A Network Emulation Tool

Kenneth Baclawski* kenb@ccs.neu.edu

August 24, 2002

Abstract

The Network Emulation Tool (NET) is a software simulation of a computer network. NET is designed for research and teaching of distributed algorithms in a computing environment that supports the Berkeley Unix operating system. Two methods are provided for customizing NET to simulate a particular computer network. One can specify the parameters of a standard network provided by NET, or one can program a new communication layer. This report describes the NET interfaces and gives examples of the use of the NET system.

1 Introduction

Networks and software designed to take advantage of networks are increasingly common. The Network Emulation Tool (NET) is a simulation of a communication network on a fixed number of computers, each with its own shared database. NET is designed to aid research and teaching of distributed operating system and distributed database concepts. Written in C, NET can be used by any computing environment that supports the Berkeley 4.2 or 4.3 bsd Unix operating system. The NET system has already been successfully used in a course on Distributed Database Systems and is being used for research in distributed concurrency control and recovery algorithms. The course and the NET system were based on [4] and [3]. The main programming assignment in the course consisted of implementing parts of a banking network such as the Cirrus Network [6]. The research is mainly concerned with performance and availability issues as in [5] and [7].

The network being emulated is a collection of sites that communicate with one another via messages. Other communication paradigms, such as rendezvous or remote procedure call are easy to simulate with NET. A detailed example is given to illustrate remote procedure call. All computation at a site is done by processes. These processes are simulated by Unix processes. They carry out their tasks by using Unix system services and by calling procedures of the external interface of NET. As far as is possible, a process is programmed like an ordinary Unix process, and the emulation is made to be transparent. For example, a process can call the printf function to display output on the screen. NET will capture the output produced by such calls and write it, one line at a time, to the screen, with each line labelled by the site and process that produced it.

In designing the NET system, it was recognized that there are many kinds of network and that their characteristics vary a great deal. NET is layered to allow one to use it to simulate particular networks very easily. A standard network is provided by NET, the parameters of which can be adjusted. These parameters control the probability distributions of message delay, loss and duplication, site failure, network partition and site insanity. By adjusting these parameters one can simulate a great variety of networks. Alternatively, one can write a new communication layer that simulates the target network more accurately.

The NET system maintains statistics of message traffic sent and received by each site. These are stored in a reserved page of the database at each site. This can be used to monitor the performance of the simu-

*College of Computer Science, Northeastern University, Boston, Massachusetts 02115

lated network as well as that of the distributed algorithms being tested.

This report is organized as follows. The architecture of NET, both internally and externally is described in Section 2. The two sections after that present the external interface and the internal interface, respectively. Section 5 gives an example of the use of NET. The last section describes some future directions of work planned for the NET system.

2 System Architecture and Design Goals

The NET system emulates a network having a fixed number of sites, numbered consecutively beginning with 0. At each site a variable number of processes can be executing. The processes at each site share a cached, shared database. The processes communicate with each other by sending messages via the network.

The architecture of NET is divided into two layers: the Service Layer and the Routing Layer. The Service Layer is used by user processes via the External Interface. Communication characteristics are determined by calls to procedures in the Routing Layer by the Service Layer via the Internal Interface.

The Service Layer provides the user processes with the External Interface described in more detail in the next section. There are facilities for network communication, database access and process control. All of these except for network communication are completely emulated in the Service Layer itself. Whenever the Service Layer receives a request to send a message via the communication network, it calls procedures in the Routing Layer to determine what happens to the message. The Routing Layer determines when the message arrives, if it arrives at all. In addition, the Routing Layer is periodically queried to determine the current network state. The network state includes information such as which sites are functioning as well as which sites can communicate with one another. When a site goes down, the Unix processes at that site are killed, and volatile storage is lost. A "reboot" process is created when the site comes back up.

The Routing Layer can be the Standard Routing Layer provided by the NET system. Alternatively, a more specific Routing Layer can be written to simulate a target network more accurately. The Standard Routing Layer already simulates a large variety of network events, the occurrence of which is controlled by adjustable parameters. The Internal Interface and the Standard Routing Layer are described in more detail in Section 4.

The implementation of NET makes heavy use of Unix system services such as pipes, signals and I/O redirection. When NET is running, each user process is implemented with a Unix process; and, in addition, a central hub process provides the NET services. All these services are done as follows. The user process calls a NET procedure in the External Interface. This procedure sends a message via a pipe to the hub process. The hub, in turn, reads this pipe and tries to perform the service requested. The hub then acknowledges or rejects the service request by sending a message back to the process on another pipe. The process, meanwhile, has been waiting for a response on the second pipe. When this occurs, the appropriate information is passed to the process, and it continues.

Each process can also write on the standard output. This is redirected to a pipe, and the hub reads it one line at a time. The hub then writes each line on the standard output together with a suitable identifying line. Input is disabled for all processes except for the initial one at just one site.

If there are n user processes running, then there will be $3n$ pipes in use. Each process has access only to one end of each of its own 3 pipes. One pipe is used for service requests, one for service acknowledgements and one for redirecting standard output. The file descriptors of the first two pipes, as well as the site and process identifiers, are passed to the user process in the `environ` global variable. The hub has access to the other ends of all $3n$ pipes. The request/acknowledgement paradigm is designed to be as close as possible to the Ada rendezvous structure.

The NET system is designed to be portable to any 4.2 or 4.3 bsd Unix system. Although perfect portability is probably not possible, a number of methods were employed to make it easier to port NET. The soft-

ware not only compiles with no warnings, but it also passes lint. Furthermore, the `NET` system has been compiled, tested and used in teaching and research on two completely different machines, a VAX 11/750 and a Pyramid 98x, using identical source code.

To begin the execution of the `NET` system, one executes the hub process, this being a program named simply `net`. This program takes three parameters. The first is the name of the database; the second is the name of the "boot" program; the third is the name of the "reboot" program. If any of these are not specified, then the default names `data`, `boot` and `reboot`, respectively, are used. The database is stored in two separate files. The materialized database has the extension `.base`, and the log file has the extension `.log`. The databases of all the sites are stored in these two files for simplicity. Each page or log entry has a header that identifies the site to which the item belongs. The user need not be concerned with the format used in these files.

When the emulation begins, a special process is created at each site, having process identifier equal to 0. This special process is the first step in the bootstrapping of the site. Process 0 has just two duties. It creates a process that executes the initial or "boot" program, after which it loops forever waiting to receive a message. When it receives one, it creates a process that executes the program specified by the message. Process 0 replies with a message containing the process identifier of the process that was created. See Section 5 for an example of how all this works. If a site goes down, and then later comes back up, the same steps are performed as above, except that the "reboot" program is used instead of the "boot" program.

3 External Interface

The functions described below implement the basic operations that can be performed by the processes. In every case, the function returns `success` if successful and `failure` if unsuccessful. The constants `success` and `failure` are integers with values 1 and 0 respectively.

3.1 Network Communication

```
send_message (site_id, process_id, message)
  int site_id, process_id;
  char *message;

send_string (site_id, process_id,
        message, length)
  int site_id, process_id;
  char *message;
  int length;
```

The `send` function transmits information from one process to another. It succeeds if the specified site exists. The two versions differ only in how this information is specified. The `send_message` function transmits a null-terminated array of characters. The `send_string` function transmits an array of characters and a length. This second version can transmit an arbitrary sequence of characters. The first version is easier to use; the second is more general. Other functions below are also in two versions, and no further comment is made about this.

```
broadcast_message (message)
  char *message;

broadcast_string (string)
  char *string;
```

The `broadcast` function performs the `send` function to all the currently active processes, except for process 0 at each site and the process doing the broadcast. The `broadcast` function succeeds if the network being emulated provides this service. The standard network can provide this service if desired.

```
receive_message (site_id, process_id,
        message, timeout)
  int *site_id, *process_id;
  char **message;
  int timeout;

receive_string (site_id, process_id,
```

```
      string, length, timeout)
  int *site_id, *process_id;
  char **string;
  int *length, timeout;
```

The `receive` function waits until a message is received or the `timeout` expires, whichever comes first. The function succeeds if the message is received before the `timeout` expires. The `timeout` period is specified in seconds. The `receive` function allocates a string using malloc, and a pointer to the string is returned. This allows one to make use of messages having a size not known in advance.

Messages from a process at one site to one at another may require a period of time for transmission. Messages sent from one site to another can arrive in a different order than they were sent and can be replicated one or more times. Messages can also be lost due to a site being down, or due to network partition. These all depend on the network being emulated. The standard network can simulate any of these.

3.2 Process Control

```
self (site_id, process_id)
  int *site_id, *process_id;
```

The `self` function returns the site identifier and the process identifier of the process that executes it.

```
create_process (program_name, process_id)
  char *program_name;
  int *process_id;

exit_process ()
```

The `create` function creates a new process at the local site, which executes the specified `program_name`. The `program_name` is a null-terminated string. To terminate a process, the `exit_process` function can be called. It has the same effect as a call to the Unix exit function.

```
enable_interrupts (interrupt_handler)
  int (*interrupt_handler) ();

disable_interrupts ()
```

When interrupts are enabled, the arrival of a message causes the receiving process to be interrupted, and the specified `interrupt_handler` procedure is called. When the `interrupt_handler` returns, the process continues where it was interrupted.

Interrupts are implemented using the Unix SIGINT signal. Interrupts are disabled temporarily while an interrupt is being processed and during a call to the `receive` function.

3.3 Database Operations: Site Database

```
allocate_page (page_id)
  int *page_id;

deallocate_page (page_id)
  int page_id;
```

These allocate and deallocate the specified page in the site database. The process that allocates a page is given a write lock on it, so it can be written immediately after it is allocated. To deallocate a page a process must have a write lock on it.

```
lock_page (page_id, mode, timeout)
  int page_id;
  enum lock mode;
  int timeout;

unlock_page (page_id)
  int page_id;
```

These functions acquire and release locks on pages of the local database. The `mode` parameter is an enumerated type that takes two possible values: `read_lock` and `write_lock`. The `timeout` is in seconds.

```
write_page (page_id, value)
  int page_id;
  char *value;

read_page (page_id, value)
  int page_id;
  char *value;
```

These functions read and write entire pages of the local database. The value of a page is a fixed-length string of length equal to the page size. A page must be appropriately locked before it can be read or written.

```
purge_page (page_id)
  int page_id;
```

Database pages are cached as in a high-performance database system. When a page is read or written, the page in the cache is used. As a result, the page in the disk file may differ from the page in the cache. The `purge` request ensures that the page in the database is brought up to date. When the emulation terminates (or a site goes down), the pages in the cache can be lost.

3.4 Database Operations: Log File

```
write_log_message (message)
  char *message;

write_log_string (string, length)
  char *string;
  int length;
```

The `write_log` function writes a new log entry atomically on the end of the log file of the site. Unlike page writes, no caching is done. The entry is immediately written to the log file. If the log entries are ascii text strings then the log file can be read as any text file would be read.

```
read_log_message (read_direction, message)
  enum direction read_direction;
  char **message;

read_log_string (read_direction,
        string, length)
  enum direction read_direction;
  char **string;
  int *length;
```

The `read_log` function reads the current log entry in the specified `read_direction`, and the current log entry is updated. Each process has its own current log entry. The initial current log entry is the most recent one written. The `read_direction` is an enumerated variable which can take the values `forward_direction` and `backward_direction`. All reading is done sequentially. The `read_log` function fails when the end (or beginning) of the log file is reached. For example, if the first `read_log` call is in the `forward_direction`, then this call will fail.

Like the `receive` function, the `read_log` function allocates space for the entry and returns a pointer to it. This permits log entries to have arbitrary length.

```
reset_log ()
```

The `reset_log` function resets the current log entry of the process to be at the end of the site's log file.

3.5 Emulation Control

```
exit_emulation ()
```

This function terminates all active processes and the emulation immediately, without waiting for the processes to terminate on their own.

```
typedef struct {
  int broadcasting_allowed;
  double message_loss_prob;
  double message_repeat_prob;
  double message_delay_mean;
  int message_delay_dist;
```

```
  double site_failure_mean;
  int site_failure_dist;
  double site_repair_mean;
  int site_repair_dist;
  double site_insanity_mean;
  int site_insanity_dist;
  double network_failure_mean;
  int network_failure_dist;
  double network_repair_mean;
  int network_repair_dist;
} network_parameters;

change_emulation_parameters
                (new_parameters)
  network_parameters *new_parameters;
```

This function allows one to alter the characteristics of the standard network. If the standard network is not being used, this function may not be recognized, and even if the function is recognized, not all the parameters will be used. The standard network allows this function to be called just once. All other calls fail and have no effect. The meanings of the various parameters above for the standard network are defined in Section 4.

3.6 Diagnostic Output and Statistics

When one is testing a new program that uses `NET`, it is helpful to be able to print diagnostics. These would normally be removed when the program is executing in a satisfactory fashion. To print such diagnostics, simply use the normal C function printf. The emulator captures the output and prints it on the standard output after a line indicating the site and process that produced the output. The initial process at site 0 is the only exception to this. Output produced at this site is sent directly to the standard output without any identification of its source.

Statistics of message traffic for each site are maintained by the hub process and stored in page 0. All values saved are integers. Each value counts events according to four attributes, each of which has two cases:

1. *The count can be the number of messages (mesgs) or the number of bytes (bytes).*
2. *The messages could be sent from the site (sent) or were to be received by the site (recv).*
3. *The messages either arrived at their destination (arri) or got lost along the way (lost).*
4. *The messages were either broadcast (bcst) or ordinary (norm) messages.*

As an example, the number of ordinary messages that were sent to the site but never arrived would be stored in the field named `norm_lost_recv_msgs`. Each of the sixteen possible names denotes one field in page 0. Page 0 has the following type:

```
typedef struct {
  int norm_arri_sent_bytes;
  int norm_arri_sent_mesgs;
  int norm_arri_recv_bytes;
  int norm_arri_recv_mesgs;
  int norm_lost_sent_bytes;
  int norm_lost_sent_mesgs;
  int norm_lost_recv_bytes;
  int norm_lost_recv_mesgs;
  int bcst_arri_sent_bytes;
  int bcst_arri_sent_mesgs;
  int bcst_arri_recv_bytes;
  int bcst_arri_recv_mesgs;
  int bcst_lost_sent_bytes;
  int bcst_lost_sent_mesgs;
  int bcst_lost_recv_bytes;
  int bcst_lost_recv_mesgs;
} statistical_information;
```

Note that a message may count as arrived from the point of view of the source site but as lost from the point of view of the destination site. For example, if the destination process exists when the message is sent but exits before receiving the message, then the message is considered lost only at the destination site.

4 Internal Interface

The architecture of `NET` permits one to specify a specific network to be emulated. Alternatively, a standard network is provided whose parameters can be

varied so as to simulate quite a variety of networks. In this section the internal interface and the standard network are described.

4.1 Standard Emulation

The standard network emulation is used unless a new Routing Layer is written. The characteristics of the standard network are determined by the `network_parameters` record. These fields of this record are now defined.

The first field is the simplest one. Broadcasting of messages is allowed when `broadcasting_allowed` is nonzero. The default value is zero.

The next two fields determine the probability that a message is lost or duplicated (for reasons other than site failure, site insanity or network failure). The probability that a message is lost is `message_loss_prob`. The default value is zero. The probability that a message is duplicated is `message_repeat_prob`. The duplicated message can also be duplicated, and so on. The average number of spurious messages produced by this process is

$$\frac{p}{(1-p)^2}$$

where p is `message_repeat_prob`.

The remaining fields define the probability distributions of times until events occur. A pure (memoryless) waiting time has an exponential distribution. If an event requires a succession of independent events to occur then the waiting time will have the distribution of a sum of independent exponentially distributed random variables. For example, in a token passing network, the a site must wait for the token before sending a message. The time taken by each site to relinquish the token can be modelled by an exponential distribution. The total time taken is the sum of as many exponential distributions as there are sites in the network.

For each waiting time, one can specify the mean (average) value of the waiting time and a field called the "distribution parameter" (or *dist* for short) that specifies how many independent, identically distributed exponential random variables are to be added to get the total waiting time. In the special case when $dist = 0$, the waiting time is taken to be constant. If the mean value is set to a negative number, then the waiting will be "infinite." In other words, in this case the event in question will never occur.

When the mean value is m, the variance will be

$$\frac{m^2}{dist}$$

the standard deviation will be

$$\frac{m}{\sqrt{dist}}$$

and the skewness will be

$$\frac{2}{\sqrt{dist}}$$

When $dist = 1$, the skewness is very large. When dist is around 4, the distribution is becoming similar to the normal distribution, except for the skewness. As *dist* increases, the skewness goes away, and the shape of the distribution does not change much, the only change being that the variance is decreasing. For large values of *dist*, the distribution is essentially a constant.

In all cases, the default value for *dist* is zero, i.e., the waiting time is the same as the mean waiting time. Except for message delay, the default value for each waiting time is "infinite." This means that the event in question never occurs. To get the default value, set the mean waiting time to a negative number.

The time it takes for a message to be transmitted from one site to another is determined by the `message_delay` fields. The default value for the mean message delay is one second.

The time until a site fails is determined by the `site_failure` fields. When a site fails, all processes at that site are terminated. The time it takes for a site to be repaired is determined by the `site_repair` fields. When a site comes back up, a process is created and the "reboot" program is executed, just as when the emulation starts. The pages in the cache are lost when a site fails.

The time until a site goes "insane" is determined by the `site_insanity` fields. Site insanity is difficult to define precisely. It can mean that messages will be garbled, repeated, lost or delayed. It can affect all messages to and form the insane site, or it might just involve messages to or from a particular other site. Once a site goes "insane" it never recovers.

The time until the network partitions is determined by the two `network_failure` fields. The partition itself can be into any number of subnetworks. The time it takes for the network to be be repaired after a partition is determined by the `network_repair` fields.

4.2 Internal Interface Specification

The Service Layer and Routing Layer communicate with one another using a shared variable called `current_state` of type `network_state`, defined as follows:

```
typedef struct {
  int broadcasting_allowed;
  int up[net_size];
  int sane[net_size];
  int reaches [net_size][net_size];
} network_state;

network_state current_state;
```

The Service Layer can read the fields of this variable, but only Routing Layer can modify it. All the fields would be boolean if C had such a type. The `broadcasting_allowed` field indicates whether broadcasting is allowed in the network. The `up` field indicates whether a site specified by the index is functioning normally. The `sane` field indicates whether a given site is still "sane," as opposed to "insane." Finally `reaches[i][j]` indicates whether site i can send messages to site j.

```
initialize_routing_layer (current_time)
  int current_time;

reinitialize_routing_layer (current_time,
        new_parameters)
  int current_time;
  network_parameters *new_parameters;

query_routing_layer (current_time)
  int current_time;
```

These functions are the only way that Service Layer can modify the `current_state`. All three functions return `success` of `failure` as in the External Interface. The first function is called when the emulation begins. If it doesn't succeed, then the emulation stops immediately. The `reinitialize_state` function is called whenever a process calls `change_emulation_parameters`. The Service Layer simply passes without change the parameter given to it by the process, and returns the result back to the process. In principle, this allows processes to communicate directly with the Routing Layer, bypassing the Service Layer. The last function, is frequently called by the Service Layer to update the `current_state`. All three functions are given the `current_time`, which is an arbitrary measure of time whose unit is a second. One can only assume that the `current_time` is a large positive number which will not overflow until several decades into the next century.

```
int message_arrival_time (sending_time,
       source_site_id, destination_site_id,
       string, length, copy_number)
  int sending_time, source_site_id;
  int destination_site_id;
  char *string;
  int length, copy_number;
```

This function determines when and if messages reach their destination in the communication network. The `string` and `length` parameters define the message being sent. The two `site_id` parameters determine the source and destination of the message. The time when the message was sent is the `sending_time`. The function returns the time when the message arrived at its destination. Since time is never negative, if the result is negative, then it is assumed that the message never arrived.

The `copy_number` parameter is the number of times the function will have been called for a given message. In other words, for the first call, the value is 1; for the second it is 2; and so on. The Service Layer will continue to call `message_arrival_time` until a negative arrival time is returned, and each such call produces a new copy of the message. This allows the Routing Layer to duplicate messages via the network. Such messages can arrive in any order that the Routing Layer may dictate.

5 Examples

The following example of the use of `NET` illustrates two of the capabilities of `NET`. One part of the example shows the use of the local database at two of the sites. The second part of the example illustrates a "remote procedure call" issued from two of the sites and executed at a third site. The remote procedure logs the "parameter" it was given and returns a "result." Both the "parameter" and the "result" are in the form of a message.

In the first part of this example, the initial processes at sites 1 and 3 allocate a page and write a random value in the first integer of the page. The page is then purged and unlocked. The preceding page is then read and the first integer on the page is displayed. By running this example several times, one can test the persistence of the data in the database.

In the second part of the example, the initial processes at sites 2 and 4 create processes at site 0, executing the program named `remote`. These are different processes, even though they execute the same program and are at the same site. The `remote` process writes the message it receives from another site to its log file, and replies with an acknowledgement.

To begin the emulation simply execute the following command:

```
net data boot reboot
```

The parameters could have been omitted since they are all default names.

The following is the actual output from an execution of the `NET` system. Another execution would produce other numbers of course.

```
This is NET Version 2.0
OUTPUT FROM SITE 1 PROCESS 7548
At site 1, the value 7548 was written to
page 1

OUTPUT FROM SITE 3 PROCESS 7550
At site 3, the value 22650 was written to
page 1

OUTPUT FROM SITE 1 PROCESS 7548
At site 1, the value 0 was read from page 0

OUTPUT FROM SITE 3 PROCESS 7550
At site 3, the value 0 was read from page 0

OUTPUT FROM SITE 2 PROCESS 7549
Remote site acknowledged with
 Thanks for the message.

OUTPUT FROM SITE 4 PROCESS 7551
Remote site acknowledged
 with Thanks for the message.

NET: Emulation completed.
```

The second execution then produced the following:

```
This is NET Version 2.0
OUTPUT FROM SITE 1 PROCESS 7556
At site 1, the value 15112 was written to
page 2

OUTPUT FROM SITE 3 PROCESS 7558
At site 3, the value 45348 was written to
page 2

OUTPUT FROM SITE 1 PROCESS 7556
At site 1, the value 7548 was read from
page 1

OUTPUT FROM SITE 3 PROCESS 7558
At site 3, the value 22650 was read from
page 1
```

```
OUTPUT FROM SITE 2 PROCESS 7557
Remote site acknowledged with
 Thanks for the message.

OUTPUT FROM SITE 4 PROCESS 7559
Remote site acknowledged with
 Thanks for the message.

NET: Emulation completed.
```

At this point, the log file is checked by executing the command:

```
more data.log
```

Here is what was produced:

```
0 7552 13 Hello remote!
0 7553 13 Hello remote!
0 7560 13 Hello remote!
0 7561 13 Hello remote!
```

The source code for the `boot.c` and `remote.c` programs is the following:

```
/*  This is the program boot.c  */

#include "net.h"

main ()
{
  int sid, rsid, pid, rpid;
  int page, *cast;
  char p[256], *m;

  self (&sid, &pid);

  switch (sid) {

  case 1:
  case 3:
    allocate_page (&page);
    cast= (int *)p;
    *cast= page * sid * pid;
    write_page (page, p);
    printf ("At site %d, the value %d ",
            sid, *cast);
    printf ("was written to page %d\n",
            page);
    purge_page (page);
    unlock_page (page);
    lock_page (page-1, read_lock, 10);
    read_page (page-1, p);
    cast= (int *)p;
    printf ("At site %d, the value %d ",
            sid, *cast);
    printf ("was read from page %d\n",
            page-1);
    unlock_page (page-1);
    exit_process ();

  case 2:
  case 4:
    send_message (0, 0, "remote");
    if (!receive_message
         (&rsid, &rpid, &m, 10)) {
      printf ("Remote program could not ");
      printf ("be executed by site %d\n",
              sid);
      exit_process ();
    }
    sscanf (m, "%d", &rpid);
    send_message(0, rpid, "Hello remote!");
    if (!receive_message
         (&rsid, &rpid, &m, 10)) {
      printf ("No message received from ");
      printf ("remote site by site %d\n",
              sid);
    }
    printf ("Remote site acknowledged ");
    printf ("with %s\n", m);
    exit_process ();

  default:
    exit_process ();
  }
}

/*  This is the program remote.c  */
```

```
#include "net.h"

main ()
{
  int sid, pid;
  char *m;

  if (!receive_message
       (&sid, &pid, &m, 10)) {
      exit_process ();
  }
  write_log_message (m);
  send_message (sid, pid,
                "Thanks for the message.");
  exit_process ();
}
```

6 Future Work

As the name suggests the NET system is a tool for further work on distributed systems. There are two research areas that are currently being developed in connection with NET. The first is the area of concurrency control and recovery in distributed and high-performance database systems in [1]. A prototype system is being developed using NET as the base system. The second area concerns a new probabilistic model for data access [2]. This model represents many observed properties of data access that earlier models do not consider. For example, data access is highly nonuniform, and this nonuniformity evolves in time. This new model has already been implemented for a centralized system. We plan to extend it to a network environment using the NET system.

The NET system is scheduled to be used in several other courses in the future. Some of these courses are project courses. We hope that some significant enhancements will result from these projects.

7 Conclusion

A detailed description of the NET system has been presented. As a teaching and research tool, NET has several advantages. Programming a NET process is almost transparent. A variety of network characteristics can be simulated, including a large variety of failure modes. Finally, NET is easy to customize.

References

[1] K. Baclawski. Database transaction options. Technical Report NU-CCS-86-5, Northeastern University, College of Computer Science, 1986.

[2] K. Baclawski. A stochastic model of data access and communication. *Advan. in Applied Math.*, 10:175–200, 1989.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[5] D. Gawlick. Processing of "hot spots" in high performance systems. In *Proc. COMPCON*, 1985.

[6] G. Gifford and A. Spector. The Cirrus banking network. *Comm. ACM*, 28:798–807, 1985.

[7] C. Thanos, C. Carlesi, and E. Bertino. Performance evaluation of two concurrency control mechanisms in a distributed database system. In *Trends in Information Processing Systems*, volume 123, pages 266–279. Springer-Verlag, Berlin, 1981.