

2

AD-A201 906

DTIC FILE COPY

RELIABILITY OF A MULTICOMPUTER DISTRIBUTED  
OPERATING SYSTEM

FINAL SCIENTIFIC REPORT

By

Amnon Barak

August 1988

DTIC  
ELECTE  
OCT 18 1988  
S H D

המחלקה למדעי המחשב  
האוניברסיטה העברית בירושלים

DEPARTMENT OF COMPUTER SCIENCE  
THE HEBREW UNIVERSITY OF JERUSALEM  
JERUSALEM, ISRAEL

DISTRIBUTION STATEMENT A

Approved for public release;  
Distribution Unlimited

88 10 18 074

UNCLASSIFIED

ADA201906

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS			
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT UNLIMITED			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION Hebrew University of Jerusalem		6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science Jerusalem, 91904, ISRAEL		7b. ADDRESS (City, State, and ZIP Code)			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION BOARD		8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code) 223/231 Old Marylebone Rd. London NW1 5TH UK		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Reliability of a multicomputer system					
12. PERSONAL AUTHOR(S) Prof. Barak Amnon					
13a. TYPE OF REPORT Final Scientific		13b. TIME COVERED FROM 87-8-1 TO 88-7-31		14. DATE OF REPORT (Year, Month, Day) 88-8-5	15. PAGE COUNT 8
16. SUPPLEMENTARY NOTATION The program manager is Mr. Mark Zonca, RADC <i>probabilistic</i>					
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)			
FIELD	GROUP	SUB-GROUP			
		Multicomputer Operating System			
19. ABSTRACT <p>The project on Reliability of a Multicomputer Distributed Operating System is a three-year research effort in reliability and availability of multicomputer systems with decentralized control. The main objective of this project is to allow a loosely coupled network of computers to cooperate in order to provide a general-purpose non-stop computing facility.</p> <p>The main issues involved with increasing the reliability are replication and multiple (concurrent) activities. This report covers the third research year: August 1, 1987 to July 31, 1988. During this period we investigated four main areas for increasing the reliability. First, we developed a set of algorithms for finding the global average load of a large multicomputer system. The second part is a study of a filing system for a multicomputer with a large number of disks. The third area is a performance study of remote vs. local system calls in the MOS distributed system. The fourth area is an effort to develop mechanisms for supporting massive parallel tasks in a distributed environment. <i>Keywords:</i>  <i>Israeli message passing, (KR)</i></p>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL		22b. TELEPHONE (Include Area Code)		22c. OFFICE SYMBOL	

EOARD-TR-89-02

This report has been reviewed by the EOARD Information Office and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication.

  
PARRIS C. NEAL, Maj. USAF  
Chief, Applied Electronics

  
JAMES G.R. HANSEN, Lt. Col, USAF  
Deputy Commander

# RELIABILITY OF A MULTICOMPUTER DISTRIBUTED OPERATING SYSTEM

## FINAL SCIENTIFIC REPORT

By

Amnon Barak

August 1988

### 1. INTRODUCTION

Research in reliability of a multicomputer distributed operating system is an investigation into the possibility of constructing a highly reliable multicomputer operating system for a cluster of loosely coupled, independent computers that are interconnected by a local area communication network.

The key issue in reliability is redundancy. Through replication at every level of the hardware and software systems, copies of files and processes have a higher probability of surviving partial failures and thus allow non-stop operation of the system. One outcome of this approach is increased availability, since several copies of the replicated object are available.

The main thrust of our current research is to develop a software layer, on top of the operating system, in order to make intelligent system decisions. The areas which we considered for this layer include reliability, recovery from partial network failure, support of atomic transactions, improved communication and the support of parallel tasking.

The implementation of this software layer is based on the availability of MOS, a multicomputer operating system which was developed by the investigator at the Hebrew University during the last four years. The main characteristics of MOS which are applicable to the current research are:

1. **Decentralized Control:** Control decisions are made independently by each processor.
2. **Network transparency:** The network appears to the user as a single machine. Communicating processes need not know if they are on the same processor or on different processors.
3. **Multiprocessing Environment:** Each processor provides a multiprocessing environment.
4. **Logically Identical Processors:** Application tasks may be assigned to any processor.
5. **No Shared Components:** No memory or other devices are shared between the processors.
6. **LAN for Interconnection:** No assumption is made about the topology of the network except that it is connected.
7. **Message Passing:** Communication is achieved by messages.
8. **Load balancing:** MOS supports dynamic process migration for load balancing.
9. **Dynamic Configuration:** Allows a machine to join or leave the network at any time.
10. **Compatibility with the UNIX user's interface:**

During the period covered by this report, Aug. 1, 1987 - July 31, 1988, work has been done in the following areas:

1. Algorithms for the global average load of a multicomputer system: a paper is now being written to be submitted for publication.
2. A filing system for a very large multicomputer configuration: a paper was written and submitted for publication. A summary follows below and copies of the paper are enclosed.
3. Performance of MOS: a thesis is currently being written on this subject. A summary of our results follows below.
4. Support of replicated parallel tasks: a thesis which summerized our results was written and is attached to this report. Also, a paper on this subject was accepted to the European UNIX systems user group conference, to be held in Portugal, October 3-7, 1988.

A brief summary of each of these subjects follows below.

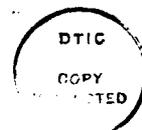
## 2. PROBABILISTIC ALGORITHMS FOR THE AVERAGE LOAD OF A MULTICOMPUTER

We develop probabilistic algorithms for estimating the global average load of a large scale, loosely-coupled multicomputer system. These algorithms are useful for load balancing between the nodes of the multicomputer. The algorithms are based on load messages which are transmitted asynchronously between the nodes. Two methods for routing the messages are suggested. In the first case, each node sends load messages to a randomly selected nodes. In the second method each node sends load messages to a fixed set of prespecified nodes. We show that based on these messages, each node can find an estimate for the global average load of the system, thus it can improve the overall performance by making better scheduling decisions through load balancing.

Consider a multicomputer system having a large number of independent nodes that are interconnected by a local area communication network. Throughout this work, a multicomputer system means a set of autonomous computers (nodes) that communicate with each other by messages. Suppose that the topology of the network allows a direct link between any pair of nodes, as for example in the Ethernet communication network. Suppose that each node maintains in addition to its local load, an estimate for the global average load and possibly load values of other nodes as well as their estimates of the global average load. Assume that this information is transmitted between the nodes in a manner which provides the most up-to-date loads and estimates to each node. The problem is then how to route the load information among the nodes and how each node can determine the global average load of the system based only on the information which is received by that node.

In this work we are interested in algorithms that can be used in a multicomputer system with a large number of nodes. These algorithms must satisfy the following properties. First, we require uniformity; this means that all the nodes use the same algorithm, that there is no central control, that all the nodes use the same unit of time and that each node is responsible to execute the algorithm independently of the other nodes. We also require low communication overhead; therefore we do not allow network broadcasts due to its high overhead, and we allow only short load messages. We also assume that the number of nodes  $n$  is arbitrarily large and that no *a priori* information about the execution time of the arriving processes is available.

One method to achieve load balancing is by finding the average load over all the nodes of the system, then to migrate processes from nodes whose load are above the average into nodes whose load are below the average. An alternative load balancing policy that on the average uses only one message for each node, during each unit of time, was developed in [3]. In this case, load balancing is achieved by reduction of the variance of loads between pairs of nodes of the multicomputer. This policy does not need the average load but requires that each node maintains up-to-date information about the load of a (small) subset of other nodes. One disadvantage of this policy is the communication



Dist		
A-1		
ion For		
RA&I	<input checked="" type="checkbox"/>	
B	<input type="checkbox"/>	
need	<input type="checkbox"/>	
ation		
ation/		
bility Codes		
ail and/or		
Special		

overhead which results from possible suboptimal process migration between pairs of nodes whose respective loads are below or above the global average load.

Due to the fluctuations of the values of the load at the nodes, an algorithm that finds the exact average load is computationally expensive because it requires an instant reading of the load of all the nodes. Alternatively, a suboptimal algorithm can be devised according to the observed state of the system. Such an algorithm attempts to approximate the exact value of the average load with less knowledge, and at a fraction of the cost that is required by an optimal algorithm.

In this work we propose a class of asynchronous algorithms by which each node can find an estimate for the global average load. The main advantages of our algorithms are low communication overhead and simplicity of the implementation, we only require that each node finds its local load, followed by a simple computation of the node's estimate of the global average load and then the transmission of this estimate to a subset of target nodes. We prove that by using this scheme, each node can find an estimate for the global average load by monitoring the values of the received loads and by correlating these values with its local load. As such, our algorithms are suitable for very large multicomputer configurations, and they confirm the observation in [4] that simple algorithms that use very small amounts of state information, can dramatically improve the performance of the system.

We suggest two alternative schemes for routing the load messages. first, we give a probabilistic algorithm in which each node sends messages to randomly selected nodes. This algorithm is also aimed to overcome node failures. Then, we present an algorithm in which each node sends messages to a fixed, prespecified set of nodes. In both cases, we prove that the variance of the estimates for the global average load, reduces during each unit of time and we also give the rate of this reduction.

### 3. PERFORMANCE OF A DISTRIBUTED SYSTEM

this project is concern with performance evaluation of MOS. In any system, performance is a vital issue. For a distributed system, the performance of various mechanisms, e.g. remote procedure calls or a memory mapping mechanism between local and remote machines, determines the feasibility of the distributed system itself and the feasibility of other mechanisms, such as those used to make a system reliable. For this reason, an extensive analysis of the mechanisms at the heart of the MOS system and several MOS application programs was carried out.

#### 3.1. Remote Kernel Procedure Calls

System calls define the basic interface between the user and the operating system. The overhead associated with the remote execution of system calls has a significant influence on the performance of the MOS system. In MOS, system calls can be divided into three groups: system calls with no remote implementation, system calls with scall versions which use the funnel mechanism and system calls with scall versions which do not use the funnel mechanism.

##### 3.1.1. Measurement Technique

The remote system calls overhead was measured by running a set of benchmarking programs. Each of the benchmarks measures the elapsed time required for local and remote execution of a system call that has a remote implementation. Each system call is executed 10,000 times on objects located on the local node and 10,000 times on remote objects. The system calls measured are all part of the standard UNIX interface. The measurements were done when the remote and the local nodes were running in single user mode to reduce interference from uninvolved processes.

##### 3.1.2. Results

The benchmarks determine the slowdown associated with a remote kernel procedure call which is computed as the elapsed time required for the remote execution of the call divided by the elapsed

time required for the local execution. It should be noted that the design of the MOS kernel allows for certain system calls to be executed locally regardless of the current position of the calling process. This type of system call has no overhead, that is a slowdown factor of 1.0. Using the relative frequency of the system calls, the weighted average of the slowdown of system calls was computed to be 2.4. Although comparing these results to measurements done on other distributed systems is of dubious value, the results show the the MOS RPC mechanism is one of the more efficient ones.

### 3.2. Interprocess Communication Mechanisms

In order to utilize the processing power of a distributed system, a distributed application is broken into several subtasks which can run in parallel. In addition to the time required for the computation of the task, each of a distributed application's subtasks must use some form of interprocess communication ( IPC ) mechanism to communicate with the other processes working on a given task. The cost of the IPC is a critical factor in determining the performance of a distributed application.

The IPC mechanisms used in MOS are really a subset of the system call interface described above and showed a similar slowdown factor, the average slowdown for the different mechanisms is around 2.3. The measurements showed the importance of using a distributed IPC mechanism: mechanisms, like the current initial of AT&T System V messages, which store messages on one machine, create a serious I/O bottleneck which hinders the scaling ability of distributed applications. By distributing the message storage structures at random between available machines, the performance of distributed applications which make heavy use of message passing was considerably improved.

### 3.3. Process Migration and Networking

The MOS load balancing algorithms use the dynamic migration mechanism to evenly redistribute processes from loaded to less loaded machines. Distributed applications, simply create several subprocesses with the standard UNIX fork mechanism and the load balancing mechanism sees to the even distribution of the processes throughout the system. This section first describes the performance of the implementation of the funnel mechanism and then the speed of process migration.

#### 3.3.1. The Lower Layer of the Network

In any distributed system, performance is highly dependent on the speed of the physical network and the networking protocols implemented. In MOS, the user has no explicit access to the network. The linker uses the network to implement the following functions: process migration, funnels and remote kernel procedure calls. The speed of the physical network is determined by the hardware used: the Pronet token passing ring has a throughput of 10 Mbit/second (1,250 Kbytes/second ).

#### 3.3.2. The Funnel Mechanism

The funnel mechanism typifies the network related mechanisms. Therefore, analysis of the networking protocols centers on measurements of the data throughput of the funnel mechanism. Local memory to memory copies are ten times as fast as a network read. The throughput over the network was 112 Kbytes/second. The time spent in the upper kernel does not have a significant impact on the throughput, in fact it accounts for only 3 percent of the time spent in the transaction.

One attempt to improve the throughput involved removing the Pronet driver's checksum which is done on every message sent. Although the current network hardware requires that the error checking be done in software, future generations of hardware will be able to do this by themselves. An added advantage of not doing a checksum is a reduction in the number of message queues maintained by the Pronet driver by one. Without the checksum, throughput was increased to 164 Kbytes/second, an increase of 46 percent. At this rate, a two machine transaction utilizes 13.12% of the Pronet's maximum bandwidth.

### 3.3.3. Process Migration

The process migration mechanism is built upon the remote kernel procedure call mechanism and the funnel mechanism. Remote procedure calls are used to request that a new procedure entry is set up at the remote site and to start execution of the migrated process on the remote machine after the migration completes. Funnels are used to transfer the procedure's text, data and stack regions. Not surprisingly, the throughput of process migration is closely tied to the performance of the funnel mechanism: the measured throughput is 114.77 Kbyte /second which is slightly better than that of the transfer rate of the funnel mechanism.

### 3.4. Distributed Applications

The preceding sections of this work measured the slowdown factor associated with various low level mechanisms of MOS. This section measures the speedup factor of distributed applications when run on the MOS system. The speedup factor is computed to be the minimum time the application requires on a one processor system divided by the time required on a many processor system. The optimal speedup of an application is linear, that is an application should have a speedup of  $k$  when run on a  $k$  processor system.

The nature of an application determines what kind of speedup can be obtained. Two types of applications were measured. The first application measured, a parallel version of the UNIX make, does a large amount of disk IO as well as computation. Without modification, each of the 60 subgoals which could run in parallel during the execution of the parallel had some measure of IO with the machine where the application started running. In this case, the speedup obtained with a four machine configuration reached on 40% of the optimal. By distributed the temporary files created by each of these subgoals randomly between the working nodes, the IO bottleneck was eliminated and the speedup obtained reached 60% of the optimal.

Other applications, like the implementation of the Traveling Salesman Problem, are basically CPU bound and obtained speedups that were roughly optimal. The basic performance of the message passage mechanism was changed as described above which added a slight performance improvement.

## 4. A HOLOGRAPHIC FILE SYSTEM FOR A MULTICOMPUTER WITH MANY DISKS

We present a "holographic" file system (HFS) for concurrent data retrieval in a computer system with a large number of disks (although it is probable that most nodes will be diskless). In such a file system it is possible to operate on a file by simultaneously utilizing many (or all) of the disks, since the file system is organized to take advantage of the multiplicity of equipment, rather than limiting access to a single disk for each file, as in most existing file systems. The main advantages of the HFS are a speed-up in data retrieval related to the number of disks, and improved availability by allowing access to parts of a file even when other parts of that file are not accessible.

Future computing needs, such as in the management of large amounts of data, will require unified computing systems consisting of a large number (possibly thousands) of loosely coupled, independent computers (nodes), interconnected by a communication network, without shared memory or shared devices. With so many processors, however, no one processor can "know" the state of the entire system (because of the overhead and possible congestion in trying to keep it up to date), and we shall no longer be willing to shut down the entire system because of the failure of one or a few processors. The "operating system" for such a configuration must be completely distributed, in some reasonable sense, and it must account for processors dropping out of the system and returning at some later time, possibly with obsolete control information and/or obsolete data.

Unfortunately, existing operating systems are unable to handle more than several dozens of nodes, due to the fact that many commonly used mechanisms and internal algorithms lead to congestion when used in a configuration beyond a certain size. One bottleneck in most multicomputer

systems is due to the use of a traditional file-system organization, in which all the records of each file are placed in a single disk. The main drawback of such an organization is that it does not take advantage of the multiplicity of the hardware components by allowing parallel operations on files. Future systems, particularly systems with a large number of disks, must find a way to speed up the access time to files; in particular, by encouraging parallel access to many records simultaneously.

In this work we present an organization for a file system that scatters records of a file into different nodes, in order to allow parallel read/write operations of different records. The advantages of this organization includes (a) the elimination of the restriction that a file system reside in a single partition of a disk, (b) enhanced throughput due to concurrent accessing of the same file system by many processors, (c) the possibility of better disk-drive load balancing, and (d) increases the availability of files and allows operations on files even when portions of the data are inaccessible due to disk or node crashes.

The fundamental assumption that we make is that in future systems many, perhaps most, of the node machines in a large configuration will not have disks. Thus, almost every file-related operation will be performed to or from a remote file server. While there is often concern over the relative cost of remote file access over local access, in Lazowska et al, it is argued that "with a well-designed file server ... the cost of remote file access is reasonable even for substantial numbers of client workstations." In effect, then, the holographic file system may be regarded as a redesign of the traditional file server so that queuing delays and congestion problems are minimized, or avoided altogether.

A technical report with the complete paper is enclosed.

## 5. DISTRIBUTED LIGHT WEIGHT PROCESSES FOR MOS

Distributed systems integrate a set of loosely coupled processors, each with its own local memory, into a single machine environment. In the distributed systems model, various user processes may run concurrently on different machines and possibly communicate to achieve a common goal. This form of concurrency encourages a programming style that uses large grain-size computation blocks. Such *distributed programs* consist of a set of execution entities (called *threads* or *tasks*) that perform considerable amount of work independently and communicate infrequently through messages. Threads are a convenient way of expressing concurrent programs and therefore, many programming languages embody thread-like entities in their syntax, e.g. Occam [IN84a] and Linda [ACG86]. However, the overhead of handling processes by the operating system is costly. For instance, it has been noted that the UNIX processes are *heavy-weight* in that they carry much associated state information. Therefore, operations on them (e.g. context switching) are slow.

Light Weight Processes (LWP) has been suggested by Kepes [Kep85] as a programming tool for supporting cooperating processes on a uniprocessor. In the LWP mechanism suggested by Kepes, a runtime support library provides the coroutine primitives within a single, heavy-weight-process (HWP). Another alternative for supporting LWPs is at the kernel level. On a multiprocessor, the kernel support version has a primary advantage of allowing real parallelism. One of the most recent operating system kernels that support LWPs is Mach [ABG86]. However, none of the kernel or user level LWP mechanisms provide concurrency in distributed environments.

The *Distributed Light Weight Processes* mechanism (DLWP) is a facility for supporting distributed programs in MOS. The goal of the Distributed Light Weight Processes mechanism is to be able to exploit concurrency in a distributed environment. The mechanism is designed to be able to support a variety of application types by supporting processes as a programming tool. It exploits concurrency up to the level available in the system and provides additional, virtual concurrency through time sharing. In this way, it can be used both for efficient utilization of concurrency and for experimenting with large scale concurrent programs.

The DLWP mechanism is implemented immediately above the operating system kernel level, in the form of a user-level runtime library. It extends the uniprocessor Light Weight Processes

mechanism through a new operation, *split*, which adapts the classical Light Weight Processes mechanism for distribution and dynamically disperses the workload among processors. A LWP *pod* within a HWP may *split* to create multiple pods that execute in different HWPs. The MOS dynamic load balancing [BaS85] automatically assigns the HWPs to different machines and provides concurrency.

The partitioning strategy takes into consideration past behavior of the LWPs, in terms of CPU consumption and communication. This profile information is used to reach a partition that splits the load evenly while incurring minimum communication overhead. For this purpose, the profile information is kept in a graph and a heuristic graph partitioning algorithm is employed.

The main features of the mechanism are:

*dynamic configuration*

The mechanism utilizes memory and CPU according to the application's needs. In particular, the binding of threads to CPUs is done dynamically by *splitting* at runtime. This property relies on the ability of the underlying operating system to dynamically assign processes to processors. In the current implementation, the MOS dynamic load balancing facility provides the required flexibility.

*massive parallelism*

The mechanism is designed to provide massive support of parallelism, free from the hardware constraints. The threads may be scheduled in a time-sharing scheme when they exceed the available concurrency in the system. Thus, the only limit on their number is their total consumption of memory.

*applicability*

The mechanism provides services via a set of general purpose interface routines. This set may be easily modified or added to, *e.g.* to allow different communication styles. This allows the mechanism to support a variety of parallel programming languages in providing the underlying runtime support for precompiled programs. For example, an occam support package has already been implemented on top of a single machine LWP system [MaS86] and is now being ported to MOS to use the DLWP mechanism.

*portability*

The DLWP library is written on a UNIX compatible system, in a high level programming language (c).

REFERENCES

- [ABG86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young, Mach: A New Kernel Foundation for UNIX Development, Technical Report, Carnegie Mellon University, August 1986.
- [ACG86] S. Ahuja, N. Carriero and D. Gelemter, Linda and Friends, Computer 19(8), Aug 1986, pp. 26-34.
- [BaS85] A. Barak and A. Shiloh, A Distributed Load Balancing Policy for a Multicomputer, IEEE Software - Practice and Experience 15(9), Sep 1985, pp. 901-913.
- [Kep85] J. Kepes, Lightweight Processes for UNIX Implementation and Application, in *Usenix technical conference proc.*, Usenix Association, Portland, Or, Summer 1985, pp. 299-308.
- [MaS86] D. Malki and G. Shwed, A Unix OCCAM+ System, Technical Report CS-87-7, Hebrew University of Jerusalem, Jerusalem, Il, June 1986.
- [MaS86] D. Malki and G. Shwed, Unix OCCAM+ Compiler, User Manual, Hebrew University of Jerusalem, Jerusalem, Il, Jan 1986.