = time_minus (now, time_sent[(last_ack + i)
                                        % telnet_window]);

                      /* actual round trip time */
            tao = make_float (diff);
                      /* estimated round trip time */
            round_trip_time = round_trip_time +
              alpha_rtt_ftp * (tao - round_trip_time);
                  /* plot data */
            make_flt_plot ("tao", tao);
            make_flt_plot ("rtt", round_trip_time);
            make_entry (tao, &rt_time[node_number]);
        }
                /* update state */
        last_ack += (pkt->seq_no - last);
    }
    hold ();
            /* get rid of the ack packet */
    free (pkt);
    release (1);
    goto test;
}
```

INT packets are generated from the trunk, and are used to indicate that the data has been sent out from the output line. When this happens, since this is a ftp node, and we always have data to send, we generate a new packet and send it. ftp_window is the window size set in the control panel.

```
        case INT:
            /* line is now free */
            hold();
            line_busy = 0;
            free(pkt);
            release(1);
test:
            /* figure out window size */
            min_window = ftp_window;
            make_plot("win", min_window);
```

If we have a retransmission pending (i.e. it was generated, but the line was busy so the packet couldn't be sent) then it must be sent before any data packet. The function num_in_q() returns the number of retransmission packets that are pending. We check to see if any of these packets have been acknowledged since the time they were enqueued, and if they were, they are discarded. Else one of them is sent out.

```
num = num_in_q (node_number);
while (num-- != 0)
   /* send the timeout packet */
   /* but only if an ack has not been received in the interim */
{
   pkt = deq (node_number);
   if (pkt->seq_no > last_ack)
      /* ok to send , even if later */
   {
      if (pkt->seq_no <= last_ack + min_window)
         /* ok to send now */
      {
         gen = pkt->gen_time;
         diff = time_minus (now, gen);
```

```
                timeout = beta_rtt * round_trip_time;
                pkt->gen_time = runtime ();
                line_busy = 1;
                num_retransmissions[node_number]++;
                ftp_safe_send (pkt, timeout, 0, &last_sent);
                        goto recv;
            } else     /* save for later */
                enq (node_number, pkt);
        } else {
            hold ();
            free (pkt);
            release (1);
        }
        /* drop pkt on the floor */
    }
```

Two things can prevent a packet from being sent out: the flow control window can be full, or the output line can be busy. The test: label tests for both cases, and if both of these conditions are absent a packet is generated and transmitted. (We need to test for a busy line since we can jump to 'test:' from more than one location, and it is not always true that the line is free at that time.) The dest field is processed so that local nodes get the correct value even in the global (DisREAL) name space. The time at which the packet is sent is remembered, and the timeout value is set. A call to ftp_safe_send() initiates packet delivery. This routine generates timeout packets and makes sure that timeouts are generated in nondecreasing order in virtual time.

```
/* must have a window, and the line should be free */
if (last_sent < last_ack + min_window && !line_busy) {
    hold();
    /* cons up a packet */
    pkt = (PKT_PTR) malloc((unsigned) sizeof(PKT));
    release(1);
    /* set parameters */
    pkt->size = ftp_size;
    pkt->gen_time = runtime();
    pkt->type = FTP;
    pkt->seq_no = (seq_no)++;
    pkt->dest = assigned_sink[node_number];
    if (pkt->dest < ABSOLUTE_MAX_NODES)
        pkt->dest += ABSOLUTE_MAX_NODES * realnum;

    pkt->alpha = alpha_f;
    pkt->source = ABSOLUTE_MAX_NODES * realnum + node_number;

    /* update state */
    time_sent[(pkt->seq_no) % ftp_window] = runtime();
    line_busy = 1;
    timeout = beta_rtt * round_trip_time;

    /* send out the packet */
    ftp_safe_send(pkt, timeout, 0, &last_sent);
}
break;
```

If a timeout is received, the packet has to be retransmitted. We first check if the timeout was spurious. If so, we ignore the timeout. Else, the retransmission counter is updated. If we are able to send the packet, we do so, else it is enqueued to be dealt with later.

```
            case TIMEOUT:
                /* may have been acked */
                if (pkt->seq_no > last_ack)
                    /* cant ignore */
                {
                    /* resend */
                    now = runtime();
                    pkt->type = FTP;
                    hold();
                    min_window = ftp_window;
                    /* can send the pkt */
                    if (pkt->seq_no <= last_ack + min_window && !line_busy) {
                        timeout = beta_rtt * round_trip_time;
                        line_busy = 1;
                        pkt->gen_time = runtime();
                                        num_retransmissions[node_number]++;
                        ftp_safe_send(pkt, timeout, 0, &last_sent);
                    } else
                        /* can't send : save it */
                        enq(node_number, pkt);
                    release(1);
                } else
                    /* dump pkt */
                {
                    hold();
                    free(pkt);
                    release(1);
                }
                break;
            case DROPPED:
                /* kludge - not used any more */
                free(pkt);
                break;
            default:
                hold();
                free(pkt);
                release(1);
                pr_error("ftp_source recvd. a pkt of unknown type ");
            }
        }
    }
```

This basic structure of a source described here is modified to implement the changes made by Jacobson and by Jain. It is best to read their papers/technical reports before reading the code.

The major addition that the telnet code makes is to incorporate handling for a WAKEUP packet that indicates that it is time for the next packet to be generated. The 'sleepy' variable indicates that the source is not active at that time. Inactive hosts are allowed to send retransmissions, but not new data packets.

Pulse sources use the pkt->alpha field to generate inter-pulse spacings. They set pkt->alpha, an otherwise disused field, to signal to a channel function that they have sent the end of a pulse. This is used by the channel function to call reliable() and send a PULSE packet back to the source to start off the next pulse.

One major limitation of Nest is that it allows only one thread of execution per node. Since a timer is a logically separate thread of execution, it is difficult to set timeouts. In REAL, timeouts are set by sending oneself a packet that will return after a known delay. When the timer packet is received, it is as if we received a timer interrupt and this event is used to trigger off the actions that should happen at the interrupt.

The problem with this is that only channel functions are allowed to call 'reliable()' which is the only function that knows how to send packets with a known delay. So, channel functions have to be involved in setting timeouts. A node function computes at what time it wants TIMEOUT, WAKEUP and PULSE packets, and tell this to the channel function. The channel function calls reliable() on behalf of the node function and things work out. The scheme is quite ad hoc, but that is all that is available at the moment. What we really need is a way to allow reliable() to be called from a node function. For example, the file *reliablenest.c* contains a modified version of reliable() that can be called from the monitor. An equivalent for node functions can probably be cooked up.

2.1.2. Router

router.new.c is the file that contains the node function for the switch. The router is complicated mainly by the fact that DEC routers need a lot of processing. The basic functionality is actually not too complicated. Let us step through the file. I will skip parts devoted to processing DEC related stuff. So, it will not appear here, even though it is present in *router.new.c*.

The file opens with a declaration of a debugging flag R_DEBUG. If this flag is set (usually in *switches.c*), then some debugging information will be printed out as the simulation proceeds. There are a number of such flags, and to use them, you should modify and recompile *switches.c*.

Following the standard includes, the state variables used by fair queueing and DEC queueing are declared. The DEC variables are complicated, and you should read the algorithm described in their technical report (part IV) to see what they mean. To save on array sizes, output lines are numbered from 1 to MAX_FAN_OUT. The mapping is stored in the array q_number and is done during the first time the node is executed. During this time, some other simulation variables are also initialized.

```
int         R_DEBUG = 0;
/* Router debug flag */
extern int    LITTLE_DEBUG;
extern int    DEC_DEBUG;

#include "../src/nest.h"
#include "../src/graph.h"
#include "../src/defs.h"
#include "config.h"
#include "types.h"
#include "parameters.i"

extern PKT_PTR  select_from_op_q ();
extern timev  *convert ();
extern int    q_num_of_conv[MAX_NODES + 1][MAX_CONVERSATIONS];
extern int    num_active[MAX_NODES + 1][MAX_FAN_OUT];
extern TABLE   so_tossed[MAX_NODES + 1];    /* Number of packets
                                  * tossed by source */
extern TABLE   so_gw_wait[MAX_NODES + 1][MAX_NODES + 1];
extern int    num_buffers_allocated[MAX_NODES + 1][MAX_FAN_OUT];

/* some code skipped */
/* set queue numbers */
for (i = 0; i < MAX_NODES + 1; i++)
     if (route[node][i] is i)
          q_number[i] = count++;
```

The main loop of the router gets a packet and processes it. After the packet is received, some state variables are set up - so: the source, dest: the immediate destination, final_dest: the ultimate destination, qnum: the line on which to send, conv_id: the conversation to which the packet belongs.

```
for (ever) {
     /* get a packet */
     sendr = recvm(&destn, &key, &pkt);
     safe_print("received a packet at the router ", pkt);

     /* set up useful values */
     so = pkt->source;
     final_dest = pkt->dest;
     if ((final_dest / ABSOLUTE_MAX_NODES) is realnum) /* local */
          dest = route[node][pkt->dest % ABSOLUTE_MAX_NODES];
     else
          dest = route[node][cc_router];
     /* dest is the actual next node - not the eventual destn */

     conv_id = hash(node, so, final_dest);
     /*
      * if this is a new conversation, conv_id will be assigned a
      * new number, else it is an old value
      */
     qnum = q_number[dest];
     q_num_of_conv[node][conv_id] = qnum;
     /* qnum is the output line on which to send */
     now = runtime();

     pkt->arr_time = now;
```

Now the packet is processed according to its type. If it is of a type that needs to be forwarded, the scheduling policy is used to decide what has to be done to the packet.

```
     switch (pkt->type) {
     case TELNET:
     case FTP:
     case ACK:
```

For fair queueing (FQ) gateways, the arrival of a packet is an event which triggers recomputation of state (the round number). Since this computation is rather involved, it is shunted to function new_arrival(). new_arrival() will also create conversation state for new conversations. Since this needs to be done for other policies as well, check_conv() is used to check for state, and if it doesn't exist, add_conv is used to add this freshly started conversation to the set of conversations. Note the implicit establishment of connections.

```
if (policy is FQ || policy is HOMEBREW)
    /*
     * discard any conversations that have become
     * invalid and also establish a checkpoint
     */
    new_arrival(conv_id, qnum, pkt, dest);
else if (!check_conv(conv_id, qnum))
    add_conv(conv_id, qnum);
/*
 * FQ will update the # of conversations
 * automatically - else we have to figure out if a
 * new conversation has started on the line, and if
 * so, add this conversation to the list of
 * conversations on the line
 */
```

Ignoring manipulations for DEC, the next step is to check if the packet can be sent out right away (if the output line is free).

```
if (busy[node][qnum])
    /* line to dest is busy , so buffer */
{
    safe_print("line was busy", pkt);
```

If it isn't we check if there are enough buffers to store the packet. If there aren't enough buffers, then we need to drop a packet. Depending upon the decongestion policy selected, we drop a packet from the head, tail or at a random position in the queue. If the scheduling policy is FCFS, instead of dequeueing, we will just drop the incoming packet (unless we have to do random decongestion, in which case, decongest_random() is called).

```
/* decongest if no more buffers */
if (policy != FCFS_FCFS_PKT && policy != DECBIT) {
    if (all_op_q_full(qnum)) {
        switch (decongest_mechanism) {
        case 1: decongest_last(qnum);
            break;
        case 0: decongest_first(qnum);
            break;
        case 2: decongest_random(qnum);
            break;
        }
    }
    /* in any case, chuck into a buffer */
    if (put_in_op_q(conv_id, pkt) == ERROR)
        pr_error("Bug in decongestion code ");
} else {      /* policy is fcfs or decbit */
    if (all_op_q_full(qnum)) {
```

```
                                   if (decongest_mechanism is 2) /* random */
                                                           decongest_random(qnum);
                                   else {
                                        /*
                                         * drop this packet
                                         * - so do nothing
                                         * ...
                                         */
                                        free(pkt);
```

Once buffers have been created, the packet is placed into the appropriate queue.

```
                                   if (put_in_op_q(conv_id, pkt) == ERROR)
                                                           pr_error("Output buffer full - CONGESTION ");
```

If there is space in the buffers, the packet is placed in it, and buffer length statistics are updated. Other state regarding time of update is also set.

If the line was free, then the packet sent out, and appropriate statistics regarding the packet are collected. If a bit needs to be set, this is done now.

```
                                   if (set_bit(conv_id, pkt)) {
                                        pkt->decbit = 1;
                                                  }
                              sendm(dest, 0, pkt);
          /* rest of file skipped */
```

If an INT packet arrives, the processing is similar to the case when the line is free. do_transition() and set_bit() do some processing needed to implement DEC bit setting.

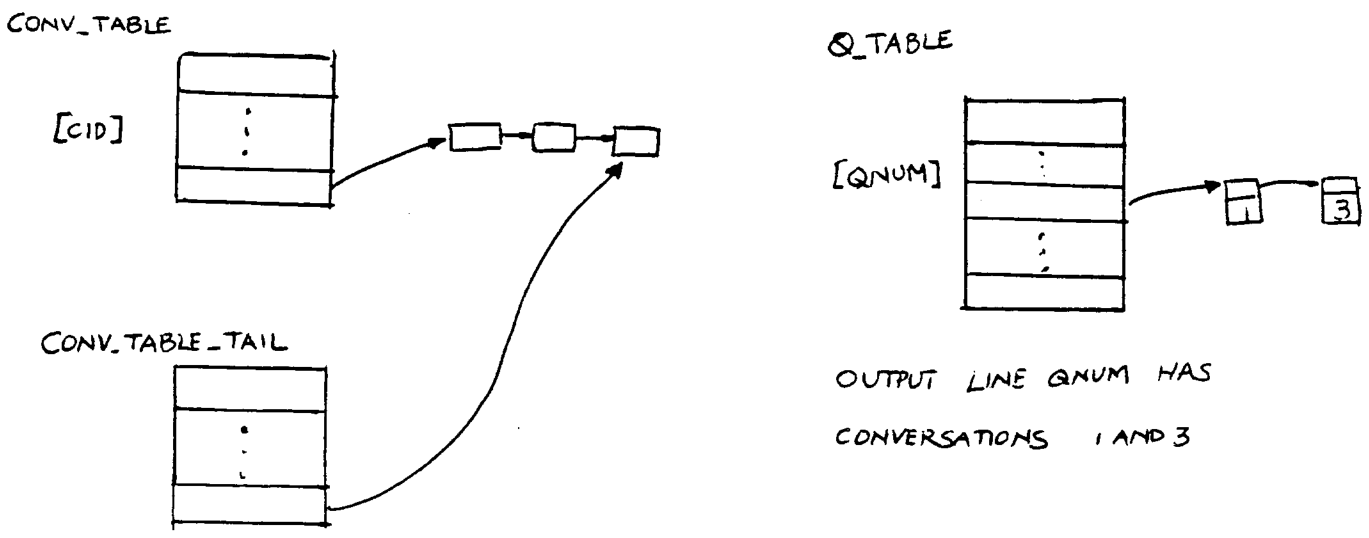
2.1.3. Sink

The sink node function is a universal receiver. It acknowledges every packet it receives with an ack packet that contains the sequence number of the last in-sequence packet it has seen. The sink uses a small trick - the packet it receives is converted into an acknowledgement merely by changing its parameters.

2.2. Queue Management and Routing

The queue management functions are written in an object oriented and layered style. The queue objects are manipulated by a small set of functions. Each layer provides services that the layer above uses to provide its own services.

The data structures for the queues are in *queues.c*. Packets are buffered in a per-conversation linked list and are accessed by two pointers: one points to the packet at the head of the queue, and another points to the tail. Each packet has a field that points to the next packet in the queue.



Conversations are indexed by a conversation identifier (CID) that uniquely identifies each source-destination pair. The CID is equivalent to a virtual circuit number, and is used to manage resources that are allocated to that conversation. The packets queued for a conversation with CID = C are pointed to at the head by conv_table[C], and at the tail by conv_table_tail[C]. q_num_of_conv[C] is the number of the output line on which C sends packets.

Given the number of an output line, we need to keep track of the set of conversations that are using it. This is managed by a linked list of conversation structures that contain the CIDs of the conversations on that output line. q_table[Q] points to the head of the linked list of conversations that share output line Q.

The state of the queues is summarized in two arrays - num_in_op_q[C] is the number of packets in conversation C's queue. num_buffers_allocated[Q] is the total number of buffers used by output line Q.

The functions defined in *queues.c* allow access to elements of queues, and to the queue state. init_q() initializes queue state. all_op_q_full() and op_q_mt() allow read access to the queue state. put_in_op_q(C) places a packet at the tail of conversation C's queue. get_from_op_q() gets packets from the head. dequeue(p) removes a packet pointed to by p from its conversation queue without deleting it. get_num_in_op_q() returns the number of packets in a conversation's queue. fc_fs() returns a pointer to the the packet at the head of a queue (if this packet has to be removed, it must be explicitly dequeue). get_num_active() returns the number of conversations on an output line that have packets queued to be sent. dump_q(C) prints out the packets queued in the queue for conversation with CID C.

The next layer handles decongestion (packet dropping) using the functions defined above. decongest_first(Q) drops a packet from the head of the queue of the conversation that has the most packets queued up in output line Q. decongest_last(Q) acts similarly, but drops a packet from the tail of the longest queue. decongest_random(Q) drops a packet chosen at random from the set of all packets queued at that output line.

Conversation management is done using three functions add_conv(), delete_conv() and check_conv(). check_conv(Q) checks if a given CID is present in the list of conversations using output line Q.

Scheduling policy management is done in layer 3 of the queueing hierarchy. get_fcfs_pkt(Q) returns a pointer to the packet that arrived first at output line Q and at the same time it dequeues this packet. Thus, to implement FCFS scheduling, the router has to make a call get_fcfs_pkt(Q), and the appropriate packet will be presented to it.

get_min_bid(Q) returns the packet that has the least bid amongst all the packets queued on output line Q. find_min_finish(Q) returns the minimum finish number amongst all the packets queued on output line Q. It is used to recompute the round number in Fair Queueing.

The top layer of queueing deals with packet scheduling and is done in *scheduler.c*. The function select_from_op_q() looks at the policy that has been selected for the simulation and calls get_fcfs_pkt() or get_min_bid() accordingly. Some policies in select_from_op_q() have been declared but not implemented. These used to be implemented in an earlier incarnation of the simulator, and have not been ported to this version. The functions new_arrival() and remove_old_conv() do some processing needed for Fair Queueing.

The entire queueing hierarchy exports a single function: select_from_op_q(), which is called by the router. Once the policy has been specified, a call to this function automagically returns with the appropriate packet.

node_queue.c provides queue management for retransmitted packets in a node. The interface is through the functions enq(), deq() and num_in_q() which place a packet at the tail of its queue, remove it from the head, and tell the queue length.

2.3. Distributed simulation

The distributed version of REAL is called DisREAL. Distributed simulation allows users to spread a simulation across several machines. The basics of distributed simulation are described in the technical report. Unfortunately, it does not seem likely that this facility will be useful in simulating networks with high speed links.

The files that fall in the category are:

cc_router.c This contains the code for real_router() which implements cross computer routing. When this node function receives a Nest packet, it places the packet in a buffer where the monitor can pick it up. At the end of the pass, the monitor converts the virtual packet to a real packet and sends it over a BSD socket to the simulation master. If the packet is not destined for a remote host (i.e. its high order digit is the same as the local REAL number) then it is merely forwarded on its way. There is a bug here - the packet is sent even if the INT packet from the previous packet has not arrived (that is, the outgoing line is assumed to be infinitely fast). This has to be fixed by inserting code from *router.new.c*, or modifying *router.new.c* to do cross computer routing. This will have to be done before DisREAL gets used in any big way.

coord.c This file contains functions that a REAL monitor links in to be able to coordinate its simulation with other REAL simulations. The function init_socks() opens a BSD socket ready to receive on MASTER_PORT. The function create_receive_connection() defined in *sockets.c* takes care of the details necessary to do this.

coord() works in two phases. In the first phase, it sends the master all the packets that had been stored by cc_router() and follows these by an END packet. It then waits for the master to send it a message. If the message is a packet, the packet is delivered using deliver_pkt(). If is a PROCEED signal, coord() returns to the monitor, and the simulation proceeds. *coord.c* also provides the queue management necessary to buffer packets stored by cc_router(). These are the functions add_cc_pkt() and get_cc_pkt().

machines.c The file *machines.h* contains a list of names of machines on which to run DisREAL and their REAL numbers. create_machine_list() creates a list of machines read from the file machine.h. step_machine() steps through the list, returning pointers to machines and their associated REAL numbers.

real_master.c The master function performs two functions: synchronization and routing. The master does simple message based synchronization: when END messages have been received from all the monitor functions in the distributed simulation, PROCEED messages are sent out to them. The communication between monitors and the master is through packets with a format defined in *real_master.c*. The main loop of real_master reads packets off from any socket that is ready (using 'select()'). The packet is then processed according to its type.

The master has to decide two things: on which socket to send out the packet, and how much delay the packet should receive. The first is done by figure_out_sock(), the the second by set_delay(). At the moment figure_out_sock() merely divides the global destination address by ABSOLUTE_MAX_NOPDES, and takes the integral part as the realnum of the destination. set_delay() assumes that the cross computer delay is always 1.5 seconds and computes delays on this basis. This value is written into the packet, and the packet is sent out to the destination simulation. For example, if a packet was generated 1 second before the pass time, then the destination computer is asked to deliver the packet 0.5 seconds after it receives it.

Clearly, these functions have to be made more general. The delay on the cross computer line should be used to determine the additional delay that the packet should be given in the destination simulation. The routing should involve a global routing table, and perhaps policy routing can be done. However, this requires something like EGP between all the monitors, and I am not too enthused about implementing that.

If the master receives an END packet, then the master checks if all the monitors have sent out END messages. If they have, then the PROCEED message is sent to all the monitors.

The basic socket manipulation code is in *sockets.c*, and most of this has been stolen from Stuart Sechrest's socket primer.

2.4. NetLanguage generation

NetLanguage has been extensively described in the NetLanguage manual. Here, I will describe how to add features to NetLanguage, and a little bit about automatic NetLanguage generation.

NetLanguage is implemented using lex and yacc, and I expect that you know how to use these tools. The files for the language are in sim/lang. lang.lex contains definitions for creating a lexical analyzer using lex. The definitions are pretty straightforward. lang.yacc is the yacc input file. Note the complications introduced by linked lists: one needs to differentiate between the first function in a list and the rest. The actions are generally printf statements to generate C code that will actually create the graph structures. This kind of indirection is needed in order to be able to link in functions whose addresses are not known at structure generation time.

The automatic generation of NetLanguage poses an interesting situation. The basic problem is that the simulator knows about function addresses, but not their names. Hence, it needs to access the symbol table and read off the function names to get back the ascii names. I initially tried to decipher the a.out file for the simulator, but that was too complicated. Instead, I invoke the 'nm' system command to extract these names. The result is filtered through an awk script *nm.awk* and read into an internal symbol table organized as a binary tree keyed on function addresses. This is then used to translate from function addresses to ascii names.

tree.c contains functions to create a generic binary tree and manage it. *symtab.c* contains functions to create a symbol table and access it. The file *writelang.c* goes through the graph structure and prints out a language file. If the language definition is changed, *writelang.c* should also be modified.

2.5. Miscellaneous

channels.c This file contains two channel functions called tx_chan() and delay_chan(). tx_chan() is used to add transmission delay to a packet. It looks at a packet size and the transmission line speed to determine the transmission delay for a packet. It then calls delay_chan() (note that I don't use the Nest chan_stack facility - I didn't understand it when I wrote this). delay_chan() looks at line characteristics and adds latency delay. It also sends ghost (INT) packets back to the sender to tell it that the line is now free.

NEST does not allow reliable() to be called from a node function. This a major problem with timeouts, since there is no way for a node to set a timeout value directly. To get around this, delay_chan() has been kludged to generate timeouts, pulse signals etc. The code is not pretty, but it is the best I could do.

config.h This file contains configuration paramters and other useful definitions. If you want to do large simulations, you must make sure that you do not exceed the limits defined in config.h. If you do, reset these values and recompile the binary.

sim.c This has main() for the simulator. It reads in command line arguments, and then calls simulate(), never to return. The file declares globals and simulation parameters that are externed to the other files through parameters.i. *sim.c* #includes *lang.c*, which is a C file created by processing a NetLanguage description. This becomes the default graph for the simulation. If a stand alone simulation (simulate -S) is specified, then this is the network to be simulated.

hash.c This file contains functions to translate from a source destination pair to a conversation ID. It used to be a hash table, but I found that a plain old array works with much less trouble, so I switched back to Neanderthal coding. The file exports two functions: hash() and invert().

init.c Some initializations of queues, tables and other simulation variables.

monitor.c This file contains custom_monitor(), which is the monitor function that REAL uses. custom_monitor() does a number of things. The basic functionality is to open a private socket to the control window (as opposed to the NEST socket), and then to listen on that socket for commands. The function also calls coord() to coordinate with other REALs to do distributed simulation. Since the monitor is called at every pass and can access all global variables, it is used to do things like printing out time ticks, printing out simulation reports etc. Here is a guided tour through the file.

After includes and declarations, the monitor checks to see if this is the first time it is executing. If so, it does some initializations, prints out the simulation parameters using dump_parameters() and tries to open a socket to the control window. The variable Log_mon_connected is set if the socket is able to open. If the socket cannot be opened right away, a half second timeout is set, and we try again.

When connection is established, the monitor listens for a message. nbrecv() from *sim/src/nbsock.c* does a non blocking receive (if a select for read fails, the recv falls through). The received message is then serviced by service(). At this point, if distributed simulation is enabled, coord() is called.

service() looks at the first byte of the message to see if it is a special control message (one of dump, restart, kill or writelanguage). If it is, it is used to execute a function. This is equivalent to a non-blocking remote procedure call. If the message is not a special command, it is assumed to be a 'send parameters' message, and the simulation parameters are set to values received over the socket.

try_to_dump() tests if the time now is a multiple of the control parameter DUMP_INTERVAL (defined in config.h). If it is, a simulation report is printed out.

set_assigned_sinks(), set_plot_options() and set_line_speeds() set these values from the parameters received over the Nest socket as part of the graph structure.

plotting.c This file contains two functions make_plot() and make_flt_plot() that are described in the user manual.

routing.c The function route_graph(g) in this file takes a pointer to a graph structure, and figures out the shortest hop-count routes between all pairs of nodes. This implements Dijkstra's all pairs shortest path algorithm, and is based on pseudocode in Bertsekas' networks book. The routing is static and centralized. If we do true EGP like exchange of routing tables between REALs, this has to be rewritten.

switches.c This file collects debug flags defined all over the place. The idea is that setting a flag will involve recompiling a small file, and a relink, instead of recompiling a large file. Also, it is easier to set a number of flags all at once.

table.c This contains the functions necessary to manipulate and print out a report of a simulation. It exports two functions - make_entry(t) and make_time_entry(t) that stores floats and timevals respectively into the table that t points to. Tables are stored with two fields - mean and num_entries. The mean stores the sum of all the entries and when it has to be printed out, this is divided by the number of entries. One major problem is that the formatting is very sensitive to changes. If you change any fields, good luck!

exptest.c This is used to check if the random number generator on the machine you are using is up to par or not. It just builds up a histogram of exponentially distributed values, and prints it out. You can eye-ball this using graph(1) or run it past your favorite randomness tester to be sure that you are not let down by the simulator's random number generator. On a Version 9 Unix, use 'grap |pic|troff|lp', on 4.3BSD use 'graph -g1 -l |lpr -g'.

Makefile The makefile is quite standard. Define CFLAGS, LDFLAGS and LINTFLAGS for cc, ld and lint options. 'make final' will get you the binary for the simulator. old: is for backward compatibility. clean: will remove droppings created by plotting. real_master: will create the real_master.

3. Cookbook

This section of the manual is for the not-so-adventurous programmer who wants to make modifications to the system without having to figure out how the whole thing works. To this end, here are some recipes for modifying things like the control window, statistics collection etc. By necessity, this section is incomplete and suggestions are welcome.

4. Modifying random number generation

The random number generator used in REAL is random(3). The macro RANDOM defined in config.h makes a call to this system utility. If you wish to use your own generator, change the definition. You can modify change the generator state calling initstate(3) and setstate(3) from the monitor.

4.1. Modifying config.h

config.h has declarations that are used to size the simulator. MAX_NODES is the number of nodes in the current simulation. ABSOLUTE_MAX_NODES is the nearest power of 10 larger than the maximum number of nodes you intend to have in your simulation. No gateway should have more than MAX_FAN_OUT outgoing trunks. MAX_CONVERSATIONS is the largest possible number of conversations in a simulation. MAX_WINDOW_SIZE is the largest possible window size.

Two definitions control report generation: DUMP_INTERVAL is the time in seconds between reports. DUMP_FILE is the name of the report file. DISTRIBUTE is defined if you would like to have distributed simulation.

4.2. Modifying NetLanguage

If you want to add a new parameter to the language, or to change it, you will need to make the following changes: First, change lang.lex to recognize any new tokens. Modify lang.yacc to add the new token, the new production and the semantic actions necessary for that production. Change lang.inp to add the new construct to the sample language input. (This will allow you to check your work.) Now, type in 'make lang.c'. If everything works correctly, the *lang.c* file should be created. Change directory to sim/sim, recreate the binary file, and run in single user mode to test that the C code generated is correct. You will need to modify *writelang.c* to reflect the change in the language syntax.

Here is an example of how this works. Suppose that you want to add a new paramter to the simulator, and want this parameter to be described in NetLanguage. I will assume that the parameter is a global that is declared in *sim.c*. The place to add this parameter is probably the real_parameters section of NetLanguage. Let us suppose that the parameter is called my_param.

First, you should decide the syntax the language addition should have. In this case, pattern matching with other parameter declarations, an appropriate syntax would be

```
        my_param = 5.5;
```

Next, let us modify lang.lex to recognize the token my_param. Simply add the rule

```
        my_param        {return MY_PARAM ;}
```

to this file, along with the rules for the other parameters. (5.5 is a floating point number and will be recognized as F_NUMBER.)

We are now ready to modify lang.yacc. Note that real_params are declared in a paramlist. Modify the 'param' non-terminal to add 'my_param' to the list. Now, add the rule for 'my_param'. Pattern matching from the rules already there, this should be

```
        my_param :      MY_PARAM '=' F_NUMBER
```

What about semantic actions ? We know that *sim.c* declares a global called 'my_param', so the action is just

```
        fprintf (g, "my_param = %f;0, $3);
```

Finally, declare MY_PARAM in the %token list at the head of lang.yacc. You are done with modification to NetLanguage.

Now, change lang.inp to add your parameter to it. Typing in 'make lang.c' will tell you if you did things right. The lang.c file produced should be correctly created, and should link in with sim.c in sim/sim correctly.

You still have to modify writelang.c Find the place where the other parameters are printed out. Add your parameter to the list, and recompile writelang.c. This will make the addition consistent.

```
        P(my_param = %f;, my_param)
```

4.3. Modifying the control window

The control window can be modified to add

1) New parameters in Layer 4
2) New action in pop up menus

Instead of describing how to make changes for each possible menu (there are 5 of these), I will describe a method to make these changes that is almost painless (and mindless). The trick is to grep in sim/display for keywords that are close to where you want your new entries. These will bring up new keywords that are related. Keep grepping till you have found all the places that have a keyword or something very close to it, and make all the obvious changes. Let me demonstrate this fuzzy technique with an example.

Suppose that you want to add an entry 'my_entry' to the entries in the menu that pops up when you press the right mouse button in layer 5 over a node. Note that the node menu already has an entry called 'Set Node Data'. So, just grep for 'Set Node Data' in sim/display. This is what you get

```
(ra 42)--> grep "Set Node Data" *.c *.h
menu.c:   "Set Node Data", {Set_N_Data, nil},
```

So, there is something to change in menu.c. Also, it looks like searching for Set_N_Data is a good idea. Let us do that.

```
(ra 47)--> grep "Set_N_Data" *.c *.h
actions.c:   action_table[Set_N_Data].opcode = Set_N_Data;
actions.c:   action_table[Set_N_Data].function = n_set_data;
actions.c:   action_table[Set_N_Data].mouse = Node;
actions.c:   action_table[Set_N_Data].fixed = No_Arg;
actions.c:   action_table[Set_N_Data].name = "SET_NODEDATA";
events.c:   Meta (Right), {Set_N_Data, nil},
menu.c:   "Set Node Data", {Set_N_Data, nil},
action.h:#define Set_N_Data    (End_Node + 1)
action.h:#define Show_N_Data    (Set_N_Data + 1)
```

Looks like pay dirt! We need to make changes in actions.c, events.c and action.h. Now, let us search for "n_set_data".

```
(ra 48)--> grep "n_set_data" *.c *.h
actions.c:extern              n_set_data (), n_show_data (),
n_set_func ();
actions.c:   action_table[Set_N_Data].function = n_set_data;
node.c:n_set_data (nodeptr)                /* interactive */
```

Ok, we add node.c to the things to change. Now, search for SET_NODEDATA.

```
(ra 49)--> grep SET_NODEDATA *.c *.h
actions.c:   action_table[Set_N_Data].name = "SET_NODEDATA";
```

No new names have come up, so we are practically done. In each of the files that have come up, find out what has to be changed, and pattern match. For example, in events.c, we see

```
eventmap      default_node_map[] =
{
    Left, {Move_Node, nil},
    Middle, {Start_Edge, (pointer) -1},
    Right, {Act_Menu, "Node Menu"},
    Shift (Left), {Delete_Node, nil},
    Shift (Right), {Show_N_Data, nil},
    Meta (Right), {Set_N_Data, nil},
};
```

Looks like we are mapping mouse clicks to actions. Just select which mouse click should do my_entry's actions. Suppose it is Shift (Middle). Then add

```
Shift (Middle), {My_Entry, nil},
```

to the list. Similarly, modify all the other files.

As another example, if you want to add a parameter to the control window Layer 4, you will need to change *display/sunvparam.c*, *display/comm.c* and *sim/monitor.c*.

There is something extra that you need to do if you want to communicate data from the control window to the simulator. You have a choice of using the Nest socket or the REAL socket. I would recommend modifying the REAL socket - it is much cleaner. Suppose that you have installed a new parameter in the Layer 4 subwindow. To send it to the simulator, you will need to add a function to comm.c that writes the parameter value (that you have obtained from the window) into a packet to be sent over the socket. When this is received at the other end by custom_monitor() (in *monitor.c*), you will need to modify service() so that the parameter is read in. You may want to make the parameter a global, in which case declaring it in *sim.c* and adding it to *paramters.i* will do the trick.