

# NEST: A Network Simulation and Prototyping Tool

*David F. Bacon IBM T.J. Watson Research Center*

dfb@watson.ibm.com

*Alexander Dupuy, Jed Schwartz, and Yechiam Yemini Columbia University  
Department of Computer Science 520 West 120 Street New York, NY 10027*

{dupuy, jed, yemini}@cs.columbia.edu

## Abstract

This paper describes Nest, a testbed which provides a simulated network environment for developing and analyzing distributed systems and algorithms. Nest has a number of interesting features, including a transparent implementation of lightweight processes under UNIX, and a distributed monitoring facility with a graphical user interface.

## Introduction

Nest (Network Simulation Testbed) is a tool for simulating and prototyping distributed algorithms and systems. Nest allows the simulation of arbitrary network topologies and communication characteristics. In addition, the network can be configured and controlled dynamically within the program, or through a mouse-driven graphical user interface which displays the state of the network and allows the user to change it.

Nest is provided as a library of functions which are linked together with the user's code, and can be used with any language that follows the standard stack discipline; this approach to network simulation has also been used by [Bac187], [Cook87], and [Xu87]. There are no built in functions for statistics gathering, but since the user can easily program the functions that run on nodes and pass data over links, relevant statistics can easily be gathered. In addition, at regular intervals during the simulation, a user function is invoked which can change the network configuration or perform any other function that requires a globally consistent state.

The entire simulation runs within a single Unix process. Nest schedules the processes associated with the nodes in the network, and ensures that messages arrive in the correct order in simulated time. There are several advantages to having the simulation inside of a single process: first, users need not concern themselves with encoding data structures for transmission over the network, since they can simply pass a pointer to a heap-allocated structure. Secondly, debugging tools like dbx which only operate on a single Unix process

can be used. Finally, it is possible to simulate systems with large numbers of processes, since Nest in effect provides a lightweight process mechanism like those discussed in [Kepe86] and [Libe87]. However, Nest provides a greater degree of transparency, since it does not require explicit co-routine functions to transfer control from one process to another, nor does it require the stack for each process to be pre-allocated or of fixed size.

Nest provides a communications interface to an abstract network through the functions `sendm()`, `recvM()`, and `broadcast()`. By default, messages are delivered reliably in a fixed amount of time associated with the particular channel on which they were sent. However, users may provide their own channel functions to model arbitrary behavior of the transmission medium. In fact, each channel has a stack of functions associated with it, somewhat like the stackable protocol modules in Dennis Ritchie's streams.

Since network dependencies are isolated in a few functions that provide the communications interface, Nest may be used to develop and debug prototypes of network applications which can then be run on real networks with only minor modification.

## The Nest User Interface

Nest applications are controlled through a separate user interface program. This user interface presents a graphical display of the network, as well as a panel of controls for simulation operations (see Illustration 1). It communicates with the simulation using a TCP/IP connection, and multiple user interface programs can connect to a single simulation. The separation of the simulation from the user interface allows the simulation to be run on a compute server, while being controlled from the user's workstation.

The main feature of the user interface is that it allows the user to dynamically create and modify the network configuration. Nodes and links can be created and destroyed, and the processes being run on the individual nodes and the functions controlling the links can be changed with the user interface. This is done with mouse button clicks in the network window, where a diagram of the current network is displayed. Clicking the left mouse button creates a new node at the current mouse position. Links are created by positioning the cursor over an existing node, then dragging the mouse with the middle button down until it is over another node.

Deletions and other modifications can be made using the pop-up menus which appear when the right mouse button is pressed. When this button is pressed while the cursor is on or near a node, a menu appears with items which allow the user to delete or modify the node, as in the illustration. It is also possible to display more detailed information in a panel above the network display, where it can be modified. A similar pop-up menu appears when the right button is pressed while the cursor is over a link. If the right button is pressed while the cursor isn't over any node or link, another menu is called up, with an

item to undo recent deletions, as well as one which sends a message to the simulation containing all the changes to the network which the user has made.

A panel above the network display allows the user to control simulation parameters, including the granularity of the simulation phases, whether the network supports broadcast messages, and the default delay for messages being sent through links. There is also a panel which allows the user to specify the simulation which the user interface is connected to, whether other user interfaces can control the simulation, and permits the entire simulation to be paused (perhaps to be resumed at a later time).

## An Example Nest Application

Having looked at the basic user interface to Nest, we will turn to the programmer interface (i.e. what someone writing a simulation, or prototyping a distributed system, needs to know in order to use Nest). In the classic Unix tradition, we will use a variant of the famous "hello, world" program to demonstrate the basic features and usage of Nest.

The program in Figure 1 is a complete Nest program in C. It can be compiled and linked with the Nest library to create a Nest simulation program. The first thing you may notice is the absence of a `main()` routine. While the programmer can supply a `main()` routine for the emulation program if desired, the Nest library contains a generic main routine which initializes the simulation with `node_main()` as the main routine for each simulated node. This main routine for each node takes a single argument, the node id assigned to it by Nest. Node ids are used by Nest to uniquely identify each node, and are used whenever a node needs to be specified for a Nest function.

An important part of a network simulation is communications between nodes. An example of this can be seen at the beginning of the `node_main()` routine. The first thing the routine does is to broadcast a message to all its neighbors. A message in Nest consists of two parts, usually called the key and the data pointer. While these are both just 32 bit quantities, they are conventionally used in different ways. The key is typically used to identify messages, either by type or by number. The data pointer is usually a pointer to the data of the message, in some format determined by the type of the message. In the example, the key is the defined constant `HELLO`, and the data pointer is just a pointer to the null-terminated string "hello, world". This simple message structure of key and data pointer is extremely flexible, since the data pointer can point to any sort of data, from character strings to complex linked data structures, such as trees and lists.

After communication, the next most important thing in a simulation is the passage of time. Since all the processes in a simulation are running in a single Unix process, and since the passage of time within the simulation is largely independent of real time, system calls such as `sleep()` and `time()` will not have the desired effect. Instead Nest provides alternate

routines, such as `slumber()`, which is the next function called in the example. After a node broadcasts the hello message, it would like to receive messages from its neighbors. But it may take a certain amount of time for the messages to arrive. So `slumber` is called to suspend the node (in this case, for five seconds) to allow messages to arrive. The `SLUMBER_NOWAKE` parameter is a defined constant which tells Nest not to interrupt the `slumber` if messages arrive.

---

```
#include

#define HELLO 1
#define ACK 2

struct timeval five_seconds = { 5, 0 };

node_main (nodeid)
ident nodeid;
{
    char *message;
    ident dest, sender;
    int msgtype;

    broadcast (HELLO, "hello, world");           /* broadcast hello me
    slumber (&five_seconds, SLUMBER_NOWAKE);    /* wait for hellos */
    while ( any_messages ( ) ) {                /* avoid blocking on
        sender = recvm (&dest, &msgtype, &message);
        hold ( );                               /* begin critical sec
        printf ("%d received
                nodeid, message, sender,
                dest == 0 ? "broadcast" : "sendm" );
        release (1);                            /* end critical secti
        if (msgtype == HELLO)                   /* acknowledge hello
            sendm (sender, ACK, "isn't that a bit cliche?");
    }
}
```

**Figure 1.**

---

Once the node has waited a certain amount of time, it calls `any_messages()` to see if any messages are available to be received. This prevents the node from blocking indefinitely

on a receive if there are no more messages which have been sent to it. While there are messages to be received, the node calls `recv()` to receive the message. It passes three pointers to variables, which are set to the original destination, key and data pointer of the message. The original destination stored in `dest` is just the node id of the receiving node, or 0 (which is not a valid node id) if the message was broadcast. The nodeid of the sender of the message is returned by `recv()`.

Once the message has been received, the example prints a diagnostic message describing the message which has been received. Before it does, it calls the `hold()` function to prevent Nest from interrupting the `printf` call and giving control to another node. This ensures that the messages from different nodes will not be mixed together, as well as preventing any problems caused by non-reentrant implementations of the `stdio` library. The `release()` function is called afterward, indicating that the critical region has ended. The parameter to `release` indicates the number of nested `hold()` calls to be released.

Finally, the node replies to each received HELLO message with an acknowledgment. Since the acknowledgment is directed to the node which sent us the HELLO, `sendm()` is called instead of `broadcast()`. The first parameter to `sendm` is a destination node id; otherwise it is identical to `broadcast`.

## Implementation of Nest

Nest is implemented as a library that is linked together with the user's code. The user sets up the network and any other global data structures required, and then begins the simulation by calling the `simulate()` function. The simulation, once started, proceeds in a series of passes. During a pass, each node is allotted a fixed amount of simulated time called the pass time. Time may be consumed by running, sleeping, or waiting for the arrival of a message.

During a pass, the network remains fixed, so that all nodes run with a consistent view of the network. Any changes to the links or the nodes are made between passes. These changes can be made from a user interface client, or from a special monitor function which is called before each pass. The `nest_monitor()` function is supplied as a default-it merely reports any changes (generally that a node has terminated) to the user interface clients, but does not alter the simulation otherwise. The simulation proceeds until all of the node functions terminate or are blocked waiting for messages (i.e. deadlocked).

The two most important functions of the Nest implementation are (1) performing the context switching between the nodes, and (2) ensuring that messages arrive in the proper order in simulated time. The context switching is done by setting a timer interrupt that goes off when the node has used up an amount of time equivalent to the pass length. In the case when the node does not perform any receive operations during its quantum, the timer goes

off and the Nest timer interrupt handler routine gets control. It copies the registers, the global variable `errno`, and the stack below the `simulate()` routine into a save area using an assembly language routine. The next node is scheduled, and this information is restored (if it is being resumed) or initialized (if it is running for the first time).

If a `recv()` call is made, the node must be suspended. This is because other nodes which are earlier in simulated time might send a message that would arrive before the time of the `recv()` in simulated time. Even if there is a message in the queue at the time of the `recv()` call, another message might be sent later in real time which arrives earlier in simulation time. In order to ensure that messages are dequeued in the proper order, the node with the earliest simulated time is always run first. A node is suspended either because it is at the beginning of its quantum, or because it was blocked by issuing a `recv()`. In the latter case, when we resume the node we know that we can safely dequeue the earliest message in the queue because there are no other nodes with an earlier simulated time that could send a message.

In the previous section the `any_messages()` function was introduced. While conceptually simple, its implementation is fairly complex because of the necessity to maintain temporal consistency. It first checks if any messages are in the queue that have already arrived (that is, that have an arrival time less than or equal to the current time). If there are, it returns true. If not, it must invoke the scheduler just like `recv()`, and for exactly the same reasons: nodes with an earlier simulation time might send a message that will arrive before the current time.

Note that `any_messages()` does not use any simulated time—conceptually it always returns "right away". However, `recv()` may or may not use simulated time: if a message arrives before the time of the `recv()` call in simulated time, the `recv()` will invoke the scheduler but when the node is resumed it will be at the same time; if a message does not arrive until later in simulated time, it will be resumed at a later simulated time, assuming that a message is eventually sent. We distinguish between these two states by calling the former waiting and the latter blocking. The `any_messages()` function only waits, while `recv()` may do either.

Since one node function will generally be used on more than one node, the node functions must be re-entrant (that is, no global or static variables). The `hold()` and `release()` calls described above must be used to bracket non-reentrant code, such as accesses to global structures or calls to non-reentrant system routines. When using Nest, `malloc()` is redefined to call the `hold()` and `release()` routines, since `malloc()` is non-reentrant.

## Advanced Uses

The existence of channel functions and the monitor function, combined with Nest's simple

yet powerful network representation scheme, allows Nest to be used for a variety of purposes. Since the monitor function is called at the start of each pass, when the simulation is in a consistent state, it can be used to collect statistics at discrete intervals. The monitor function can also be used to dynamically alter the network topology under program control.

In addition, the channel functions can be written to model any kind of transmission behavior. The default function, `reliable()`, models the kind of connection provided by TCP, providing a fairly high-level abstraction of a network. However, channel functions can be written to model the network at a lower level, including such things as noisy lines, transmission delays, etc. Similarly, the monitor function can be written to model unreliable processors by "crashing" nodes.

In fact there is a stack of functions associated with each channel, and each function can filter the message as desired. This makes it easy to combine several different channel behaviors implemented by different functions, and to change the behavior of the channel dynamically.

The next section illustrates some of the uses for Nest by presenting a number of applications which have been developed using Nest. These range from a prototype of a complex distributed algorithm for position location to a detailed simulation of the ARPANET routing and topology update algorithms.

## Applications of Nest

### IPLS - A Distributed Incremental Position Location System

The first application to be developed using Nest was IPLS. The basic idea of IPLS is that nodes in a mobile packet radio network can determine their relative positions from communications delays. Distance information between nodes can be derived from fine measurements of the delays, and this distance information, combined with topological analysis of the network, is sufficient to recast the problem of relative position location as a problem in graph rigidity. Once the relative positions are fixed, absolute positions can be determined, given that the absolute positions of three of the nodes are known.

However, since we had no packet radio network on which to develop a distributed implementation of IPLS, we developed Nest in order to give us a testbed in which to do this work. A number of features of Nest helped tremendously in developing IPLS. In particular, the ability to ignore all the lower layers of the network made the implementation of the basic protocol extremely simple. The ability to pass complex data structures as messages between nodes made the implementation of the second phase much easier, since the intermediate results could be passed "as-is", without serialization. Additionally, when

the data structures used to implement the intermediate forms changed, which happened a number of times, none of the communications routines needed to be rewritten.

Since the raw data for the IPLS algorithm is the network topology itself, we needed to be able to test IPLS with a large number of network configurations. The ability of Nest to specify network configurations interactively, via the user interface, made this task much easier.

## **Simulation of Microeconomic Model for Load-Balancing**

Nest has also been used to develop a simulation of a load-balancing system based on microeconomic principles. The basic idea of the work is that the laws of supply and demand could be used to equitably distribute CPU and communications loads among a number of processors. Each process which has work to do also has a certain amount of "money", which is used to bid for CPU and communications resources. If a processor is overloaded, while neighbors are underutilized, bidding will force up the "price" of CPU time at that processor, and processes will decide to migrate themselves to neighbors where the price of CPU is lower.

A number of alternatives were explored in deciding what system would be used to implement this simulation, and there were several reasons why Nest was chosen over more traditional discrete-event type simulations. The microeconomic model presumes a certain amount of basic intelligence in the processes themselves, since it is they who decide when and whether they will migrate from one processor to another. With Nest, it was easy to write a number of C functions which implemented various degrees of intelligence for the processes; these were simply called by each node's main routine when appropriate. Additionally, the somewhat complex bidding procedure used on each machine to determine pricing and allocation of resources was quite naturally built in to the main routines for each node. In a discrete-event based simulation package these sorts of behaviors would have been more awkward to simulate.

## **A Study of the Topology Update Problem in the ARPANET**

Another project which used Nest was a study of the ARPANET topology update problem. The topology update problem is a phenomenon which exists in store and forward networks with dynamic routing of packets. Since each node maintains a local database of network connectivity and delays, it is possible for inconsistencies between nodes to cause problems such as looping and forms of lock-up.

This project used Nest to build a detailed model of a small ARPANET-type network, with hosts and end-node and intermediate IMPs connecting them. Both normal (data) messages and topology update messages were simulated. Extensive use was made of Nest's ability to

run a user-supplied function at the start of each phase of the simulation. This monitor function was used both to collect statistics from the previous simulation phase, and to generate dynamic events, such as crashes and reboots of hosts and IMPs. The ability to dynamically alter network topology and crash and restart nodes in Nest was extremely useful in this project.

## **Other work and availability of Nest**

Nest has been distributed to a large number of other sites. One site which has done extensive work with Nest is the Northrop Research and Technology Center, where Nest is being used to simulate some of the Noahnet protocols. The Noahnet is a network architecture based on flooding protocols. More details can be found in [Rose87].

The current release of Nest version 2.4 runs on Dec VAX minicomputers and 68000 workstations including Sun and Integrated Solutions 68K, under any 4.2bsd Unix derivative (including 4.3). A port to System V should be available shortly. The user interface client runs on Suns with 2.0 or later release of SunView. An X client developed at Northrop exists, but it depends on their own modifications to Nest. For details on obtaining Nest, contact the authors at the electronic or postal addresses above.

## **Conclusions**

### **Results**

As the previous examples have shown, Nest is a useful tool for a variety of network simulation problems. There are a number of reasons for its success as a network simulation tool. The major reason is that Nest has a very general, and programmable, model of network architecture. The division of Nest into a simulation server and user interface clients is another important reason.

Since Nest doesn't assume a specific network architecture or modeling method for the network, it can be used not only for studying problems using standard queuing models, but also for problems where other considerations are more important, or other approaches provide more useful results. The fact that Nest is a library designed to be used with a standard language (in this case C, although interfacing to Fortran or Pascal would be possible), rather than a simulation package with limited programming capabilities, gives the simulation builder the ability to fine-tune the behavior of the simulation program, and the interaction between simulation objects.

The concurrent processing provided by Nest allows a more natural approach to the simulation of distributed systems, without the need for explicit co-routine structures. The

provision of a monitor function which is run at the start of each phase of the simulation allows centralized functionality for statistics and logging, or code which deals with global data or file descriptors.

The simple communications facilities provided by Nest allow the user to be concerned with only the the highest (application) level of a protocol or distributed system, without worrying about lower-level issues. On the other hand, the user can also customize the behavior of these communications facilities to reflect the realities at any of the lower layers. The combination of these features with the ability to run ordinary C code in a concurrent processing environment makes Nest ideal for prototyping distributed systems.

The division of Nest into simulation server and user interface clients has proved to be a useful technique. It allows the simulation to be run on a large machine with more CPU resources, while the user interface can run on a workstation. More importantly, it separates the functions of computation and user interaction in a way that simply using a window server system such as X would not achieve, since Nest can have multiple user interfaces running at once, and users can connect and disconnect from the simulation at any time. This is especially useful for long-running simulation programs.

## **Future Directions**

Our project group has taken Nest as a starting point for the development of a system for network monitoring and management called Net-MATE. This work takes the basic Nest components of simulations and user interfaces, and extends the structure to include not only multiple user interfaces, but also multiple sources of network information and control (for both real and simulated networks), all coordinated through a central network model database.

Unusual features of this project will include support for tracking multiple layers of network protocols, from physical to application, as well as support for user interfaces which deal with these multiple protocol layers in large and complex networks of hundreds to thousands of nodes.

There are also a number of areas more directly related to Nest in which future work could be done. One possibility would be to integrate Nest with an object oriented language, such as C++ or Objective-C, in order to support simulation programs with much of the flavor of Simula. The concurrent nature of Nest implies that substantial speedups could be achieved by a parallel implementation for shared-memory multiprocessors. While this would require a major overhaul of Nest, the resulting performance increases could well be worth it.

## **References**

[Bacl87] Kenneth Baclawski, A Network Emulation Tool, Proceedings of the Symposium on the Simulation of Computer Networks, 1987.

[Cook87] Robert P. Cook, Starlite, A Network-Software, Prototyping Environment, Proceedings of the Symposium on the Simulation of Computer Networks, 1987.

[Kepe86] Jonathan Kepecs, Lightweight Processes for UNIX Implementation and Applications, Proceedings of the Summer Usenix Conference, 1985.

[Libe87] Don Libes, Multiple Programs in One UNIX Process, Usenix Association Newsletter, Volume 12, No. 4, 1987.

[Rose87] Marshall T. Rose, "The Nest Simulation Testbed at NRTC", Northrop Research and Technology Center Technical Paper, 1987.

[Xu87] Chong-wei Xu, A Simulation Test Bed for Computer Networks, Proceedings of the Symposium on the Simulation of Computer Networks, 1987.

---

This research was supported in part by the Department of Defense Advanced Research Project Agency, under contract N0039-84-C-0165, and by the New York State Science and Technology Foundation, under contract NYSSTF CAT (86)-5.