DESIGN PRINCIPLES OF OPERATING SYSTEMS FOR LARGE SCALE MULTICOMPUTERS

Amnon Barak and Yoram Kornatzky

Department of Computer Science

The Hebrew University of Jerusalem

Jerusalem 91904, Israel

ABSTRACT

Future multicomputer systems are expected to consist of thousands of interconnected computers. To simplify the usage of these systems, multicomputer operating systems must be developed to integrate a cluster of computers into a unified and coherent environment. Using existing multicomputer operating systems is inappropriate as many commonly used techniques get clogged and lead to congestion, once the system is enlarged over a certain size. This paper deals with the various issues involved with designing an operating system for a large scale multicomputer. We identify the difficulties of using existing operating systems in large multicomputer configurations. Then, based on insight gained in the design of several algorithms, we present eight principles which should serve as guidelines for the designer of such systems. These principles include symmetry, customer-server protocols, and partiality. Another component of our approach is the use of randomness in the system's control. We present probabilistic algorithms for information scattering and load estimation. Tolerating node failures, and garbage collection due to node failures, are part of a distributed operating system routine operations. We present a robust algorithm for locating processes, and an efficient algorithm for garbage collection in a large scale system, which are in line with our principles.

1. INTRODUCTION

The use of multiple, loosely-coupled computers to construct a unified computing system offers many potential benefits, including good cost/performance ratio, improved availability, and the ability to dynamically change the system capacity by adding or removing computing elements. Future computing needs will require multicomputers consisting of thousands of nodes. Existing multicomputer operating systems are unable to handle more than several dozens of nodes. This inability is caused by the fact that many commonly used mechanisms and internal algorithms lead to congestion, when used in a configuration beyond a certain size. A feasible type of an operating system for large scale multicomputer configurations is an

integrated system, which presents to the users a view of a single machine environment.

In this paper we study integrated operating systems for a large scale multicomputer configuration. We identify some of the difficulties of existing mechanisms to handle large configurations, then present a set of principles that provide a unified framework to the design of algorithms for such multicomputer operating systems. The practical viability of our framework is demonstrated through a class of algorithms that we have developed.

The multicomputer configuration we envision consists of hundreds or even thousands of loosely coupled, independent computers (nodes) that are interconnected by communication networks, without shared memory or other devices. For simplicity we assume that the network topology is a complete graph, i.e., all nodes are equally distant.

The two extremes of the spectrum of distributed operating systems currently studied, are integrated systems, which present to the user a single logical machine formed from a cluster of multiple computers, and network systems. We discuss here the former. In particular, we consider symmetric architectures, as opposed to asymmetric or functionally-dedicated server systems. The purpose of an integrated operating system is to unify the independent computers into a single machine computing environment, with complete network transparency, and a high degree of resource sharing.

Ideally, the performance of a system should expand linearly with an increase in the number of nodes. No existing operating system has achieved this goal yet. In fact many systems will grind to a halt when required to handle even a hundred nodes.

In this paper we investigate the reasons for these difficulties, and present a unified framework for the design of an operating system for a large multicomputer. This framework is derived from a set of mechanisms we have developed, whose performance in terms of quality, and overhead per node is insensitive to the number of nodes.

The rest of the paper is organized as follows: In the next section, the class of integrated multicomputer operating systems is defined. In section 3 we give an outline of the envisioned hardware architecture of large scale multicomputers. In section 4 we examine the difficulties of existing operating systems in such a configuration. We then present a set of principles for overcoming these difficulties. This is followed by four sections demonstrating, through algorithms we have developed, the viability of these principles. An algorithm for estimating the global average load of a distributed system is given in section 5. A probabilistic algorithm for scattering information is described in section 6. Section 7 presents a robust method for locating migrating processes. In section 8, we discuss a method for garbage collection in a multicomputer system. Our conclusions are given in section 9.

2. TYPES OF MULTICOMPUTER OPERATING SYSTEMS

Although there are many ways to implement an operating system for a multi-computer configuration, two major types emerge: network systems and integrated systems. The two are opposite extremes with respect to the visibility of the network. The former require the user to explicitly consider the network barrier between machines, due to the usage of special remote execution and file access protocols. The latter provide the user with a single machine environment, together with transparent remote file access, and invisible (machine-controlled) choice of execution sites. In contrast to network systems, integrated systems are better suited to support concurrent processing through automated process migration and transparent (across machines) and easy to use IPC mechanisms. Network systems usually lack convenient mechanisms for inter-machine IPC, and execution of concurrent programs, thus making distributed applications rare.

In this paper, we will concern ourselves only with integrated multicomputer operating systems. These can be further divided depending on whether they possess a symmetric or an asymmetric architecture.

In an asymmetric architecture, also called a functionally-dedicated server system, specific machines are assigned service functions while other machines, particularly personal work stations, are used to interact with the user and execute his tasks. In many cases, the individual work station has limited capabilities and often cannot function independently because it lacks some necessary functions performed by the server machines. Due to this structure and because the emphasis is on user control, the network in server systems is to some extent visible and many operations require the user to provide a complete set of machine location parameters. For example, when accessing a remote file, the user is responsible to know the complete path name of the requested file. The Cambridge distributed system [13], and the V-system [5] which supports diskless work stations, are examples of functionally-dedicated server systems. Many aspects of the operation of these systems are controlled in a centralized way. For example, in Charlotte [7] the file system is handled by a single (central) site and the computing processors are diskless. This implies that all the operating system layers related to the file system are controlled in a centralized manner.

The main characteristics of symmetric multicomputer operating systems are complete network transparency and symmetry. Complete network transparency ensures that machine boundaries can be ignored with respect to all user's operations. Symmetry means that each machine is a complete and independent system. It can function independently of the other machines, but may also cooperate with them to provide services to remote processes (residing in other machines). Several levels of integration are possible, the highest of which includes automatic allocation of resources and transparent load balancing, resulting in improved performance. MOS [2] is an example of an integrated system with decentralized control, which supports dynamic process migration [3].

A major advantage of symmetric systems over asymmetric ones, with respect to large scale configurations, is their uniform structure. Consequently, when more machines are added, the user can automatically benefit from the added computing and storage capacity, through the (transparent) resource sharing facilities. In a server system, adding more machines complicates the user's view of the system, as more entities have to be considered when deciding on execution and storage sites. The scaling advantages of systems having identical components were long ago discovered by hardware designers. Furthermore, since in symmetric systems the computers are independent, it is possible to gain a high degree of reliability by exploiting the redundancy of the hardware and by limiting the scope of the effect of a failure of any single component. Using software methods, a high degree of node isolation can be achieved, so that faults will be confined to a node and will not propagate across machine boundaries. In addition to these properties, symmetric systems enable a higher degree of availability and recoverability than asymmetric ones, as replication of items is easier.

In the remainder of this paper, we will consider only symmetric operating systems, though much of our results are also applicable to asymmetric ones.

3. THE ARCHITECTURE OF A LARGE MULTICOMPUTER

In this section we present the relevant hardware characteristics of large multicomputers. The configurations we envision, consist of thousands of processing elements (nodes) interconnected through various communication media.

In contrast to other massively parallel architectures for special applications, i.e., the Connection Machine [10], the multicomputer is intended to be used as a general-purpose computing facility. Thus, most of its nodes are full-fledged, general-purpose computers with local memory, and optional peripherals. In general, nodes have no shared memory, though individual nodes may be tightly-coupled shared memory multiprocessors.

The second component of the architecture is the communication network. Two broad classes of plausible communication facilities can be projected: a flat (or single level) network, and a multi-level one. Within a single machine room it may be possible to build a communication switch for a large system, such that communication delays between any two machines are of the same order of magnitude. We term such a topology a flat or single-level topology. For example, the TF-1 (Teraflop-1) supercomputer, a massively parallel MIMD (32768 node) machine being designed at IBM T.J. Watson Research Center, has a communication switch that allows each pair of nodes to be equally distant (i.e. a complete graph) [11]. Suprenum, another large multicomputer under development by the GMD, uses a hierarchy of communication busses such that the communication delay between each pair of nodes is roughly the same.

For systems dispersed over areas larger than a room, a switch such as the one used in TF-1 is infeasible. In this case a multi-level topology, composed of several interconnected node clusters can be used. A typical example for this topology is a

collection of personal work stations partitioned between several LANs, where the LANs communicate over long-haul lines through attached gateways. In a multi-level topology, communication delays between different pairs of nodes may differ by several orders of magnitude, unless fast inter-cluster lines are used (which is currently possible only when clusters are in close proximity).

Clearly, constructing an integrated operating system on top of a multi-level communication topology is more complicated than the same task for a flat topology [15]. As research into the design of operating systems for large multicomputers has barely begun, we will confine ourselves to multicomputer configurations having a complete graph, with a uniform delay communication topology.

4. THE CHALLENGES OF SCALING

Ideally, the operating system's performance, measured in terms of throughput, response times and availability, should expand linearly with the increase in the number of nodes. No existing operating system has achieved this goal. In fact many systems will grind to a halt when required to handle even a hundred nodes. Multicomputer systems which currently achieve a large size (in the number of nodes), execute only special applications, like electronic mail systems [4], but do not operate as general-purpose computing facilities in a system-wide manner.

Currently, only few papers have addressed the problem of constructing a general purpose, integrated operating system for a large scale multicomputer system [14]. To our knowledge no such system has even been designed. In this section, we first identify the main difficulties associated with scaling, then develop some principles for the design of algorithms for such systems. These principles are intended to serve as a framework for further research into the development of integrated operating systems for large scale multicomputer configurations. We note that the derived principles result from a set of algorithms that we have already developed, to be described in the following sections.

A close examination of the causes for the inability of existing operating systems to function properly in a large configuration, reveals that the dominant cause is the use of inappropriate algorithms and control mechanisms. The primary example are the bottlenecks created in some systems because they are controlled in a centralized manner. For example, in Locus, consistency of replicated files is handled by a single file synchronization site, for each file group [16]. Such a central controller of an otherwise distributed mechanism, becomes a performance bottleneck when more than a few sites are used.

A second prominent example, are communication and information dissemination schemes which use broadcasts. These become completely clogged when more than tens (possibly a hundred) nodes are used. Yet many papers suggest the use of broadcasts to disseminate information, e.g., [12].

These two recurrent themes are both due to an elementary property of computer and communication hardware, that unfortunately is often overlooked when designing

small systems. The functional capacity of any component in the system, be it a processing element, a communication channel or a whole computer, is bounded and does not grow when the system in which it is embedded is enlarged. Consequently, any algorithm that requires from a single component an amount of service proportional to the number of nodes, is destined to become clogged, once the system grows beyond a certain size. This observation becomes our first principle of scalability:

Principle 1: Bounded Resources - *The service demand from any component of the system should be bounded by a constant. This constant is independent of the number of nodes in the system.*

A centralized control scheme is not only an inherent performance bottleneck, but may also create a single point of failure. These two disadvantages stem from the same source, a functional asymmetry of the nodes with respect to the execution of the operation. The alternative is of course symmetry, which becomes our second principle of scalability.

Principle 2: Symmetry - *All nodes should have an equal capability to execute all the system's functions, and each node should be able to execute every such function independently of the other nodes.*

The most important ramification of symmetry is a decentralization of control. This means that as all nodes can have an equal role in the execution of every operation, each node becomes a loci of control. Protocols structured in a symmetric way lead to independence of nodes, as no node relies on others for the execution of its operations. Thus, nodes may join and leave the system at any time without affecting the others. Symmetry has an additional benefit from a software engineering perspective, as only one type of each system module has to be developed.

A naive implementation of a mechanism in a symmetric way may require global coordination of nodes, as no node controls others. This requires from each node a coordination effort that is linear in the size of the system, a violation of principle 1. A more sophisticated form of symmetry that avoids this problem can be derived by observing the relationship between the kernels of two machines that participate in the execution of some joint operation. Suppose that the kernel of one machine requests a remote service by executing a remote procedure call, or equivalently, by sending a request message that activates a server process in another machine. Then, a customer - server relation is formed between the two machines. The duration of this relation is limited, and normally ends when the operation is completed.

The possibility to perform system operations by forming pairs of requesters and servers, implies that a finer notion of symmetry that prevents the need for global coordination could be developed. In this scheme, all operations are short interactions between two parties, a requester and a server. To allow all nodes to possess equal functional capability, all remote operations are performed by this protocol. Thus each node functions as both a requester and a server, depending on whether it

has initiated a request, or is responding to one. Since in a multiprocessing system many of these operations are performed concurrently, each node should be able to function simultaneously as both initiator or server in several operations. This design implies that the resulting control overhead for executing each operation is independent of the number of nodes. We can now identify the next design principle:

Principle 3: A Customer - Server Protocol - *Each customer - server interaction should involve at most two nodes.*

A two-party interaction implemented according to the last principle is a highly efficient mechanism for a large scale configuration, because it incurs the minimal amount of inter-node coordination and communication. An immediate outcome of principle 3 is that no global operation or synchronization should be attempted within the system. Rather, any such system-wide operation should be the collective result of many separate local interactions.

In order to improve performance, decisions made by a node should sometimes be based on information on the state of other nodes. Consider for example the decision to migrate a process from one node to another, in order reduce the workload of the first. These decisions and the associated information gathering operations can be performed as a sequence of two-party interactions, as suggested by the last principle. However, a decision based on information gathered from more than a single node, allows a faster convergence to good performance, e.g. balanced load. On one hand, it might seem that decisions based on information on the state of all nodes, achieve much better performance (e.g., response time in our example) than the case where a decision is based on information from only a subset of the other nodes. This demands a large portion of the available resources (i.e., processing, communication bandwidth, etc.). However, there is evidence that even for relatively small systems, usage of partial information achieves performance which is as good as that obtained using complete information [9]. The salient feature of this example is that each node decision making, should be based on information obtained from a limited subset of other nodes. This idea is a further generalization of the principle of symmetry, in order to make it compatible with principle 1. Its applicability makes it our next principle:

Principle 4: Partiality - *Every decision should be based on information from a bounded subset of the other nodes.*

In order to implement the partiality principle each node must choose a bounded subset of other nodes to interact with. As all nodes look equal, this requires each node to break the symmetry between the other nodes. Breaking the symmetry can be done in a variety of ways, the simplest of which is a static partitioning into subsets. This however, has two disadvantages, the more intuitive one being its inherent unreliability. A more delicate disadvantage is that dependence on a fixed subset may lead to performance deterioration due to the unpredictability of users resource

requirements. For example, in the load balancing case, since the length of process execution times is unknown beforehand, any static partitioning of nodes into resource sharing groups, may lead to uneven performance.

From these remarks it appears that a flexible scheme for breaking the symmetry is required. A natural way for its implementation is through the use of randomness in choosing the subset of the other nodes. We thus arrive to our next principle:

Principle 5: Symmetry Breaking Through Randomness - *The set of nodes with which a node interacts is chosen at random.*

Actually, randomness is not enough, as a random choice made only once has all the disadvantages of a single, fixed choice. A limitation of the duration of interactions should supplement the random choice. This will be our next principle:

Principle 6: Limited Duration of Interaction - *Any interaction that involves more than one node is limited to the duration of a single system operation.*

The last principle should apply also to the execution of service operations in accordance with principle 3. For example, when allocating objects to remote processes, some mechanism must ensure that this allocation is terminated when the remote process dies. One such mechanism is described in section 8.

To illustrate that one can indeed implement algorithms for collecting information which embody all the principles discussed so far, let us describe the algorithm for disseminating information, to be further analyzed in section 6. Assume that we would like each node to reach its decisions (e.g. on process migration) by considering the current state of r other nodes. The following symmetric algorithm, which is also robust to failures can be used: Each node starts by measuring its own state. Then, every unit of time each node transmit all the state measurements it has collected so far, to some randomly chosen node. As shown in section 6, after $O(\log r)$ time units, all nodes possess an expected number of r different measurements. Note that the information is dispersed through routine transmission of messages by each node. There is no usage of polling, and hence the algorithm does not use timeouts, with their associated delays. The routine transmission of messages is an indirect alternative for using timeouts for limiting the waiting period for information from any single node, which may have failed. Thus, it is an embodiment of principle 6.

We have briefly remarked above on the importance for symmetry breaking through randomness in optimizing performance. Intuitively, it is clear that a random choice of the subset of nodes with which one interacts in an effort to share resources is better than interaction with a fixed subset. However, random choice has another important role. It breaks the symmetry between the nodes requesting an interaction in a way which avoids congestion. This simply follows from the fact that the probability of many nodes making a request to the same subset of nodes is very low. For example, the hashing method for allocating process homes, described in

section 7, breaks the symmetry between processes by uniformly distributing homes among all sites.

Naturally, we desire to limit as much as possible the subset of nodes involved in an interaction. However, we are sometimes forced to introduce some redundancy if we desire to increase the degree of fault-tolerance. Redundancy is perfectly compatible with partiality as long as the method of replication requires a fixed amount of resources, that does not depend on the size of the system. This allows for an implementation, whose probability of failure is independent of the system's size. The usefulness of the last idea merits its becoming our next principle:

Principle 7: Invariant Replication Degree - *The replication degree of any object, information or activity should be invariant to the number of nodes in the system.*

Another aspect of scaling is the increased complexity of handling the more varied failure modes of a large system. Simply stated, a large system can have many more types of failures than a small system. While no easy solution is known to these problems, one can limit their scope by properly structuring the system's kernel, e.g., as loosely coupled modules with minimal and well defined inter-module interfaces, and by using hierarchical kernel organization with high degree of information hiding. Another measure for limiting the scope of failures is isolation. Each machine should be suspicious to any incoming remote request and should validate it before servicing. A simple method to implement isolation is used in MOS through the use of a dynamically generated password for each remote interaction. For example, when a process running in one machine opens a file in another machine, the later assigns a password to this file entry. Then any operation on the file is validated against this password.

Carried to the extreme, isolation, information hiding and independence may lead to complete hostility between the machines. This however opposes the goal of an integrated system to enlarge the amount of resources available to the user along with the increase in the configuration size. An important principle which should deeply be rooted in all functions of the kernel is cooperation. This means that once a machine is being approached by another machine, then after validating the request, the later will treat the remote request as it would treat a local one. The above observations can be summarized as our last principle:

Principle 8: Isolation, Cooperation and Information Hiding.

One might wonder whether our principles enable the design of all aspects of an integrated system. No conclusive answer exists in this stage of our research. For example, we already know of one case where it might seem necessary to violate one of our rules. This apparent violation is however misleading. Suppose that it is necessary to find the value of some global average quantity of the system, e.g., the global average workload. Computing such a quantity might appear to require interaction with all the nodes, thus violating the principle of partiality. It turn out that a

probabilistic algorithm which obeys all our principles can be devised. This algorithm is based on having each node send every time unit its estimate for the average to a randomly selected node. Thus, by collecting and locally averaging values, each node converges to the actual value of the global quantity without violating the principle of partiality. More about this algorithm in the next section.

In the next four sections we discuss in detail four algorithms demonstrating the usefulness of our principles. Some of their features, stemming from our principles, which were not mentioned so far are:

1. No global static knowledge is required of the nodes in order to execute the algorithm, except N , the maximal possible number of nodes.
2. The length of the messages is $O(\log N)$ bits. (Note that the address part of each message already requires $\log N$ bits).
3. The amount of information (memory) kept by each node, as required by the algorithm, is $O(\log N)$.
4. Nodes are not required to be synchronized. To simplify the presentation, we require nodes to use the same unit of time t . However, all the algorithms work under the weaker assumption that time units may differ by a bounded amount.
5. Information is gathered through the receipt of messages routinely sent by nodes, without using polling.
6. Nodes may join and leave the system at any time.

5. A PROBABILISTIC ALGORITHM FOR THE GLOBAL AVERAGE LOAD

In this section we present an asynchronous algorithm by which each node can find an estimate for the global average load. Load balancing is then achieved by migrating processes from nodes whose load is above average, into nodes whose load is below average. We note that this algorithm can be used to find the average of any global additive quantity.

Consider a multicomputer having N independent nodes, where N is a large integer. Suppose that the topology of the network allows direct message exchange between any pair of nodes, either directly or via gateways. Assume that each node maintains a reading of its own load (averaged over a unit of time t , and divided by the processor speed), and an estimate for the global average load.

Let T be the (absolute) time, and let t be a unit of time increment. Note: the nodes are not synchronized, but during the time interval $(T, T + t)$, each node performs the algorithm exactly once, as they use the same time unit t . Let $L_i(T)$ be the load of node i at time T , $i = 1, 2, \dots, N$. Let $L(T)$ be the global average load of the system at time T . Then:

$$L(T) = \frac{1}{N} \sum_{i=1}^N L_i(T).$$

Let $A_i(T)$ be the estimate for $L(T)$ by node i , $1 \leq i \leq N$, at time T . Let $A(T)$ and $V(T)$ be the average and variance of $A_i(T)$, respectively. The following algorithm is executed by each node to update its estimate of the global average load.

ALGORITHM 1:

Step 1: Find the new local load $L_i(T + t)$.

Step 2: Suppose that node i receives m estimates, $A_{i_j}(T + \tau_j)$, for $0 < \tau_j \leq t$, $j = 1, 2, \dots, m$, from nodes $i_1, i_2, \dots, i_m$, respectively, which have selected node i during its last unit of time. Then its new estimate is:

$$A_i(T + t) = \frac{A_i(T) + A_{i_1}(T + \tau_1) + \dots + A_{i_m}(T + \tau_m)}{m + 1} + L_i(T + t) - L_i(T).$$

Step 3: Choose a random integer l , $1 \leq l \leq N$, and send $A_i(T + t)$ to node l .

Step 4: After t seconds, return to step 1.

Note that the algorithm is executed asynchronously and that each node takes into account all the load messages that were received during the last time interval t , regardless of their order of arrival.

Let $p_0, p_1, \dots, p_N$ be the probabilities that a node gets $0, 1, \dots, N$ messages respectively during a given unit of time t . Let $E(X)$ denote the expected value of X . Then the following theorems are proved in [1].

Theorem 1: If $A(T) = L(T)$, then $E(A(T + t)) = L(T + t)$.

Let $\vec{A}$ and $\vec{L}$ be vectors with elements A_i and L_i respectively, $i = 1, \dots, N$. Let $V_L(T) = V(\vec{L}(T + t) - \vec{L}(T))$. Assume that $\vec{L}(T + t) - \vec{L}(T)$ is independent of $\vec{A}(T)$. Theorem 2 gives a measure for the effectiveness of Algorithm 1.

Theorem 2: The expected variance due to the random routing of the load messages is:

$$E(V(T + t)) = \sum_{k=0}^N \frac{p_k}{k+1} V(T) + V_L(T).$$

Theorem 3: The random routing in Algorithm 1 implies:

$$E(V(T + t)) \approx (1 - 1/e) V(T) + V_L(T).$$

As a result, it is further proved in [1] that for large values of N , and under stable conditions i.e., the $L_i(T)$ for $i = 1, \dots, N$, remain constant, the load estimates converge to the global average load and that the variance of the estimates of the global average load is reduced by a factor of $(1 - 1/e) \approx 0.632$ during each unit of time.

6. A PROBABILISTIC ALGORITHM FOR INFORMATION SCATTERING

In this section we present an algorithm allowing each node to collect information from a subset of r out of the N nodes, in approximately $1.7095 \log_2 r$ steps.

A probabilistic algorithm for scattering information in a multicomputer system was presented in [8]. The algorithm's goal is to scatter to all nodes information initially stored in a single node. Assume that all nodes use the same unit of time, but that they are not synchronized. During every unit of time, each node selects another node at random and sends it a message (color). Initially, one node is colored. Each node that has the color includes it in its future messages. However, since there is no synchronization between the nodes, we assume that each node sends the color only if the color was available to the node at the beginning of the current unit of time. Thus, a worst case analysis is used.

Denote by p_j the fraction of the nodes which are colored at the beginning of the j -th step. The following theorem relates successive values of p_j .

Theorem 4 [8]: Assume that at the beginning of an iteration there are $p_j N$ colored nodes, $0 \leq p_j \leq 1$. Then the expected number of colored nodes at the beginning of the next iteration is $p_{j+1} N$, where:

$$p_{j+1} = 1 - e^{-p_j} (1 - p_j).$$

The algorithm is completely symmetrical, and hence is insensitive to the identity of the initial source of information. It uses random routing of messages in order to avoid blocking of information propagation due to failed nodes. This robustness results from obeying the principles of symmetry breaking through randomness, and limited duration of interaction.

6.1. COLLECTING INFORMATION IN A NODE

To collect the r information items in each node, we apply the above algorithm in parallel to every color, using multi-colored messages. This results in N overlaid activations of the algorithm. Note that all these parallel activations use the same messages. Hence, the number of messages and iterations is identical to those used by the above described algorithm.

Since each node is expected to collect only r different colors, where r is usually smaller than N , there is no need to run the above algorithm until every color has spread to all the nodes. The following theorem estimates the number of steps required to collect r different colors in each node.

Theorem 5: If after some number of iterations, for every color, the expected number of nodes having that color is r , then the expected number of colors in each of the N nodes is r .

Proof: By the assumption, for every one of the N colors, the expected number of nodes having that color is r . Due to the symmetry of the algorithm, each node is expected to have an equal number of colors. Hence, the expected number of colors in each node:

$$\frac{\text{expected number of copies of every color} \times \text{number of colors}}{\text{number of nodes}} = \frac{r N}{N} = r \quad \square$$

From Theorem 5, it follows that the algorithm is terminated when for every color, the expected number of nodes having that color is r . Consider now a particular color. Assume that $r \leq \alpha N$, for $0 < \alpha < 1$. Then we estimate how many iterations S are required to get an expected number of r colored nodes (in that color).

Theorem 6: The expected number of iterations, S , to get an expected number of r colored nodes (in a particular color), is:

$$S \approx \frac{1}{\log_2 (2 - \alpha)} \log_2 r .$$

Proof: From Theorem 4, if at the beginning of the j -th iteration the fraction of colored nodes is p_j , then the expected fraction of colored nodes at the beginning of the next iteration is:

$$p_{j+1} = p_j(1 + (p_j - 1)e^{-p_j}) / p_j .$$

From the Taylor series for e^{-p_j} , because $0 \leq p_j \leq \alpha < 1$, we obtain that $e^{-p_j} \geq 1 - p_j$. Substituting in the expression for p_{j+1} above, we get:

$$p_{j+1} \geq p_j (2 - p_j) \geq (2 - \alpha) p_j .$$

Thus, as $p_0 = 1/N$, and $p_S = r/N$, repeated usage of the last inequality yields

$$S \approx \frac{1}{\log_2 (2 - \alpha)} \log_2 r \quad \square$$

Corollary: For $\alpha = 0.5$, $S \approx 1.7095 \log_2 r$.

7. A ROBUST ALGORITHM FOR LOCATING PROCESSES

In an integrated system, a user's process may initially be assigned to one node, and later, as a result of load balancing considerations, be migrated to another node. In order to support IPC and to preserve the user's interface, the system must maintain up-to-date information about the location of all processes. A simple scheme to maintain such information is to establish a "home" site for each process, say in the node where it was created, and to continuously maintain in this location information about its whereabouts.

Suppose that a node which contains the home of a process is crashed, losing as a result all information about that process. Despite the crash, if the process has migrated to another site, then it may not be affected, and normally there is no reason to stop its execution. Such a surviving process is called an orphan process. The existence of an orphan process has several undesirable implications. First, if an orphan process cannot be located, then this may lead to a violation of the user's interface, since the user may lose control over his processes. Similarly, IPC to an orphan process can not be performed despite the fact that this process exists and may communicate with other processes.

7.1. HOW TO PREVENT PROCESSES FROM BECOMING ORPHANS

We now present a robust algorithm for locating active processes. As suggested previously, the (fixed) home site scheme offers a simple solution to the process detection problem but it does not overcome crashes of the home site node. Despite this, the use of a home site seems to be fundamental to large systems. To overcome the single point of failure, we suggest the use of multiple homes in fixed set of nodes. It is interesting to find out if an efficient algorithm that does not use a fixed set of sites is possible.

Consider an N node multicomputer system. Let P denote a globally unique process number, obtained by concatenating the creation site number with a local unique process number. Note that each node is responsible to assign unique numbers to its processes.

Let $m > 1$ be an integer which specifies the number of homes for each process. Then m depends on the hardware reliability and the required degree of fault-tolerance, and does not depend on the size of the system. Note that if the probability of a single site failure is q , then the probability of failure to locate a process is q^m .

The information kept in each home includes the process number, its current location, and the (local) time of its last update. Let $H(P, m, N)$ be a hash function which maps the process number P to a set of $m - 1$ distinct integers in the interval $[1, N]$. Assume that a universal hash function is used, i.e., all the nodes use the same hash function. Then the following algorithm can be used by the operating system of each node, on behalf of each local process, to establish and maintain communication between the process and its homes.

ALGORITHM 2:

- Step 1:** *When creating a new process P , establish a home in the node where the process is created. Then send $m - 1$ update messages to nodes whose numbers are given by $H(P, m, N)$.*
- Step 2:** *Every unit of time, each process sends an update message, with its present location, to all its homes.*
- Step 3:** *When a node receives an update message, then if a home exists for the sending process, it registers the location of the process and the (local) time of the update. Otherwise, the node validates (by using the process number and the hash function) if it should have a home for that process. If true, then a home is created and updated, otherwise the message is ignored.*
- Step 4:** *If a home entry is not called by its process for several (a parameter) consecutive units of time, then it is destroyed.*
- Step 5:** *When a process is terminated, it removes its homes.*

We note that step 1 of the algorithm guarantees that at least one home is created for each new process. The creation of the remaining homes depends on the status of the specific set of nodes determined by the hash function. Also note that the number of messages which are sent each unit of time is linear in the total number of processes in the system, but is independent of the size of the system. This is a benefit obtained from designing a redundancy mechanism which obeys the principle of the invariance of replication degree.

Given a process number P , the following algorithm finds its location.

ALGORITHM 3:

- Step 1:** *Send an inquiry message to the creation site of P . If no answer is received, then using the hash function $H(P, m, N)$ send inquiry messages to all the other homes of P .*
- Step 2:** *If all the the inquiry messages fail, then wait one unit of time and return to step 1. After several (a parameter) retries, if the process is not found then respond negatively to the calling process.*
- Step 3:** *When a node receives a request for information about a location of a process, and the information is available, then the node provides the information to the requesting process.*
- Step 4:** *When a node receives an inquiry message about a process and it does not have a home entry for that process, then it responds negatively.*

Note that a node is assumed crashed if it does not respond within a given period of time. Therefore, the unit of time used should be greater than the time-out period.

8. GARBAGE COLLECTION

In an integrated system, processes executing in one site may be allocated resources or objects in other sites. Examples for such resources and objects are: files, devices, servers, and entries in system tables. As sites and communication links are inherently unreliable, failures may disconnect processes from such remote objects. After such a failure, these objects are futilely allocated, without actually being used, thereby wasting their site's resources. Essentially, they have become garbage in the same sense that inaccessible objects allocated from a heap storage are termed garbage [6]. As crashed sites are seldom able to notify other sites of their failure, these resources have to be actively detected and reclaimed for reuse.

Garbage collection due to failures is much more complicated than the same operation in the context of heap memory. Intuitively, this results from the fact that a remote process which is slow in accessing a resource (perhaps due to a lost message), is indistinguishable from a process in a crashed site. This problem is similar to the one arising when detecting lost messages in communication protocols through the usage of timeout. Thus, declaring an object as garbage due to failures is inherently bound to err. On the other hand, by its very nature, garbage collection is an irreversible operation. Once done, a process is no longer allowed to access the reclaimed object. Any practical scheme for garbage collection must nevertheless be performed within a finite period. Hence, it can only strive to lower the probability for error, but can never achieve complete certainty.

The following garbage collection scheme is implemented in MOS [2]. When an object is allocated, a timer is attached to it. Processes are responsible for resetting the timer of the objects they use. This is done whenever an object is accessed, or when it receives a special *keep alive* message. A garbage collection process periodically scans all the objects, and releases those whose timers have expired.

The timers are implemented by tagging each object with a unique creation time, and using a fixed upper limit on objects lifetimes. Resetting the timer is done by resetting the tag. Since the probability of a single machine failure is low, the garbage collection algorithm is run infrequently.

The special *keep alive* messages are periodically transmitted by each site to all accessed objects. These messages are required to reduce the probability of erroneously reclaiming objects which are still needed. Note that the garbage collection method is a costumer-server protocol.

8.1. EVALUATING THE GARBAGE COLLECTION PERFORMANCE

Assume all sites use the same unit of time, but are not synchronized. Define the following parameters:

$1/K$ – The frequency of the sending the *keep alive* messages.

T – The maximal lifetime allowed for an object.

$1/G$ – The execution frequency of the garbage collection process.

For example, the parameters used in MOS are: $K = 1$ minute; $T = 2$ minutes, and $G = 10$ minutes.

The fact that the garbage collection scheme is a two-party protocol simplifies its evaluation, as its performance is independent of the number of nodes. Each process-object pair is treated separately, with respect to *keep alive* messages transmission, timer resetting, and object releasing decisions. Consequently, it is sufficient to consider the case of one process accessing a single object.

In the analysis, we ignore link failures, and consider only site failures. We assume that sites inter-failure time is exponentially distributed with mean λ . The time between successive accesses to an allocated object is assumed to be exponentially distributed with mean μ . For a random variable X , let $E[X]$ denote its expected value.

8.1.1. Computation and Communication Overhead

The communication overhead, including interrupt handling, buffer management, timer resetting, etc., is due to the special *keep alive* messages. For a single process-object pair, it is $1/K$ message handling operations per unit of time, for each of the two sites involved. The computation overhead of the object's site due to the garbage collection process, is $1/G$ operations per unit of time.

8.1.2. Probability of Incorrect Release

Normally the period T allocated to an object is longer than the period K . Due to network delays or local load, a *keep alive* message to an object may be delayed, beyond the expiration time of its current lifetime period. If garbage collection is performed during this "unsafe" period, then the object is erroneously released. To compute the probability for this release, we first compute the length of the unsafe period.

Assume that the object receives the *keep alive* messages with frequency $1/k$, where normally, $T > k \geq K$. However, due to the above delays, we are interested in the case where $k > T$. Note that the unsafe period is shortened whenever the object is accessed. This may occur either before or after the timer has expired. Let w be the expected number of accesses during the unsafe period. As the time between successive accesses is exponentially distributed with mean μ , we get that $w = (k - T)/\mu$. Consequently, the length of the unsafe period U , is:

$$\begin{aligned}
 U &= k - T - w \times (Pr[\text{access before timer expired}] \times E[\text{time between accesses}] \\
 &\quad + Pr[\text{access after timer has expired}] \times T) \\
 &= k - T + \frac{k - T}{\mu} [(1 - e^{-\mu T}) \mu + e^{-\mu T} T].
 \end{aligned}$$

Because sites are not synchronized, garbage collection is equally probable to occur in any of the unsafe periods. Since its frequency is $1/G$, the probability for erroneous release is U/G .

8.1.3. Expected Delay in Collecting Garbage

Due to the memoryless property of the exponential distribution of inter-failure times, the expected delay in reclaiming an object from the time it became garbage (when the remote site crashed), until it is released is:

$$\begin{aligned} E[G - R \mid R < G] &= \frac{1}{F_R(G)} \int_0^G (G - r) f_R(r) dr \\ &= \lambda + G - \frac{Ge^{-\lambda G}}{1 - e^{-\lambda G}}, \end{aligned}$$

where R is the inter-failure time, and $F_R(\cdot)$, $f_R(\cdot)$ are its probability distribution and density functions, respectively.

9. CONCLUSIONS

In this paper we considered the design of an operating system for a large multi-computer configuration. We identified some of the difficulties associated with using existing operating systems, and then defined a set of principles for constructing algorithms for large scale multicomputers. These principles include symmetry, customer-server protocols, partiality, and randomness.

To demonstrate the effectiveness of our principles, we have presented several probabilistic algorithms for collecting information on the state of the system, and for management and control of resources. The interesting properties of these algorithms are their completely decentralized nature, their demand for only partial information about the global state of the system, as well as limited degree of node coordination. The performance of the algorithms in terms of the quality of results, and overhead per node is insensitive to size of the system,

Most of the the algorithms that we presented have been implemented in MOS, a multicomputer operating system running at the Hebrew University of Jerusalem.

This work is part of an effort to develop new concepts and algorithms on which to base the design of operating systems for large scale multicomputer systems. Further research is needed in the areas of internal kernel structures and algorithms, and the organization of the file system. We believe that special emphasis should be given to designing probabilistic algorithms for inter-node interactions. Performance evaluation of such systems and a debugging methodology for their development, are other areas which must be investigated.

REFERENCES

- [1] Barak, A. and Drezner, Z., "Distributed Algorithm for the Average Load of a Multicomputer," Computing Research Laboratory, CTL-TR-17-84, The University of Michigan, Ann Arbor, Michigan, March 1984.
- [2] Barak, A. and Litman, A., "MOS: A Multicomputer Distributed Operating System," Software Practice & Experience, 15, 8, 725-737, Aug. 1985.
- [3] Barak, A. and Shiloh, A., "A Distributed Load Balancing Policy for a Multicomputer", Software Practice & Experience, 15, 9, 901-913, Sept. 1985.
- [4] Birrell, A.D., Levin, R., Needham, R.M. and Schroeder, M.D., "Grapevine: An Exercise in Distributed Computing," CACM, 25, 4, 260-274, April 1982.
- [5] Cheriton, D.R. and Zwaenepoel, W., "The Distributed V Kernel and Its Performance for Diskless Workstations," Proc. of the Ninth Symp. on Operating System Principles, 17, 5, 129-140, Oct. 1983.
- [6] Cohen, J., "Garbage Collection of Linked Data Structures", ACM Computing Surveys, 13, 3, 341-367, Sep. 1981.
- [7] DeWitt, D.J., Finkel, R. and Solomon, M., "The Crystal Multicomputer: Design and Implementation Experience", IEEE Trans. on Software Engineering, SE-13, 8, 953-967, Aug. 1987.
- [8] Drezner, Z. and Barak, A., "A Asynchronous Algorithm for Scattering Information Between the Active Nodes of a Multicomputer System", J. of Parallel and Distributed Computing, 3, 3, 344-351, Sep. 1986.
- [9] Eager, D.L., Lazowska, E.D. and Zahorjan, J., "Adaptive Load Sharing in Homogeneous Distributed Systems", IEEE Trans. on Software Engineering, SE-12, 5, 662-675, May 1986.
- [10] Hillis, D., "The Connection Machine", MIT Press, Cambridge, MA, 1985.
- [11] Hochschild, P., Denneau, M. and Shichman, G., personal communication, 1987.
- [12] Krueger, P. and Finkel, R.A., "An Adaptive Load Balancing Algorithm for a Multicomputer", Technical Report 539, Department of Computer Science, University of Wisconsin-Madison, Madison, Wisconsin, April 1984.

- [13] Needham, R.M. and Herbert, A.J., "The Cambridge Distributed Computing System", Addison-Wesley Pub. Co., 1982.
- [14] Schroeder M.D. Birrell A.D. and Needham R.M., "Experience with Grapevine: The Growth of a Distributed System", ACM Trans. on Computer Systems, 2, 1, 3-23, Feb. 1984.
- [15] Sheltzer, A. and Popek, G., "Internet Locus: Extending Transparency to an Internet Environment" IEEE Trans. on Software Engineering, SE-12, 11, 1067-1075, Nov. 1986.
- [16] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G., "The Locus Distributed Operating System", Proc. the Ninth Symp. on Operating Systems Principles, 17, 5, 49-70, Oct. 1983.