Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1989

DEX: A High Level Tool for Distributed System Experiments

Niraj K. Sharma

Jagannathan Srinivasan

Report Number:
89-849

Sharma, Niraj K. and Srinivasan, Jagannathan, "DEX: A High Level Tool for Distributed System Experiments" (1989). *Department of Computer Science Technical Reports.* Paper 723.
https://docs.lib.purdue.edu/cstech/723

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

DEX: A HIGH LEVEL TOOL FOR
DISTRIBUTED SYSTEM EXPERIMENTS

Niraj K. Sharma
Jagannathan Srinivasan

CSD-TR-849
January 1989

DEX: A High Level Tool For Distributed System Experiments

Niraj K. Sharma
Jagannathan Srinivasan
Dept. of Computer Sc.
Purdue University
West Lafayette, IN 47907

ABSTRACT

DEX is a high level tool designed to aid distributed system experimentation. It supports a very general model of distributed system and provides features to simulate different communication networks, work loads, failure patterns, and communication delays. It uses UNIX processes and communication mechanisms for implementing a distributed system testbed. The intermediate approach (i.e., neither simulation nor experimental) adopted by DEX makes it a useful tool for implementing distributed system testbeds at l ow cost and effort.

1. INTRODUCTION

To study and analyse distributed systems more and more researchers are moving towards experimentation. A large number of prototypes and distributed system testbeds [2,4,5,8,10] have been developed. A major advantage of this approach is that it gives performance data for a real system. On the other hand, it limits the range of experiments one can perform with a given system. Further more, there is a high cost involved in establishing the experiment. An alternate approach is to do simulation performance mode lling [9,11,12], which allows to model a distributed system to any arbitrary level of detail. A disadvantage of the simulation approach is that a detailed simulation model can be as complex as a real system being analysed. Secondly, the distributed system model is not directly representable.

The disadvantages of both the approaches motivated us to adopt an intermediate approach, which led to the design of DEX. On one hand, DEX is similar to experimentation approach as it uses UNIX processes and communication mechanism. On the other hand, it is similar to the simulation approach as DEX allows users to simulate communication networks, work load, failure patterns, and communication delays. DEX is a

linguistic tool which provides features to set up distributed system testbeds with low cost and eff ort.

DEX can express a very general model of distributed system. The major features of DEX are as follows:

1. It includes features for specifying different network topologies and has communication primitives to allow communication between any two arbitrary processes in the system.
2. It provides features to support generation of different workloads and failure patterns.
3. It maintains both message count and delays associated with messages. This feature is particularly useful for measuring message complexity of algorithms during failures which is difficult to predict otherwise. In addition, there are features to monitor parameters specific to an experiment.
4. It provides means for specifying constraints to test the correctness of algorithms. During the execution, if a constraint is violated a prespecified routine is executed.
5. It allows specification of code on each node in C language, Thus, a user has all the power of C for expressing an algorithm.

Section 2 describes the model of distributed systems. Section 3 gives rationale and description for DEX features. Section 4 illustrates use of DEX with a replication control experiment.

2. DISTRIBUTED SYSTEM MODEL

In this section, we describe the distributed system model as seen by a designer of the system. If a tool to analyse the system supports the same model, it will be very convenient for the user to express the distributed system.

A distributed system consists of a collection of nodes connected through communication channels. In such a system, there may be either point to point connections

or a broadcast channel. It is also possible to have both types of communication channels in a system. Each node executes a set of processes. Processes communicate and synchronize with other processes on the same node using shared memory based primitive. To communicate and synchronize with the processes on other nodes, a process can use only messages. Nodes and communication channels may fail any time. Messages may get lost even when there is no failure in the system. Message communication can be made reliable by using timeouts and explicit acknowledgements.

The design of DEX is based on this model and it supports convenient features to express both large and small types of networks.

3. DEX

For any tool to express distributed systems, the following questions should be answered satisfactorily.

- Is it complete? (Can it describe a wide variety of distributed problems?)
- Is it easy to interpret?
- Is it easy to understand and express the distributed computation?
- Is it compact?
- How easy it is to express a distributed system having a large underlying network?
- Can pieces of computation already built be used to compose more complicated systems?

While designing DEX, we considered all these questions. The model of an experiment in DEX as seen by a user is shown in Fig. 1. It consists of two subsystems, namely, *Global Control & Monitoring Subsystem* and *Experimentation Subsystem.* Global Control & Monitoring Subsystem contains functions useful for control and monitoring various parameters of the experiment. Experimentation Subsystem contains the code corresponding to the distributed computation being analysed. It consists of the specification of a communication network and the code to be execut ed at each node.

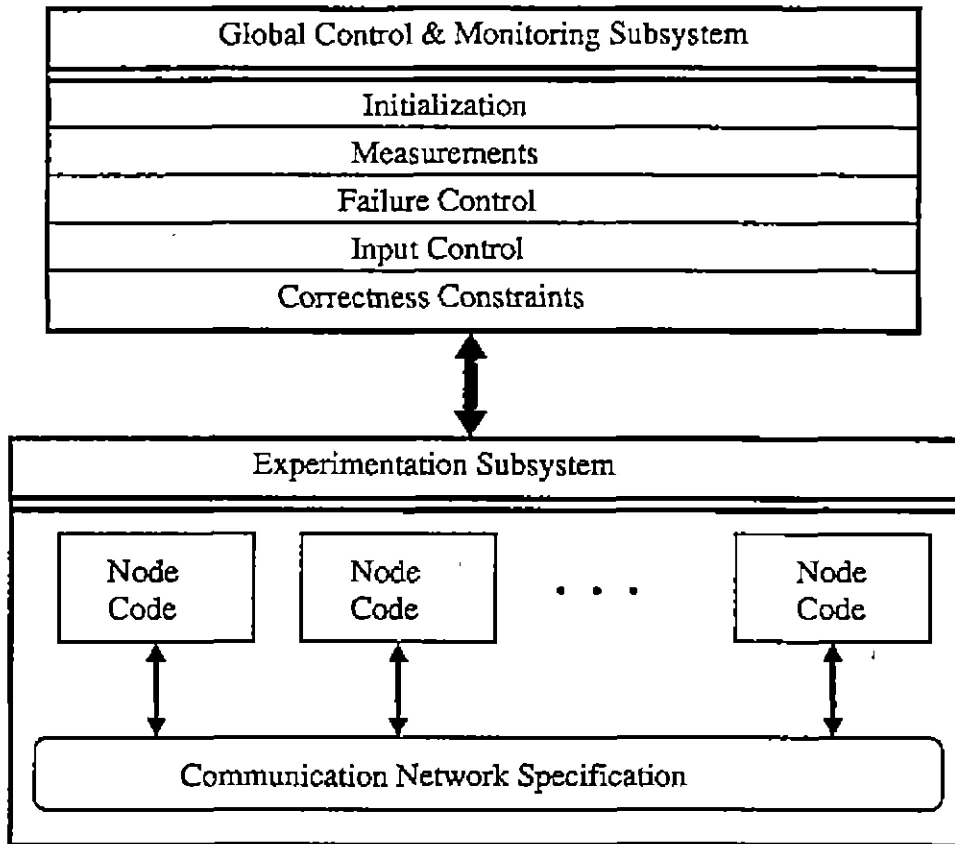

Fig. 1 Model of an experiment in DEX

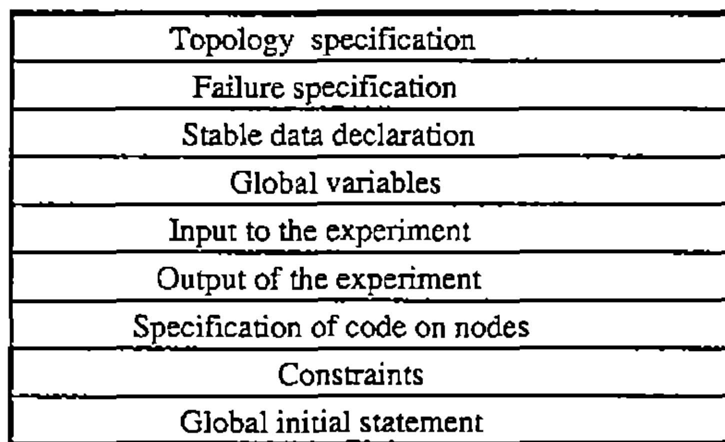

Fig. 2 Components of a DEX program

The correspondence of the model of Fig. 1 with the components of a DEX program (Fig. 2) is as follows.

- *Initialization*: Global initial statement initializes the state of an experiment. This statement is executed first when an experiment is started.
- *Measurement*: Global variables are defined for measuring system parameters. They can be either system defined or user defined. System defined variables, e.g., *faultylinks*, *faultynodes*, etc., are maintained by the system. User defined variables has to be maintained by the user. Output of an experiment specifies the the output to be generated from the experiment which can later be used to form graphs, tables, etc.
- *Failure control*: It is done through the failure specification clause. It contains statements to specify failure and repair rates for nodes and links.
- *Input control*: The input clause describes the input to be provided to the code being executed at each node. It usually contains workload specifications, e.g., generation of transactions at all the nodes with some prespecified rate.
- *Correctness constraints*: The constraint clause in a DEX program contains a set of boolean expressions involving global variables and the variables defined on nodes. Violation of these constraints is treated as an exception. In such an event, the code associated with the constraint is executed. This feature is useful to check the correctness of computation.
- *Communication network specification*: The topology specification clause in a DEX program consists of the declaration of nodes and links. There are special features to express large networks easily and it avoids explicit enumeration of nodes and links.
- *Node code*: Stable data declaration specifies the data to be maintained on stable storage at different nodes in the distributed system. Data stored on stable storage is available when a node comes up after being down for some time. Specification of code on a node consists of local system defined variables (e.g., an array containing the id's of all the neighbors of a node), user defined variables, a set of process declarations and

a initial statement at the end, as shown in Fig. 3. At a node, t he initial statement is executed first.

Local system defined variables
Local user defined variables
Processes
Initial statement

Fig. 3 Code at a node

A process in turn consists of the declaration of variables, procedures, guarded procedures, and a process initial statement as shown in Fig. 4. Guarded procedures are just like ordinary procedures except that they are preceded by a guard which consists of a boolean expression. After executing the initial statement, the process selects and executes a guarded procedure with a true guard. If more than one guarded procedures have true guards one of them is randomly selected for execution.

Variable declaration
Procedure declaration
Guarded procedure declaration
Initial statement

Fig. 4 Model of a process

The source code of an experiment is stored in a file. The DEX translator reads the source code of an experiment and translates it into standard C code which is then translated to object code by the C compiler. In the source code file of an experiment, the specification of various optional components should be in the order they are presented below.

3.1. Topology specification

Topology of a network is expressed using the clause

topology {<contents>}

where <contents> contains the specification of nodes and links.

3.1.1. Node specification

It consists of a definition of node id's to be used in the experiment. A node id is an integer. There can be only one node specification statement. For example, consider the statement

nodes: 1, 2, 5..9, 10..100 **by** 10, 101..110;

One can explicitly enumerate the id's or a range of id's. The purpose of providing the range facility is to express large number of nodes easily. In the absence of the range facility, a user will end up enumerating each node id explicitly.

A node can not be defined twice. Id's should be in the ascending order. It is also possible to specify the message processing time on a node by inserting a **time** statement before the declaration of the node, e.g.,

nodes: **time** 3.1, 1, 2, **time** 0.0001, 10..100;

In the above statement, the message processing time is 3.1 seconds on nodes 1 & 2 and on nodes 10 to 100 it is 0.0001 seconds. A **time** statement is effective upto the next **time** statement or the end of the node specification.

3.1.2. Link specification

There can be only one link specification statement and it is of the form :-

link: [<link_set>], [<link_set>], ..., [<link_set>];

Here <link_set> consists of <from> <type_of_connection> <to>, where <from> and <to> are node sets and <type_of_connection> specifies whether the connection is bidirectional (represented by '-'), unidirectional (represented by '->'), or broadcast (represented by '<>'. In this case the set <to> will not be present.)

The sets of nodes <from> and <to> can be specified in the following different

ways.

- By a range. e.g., 1..100.
- Enumeration: Explicitly enumerate the node id's, e.g., 1,2,10.
- Mixed: Range and enumeration method both can be used together, e.g., 1, 2, 3..10, 11, 12..100 by 2.

In some cases if the number of nodes in a network is very large, the user may not want any specific topology. To deal with this situation, there should be a feature using which a user does not have to explicitly enumerate all the links in the network. The feature in DEX to create links automatically is to specify <link_set> by.

<from> => **rand**(i..j) <type_of_connection> <to>

or

<from> => i <type_of_connection> <to>.

where **i** and **j** are integers. This form of specification will fix the number of links coming out of the nodes in the set <from>. In the case of rand(i..j) the number of links will be a random number between **i** and **j**, which will be calculated separately for each node in the set <from>. In the case of just **i**, the number of links coming out of every node in <from> is i. The selection of nodes from <to> is always done randomly depending upon how many links are coming out from the nodes in the set <from>. The system will never generate disjoint partitions.

Examples:

The link specification

links: [1..10 - 1..10];

means that every node in <from> is connected to every node in <to> ,i.e., there is a fully connected network with bidirectional links. The specification of a link from a node to itself is ignored by the system.

links: [1 - 10];

means just one link and

links: [1..10<>];

means there is a broadcast channel connecting 1st to 10th node.

links: [1..10 => rand(2..4) - 1..10];

means the number of links connected to a node is a random number between 2 and 4.

links: [1..10 => 2 - 1..10];

means the number of links connected to a node is fixed to 2.

Time to process messages can also be specified with link definitions using the **time** clause as used in the case of nodes, e.g.,

links: **time** 0.00001 [1..10 => 2 - 1..10];

3.2. Failure specification

There can be two methods to express failures/repairs. The first one is that user wants some failures/repairs to occur without bothering about their timings. In the other case, a user would like to specify the instances of failures/repairs at some specific moments while executing the algorithm. The later case will be dealt with in the section on Specification of Code on Nodes. Here, we deal with the first case. A user can specify failures/repairs using the clause

failure { <failure_pattern> }.

Within curly braces one can write a sequence of the following statements.

i : **downlink**: j : <links>;

It is used to specify faults in links. **i** is an integer label which can be used in a repair statement to refer to a specific **downlink** statement. It means fail **j** links randomly from the <links> set. The absence of the clause <links> means all the links in the system and the absence of **j** means user wants to fail all the links specified by <links>, e.g.,

10:**downlink**:2:;

means fail randomly two links out of all the links.

20:**downlink**::1,2..6 - 1,2..6;

means fail all the links specified. To repair 2 links randomly out of the links failed by the above statement one can use the statement :-

up:2:$20;

To repair 4 links out of some specific failed links the statement is :-

up:4: 2..7 -> 2..7;

To repair all the links, the statement is :-

up:: 2..7 -> 2..7;

To repair all the links failed by the statement having label 20, the statement is:-

up:: $20;

The general format of **up** statement is :-

up: j : <links>; or up: j : $<label>;

If a link is repaired twice, the second repair statement is ignored. The statement

i : **downnode** : j : <nodes>:

can be used to create faulty nodes. Its semantic is just like **downlink** statement. To repair node failures, the same **up** statement can be used. To introduce time delay between failures and repairs, the statement

wait r;

can be used where 'r' is a real number. A particular pattern of failures and repairs can be repeated by enclosing it in a **repeat** statement, e.g.,

```
repeat 100
{ 1:downlink:2::;
   wait 3.2;
   up:1: $1;
   wait 4.0;
   up:: $1;
};
```

will repeat the contents enclosed within { } 100 times. Instead of 100, if there is a *, it means that the **repeat** statement is to be repeated for the entire duration of the

experiment.

3.3. Global Stable Data Declarations

Stable data on a node are the data items which retain their values even if the node fails. When the node is up again, it can access the stable data items. To declare stable data the clause

stable_data { <data>; <data>; ....; <data> }

is used, where <data> consists of

<any_valid_C_declaration>; <any_valid_C_declaration>..... : i : <nodes>

Here the variables defined are replicated **i** times on randomly selected nodes out of the nodes specified by <nodes>., e.g.,

#define N 1000;**inta**, d[N]; float r : 2 : 1..100;

means out of 100 nodes select any two nodes to contain **N**, **a**, **d**, and **r**.

int a, d : : a, b, c;

means variables **a** and **d** are replicated on all the three nodes.

int a : : ;

means **a** is replicated on all the nodes.

In the code for a node, it is possible to check whether a local copy of a stable data item is present or not by using the function

local ("<variable_name>")

It will return -1 if a local copy is not present.

3.4. Global Variables

Global variables are specified using the statement

global_variables { <C_declaration>; <C_declaration>;.... }

where <C_declaration> is any valid C variable declaration. The variables declared here

can be accessed while writing code for any node. These variables are for the purpose of simulation. Three variables **faultynodes**, **faultylinks**, and **messages** are supported by the system. Their values are updated by the system automatically.

3.5. Input and Output of An Experiment

The input to an experiment is expressed by the clause

input { <statements> }

and output by the clause

output { <statements> },

where <statements> is a sequence of the following types of statements.

- Any valid C statement, including sleep statement.
- TERMINATE statement: It starts distributed termination of the experiment, This statement can also be used inside the processes to be executed at nodes.
- **repeat** statement: It is of the form

repeat <integer> { <sequence_of_C_stat> }

Instead of <integer> if a * is present then the loop is repeated for the entire duration of the experiment.

- **send** statement: This statement is described in the next section.

3.6. Specification of Code on Nodes

The code to be executed on different nodes is expressed by the statement

code_on_nodes {<data> <processes> <process_to_node_mapping>

<data_to_node_mapping>}

where <data> consists of variable or/and port declarations which can be accessed by any of the processes on a node. <processes> is the specification of code for different processes and <process_to_node_mapping> describes the mapping of processes onto nodes. <data_to_node_mapping> specifies the placement of data on nodes. A process

can access the standard variables NEIGH and ALLNODES which are arrays containing the id's of the neighbors and all the nodes in the system, respectively. NGH and N represent the number of elements in NEIGH and ALLNODES, respectively.

The specification of <data> is as follows.

decl <data_group_name> (<integer_var>) { <data_declaration> }

where the parameter <integer_var> is optional. It can be used to contain the node identification number at the time of assigning it to nodes. For example,

assign d (i) to i: <nodes>

will create several instances of the variable **d (id)** at the specified nodes where **id** is the identification of the node on which the variable is being created.

The design of the structure of processes was influenced by the work in [1,6,7]. The syntax of a process is as follows.

```
process <name> ( i );
{<any valid c declaration>;
<procedure declaration>;
<guarded procedure declaration>
begin <initial statement> end
}
```

The parameter to the process name is an integer and it is optional. A guarded procedure is just like ordinary procedure except that there exists '<guard> ->' before the keyword **procedure**. A guard may consist of a boolean expression or a **receive** statement. A guard is true if the boolean expression is true or a **send** statement corresponding to the **receive** statement in the guard has been executed somewhere.

When a process is started, it executes its initial statement first. After the initial statement, it waits for a guard to be true. If there are more than one true guards, one of them is randomly selected and the corresponding procedure is executed. When the execution of a guarded procedure is over, it waits for another true guard.

send and **receive** statements exchange data through ports. A port might have a buffer to store messages. A port can receive different kind of messages. A port can be declared using the statement:

port <port_name> **of** <message_type>, <message_type>, .. **buffer** i

The **buffer** clause is optional. If it is not present it means communication through the port is synchronous. **i** is an integer which represents the buffer size. Buffer size is equal to the largest message to be handled by the port multiplied by . A port may be declared within a process or outside a process. In the latter case, all the local processes can read messages from the port. The **send** statement has the form

send [<message name>, <parameter>, .., <parameter>]

to <port_list>:<nodes>:<process_list>

In the <to> clause, if <nodes> is not present, it means, the message is to be sent to all the nodes. Absence of <process_list> means that the port is out side processes. A number of parameters can be enclosed within {} followed by an integer to represent a repetition count. The corresponding parameters in the **receive** statement will be arrays. A parameter in a **send** statement can also be the key word **time**. The corresponding parameter in the **receive** statement will receive the time taken by the message to reach the destination. For example,

send [I_am_ok, { rand(10), rand(200)} 12, **time**] **to** :T:;

means send 25 values associated with the *I_am_ok* message to the port *T* on all the nodes. In the corresponding **receive** statement there will be three parameters. First two parameters will be arrays each of size 12 and the last one will be of type real.

The **receive** statement has the form

receive [<message name>, <parameter>, .., <parameter>] **from**

<port>:<node>:<process_name>

Specification of <node> and <process_name> is optional. If they are not present the **receive** statement will read the first message of the given type. . To check whether a particular type of message has arrived or not the statement is:

check <message_name> **from** <port>:<node>:<process_name>

where the specification of <node> and <process_name> is optional. If the message is

present, this statement will return 0 otherwise -1.

Inside processes one can write **downlinks, downnodes** and **up** statements too. The scope of the labels of **downlinks** and **downnodes** statements will be global, that means, failures created in one process can be repaired in any other process in the system using **up** statement.

Assignment of processes to nodes is done by the statement

assign <process name> (i) **to** *i*: <nodes>.

In this statement, *i* gets the value of the node id in different instances of the same process. If *i* is not present, the format of the **assign** statement will be

assign <process name> to <nodes>.

There can be several **assign** statements.

3.7. Constraints

Constraints can be expressed using the statement:-

constraints { <be>**do**<code>, <be>**do**<code>,..., <be>**do**<code>}

where <be> represents a boolean expression involving global variables and the variables local to nodes. Local variables are expressed using the convention

<nodes>:<variable name>

For example, the expression

1..100: A == 10

means that the value of the variable A on the nodes 1..100 should be equal to 10. When a boolean expression becomes false, the corresponding code will be executed and after that the experiment will be stopped with a message declaring the violation of a constraint.

3.8. Global Initial Statement

It is expressed using the statement :-

global_initial_statement { <statement_list> }

The statement list may include any valid C statement that refers to global variables. When the experiment is started, first the global initial statement is executed.

4. Example: Replication Control Experiment

To illustrate the use of DEX, we present an outline of DEX code of a system similar to SETH [3]. SETH is implemented in C to perform experiments with quorum based replication control algorithms. The code on a node for the replication control experiment is organized as follows.

- A port to receive transactions.
- The process TM (transaction manager) to execute a transaction.
- The process Vote to reply to vote requests from local and remote TM's.
- The process Commit to handle messages related to transaction commitment.
- An initial statement.

The outline of the experiment is as follows.

```
topology
{ nodes : 1..10;
  links: [1..10 => rand (2..4) - 1..10] /* system generates a random network */
}

failure
{
       repeat *
       { 10: downlink: 3 : ;
         20: downnode: 2 : ;
            sleep 0.5;
            up: 1 : $20;
            up: 2 : $10;
            sleep 0.5;
            up: 1 : $20;
            up: 2 : $10;
       }
}

stable_data
{ int a[200] : : ; /* 200 items on all the nodes */
```

```
  int wt[200] : : ; /* weight associated with items */
  int r_quorum, w_quorum;
  int version_no [200]; /* version numbers associated with items */
}

global_variables { int no_of_trans_finished; }

input
{int id;
 id = 1;
 repeat 50 /* generate 50 transactions */
      { send [trans, {rand(2), rand (200)} 5, id] to T: : ;
      /* Send transactions to all the nodes. A transaction consists of 5 read or write operations,
      represented by 1 or 2, respectively. The first call to the function rand generates read or write
      operation and the second one generates item number. id represents the transaction id. */
      sleep 1;
      id++;
      }
 TERMINATE;/* statement to start distributed termination */
}

code_on_nodes
{ decl global_port {port T of trans buffer 20};

 process TM
 { int ops [4], items [4], sites [N], sites1 [N], sites2 [N], id;
      /* ops and items arrays will receive the operations in an transaction. sites, sites1 and sites2 will be
      used to receive the participating sites' id's. First element will always be the number of participating
      sites.*/

      abort_trans(id);
      { print appropriate message, and update statistical data}

      successful_trans (id)
      { print appropriate message, and update statistical data}

      build_R_quorum ( item, sites)
       int sites[];
      {talk to the Vote process at various nodes and collect votes.
       Store the id's of the participating sites in sites. }

      build_W_quorum ( item, sites)
       int sites[];
      {talk to the Vote process at various nodes and collect votes.
       Store the id's of the participating sites in sites.}

      commit ()
      {talk to the process Commit at the participating sites to
       commit a transaction}

 receive [trans, ops, items, id] from T -> procedure process_trans ();
      { int i, failed;
       failed = 0;
       for (i = 0, i < 5, i++)
```

```
        {if ( ops[i]==1) {if (failed=build_R_quorum (item[i], sites1) == 0) break};
        else {if (failed = build_W_quorum (item[i], sites2) == 0)break}
        }
       if (failed <> 0) { if commit() == 0)abort_trans(id);
                              else successful_trans (id)};
       else abort_trans (id)
        }
 }/* end of the process TM */

  process  Vote;
  { contains two guarded procedures, one to reply read vote messages and the
    the other one to reply to write vote messages}

  process Commit
  { handle the message comming to commit a transaction }

 /* Assign port T and different processes to all the nodes */
 assign global_port to 1..10;
 assign TM to 1..10;
 assign Vote to 1..10;
 assign Commit to 1..10;
 } /* end of the code_on_node section */

global_initial_statement
 { int i;
  for (i = 0, i < 200, i++)
  { lock [i] = 0;
    version_no [i] = 1;
  }
 }
```

5. Discussion

We have presented a design of a distributed system experimentation tool called DEX. It provides features for simulating a wide class of distributed systems and uses UNIX processes and communication mechanisms. It is suitable for testing correctness of distributed algorithms through experimentation. Another attractive application of DEX is to study performance of distributed algorithms under failures. However, the data obtained using DEX will not be as accurate as that obtained from a real system. Hence, it

needs careful interpretation and validation. The implementation of DEX is currently underway.

References

[1] Arora, R.K. and N.K. Sharma, Guarded Procedure: A distributed programming concept, Information Processing Letters, 13, 4 and 5, (1981), pp. 199-203.

[2] Bhargava, B. , et.al., RAID distributed database system, Technical Report# CSD-TR-691, Dept. of Computer Sc., Purdue Univ., (Aug. 1987).

[3] Bhargava, B., et.al., SETH: A quorum-based replication database system for experimentation with failures, to appear in Data Engineering'89.

[4] Dasgupta, P., et.al., The Clouds project: Design and implementation of a fault tolerant distributed operating system, Technical Report GIT-ICS-85/29, Georgia Institute of Tech., (Oct. 1985).

[5] Franta, W.R., et.al., Issues and approaches to distributed testbed instrumentation, IEEE Computer, (Oct. 1982), pp. 71-81.

[6] Hoare, C.A.R., Communicating Sequential Process, Comm. of ACM, 21, 8, (Aug. 1978), pp. 666-677.

[7] Hansen, P.B., Distributed processes: A concurrent programming concept, Comm. of ACM, 21, 11, (Nov. 1978), pp. 934-941.

[8] Jenq, B., et.al., A queuing network model for a distributed database testbed system, IEEE Trans. Software Eng., vol. 14, no. 7, (July 1988), pp. 908-921.

[9] Rubinovitz, H.H. and F.J. Maryanski, A software tool for distributed database simulation, TR-88-20, Computer Sc. and Eng. Dept., U. of Connecticut, Storrs, CT 06268.

[10] Spector, A.Z., et.al., The Camelot project, Database Engineering, vol. 9, no. 4, (Dec. 1986).

[11] Wang, R.T. and J.C. Browne, Virtual machine-based simulation of distributed computing and network computing, Perform. Eval. Rev. 10, (1981), pp. 154-156

[12] Chandy, K.M. and J. Mishra, Distributed simulation: A case study in design and verification of distributed programs, SE-5, (Sept. 1979), pp. 440-452.