Nest User's Guide

Alexander Dupuy
Jed Schwartz

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

Wednesday June 15th, 1988

Abstract

This guide describes building simulations using 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. Overview

1.1. Source files

All of the source files discussed in this guide can be found in the nest directory. You can work in this directory, or you can copy the relevant source files and makefile into a working directory, and simply link the objects with the Nest library.

In order to implement your own application to run on Nest, you will need to provide a certain number of functions which are not in the Nest library. A sample source file containing these functions is **skeleton.c**. You can create a simulation by modifying or replacing skeleton.c. This file contains the code for a minimal example simulation which we will present below. It is designed so that you can simply delete all or much the bodies of the functions included and plug in your own code. In addition to this minimal skeleton example, you may want to refer to the code for the more complex mapper demo. This code resides in **mapper.c**, and uses the default main() routine provided by the Nest library module **main.c**.

In main.c you will find the default main() function for the Nest simulation, which does some initialization and then initiates the simulation by calling the function call simulate(). While the simulation runs, control passes between the different simulated nodes. When all of these nodes finish, or when there is a deadlock, the simulation finishes.

In a simulation, there are one or more *node functions*. One of these is associated with each node. These functions are the code which runs on the simulated nodes. If the default main() is used, as in the mapper demo, the function node_main() is the only function associated with any of the nodes. Thus all of the nodes run the same function. However, when node_main is called for each simulated node, it is passes as a parameter the number of that node, so that it may behave differently depending on the node it is running on.

Many distributed systems will fit naturally into this model, with each node running the same function. If you want to run different functions on different nodes, you can do this in much the same way that the skeleton demo does. You create a file containing a main() routine which sets up the simulation, as well as the several node functions which you want to provide.

Also, if you want to be able to associate more than one function with certain nodes, and to alternate which function runs on a particular node during the course of a simulation, then place these functions in this file. In the simple skeleton example which we provide, there are several node functions. All of these functions, in addition to the main() function, reside in the file skeleton.c. You can build a complex Nest simulation by replacing the node functions in skeleton.c by your own node function(s). In addition, you will have to modify the main() function in skeleton.c to initialize your own network configuration properly.

In addition, you may modify the file called "nestmon.c", if you want to provide automatic network reconfiguration, or periodic processing of any global values or non-node-resident tasks, including custom monitoring features. In this file there is a function called nest_monitor() which is called once during each simulation pass. At each pass, this function exchanges data with any nest interface which is trying to communicate with it, and modifies the simulated network if directed to by one of these data exchanges. You can extend this function to automatically modify the network under program control, or to perform any other periodic task such as gathering data on system-wide basis, or communicating with a custom interface which you have written.

Another alternative is to create a function called auto_monitor() which does any processing you want to do, and then calls nest_monitor(). This works quite easily if you are using the default main(), since it will use auto_monitor() as the monitor function.

1.2. Include files

- **nest.h**: You will need to include nest.h in any nest application file which you create. Nest.h defines some basic values, types and macros, and in turn includes other important include files.
- **graph.h**: You will need to include graph.h if you want to access the your simulation's global graph structure, or any other graph structure which you create, because it is in this file that the graph data structure is defined.
- **defs.h**: You may want to include this file which defines many convenient values, alternate C syntaxes, data structures and macros.

2. Implementing a Simulation

2.1. Main

In a Nest simulation, the main() function initializes a simulation graph, a limit on the total stacksize and number of nodes, and a network portnumber for addressing the server by client displays. These parameters are then passed to the function simulate(), which runs the simulation until it exits, at which point control passes back to main() for any post-processing that might be desired.

A generic main() is provided in the Nest library, which will be sufficient for some applications. If you can use the generic main() as is, you will not need to write any main() function of your own; the generic main() will simply be linked in from the Nest library by default. The generic main will initialize the simulation graph on the basis of a stored file, or interactive alphanumeric input from the terminal (see Nest Reference Manual for details). Its major limitations are that the simulation it spawns can run only a single function [which you must name node_main()] on all nodes; that it will only utilize a single channel function [which must be named channel()] on all channels. Note also that the internal monitor function which is put into place by the generic main function is called auto_monitor, so if you use generic main() and you write your own internal monitor, you must call it auto_monitor().

If your simulation will include more than one node function or channel function, you will need to implement your own main() which assigns these functions to the appropriate fields in the global graph header. If you provide your own main() function, then when you link your code with the Nest library, your main() will be used instead of the generic one.

2.2. Node functions

In nodemain.c you will find a function called node_main. You will need to write one or more functions like node_main. They will all have the same parameter list as node_main, i.e. they will accept a single ident argument which corresponds to the id of the node they are running on. However, the code in your node function(s) will be entirely your own. In skeleton.c you will find two node functions like node_main(), which are called producer() and consumer().

Main(), or a function which it calls, will assign the appropriate fields in the graph header to your node function(s), so that they will run on the proper nodes. See the initialize_graph() function in skeleton.c to see how this is done.

When a node function is called it is passed a single argument which is the node it is running on. It can pass the value to functions that it in turn calls, if necessary, or else these functions can discover this value themselves by calling get_node_id(). A node function can find out the location of the node it is running on by calling get_location(). It can find out which nodes it has a communication link to by calling get_neighbors().

A node function will ordinarily carry out some communication with its neighbors. It can do so by calling the function sendm and passing to this function an argument specifying which node(s) to send a message to. The destination specified can be all neighbors (destination = 0), or the (positive) node id of some node which it is linked to. The set of nodes which it is linked to can be gotten by calling the function get_neighbors().

A node function can find out if any messages sent by other nodes have been delivered to it by calling the function any_messages(). It can receive these messages by calling recvm().

The simulated cputime and total runtime for a node can be gotten by calling cputime() and runtime() respectively. A node can be put to sleep in simulation time with slumber(), and its cputime and runtime can be directly incremented with advance().

2.3. Channel functions

The fundamental channel function used by the Nest library to deliver messages between nodes is called reliable(). When you initialize the simulation graph for your application, you can place one or more channel functions on the channel stack of each edge in the graph. At the end of each of these functions there will usually be a call to channel_sendm(). If there are functions remaining on your channel stack, then channel_sendm will pop and call the topmost of these. If and when all of the functions on your channel stack have been called, channel_sendm() will then call reliable() to deliver the message to the destination node's queue.

Thus, reliable need never be explicitly placed on a channel function stack. It would be a bad idea to do so, because reliable simply delivers a message and exits, ie. it doesn't call channel_sendm() continue down the stack, so any functions placed on the stack below reliable() would never be called.

The minimal channel function which can be placed on a channel stack is just a "wrapper" which simply calls channel_sendm() with the same arguments that are passed to it. An example of such a wrapper function is "channel()" which can be found in channel.c.

In our mapper example, as in most Nest applications, all that is desired is the reliable delivery of all messages on all edges, without modification or duplication of message data. Thus, all that is required is to use the channel() wrapper function as the sole function on the channel stack of every edge. If you are using the default main() the channel() function will be used as the channel function for all edges. If you are providing your own main() you can explicitly place channel() on the channel stack for each edge, or you can leave the channel stack empty, in which case, reliable() will be called.

Another simple and common alternative to this approach would be to use, instead of the function channel() which does nothing, the function safe_string() which makes a copy of the message data before calling channel_sendm(). If a message is sent with safe_string(), then the sending node may safely modify or free its copy of the message data, and the destination node may do the same after it is received.

If, on the other hand, you want to do some custom manipulation of message data, as we did in our skeleton example, you should include your own channel function(s) in your source file. These must be properly assigned to fields in your simulation graph during initialization in main(). See skeleton.c for the custom channel function translate(), and the way that it is placed, along with safe_string(), on a channel function stack.

2.4. Internal monitor function

One or more internal monitor functions can be defined, and included in the list of available monitor functions in the simulation graph header. Of these, one must be assigned as the current internal monitor function, in the graph header, and it is this function which is called at the end of each simulation pass. Other internal monitor functions (if there are any), serve as alternates which one can switch to interactively or under program control by assigning the current monitor field in the graph to be of another function from the list.

The general purpose monitor function provided by the nest library is called nest_monitor(). If you are only using our sunvclient for simulation monitoring and dynamic configuration, then nest_monitor() should be adequate. However, if you want to communicate with a custom interactive monitor which receives application level data, or if you want to reconfigure the simulation or collect statistics under program control, or if you want to automatically perform any other sort of non-node-resident task, you will need to write your own monitor function.

The approach that is best for most customizations is to write a function which performs your custom monitor functions and then calls the generic nest_monitor() function before exit. Thus, after your custom code performs its

tasks, nest_monitor() services interactions from sunvclient monitors. This is the approach we have taken in skeleton.c.

In the mapper example, there is not, in fact, any monitor customization, so we define a monitor function called auto_monitor() which is simply a wrapper function that calls nest_monitor(). The auto_monitor() function resides in automon.c, and our generic main initializes "auto_monitor()" to be the sole internal monitor function for the simulation. Thus, if your simulation requires only a single internal monitor, as most simulations do, you can customize your monitor by adding code to our auto_monitor() function in automon.c. You will still be able to use the generic Nest library main(), if desired, if you do not rename "auto_monitor()". If you are writing your own main(), you should assign the appropriate fields in the simulation graph to include whatever monitor or monitors you like. If you simply want to use nest_monitor(), you can assign nest_monitor() to the appropriate fields directly, i.e. there is no need to use the auto_monitor() wrapper.

In skeleton.c you will find an example of a simple customized monitor function. It is called user_monitor(). Notice that it exits by calling return(nest_monitor(newgraf)), thus invoking the generic nest_monitor() and ensuring that its return value will be returned. Notice also how user_monitor() is established as the current internal monitor function, and both user_monitor() and nest_monitor() are established as available monitor functions by the initialize_graph() function in skeleton.c.

Nest_monitor(), like any custom internal monitor which you may write, is passed a pointer to a graph as a parameter, and returns a graph pointer. Nest_monitor() does not modify the graph which it is passed, and it returns this same graph, unmodified. Thus, nest_monitor() does not change the simulation state, it merely sends updated simulation information to client monitors. However, you may require a custom internal monitor to modify the simulation state. Note first of all that the internal monitor function, unlike a node function, is non-preemptible, i.e. it runs until completion, so you do not have to concern yourself with critical sections of code. There are two ways that you can modify the simulation state within a monitor function.

The first is by returning a pointer to a simulation graph which is a modified version of the graph which is passed to the function as a parameter. If you choose this way, then you may either modify the actual graph passed to the function and return it, or else you may make a duplicate of this graph, modify this duplicate, and return the modified duplicate. If you do the latter, you will probably want to free the original graph which is passed to the function using the function graph_free(). In either case, you can call return(nest_monitor()) as we did in skeleton.c, passing nest_monitor() a pointer to your modified graph. Nest_monitor will then return this modified graph when it exits.

The second method of modifying simulation state within an internal monitor function is to directly modify simulation globals (see below, Section 4.2, Page 9), rather than the graph passed to the function, and then return nil, instead of a pointer to the graph. It is essential that you return nil if you want Nest to retain your direct simulation global changes rather than modifying the state to correspond to a graph that you return. Thus these two methods, that of modifying and returning a graph, and that of modifying simulation globals and returning nil, are mutually exclusive and cannot be mixed. Specifically, if you do the latter, then your custom monitor function must not invoke nest_monitor(), which is expecting a pointer to an actual graph and will return one. Note also that if you choose this method you should *not* free the graph passed to the monitor function using graph_free().

One important feature of nest_monitor() is that if there is a deadlock in the simulation (i.e. all nodes are waiting for events which will never occur) and there is a user interface client connected to the simulation, nest_monitor() will pause the simulation. This is important because the nest simulation would exit on this condition if nest_monitor() did not trap it. Thus, one paradigm for nest use is to establish several available monitors, one which is, or which invokes, nest_monitor(), and one or more which do not. The first monitor is used initially, but when and if it traps a deadlock, another monitor which specially handles deadlock can be switched to.

3. Implementing Your Own Interface Client

A Nest client is an independent program, usually interactive, which communicates with the Nest simulation process via socket ipc. There are several approaches to writing an interface client. Sunvclient is an example of client which communicates with Nest's built in server. The advantage to writing this sort of client is that you do not have to write any server-side code; you simply make use of the built-in server facilities. The built-in server handles multiple connections and services asynchronous requests without disturbing the simulation.

You may be able to create the custom monitor you require by simply modifying sunvclient. This is certainly the easiest way to go since it would not require you to recode any communications functions. Sunvclient, and any client which interacts with the built-in Nest server, must do so using the Graphlanguage protocol described in the Nest Overview and the Graphlanguage document. If you write your own client to interact with the built-in server, it must obey this protocol.

Sunvclient, and other clients which interact with the built-in server, can only exchange information regarding the basic simulation state and network topology. Application level information is not available from the built-in server. If you want a client which handles application-specific information, you will need to handle the communications yourself both on the client end and on the simulation end. This can also be done with sockets.

One approach which is convenient in many cases is to place in a custom internal Nest monitor the code required on the server side. This is the approach taken in the skeleton.c example. This internal monitor function handles such things as accepting socket connect requests made by the client, and reading data sent by the client. It is best to have such input operations done by the internal monitor function for Nest, rather than by individual Nest node functions, because the internal monitor is non-preemptible, and it can easily distribute any data destined for individual nodes via global variables.

As far as output goes, this may be done by the internal monitor function, or by nodes themselves. Global statistics and state information should certainly be sent by the monitor function. However application level data possessed by a certain node function could be sent directly over the socket by the node function, or it could be placed in global storage where the internal monitor would later grab it and send it out to the client. Note that if the node is going to send over the socket directly, it must bracket the socket calls as a critical section using hold() and release() since the socket calls are non-reentrant. Note also that a node should check some global flag set by the internal monitor indication whether or not a socket it might send to has yet been connected. See the user_monitor() and consumer() functions in skeleton.c for this approach.

4. Global Simulation Variables

4.1. Read-only variables

The following values are set at the initialization of the simulation and should not be modified. They should be treated as read-only values during the simulation:

struct graph *Graph -- The graph structure which contains simulation state and topology. This structure should not be modified directly. Instead, this structure is passed to the internal monitor function when it is called. Modifications made to this copy by the internal monitor function are automatically copied back to Graph when the internal monitor exits and returns its modified duplicate graph.

unsigned Nodes -- the maximum number of nodes which may exist at any time during the simulation.

long StackSize -- amount of memory (in bytes) which is initially allocated to contain the combined stacks of all node functions. This is only a soft limit on total stack space -- additional space it is dynamically allocated if this space is filled.

int PortNumber -- the host port to which client monitors must connect in order to communicate with the simulation server.

4.2. Modifiable variables

The following values may be modified during the course of a simulation by an internal monitor function (see above, Section 2.4, Page 5 for details of how and when globals can be modified by an internal monitor):

boolean Altered -- set if there any nodes have exited or aborted on error during the current simulaion pass.

boolean Paused -- When true, execution of the simulation is temporarily halted, and all that goes on is communication with client monitors.

boolean Logging -- if set to true, reports of each context switch and each simulation pass completion is printed on stderr. (Note: if the Nest library has been compile with the preprocessor symbol "debug" defined, additional information useful for debugging Nest itself will be printed.

boolean Broadcast -- if true, then messages are sent from a sender to all of its neighbors, regardless of designated destination. (Note: this flag and Point2Point are mutually exclusive).

boolean Point2Point -- if true, then all messages must be addressed to individual node destinations, not to the "all-of-my-neighbors" destination indicated by 0. (Note: this flag and Broadcast are mutually exclusive).

long Delay -- default global delay used to set message delivery delay on links which have weight set to 0.

fd_set Clients -- bit mask which indicates which file descriptors correspond to sockets with clients monitors connected to them.

fd_set Writes -- bit mask which indicates which file descriptors correspond to sockets for which data to be sent to a client monitor via nbwrite() is still pending completed transmission by a subsequent call to nbsend().

5. Miscellaneous Hints

5.1. Shared Data

The Nest programmer must be careful with storage which is accessed by more than one node. Such storage falls into two categories, static or extern data, and heap data. It is suggested that global data be allocated in the form of arrays with one entry per node, so that each node only access its own entry. If this discipline is followed there can be no corruption of one node's data by another node. In cases where this is too restrictive because several nodes need to share a single global data location, critical sections of code should be bracketed by calls to hold() and release().

It is common for nodes to share access to data structures in the heap to which a pointer has been passed from one node to the other in a message. As with globals, hold() and release() might be needed to enclose critical sections if both sender and receiver node were concurrently accessing the data structure. However, this is a very bad practice which should be avoided, not only because creates programming difficulties, but because it compromises message-passing semantics; ie. a message should be readable in the state it was in when it was sent. Thus, concurrent access to message data should be prohibited by either synchronizing access through message passing, or else by passing sending pointers to copies of data structures, not pointers to the original data structures, in messages.

In addition to the problem of concurrent access, there is the problem of ensuring that all storage allocated on the heap is freed. If a sender sends a pointer to a data structure to several recipients, no recipient, nor the sender, should free it unless it is somehow sure that all recipients have read it. Once again, message passing can be used to provide the necessary synchronization for this, but a better solution is to duplicate data structures, either before sending them, or in transit by a channel function. When this is done, all recipients can and should free messages after they get them, and senders can free their original copies of the data whenever they like.

5.2. Debugging

You can use your favorite debugger, eg. dbx, dbxtool, adb, to debug a Nest simulation. One factor that complicates debugging is that, while you step through a the function running on a certain, Nest may switch context on you, so that you are suddenly thrown into the middle of a function running on another node. As a result, it is important to be cognizant of the current node id at all times, and to make sure that you notice whenever it changes as a result of a context switch. One very convenient way of doing this is provided by dbxtool on SUN workstation. In dbxtool you can ask to "display" the variable node_id, and the current value of node_id will be constantly displayed in a subwindow so that it will change whenever the context is switched.

Note that in order to debug Nest in dbx, you should have dbx ignore the signals SIGPROF, SIGALRM and SIGURG. You can set this in a .dbxinit file.