Nest System Overview

Alexander Dupuy
Jed Schwartz

Computer Science Department
Columbia University
New York, NY 10027-6699

Wednesday June 15th, 1988

Abstract

This manual describes the simulation library provided with Nest Version 2.5. Nest is available from Columbia University. For information, please contact the authors.

This research was supported in part by the Department of Defense Advanced Research Project Agency, under contract F29601-87-C-0074, and by the New York State Science and Technology Foundation, under contract NYSSTF CAT (87)-5.

Table of Contents

1. Introduction

Nest is many things; It is a network simulation library for Unix* and a development environment for distributed software, It is also a testbed for systems in the simulated network and a tool with a graphical user interface for monitoring the simulation or creating network descriptions.

1.1. Reasons for single process simulation library

In the development of a distributed system one often needs to debug the system, or test its behavior on a network, even though the development may be taking place on other machines, on a different type of network, where fewer processors may be available. In these situations, one needs some method of simulating a network of processors to allow testing and debugging.

If the number of nodes in the system is small enough, one may be able to simulate them with separate tasks on a few multitasking processors. These tasks, however, will run in an environment which may be quite unlike the one which is being simulated; the network topology and communication facilities could vary greatly, requiring a network simulation package which would be included in each task. Also, if the number of nodes is more than a hundred, the system overhead of these tasks may become too great, even if they are spread out over several machines.

Debugging any distributed system can be difficult even if the tasks for all the nodes are on one machine. One ends up constantly switching between tasks running under separate debuggers (confusing even with multiple windows), since there is no way to issue global debugging commands or select nodes to receive debugging commands.

What is needed is a way to simulate a distributed system in a single task, allowing one to simulate arbitrary numbers of nodes and do global debugging. The only limitation is that they all fit in one address space (this is not too great a restriction with 24 to 32 bit addresses) allowing possibly several thousand nodes to be simulated.

Nest was created because we needed these facilities for the development of a distributed system known as the Incremental Position Location System (IPLS). The features and components of Nest were created for IPLS, but have been designed so that Nest can be used for work with distributed systems in general.

1.2. Nest features

The ability to simulate many different network topologies should be quite helpful for someone working on network routing algorithms, or studying network traffic flow patterns. Those developing distributed systems for many different kinds of networks, some of which might not even be available, or wishing to work on algorithms without having to deal with the network itself, should find the ability to bypass the actual hardware and operating system interfaces invaluable.

1.2.1. Simulation of Networks

Because of the importance of network topology and behavior for IPLS, the ability to model networks was critical; the major goal of Nest was to model and simulate them in some detail. However, this is not too closely tied to any particular network; instead we did this in a generalized fashion which could be easily adapted to any kind of network.

Our model includes a number of basic operations on the nodes and edges of a network, such as specifying the function to be run on a node, causing nodes and/or edges to fault, and modeling basic transmission characteristics like noise or lost messages.

1.2.2. Testbed for distributed programs

Nest is designed to run the same code in the simulation that one would run in a real system, since the simulation is independent of the language used, and more or less transparent to the application program. The communications functions are different in the details, but support the same communications style used in most real systems.

1.2.3. Development environment for distributed programs

It is possible to use ordinary debugging tools on the functions running under the simulation, and specialized debugging tools which know about the simulation may be developed for future releases.

Nest also allows one to bypass the low level nature of real communications mechanisms while developing algorithms, so that if data structures change, one does not have to rewrite all the communications procedures before testing it again.

1.2.4. Performance monitoring of distributed programs

Nest allows one to see how the performance of algorithms are affected by various network configurations, as well as generally being able to view the distribution of computation and communications between the nodes. Nest also provides a method of monitoring and logging the message activity in the network.

2. System Structure

2.1. Control

The heart of Nest is the simulation library which contains the routines necessary for the simulation of a distributed network. Simulation programs are created by linking this library with the functions which will be run on the nodes. For more specialized simulation programs, routines that specify initial configurations, cause dynamic modifications, or provide additional interfaces to users or other programs can be written and linked in as well. Control is passed between the simulation library, the functions for the simulated nodes, and auxiliary routines; once a simulation has begun, certain library routines coordinate the actions of all three, until the simulation is finished.

2.1.1. Main routine and library generic main()

Since the simulation package is a library rather than a program, users can write main routines which provide any desired program interface for command line arguments, interactive initialization or signal handling. A user-written main routine could specify a starting network configuration, establish communications with other programs, open logging files and so forth.

In some cases, no special handling may be needed, so the library includes a "generic" main routine which parses standard command line arguments for the simulation, optionally prompting for network configuration, and invokes the simulation library control loop. The routines which parse command line arguments and prompt for network configuration can also be used by a user-written main routine.

2.1.2. Initialization, simulate() parameters

The network simulation is started when the simulate() function is called, which in turn invokes the top-level simulation loop. Once this function has been called, it will return to the main routine only when the simulation is finished, as described below. The simulate() function takes arguments which determine the initial state of the simulation, as well as several runtime limits.

One argument sets the absolute limit on the number of nodes in the simulation; fewer nodes may be present, but never more. This allows Nest internal data structures to be indexed by node ids for fast access.

Another argument gives a size for combined stack area of the nodes, since space for them is allocated at the start of simulation. This number is a soft limit on the total size of all the nodes' stacks at any one time. If this limit is exceeded, Nest will try to allocate additional space; however this is much less efficient than allocating enough space at the start, as the entire stack area may need to be copied.

Two others give the port number for the simulation if user interface clients are to be used, and an initial network configuration in the form of a graphlanguage structure, which will be discussed in detail later on. The latter gives the interconnection of nodes as well as a number of global parameters.

2.1.3. Scheduler behavior/function

The top-level simulation loop invoked by simulate() manages all scheduling of nodes in the simulation. The simulation scheduling is organized in cycles (or passes) of some user specified time length, called the pass time. In each pass, each node which has been started and is not blocked or stopped will be allowed to run for this length of time. Nodes which are stopped, or blocked awaiting messages will not be run, although they will use simulated cpu time.

During each pass, the network connection structure will not change, and the state of any node will be changed only by simulated events, such as messages or function returns. In other words, no manipulation of the simulation can take place during a pass. Between passes, however, changing network connections, as well as stopping, starting or resetting of nodes is possible. This can be done through a user interface client, or through a monitor function.

2.1.4. Monitor function hook

Before the beginning of each pass, after any changes received from clients have been made to the graphlanguage model of the simulation state, a monitor function is called with the graphlanguage structure as an argument. This function can then make changes to the state model, log unusual events, send updates to the clients, or look at global or heap data used by nodes. This function returns a graphlanguage model which is used to update the simulation state.

A generic function, nest_monitor(), is provided, although the user can specify any number of others, and switch between them. The nest_monitor() function sends an update to the clients if any changes have occurred in the simulation, as noted by the setting of a global variable, as well as when a client first connects to the simulation.

2.1.5. Termination, statistics

The simulation will finish and return control to the main routine when the following conditions are met: All node functions have returned or are blocked waiting for messages which have not been sent (deadlocked), and the simulation is not paused. Before the simulation terminates, the monitor function is called, and may pause the simulation to prevent it from finishing. The generic nest_monitor() function will pause the simulation if any clients are connected to the simulation at the time.

When the simulation finishes, it can collect and compile statistics on various simulation measures such as number of passes, time simulated, cpu usage for each node, and messages transmitted or received by nodes or edges. A log of messages, in chronological order, can also be produced.

These logging and statistics gathering facilities are not implemented in the generic nest_monitor() function included in the Nest library, and must be added by the user. This basic functionality, as well as more detailed or simulation specific statistics, could be provided by a custom monitor function.

2.2. Server

The Nest simulation library includes "server" routines which provide a standard communication facility for simulation programs which act as servers for user interface clients such as the Sun client, or other user interface clients written to follow the same protocol. Some of the server routines can also be used by custom monitor functions to communicate with user interface programs using other protocols for interaction with the specific "distributed" system being simulated.

2.2.1. Server/client control protocol

The protocol used by the server routines and the client is based on stream sockets. Connections are established in the conventional manner, with the simulation listening to a socket bound to the port specified in the arguments to simulate(). The server routines periodically check for connection requests from clients on this socket, and will accept them if possible. More than one client can be connected to a simulation at one time; the only limit is the number of file descriptors available for sockets.

The connected sockets are used in two distinct ways: graphlanguage messages describing the simulation state (possibly with changes) and out-of-band control codes and acknowledgements are exchanged by server routines and the clients. These two uses are logically independent, and control codes can be sent at any time, even while graphlanguage messages are in progress.

The communication between simulation and client is asymmetrical; the two use graphlanguage and control messages in different ways. Since the simulation may have state information which isn't yet known by the clients, messages from the simulation to clients represent a complete network to replace the existing one, while messages sent from clients to the simulation only indicate changes in the network. Similarly, control codes have different meanings depending on who sends and who receives, and certain codes are only sent from one to the other, not vice versa.

2.2.2. Structure of control messages

Control messages consist of a single byte (ASCII) code, and are sent as out-of-band data on the stream socket. Because of this, they arrive after very short delays, and are used to provide immediate control over the simulation program, even during a pass, when changes to the simulation are not allowed. The simulation can be locked to prevent changes, suspended or continued.

Unfortunately, the implementation of out-of-band data under BSD Unix suffers from a number of implementation problems. As a result, all control messages are send both as out-of-band data and as in-band data, between graphlanguage messages. The current server and client discard the out-of-band data, and only act on the in-band copies. Other implementations can act on either in-band or out-of-band, but not both. All implementations should generate both kinds of control messages.

When a client first connects to a simulation, it is sent a control message consisting of an ASCII digit (currently '0','1','2', or '3') which indicates the current status of the simulation. If the simulation is paused and locked, the code will be '3', if paused only, the code will be '2', if locked only, the code will be '1', if neither, the code will be '0'.

All control or graphlanguage messages from clients to a simulation are acknowledged by one of these codes: "Okay", "No Way", "Already", or "Huh?". "Okay" indicates that the control code was accepted, "No Way" that it was refused because the simulation is locked by another client, "Already" that the code is pointless (e.g. unlocking an already unlocked simulation), and "Huh?" that the code was not recognized. Messages from a simulation to a client should never be acknowledged, as this may lead to looping "Huh?"s.

Since several clients can be connected to a simulation at any time, a locking mechanism is needed to prevent conflicting changes to simulations. If a simulation receives a "Lock" code, and isn't already locked, it will respond "OK", and send all other connected clients a "Lock" code. After this, the simulation will no longer accept messages from other clients, and will always acknowledge with "No Way". A later "Unlock" code from the locking client will be acknowledged, causing "Unlock" codes to be sent to others, and messages to be accepted again.

2.2.3. Structure of graphlanguage messages

The graphlanguage is a standard way of sending graphs or networks between communicating processes. It allows application specific data to be included in the framework, and provides routines for converting graphlanguage messages (character streams) into graphlanguage structures. Both are composed of three parts; a header which contains global information about the graph, a list with information about each node, and a list with information about each edge. The basic framework has no information associated with nodes, and only two nodes associated

with each edge, but Nest graphlanguage has state information for nodes and edges, as well as global variables for the header.

A Nest graphlanguage header specifies a number of global variables, such as simulation pass length, realtime wakeup interval, broadcast or point-to-point communications. The selected monitor function, current simulation time and pass number are included in the header. One "constant" is included; the maximum number of nodes allowed, as passed to simulate(). There are also three lists, and defaults, for functions. These lists include all valid functions' addresses and string names for the monitor function, for functions to be run on simulated nodes, and for channel functions (see below).

A Nest graphlanguage node entry contains the id number, location, the function for the node, the cputime used by the node since the start of the simulation, and the current status of the node (running, blocked, stopped, etc). It also includes markers indicating changes to the status of the node (start and halt) which are only used in messages from the monitor or client to the simulation. An edge entry, similarly, contains a weight (usually used as the delay time), a list of channel functions, and a marker indicating that it should be deleted (the edge endpoints are specified by the graphlanguage framework).

2.2.4. Server timing and socket behavior

While no changes can be made to the simulation during a pass, as noted above, communications between the client and the server routines of the simulation library do take place during passes. To prevent this from blocking the entire simulation, non-blocking sockets are used for communications between them. This requires that communications with the client be separated from the actual modification of the simulation specified.

In order to maintain reasonable response, the server routine that manages these communications runs several times during a pass, at the time when the scheduler determines the next node to run. Reads and writes are buffered, possibly over several passes, until complete graphlanguage messages are received. These are then queued for execution at the end of the pass.

2.2.5. Routines for monitor use

The non-blocking communications routines mentioned above can be used by the monitor for communication with other programs. There are also a number of standard graphlanguage routines which modify the graphlanguage structure of the network; these can be used to mark changes in the network in more or less the same way the Sun client does.

2.3. Sun user interface client

While any program which follows the protocol described above can be used to monitor and control a simulation programs, only one such program currently exists. While its user interface is not necessarily the only possible one, or even the best, it is a good example of how a user interface using graphlanguage protocol can work. Note that the user interface is typical of Sun programs; a tool for another machine with a different user interface style should probably use that style, rather than trying to imitate this one.

2.3.1. Client tool structure

The Sun user interface client uses the SunView libraries and follows most of the user interface conventions for SunView programs. It uses the mouse for most input, allowing the user to enter graphs directly, by drawing the nodes and edges on the screen. Internally, the client, like most SunView programs, is event-driven; all processing is done in response to user requests and input.

In the case of the client, there is an additional source of input, the simulation program. The client waits for input from either of these sources; when there is a message from the simulation, it updates its state, and the screen display which reflects this. When there is user input, the client will update its state and display, and possibly send a message to the simulation program.

2.3.2. Network manipulations

The primary area of user interaction is the network window, where a diagram of the current network is displayed. Nodes and edges can be added quite easily using the left and middle mouse button. To create a node, simply position the cursor and click the left button. To create an edge, position the cursor over an existing node, then press and hold the middle button. Then move the cursor to another node (an edge line will follow the cursor) and release. It is even possible to combine the two actions by clicking the left button while an edge is being made with the middle button held down. This creates a node, which can be connected to the edge by releasing the middle button.

Deletions and other modifications are made using the right, or menu button. When the menu button is pressed while the cursor is on (or very near) a node or edge, an appropriate menu is called up. The menu for nodes includes options to remove the node, stop or start it, or to display detailed information. The detailed information can then be modified in the pop-up node window. A similar procedure allows edges to be removed and/or modified. If the menu button is pressed while the cursor isn't on any node or edge, a general menu is called up, which allows recent deletions to be undone, or a graphlanguage message containing the modifications made to be sent to the simulation.

There is also support for altering the actions bound to the mouse buttons. This is described in more detail in the Nest User Interface Manual.

2.3.3. Simulation parameter control panels

These three panels provide "buttons" "switches" and text parameters which can be edited; these allow the user to modify any part of the simulation which is represented in a graphlanguage message.

2.3.4. Interacting with simulation

As with the control panels for simulation globals, there is a panel for immediate control of the simulation, using control messages, as well as specifying the host and port number of the simulation which is to be monitored.

2.4. Multiprocessing

In order to use the Nest simulation library effectively, it is useful to have a fair understanding of how the library routines provide a form of multitasking for the nodes in the simulation. While it is not completely invisible, the simulation multitasking should be relatively transparent to well-structured programs, and it allows some shortcuts which make the prototyping of distributed systems easier.

2.4.1. Memory use breakdown when using Nest

A simulation program has three kinds of data areas; since the code segment is read-only, sharing it between nodes is invisible. There is global data, a dynamic heap area above this, and a stack at the top of memory. The global data area is, of course, global, and as such must be shared by all the nodes. This is efficient as far as minimizing consumption of system resources, but has unpleasant side effects when using global or static variables carelessly. Preprocessor macros can be used to prevent unintentional sharing of globals, and with structured programming techniques which avoid the unnecessary use of globals, this is not a major problem.

The heap area is also shared by all nodes, but since it is allocated at runtime on a piece by piece basis by malloc(), there are no problems with overlapping data. Each node will make a separate request to malloc() for memory, and will receive one for its exclusive use. It suffices to provide checking in malloc() to ensure that one node doesn't get allocated memory which another node has free'd, but is not finished with. However, a restriction is imposed that only the last free'd block is guaranteed to be unaltered for each node. This covers the vast majority of uses for referencing free'd data (a questionable practice in many cases).

While the stack space in memory is shared, the ordered and functionally related nature of stack data make it possible to simulate completely separate stacks for each node, which is an absolute necessity for a reasonable simulation. A copy of the stack for each node is kept in the heap area, and before a node is run, this is copied into the stack area. When a node is interrupted, the heap copy of the stack is updated, and its size is adjusted. This is done just before the machine state (registers and so forth) is saved or restored.

2.4.2. Context switch description

In a context switch, the node's stack is copied to a storage area on the heap, and an assembler function somewhat like _setjmp() saves the internal machine state. In order that meaningful error codes be returned by interrupted system calls, the variable errno is also saved (an exception to the sharing of global variables). A longjmp() call is made to truncate the stack and return control to the top-level scheduling routine.

When a node is continued, a small assembler routine first switches the stack pointer to another area, so that the node's saved stack can be copied into the stack area safely. Then another assembler routine somewhat like _longjmp() is used to restore the machine state which was preserved earlier, and return control to the signal handler, which returns control to the interrupted node function.

2.4.3. Stack management

While the saved stacks are stored in the heap area, certain efficiency considerations, and a bug involving reclamation of free'd space in some versions of malloc(), make it impractical to simply allocate space for stack copies as needed. Instead, a large block of memory is allocated when the simulation is started, and this space is divided up as needed for the stacks. If the simulation library runs out of stack space, it will attempt to allocate more as needed; however, this is less efficient than allocating it in the first place.

2.5. Network

Multitasking is only half of a network simulation; the network itself must be represented in the simulation. Nest represents networks in a way which can be used for networks ranging from broadcast radio links to dedicated point to point lines. There are also provisions for special characteristics of networks such as transmission errors or delays.

2.5.1. Global network characteristics

Networks can be broken up into a number of types not only by their interconnection patterns, but by the nature of communications within them. This categorization by communication behaviors can depend not only upon underlying hardware, but upon the software level at which one considers them. An ethernet is based on broadcast communication, but at a certain level messages are blocked out unless they are intended for the receiver, giving point-to-point behavior. The simulation library tries to provide a general method of describing this behavior without being tied to details.

The broadcast or point-to-point nature of a network is reflected by two global flags in the graphlanguage header. Either broadcast or point-to-point can be set, or neither, but not, of course, both. If broadcast is set, all messages are broadcast, regardless of the destination, while if point-to-point is set, no broadcast messages are allowed. The default, when neither is set, is to allow both; sending ordinary messages point-to-point, and broadcasting any messages with null destination fields. Note that this feature is limited to homogeneous networks; i.e. it is impossible to specify a network which is partly point-to-point and partly broadcast.

In most networks, communications from A to B have the same character and are subject to the same constraints as those from B to A. While the simulation library keeps distinct entries for each direction of a communications link, the graphlanguage header has a directed flag which, when clear, indicates that all entries define edges for both directions. The graphlanguage header also has defaults for the properties and behavior associated with edges described below. These defaults are used if no other behavior is specified for an edge.

2.5.2. Edge properties and channel functions

Network behavior for specific edges is also in the graphlanguage specification. One important parameter which is kept for each edge is the weight, which is often used as the transmission delay for messages which are sent across it. Weights are long integers; since delays are given in microseconds; a delay of up to 30 seconds can be specified.

More involved behavior, such as noise or transmission errors, or behavior which is dependent on the information being transmitted, can be modeled by channel functions. Each edge has a list of channel functions which provide logging of messages, modeling of transmission lossage, or copying of data.

When a message is sent from one node to another, the first channel function in the list is called with the message, sender, destination and weight, as well as the remainder of the channel function list. This function may alter the message or envelope, and either pass the message to the next channel function, deliver the message itself, or simply drop the message entirely. A channel function can deliver the message to any recipient which is connected to the sender, at any time in the simulation, although if the delivery time is in the past, it won't be modeled accurately.

2.5.3. Common channel functions

As with other functions in the simulation library, a generic channel function, reliable(), is provided for use as the default in cases where no special behavior is needed. This function delivers the message, unchanged, "weight" (simulated) microseconds after the time it was sent. If the weight is zero, the default delay in the header is used instead. The reliable() function can also be used by a channel function which wants to deliver the message itself.

Another channel function, safe_string(), assumes that the data is a null terminated character string, and replaces the message data with a copy to prevent data sharing, before invoking the next channel function in the list.

Note that while reliable() interprets the weight parameter as a transmission delay, user-supplied channel functions aren't required to do so, and can use the weight as a variable for anything they want, passing a different value to the other functions on the channel stack.

2.6. Communications

The actual means of communication between nodes under Nest is a simple but flexible message type which can be used either to take advantage of the single address space of Nest, or to simulate more realistically the usual byte-string packet nature of most communications media.

2.6.1. Message description

Nest message packets are very general, with specific message formats left for user functions to implement. The message data is represented by two generic fields: a key, and a data pointer. The key field is an integer code which may be used to represent the message type, count, or other simple data. The data pointer field allows passing of more complex messages; it is a pointer to data, which may be anything except a local variable (automatic or register). Typically it might be a character string, or complex data structure, the exact type being indicated by the message key.

Each message has a header associated with it, consisting of the node id's of the sender and destination. A message also has an implicit header associated with it, consisting of the time of sending and arrival at the destination.

2.6.2. Sending mechanism

Sending a message is done by calling the function sendm() with the id of the destination, and the two message fields described above. The destination argument is the node id of the (intended) receiver; it may be zero if it is desired to send to all neighbors. If one of the Nest channel functions is used to send the message, it will return a result: a positive value if the message is successfully transmitted, or an error indication if the sender is not connected to the destination or a broadcast message was attempted and point-to-point was set. If a user-supplied channel function is invoked to send the message, the result of that channel function is returned instead.

2.6.3. Receiving mechanism

Receiving a message is done by calling the function recvm(), with pointers to variables to hold the values of the destination, key and data pointer fields as specified in the original sendm() call. If a message is available, these variables are set appropriately, and the node id of the sender is returned. If no messages are available yet, as determined by the arrival time in the message headers, nodes which call recvm() will block until one becomes available (that is, they will stop being scheduled, but will appear to be using simulated cpu time). It is also possible to check whether any messages are available with the function any_messages().

2.6.4. Channel function behavior

When a message is sent, the channel function associated with the edge connecting the sender and the destination is invoked to deliver the message. The channel function is called with arguments generated from the original sendm() call, as well as others. The key and datapointer are taken from the sendm() arguments, as well as the destination id. The receiver id, i.e. the node which should actually receive the message (not necessarily the destination for which it was intended, is taken from the destination argument). The sender's id is taken as the id of the node which called sendm(). The weight is taken from the weight associated with the edge.

After the channel function has done whatever it wants with its arguments, including changing any or all of them, consistently or inconsistently, it should pass them to the next function on the channel stack by calling channel_sendm() with the arguments. This function will invoke the next function on the list, if any, with the arguments it is given. If there is no other function on the list, it will invoke reliable() to deliver the message. It is

possible to skip the other channel functions by calling reliable() directly, or to lose the message "in transit" by doing nothing.

2.6.5. Effects on scheduling

In order to accurately simulate the timing and order of message arrival, when a node calls the recvm() function, it will block until the time of arrival of the first available message, or indefinitely if none exist. Even if the first available message arrived in the (simulated) past, the node will block, allowing others which have not yet run in that time period to send messages which might arrive even earlier. All this should be invisible to the application functions, and serves only to ensure arrival of messages in the proper order.

3. Hints on use

Using the Nest simulation library to develop a distributed application can be somewhat tricky. There are a number of traps which the unwary can fall into which can cause errors which may be sporadic or otherwise hard to track down. This is not intended to be a complete guide to writing node or monitor functions for Nest, but is the start of a collection of Nest lore which may make it easier for new users to develop distributed applications.

3.1. Warnings

3.1.1. Global variables

Since all extern or static variables are shared between all the nodes, it is generally unwise to use them for anything other than read-only constants. If you need to port existing code which uses extern and static variables, you can define macros which turn references to "myextern" (for example) into "myextern[get_node_id()]" and redeclare myextern as an array of whatever it is. Be sure to check the node limit argument to simulate to make sure that it isn't greater than the size of these arrays.

A worse problem is system library routines which use extern and static variables. The worst offenders are routines which return pointers to a static area containing some information which is overwritten on each call. (There is usually a note in the BUGS section if this is the case). Judicious use of the hold() and release() functions, and copying the data to a local variable can help here. In general, try to limit system library calls to the main() or monitor routines wherever possible.

3.1.2. Nest messages

When sending data pointers between nodes, some sort of conventions should be observed to avoid fatal side-effects of sharing allocated memory. There are no conventions which work best for all programs, but some typical ones are described below.

- Shared string-data: When a message is sent, the sender should not make any future references to the string which is passed, or free it. The recipient may free it (but not if you may be sending string constants). If a message is sent to all, recipients should not free it, or write to it, although all (including the sender) may read it.
- Copied string data: When a message is sent, safe_string() is used to copy the string. All nodes may treat message strings as if they are their own. This is generally the most realistic convention.
- Shared data-structures: This convention is much the same as shared string data, with the same caveat about sending globals which other nodes may attempt to free. Additionally, some sort of convention on components of the structures should be observed.

3.1.3. System objects and calls

Dealing with system objects like file descriptors can be somewhat painful, especially if you have more nodes than fd's. If it's possible, you should isolate the file handling code to a monitor routine, or a node running a special logging function, and have these routines look at global data set up by the other nodes.

Since the nest library controls the swapping of nodes by timer interrupts, it is possible that system calls will be interrupted, and will therefore fail, returning error EINTR in the extern variable "errno" (which *is* maintained for each node, unlike other extern variables). If you call system functions such as read() or select() which may block, you should check for this error, and ignore it.

Finally, when a function running under nest is done, it should *not* call exit(), but simply return(). Calling exit() will abort the simulation.

3.2. Debugging

3.2.1. Diagnostic output

The most basic form of debugging is to insert printf statements into the code you are debugging. This can be used with Nest, but it is a bit awkward. You should use hold() and release() to keep the diagnostic output from several nodes from getting all mixed up, or only enable output on one node. It is also a good idea to identify the current node in the diagnostic output.

3.2.2. Dbx and Dbxtool

These are generally much better tools for debugging than printf statements. You can use these with Nest simulation programs much as you would any other program, but there are a few things to watch out for. Nest does context switches when it receives signals of various kinds. Usually the signals are generated by timer interrupts, but control messages from clients and faults in node functions will also cause context switches.

Generally you will want to ignore the timer and control message signals, so that context switches can take place transparently. This can be done automatically by placing the ignore commands in the .dbxinit file. However, if these signals are ignored while you are single stepping through code, a context switch may take place without your knowing it, and you may be somewhat confused by the fact that variables have suddenly changed.

You may want to trap these signals instead, so that context switches are prevented; this can be done with the "catch" command. You can then single step through the code on a single node, and when you are done, ignore the signals. If a signal was caught while you were stepping through the code, you can simply "continue 16" which will cause a context switch. Bear in mind that certain Nest functions such as recvm(), any_messages(), and slumber() require a context switch for synchronization, and may fail if one does not happen.

A useful thing to do if you have the Sun dbx or dbxtool is to "display _current_node", which will cause the current node id to be displayed whenever the debugged process has stopped.

3.2.3. Core dumps

If an addressing or instruction fault occurs while Nest is running a node, the signal generated (SIGILL, SIGIOT, SIGBUS, or SIGSEGV) is trapped by Nest. Nest generates a core dump, then marks the node as dead, and switches to another node. The core dump is renamed to core.%d where %d is the id of the node which failed.