Implementation of an efficient load balancing strategy for a local computer system

Katherine M Baumgartner, Ralph M Kling* and Benjamin W Wah*

Load balancing (or global allocation) has been shown to be effective in reducing the average response time of jobs in local computer systems. This paper describes the development and implementation of the GAMMON (Global Allocation from Maximum to Minimum in cONstant time) load-balancing strategy in an existing networked computer system. GAMMON uses the available broadcast capability of multiaccess networks to implement an efficient search technique for finding the extremum of a set of numbers, which reflect the workloads in the computers. The search technique has average overhead which is independent of the number of participating stations. A description is given of the development of a window search protocol, the development of a load-balancing strategy using the window search, and a practical realization of the GAMMON strategy on an existing network of Sun workstations. The transition from the theoretical concept to practical issues encountered during the actual implementation is described in detail. Emphasis was placed on usability, reliability and performance which was measured on two different networks.

Keywords: broadcast datagrams, distributed search, Internet protocol, load balancing, Unix interface, transparent operation

The decreasing cost, the growth in technology, and the diversification of applications have caused computer systems to evolve from being centralized to being distributed. A distributed computer system (DCS) may possess a large number of general and special-purpose autonomous processing units interconnected by a network.

Load balancing uses communication facilities to support remote job execution in a user transparent fashion to improve resource utilization and reduce response time. A decision to load balance is made if the job is likely to be finished sooner when executed remotely than when executed locally. Load balancing is beneficial, since a job will almost always be waiting for service at one processor while another processor is idle in a DCS with ten or more processors[13].

Load-balancing decisions can be made in a centralized or a distributed manner. A centralized decision implies that status information is collected, and decisions to load balance are made at one location. Theoretical studies on centralized load balancing have been made by Chow and Kohler[4] and Ni and Hwang[15]. The disadvantage of centralized scheduling is the overhead of collecting processor and job status information. If this overhead is large, scheduling decisions are frequently based on inaccurate and outdated information. In contrast, a distributed load-balancing scheme does not limit the scheduling intelligence to one processor. It avoids the bottleneck of collecting status information and jobs at a single site and allows the scheduler to react quickly to dynamic changes in the system state.

Load balancing can also be classified as state-dependent or probabilistic[4]. A decision based on the current state of the system is called state-dependent. A decision is probabilistic if an arriving job is dispatched to the processors according to a set of branching probabilities that are collected from previous experience or are based on system characteristics.

Digital Equipment Corporation, 146 Main Street ML03-5/T37, Maynard, MA 01754, USA
* Computer Systems Group, University of Illinois at Urbana-Champaign, CSL, 1101 W Springfield Avenue, Urbana, IL 61801, USA

 0267 6192/89/040216-11 $03.00

Other research on load balancing include studies characterizing state-dependent load balancing, determining appropriate state information, and proposing efficient algorithms[3,6,11,13,21] and topology-dependent strategies[5,8–10,18,19].

Distributed, state-dependent load balancing is implemented on the Purdue Engineering Computer Network, which is a system of computers connected by a hybrid of Ethernet and point-to-point links[7]. Each processor polls other processors for status information about their loads, decides which machine has the lowest load, and sends the job for remote processing if the turnaround time is shorter.

Some results of these previous studies are as follows.

- A network with load balancing performs better than a network without load balancing.
- State-dependent load-balancing strategies result in better performance than probabilistic strategies, but the overhead associated with implementing them is higher.
- Probabilistic strategies are sometimes insensitive to dynamic changes in system load and may result in suboptimal performance.
- Load-balancing decisions considering the state of the source only do not have the potential for performance improvement that decisions considering the state of the server do[21].
- Extensive state information is not needed for effective load balancing and can be detrimental to system performance[6].
- Status information used in a state-dependent decision must be readily available. Decisions based on outdated or inaccurate status information could degrade performance.
- Load balancing increases network load which may impede message transmissions.

An efficient load-balancing strategy will result in a minimum in response time and send a minimum of state information across the network. Since the common bus only allows one transmission at a time, one such strategy is to send a job from the processor with the maximum load to the processor with the minimum load. The overhead of sending these jobs and identifying the participating stations should be kept low. This paper proposes a strategy for load balancing that can be implemented on existing networks. The strategy uses an efficient technique to identify the minimally and the maximally loaded processors with constant average overhead. The strategy is called GAMMON: Global Allocation from Maximum to Minimum in cONstant time[2].

The organization of this paper is as follows. The subsequent section gives an overview of GAMMON, followed by a section presenting a window-broadcast protocol for distributed extremum search, describing its development, and demonstrating its feasibility. The next section describes the transition from the theoretical concept to a practical implementation. Subsequently, implementation details are discussed. Performance results are presented in the penultimate section, followed by conclusions.

GLOBAL SCHEDULING STRATEGY

In this section, an overview of the scheduling problem on a broadcast bus is presented. Figure 1 shows a diagram of the system under consideration. There are multiple identical processors connected by a broadcast bus. Each processor can have arrivals external to the system or from the bus. Jobs are modelled as independent tasks. If jobs are migrated to a processor across the bus, the results must be returned to the originating processor when execution is completed. Moreover, the queue at each processor is finite; only a limited number of jobs may be waiting for execution.

Development of the strategy

The first step in defining a procedure is to find those system states in which job redistribution can result in improved performance. Obviously, the occurrence of the idle-while-waiting condition should be decreased as well as the occurrence of any state that makes idle-while-waiting more likely. In multiprogrammed systems, assuring that processors are busy is not sufficient to minimize the occurrence of the idle-while-waiting condition. Since the response time for each job is degraded as more jobs are added to the active queue, it is important to distribute all available jobs evenly.

The strategy discussed here uses the queue length of active jobs at a processor as a metric to indicate workload. A queue-length imbalance will make the idle-while-waiting condition more likely. Hence, a load redistribution action is needed when there is significant difference between queue lengths, such that the estimated total overhead of migrating a job, queueing delay at a remote processor and later returning results is less than the delay a job would experience at its source queue.

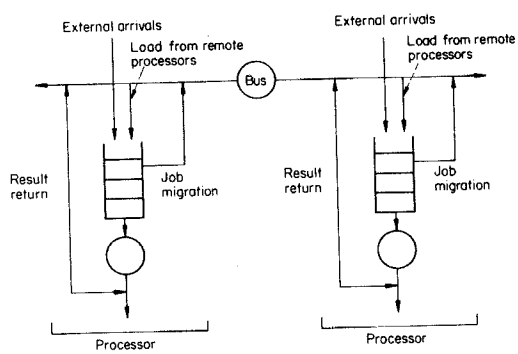

Figure 1. Queuing diagram of a system of processors connected by a broadcast bus

It should be noted that the number of jobs at a processor, while frequently a good reflection of load, is not always adequate. Other factors that may contribute to the workload at a processor include physical differences of processors (such as speed or size of main memory), paging activity, and the ratio of processing activities and input/output activities in jobs. Future work will involve investigating a more inclusive measure of processor load.

An ideal redistribution of jobs, given that the metric used is the queue length at each processor, is to have equal number of jobs at each processor. Since a single bus is used to connect all the computers, only one job can be migrated at any one time, and it is not possible to perform such an ideal redistribution in one step. The movement with the most impact is to take a job from the maximally loaded processor, and send it to the minimally loaded processor. Such a strategy is both source- and sink-initiated. In addition, minimal status information is needed because this strategy only requires the identification of the maximally and the minimally loaded processors.

Three scheduling operations are required for this redistribution on a bus network: identification of the maximally and the minimally loaded processors, job migration, and result return. Migrating jobs and returning results are straightforward because existing communication facilities can be utilized; however, identifying processors with the load extremes efficiently is more difficult. Such an operation should have very low complexity, preferably independent of the number of processors connected to the bus. Any centralized scheduling algorithm, such as polling, is not suitable here.

The priority ordering for tasks using the bus network is (i) regular message transfer, (ii) result return, (iii) job migration, and (iv) max/min identification. The intuitive reasons behind the choice is that the network was designed for message traffic, which has the highest priority in using the network. Early return of results will contribute to reduction in average response time, while early migration of jobs may not because jobs may be queued at the remote processor; hence, return is given higher priority to job migration. Lastly, identifying candidates for load balancing should be given a lower priority relative to job migration because the bus network is shared among all competing jobs; therefore, migrating multiple jobs simultaneously will delay the initiation times of all jobs concerned.

The scheduling strategy, GAMMON, consists of two steps that are executed repeatedly. The first is to determine which of the current tasks has the highest priority, and the second is to execute that task.

WINDOW BROADCAST STRATEGIES FOR DISTRIBUTED EXTREMUM SEARCHES

Contention resolution

Carrier-sense-multiaccess networks with collision detection (CSMA/CD) are a type of local-area network with packet switching and a bus topology[16]. CSMA/CD networks evolved from CSMA networks that have *listen-before-talk* protocols to avoid overlapping transmissions. The collision-detection ability of CSMA/CD networks allows processors to additionally *listen-while-talk*, so collisions resulting from simultaneous transmissions can be detected and stopped immediately.

There are three types of protocols for contention resolution in CSMA/CD networks. Collision-free protocols strictly schedule bus accesses, so no collisions occur. Contention protocols function at the other extreme, allowing processors to transmit whenever they find the bus idle. When collisions occur because of simultaneous transmissions, processors stop transmitting, wait for some prescribed amount of time, and try again. The backoff algorithm of Ethernet[14] is an example in this class. The disadvantage of collision-free protocols lies in the overhead of waiting for transmission, while the disadvantage of contention protocols is the time wasted during collisions. A third type of contention-resolution protocol is the limited-contention protocol. This type of protocol chooses a processor for transmission from among those waiting to transmit based on a priori information, such as the channel load. The Virtual Window Protocol proposed by Wah and Juang[9,18,20] is an example of a limited-contention protocol.

The Virtual Window Protocol assumes that three-state network information is available to each processor: (a) collision is detected on the bus network, (b) bus is idle, and (c) exactly one station broadcast on the network. All participating stations maintain a global window and progressively reduce the size of the window until exactly one station broadcast on the bus. Only stations with contention parameters inside the window are allowed to broadcast in each contention slot, which is the maximum round-trip propagation delay of the network. The advantage of the Virtual Window Protocol as compared to the backoff algorithm of Ethernet is that the time between successive trials is always bounded by the size of the contention slot, while the time between successive trials in the backoff algorithm is random. Moreover, by precomputing the optimal window to be used in each iteration, it is possible to identify the winner in 2.4 contention slots on the average, independent of the number of contending stations[9,18,20]. However, hardware modification to existing Ethernet interfaces is necessary because the three-state network information is normally not accessible to the operating system.

Window-broadcast strategy with two-state network information

The extremum-search strategy developed in this work is similar to the Virtual Window Protocol described above, except that two-state network information is assumed: (a) a station successfully broadcasts in a contention iteration, and (b) no station broadcasts in the broadcast slot. A broadcast slot consists of the time for all stations to contend for the channel and the time for the winner

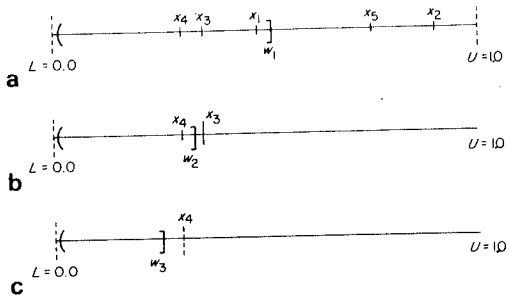

Figure 2. Example of the window protocol using the one-broadcast strategy

to broadcast its message. As a result, such information is available to the operating system, and our design does not require any changes to existing Ethernet hardware interfaces.

Since the information available for window selection is different at the applications level, the decision process has to be modified. The one-broadcast strategy for identifying the minimum is described in detail below; the identification of the maximum is similar and is not described. Further details about other strategies can be found elsewhere[1,2]. The performance is evaluated using the number of broadcast slots the strategy requires to isolate the minimum search parameter. The one-broadcast strategy described assumes that information about the distribution of search parameters is available. To justify the assumptions made on the distribution, measurement results of load distribution are presented in the final portion of this section.

One-broadcast strategy

The one-broadcast strategy allows a maximum of one broadcast slot per iteration. Starting with an interval $(L, U]$, each station has a search parameter x_i in the interval. The stations maintain a global window on the interval. Stations with parameters within the window attempt to broadcast their search parameters, and if there are one or more parameters in the window, there will be a contention resolution followed by a broadcast of one of the search parameters. In that case, the upper bound of the interval will be updated to the value broadcast. If there are no parameters within the window, the lower bound of the interval is updated to the upper bound of the window used, and the protocol continues. The minimum is identified when the lower bound of the interval is equal to the upper bound.

An example of the one-broadcast strategy is shown in Figure 2. In this example, $x_1 = 0.48$, $x_2 = 0.90$, $x_3 = 0.35$, $x_4 = 0.30$, and $x_5 = 0.75$. For the first iteration (Figure 2a), the upper bound of the window chosen is 0.51. Stations 1, 3, and 4 attempt to broadcast their parameters. Suppose that station 3 is the winner and transmits. The next interval to be searched is $(0, x_3)$. Let the upper bound of the next window chosen be 0.33. Only station 4 trys to transmit its parameter, and x_4 is broadcast. The search has not concluded even though x_4 is the minimum because the fact that it was the only station broadcasting is not available to station 4 or to the other stations. The next window chosen is 0.25. There is, of course, no broadcast. This process will continue until the bounds of the window isolate x_4, and the minimum is globally known.

The choice of the window in each broadcast slot is based on the probabilities of the two states, transmission and idle, which are dependent on previous broadcasts. If a previous broadcast slot resulted in the transmission of a value, say x_{b1}, then any subsequent transmissions must be less than x_{b1}. This implies that any subsequent x_is broadcast were eligible to broadcast during the iteration that x_{b1} was broadcast, but lost the contention. The probability of the subsequent transmissions must be conditioned on the fact that any x_is in the current window did not broadcast when they were eligible during previous iterations. The choice of the window is, thus, dependent on previous broadcasts, hence, the choice cannot be optimized by dynamic programming methods because the Principle of Optimality is not satisfied.

Assume that station i has an independent search parameter x_i with distribution $F(x)$ and density $f(x)$. The following definitions are used to formulate an approximate solution of choosing the upper bound of the next window, assuming that the probabilities of transmission and idle are independent of previous broadcasts and are computed without information from previous broadcasts. The following definitions are used.

$N_{A1}(a, b)$ the minimum expected number of broadcast slots to isolate the minimum x_i on the interval $(a, b]$ using an approximate solution of the one-broadcast strategy, given that all x_is are in $(a, U]$, and that at least one x_i is in $(a, b]$;

$\phi_A(a, b, w)$ the probability of a transmission on the interval $(a, w]$, given that all x_is are in $(a, U]$, and that at least one x_i is in $(a, b]$;

$\theta_A(a, b, w)$ the probability of no transmission on the interval $(a, w]$, given that all x_is are in $(a, U]$, and that at least one x_i is in $(a, b]$.

It is obvious that

$$\phi_A(a, b, w) + \theta_A(a, b, w) = 1.0. \tag{1}$$

There are two cases to consider when calculating $\theta_A(a, b, w)$, namely, $b = U$ and $b \neq U$. When $b = U$, it is uncertain whether there is a x_i at b, and the arrangements of the n x_is must be considered, so

$$\theta_A(a, U, w) = \frac{(F(U) - F(w))^n}{(F(U) - F(a))^n} \tag{2a}$$

When $b \neq U$, there must be a station at b, since b is only updated to a value of x_i in the event of a transmission. In this case we are only concerned with the placement of

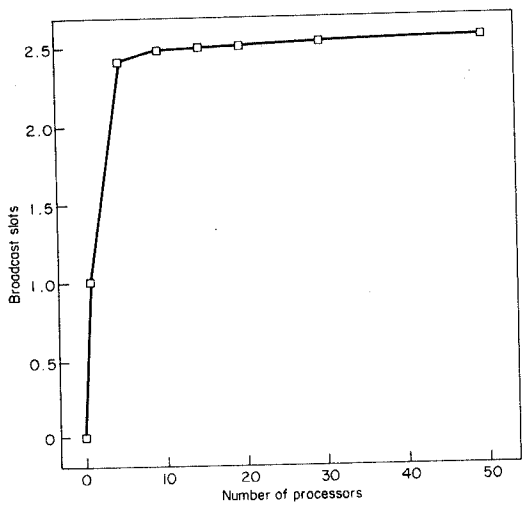

Figure 3. Results of the simulations for the one-broadcast window-search strategy

at most $(n-1)$ of the x_is,

$$\theta_A(a, b, w)|_{b \neq U} = \frac{(\mathrm{F}(U) - \mathrm{F}(w))^{n-1}}{(\mathrm{F}(U) - \mathrm{F}(a))^{n-1}} \tag{2b}$$

The recurrence for choosing the window is

$$N_A(a, b) = \min_{a<w<b} \left\{ 1 + \phi_A(a, b, w) \times \left[\int_a^w \mathrm{f}_c(a, w, x_b) N_A(a, x_b) \mathrm{d}_b \right] + \theta_A(a, b, w) N_{A1}(w, b) \right\}. \tag{3}$$

The three terms on the right-hand side of the above equation count the current broadcast slot, additional broadcast slots in the event of a transmission, and additional broadcast slots if the current broadcast slot is idle.

The assumption that contention can be resolved in one step when the window size is smaller than δ holds, so the boundary condition is

$$N_{A1}(a, b) = 1 \quad \text{for all } (b-a) < \delta \tag{4}$$

The data structure for storing the windows is a two-dimensional array. The number of decision points is determined by the values of a and b. The total number of unique nodes with $\delta = 1/(10n)$ is $((10n)^2 + 30n)/2$, which is determined by counting the decision points indicated by the above recurrences. For $n = 5$ and $n = 6$, the numbers of decisions points are 1 325 and 1 890, respectively.

The simulation results for the distributed window search using the approximate one-broadcast strategy are shown in Figure 3. The broadcast parameters were generated from a uniform distribution in (0, 1], and sufficient cases were simulated until a confidence interval of 0.95 was reached. The number of broadcast slots is bounded on the average by 2.6.

The proposed scheme is practical as a result of the constant expected number of broadcast slots. The time required for a contention slot is approximately 50 μs, and the time required to broadcast a search parameter may be estimated at approximately 100 μs. It follows that each broadcast slot would require on the order of 220 μs if an average of 2.4 contention slots[19] were required to resolve contention. If it takes 120 μs to resolve contention and 100 μs to transmit a 1 kbyte packet, then the overhead of each load balancing decision to identify the maximally and the minimally loaded processors is equivalent to transmitting 5.2 1-kbyte packets.

Distribution of load averages

The knowledge on the distribution of workload (or load averages) is needed in the distributed search in order to choose the windows. The load averages were measured on a system experiencing a real workload in order to determine its characteristics and whether global scheduling has potential to improve performance. The study consisted of measuring the load on a system of 10 Sun workstations (servers and clients). Every sixty seconds, the one-minute load average was measured and logged. The load data was analysed using an adjusted Komolgorov–Smirnov test[12,17]. This goodness-of-fit test can be used to detect differences between a normal distribution and the empirical distribution indicated by the measured data. The agreement was measured over time. The results indicate that the distribution of load averages can be estimated using a normal distribution, as 80% of the time that the system is active, the distribution is within 0.215 of a normal distribution.

The results of the study are shown in Figures 4 and 5. Figure 4a shows the maximum and minimum load averages over time. The minimum is almost always zero, and the maximum varies. The peak utilization is between sample number 700 and 1 400 which reflects the load from 1 p.m. until midnight. From 1 a.m. until 11 a.m., the loads measured were uniformly low ($\leqslant 1$). There is no potential benefit from global scheduling during that time. Figure 4b shows one hour of the minimum, average, and maximum load averages. The differences between the normal and empirical distribution is shown in Figure 5.

PRACTICAL REALIZATION

It is now time to focus on the transition from the theoretical concept described above to an actual implementation. Several options and limitations imposed by the target system architecture have to be considered.

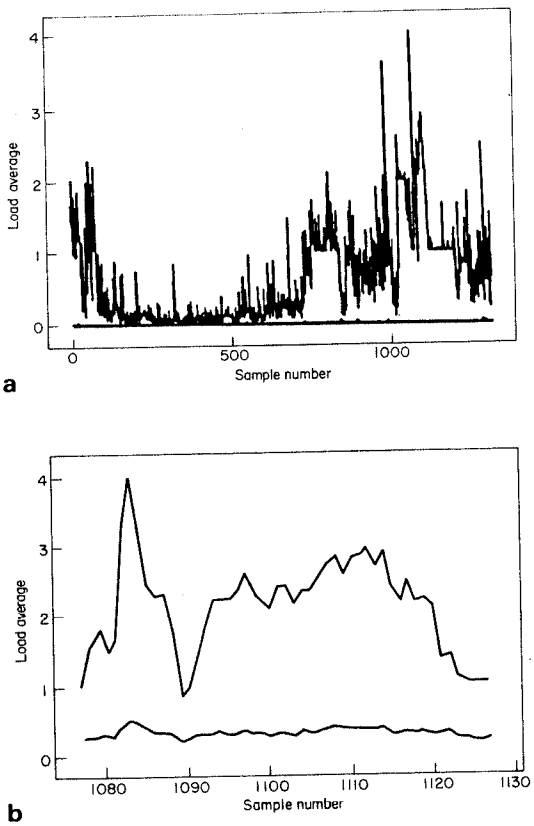

Figure 4. Load maxima and minima (a) for one day; (b) for one hour

Host system

The two distributed computer networks on which GAMMON has been implemented connect Sun 3/280 servers and Sun 3/50 discless clients by Ethernet. The first system consists of one server and two clients whereas the second one has one server and ten clients. A client can access a server's disc via the network, and is allocated a portion of the server's disc for swap space. Swapping over the network is a part of the regular message transfer. The NFS (Network File System) supports transparent access to remote file systems. This mechanism allows uniform access by the clients to secondary storage.

All hosts run Sun Unix version 3.4. When a process is initiated on a server or a client, a core image of that process containing run-time information exists in the swap space associated with that server or client. This core image may be quite large (on the order of several megabytes). Since the operating system maintains a separate swap partition for each host, there is no straightforward way to load-balance jobs already in progress without explicitly copying their core images from one swap partition to another. Due to the size of the core image, this transfer would have high overhead (on the order of seconds). Owing to this overhead, the current implementation migrates jobs only at their entry point. Future enhancements to GAMMON will study pre-emption strategies, especially in combination with Sun Unix version 4.x which allows a common swap area for several hosts.

Adaptation of the GAMMON strategy

The search for the minimum load as described in the previous section takes 2.7 broadcast slots on the average. A practical implementation of the strategy within the constraints of the operating system poses several difficulties. There is a high overhead for all stations to synchronize the updating of the search window and to resolve priorities of broadcasts. A number of requirements in the protocol have been relaxed to result in a practical implementation. A number of strategies that allow a practical implementation but do not significantly degrade performance are discussed in this section.

The first strategy is to truncate the search after one iteration to reduce network traffic. This is equivalent to the use of a precomputed static search window instead of a dynamic one as presented in a previous section. As shown later on, this does not severely affect global performance since workload information is heuristic in nature and small errors in identifying processors for load balancing are not critical. An important tradeoff we have achieved is that reasonable results are obtained at substantially lower overhead.

The second strategy to reduce computational overhead is to deliberately not enforce priorities of network usage as outlined before. A strict enforcement would have required changes to the operating system kernel which

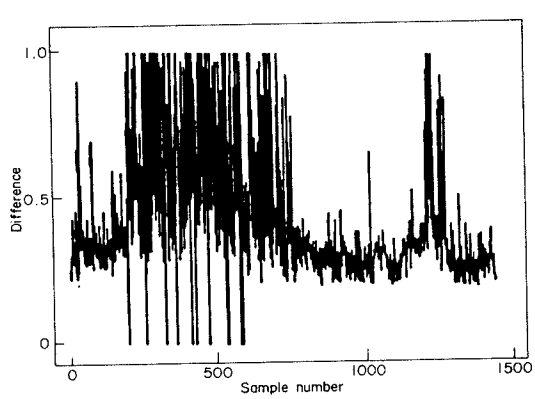

Figure 5. Difference between empirical and normal distributions

would have greatly reduced the portability of our system. Moreover, the effect of interleaving packets containing results and migrating jobs will not have a significant impact on the overall performance. The problem of too many jobs being migrated at the same time is considered subsequently.

In the third strategy to reduce overhead, the search for the maximum load is not performed explicitly. It was observed that if a processor's load was above the initial window in the minimum search, its load was the maximum the majority of the time. As a result, processors with loads above the initial window can migrate the current job if the minimum load is current, and an explicit search of the maximally loaded processor is not needed.

A potential problem with not explicitly identifying the maximally loaded processor is that a lightly loaded processor may be swamped by jobs from more heavily loaded ones. There are two solutions to resolve this problem. First, a processor may only be allowed to migrate jobs if it has a load higher than the upper bound of the initial window and a new arrival. Second, a processor may be allowed to migrate at most one job between searches. Our performance data indicate that swamping is not a problem for a moderate number of participating processors. However, if pre-emption were implemented, it would be necessary to identify a unique maximum, as a large fraction of the processors may be pre-empting jobs at any time.

Finally, in order to avoid synchronization overhead as much as possible, the hosts participating in the search are only loosely synchronized by means of alarm timers which are reset upon arrival of a data packet. It was found that this type of synchronization, which is less stringent than assumed previously, is sufficient for our purposes and does not lead to adverse effects. On the contrary, a lower program complexity and less network overhead were achieved.

Implementation overview

During the program development, a number of decisions were made concerning the implementation of certain functions. On the highest level, the search for the minimally loaded processor should be running periodically and independently of eventual job migrations. This will minimize job startup times. The underlying search algorithm is the previously described window broadcast strategy.

The search daemon is a background process that periodically participates in a search for the minimum load among all network hosts. A search is initiated by a periodic timeout signal on each host, or in response to an incoming data packet. In any of those events, each host will send out a data packet containing its own load information, provided it is within the search window. Incoming packets are used to update the local information about the minimally loaded host.

This algorithm proved to be the most efficient in terms of network load and processor overhead. Finally, the program was designed to be modular, therefore extensions can be easily made. These include a module that emulates NFS calls on a non-NFS machine, and a module which handles migration of jobs already in progress.

The second part of the GAMMON load-balancing software is the remote execution handler, which operates asynchronously from the search. Upon a user request to execute a certain program, the load-balancing software decides if the job will be executed locally or remotely. Job migration is performed by sending the necessary information of a job at its entry point to the remote processor. The input and output channels of the user's shell and the remote execution shell are linked to ensure transparent operation.

IMPLEMENTATION DETAILS

Distributed search

A search is initiated, as mentioned before, by a periodic alarm signal, or by a packet arriving from another search module. When an alarm is received, the processor's current load is compared to the lower bound of the window. If the local workload value is smaller than the currently known lower bound, it is broadcast with a timestamp and processor address. If a packet is received, the load is accepted as the minimum and is stored with the current time. If more than one processor sends a load packet due to their both receiving an alarm simultaneously, the minimum load is accepted. If the loads are identical, the processor address is used as a tie breaker. The alarm timer is reset when packets are received, therefore, the process is loosely synchronized.

The contents of received packets are written to a log file to monitor the current status of the network. Along with the status (the minimum load, and the location of the processor with the minimum load), the current time is also stored. When the status information is retrieved, the time stamp is checked and is used to determine if the load value is out of date. In that case, the processor considers itself ineligible for job migration until it receives another status packet. This time stamping allows an unreliable communication mechanism (broadcast datagrams) to be used to communicate load information. This reduces network traffic since no circuit has to be established and no acknowledgements have to be sent. In case that load information is lost by one processor, that processor does not migrate jobs. This has a minimal effect on the overall performance of the scheduling strategy. Moreover, since packet loss is a relatively rare occurrence, acknowledgements are unnecessary.

The search routine used to find the minimally loaded host can be implemented on single broadcast networks connecting independent workstations or a file server and a number of clients. It can also be run in a hierarchical manner, for example, on a system having a number of the above mentioned networks plus a backbone net connecting the file servers. In this case, the search would be run on each subnetwork independently. In addition, there will be a global search among the file servers on

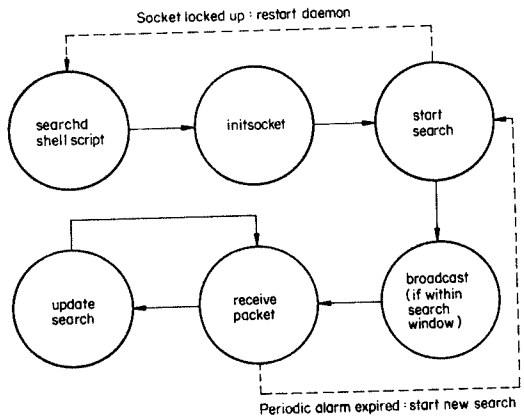

Figure 6. Simplified state diagram of the searchd *daemon*

the main net; however, this search would use the values found by all the local searches. The remote execution handler could then either use the local result, if the scope of the job migration should be limited to the local network, or the global result, if load balancing among the networks is allowed.

Search shell script

The *search* shell script is used to start the search process. It has two purposes. First, it clears the status file containing a trace of all packets sent or received, and the resulting actions taken. Subsequently, the *searchd* process is initiated, which handles all the necessary interactions required for the search process. This includes sending, receiving and processing the load information.

Unfortunately, experiments have shown that the broadcast sockets used for the Internet communication between the hosts participating in the search can lock up. It was found out that this error occurs primarily on heavily loaded networks with heterogeneous machines. A locked-up socket will prevent the corresponding machine from receiving packets, thereby severely disturbing the search process.

Searchd uses an interrupt timer to deter such a condition. If it expires a certain consecutive number of times, the daemon has to be restarted in order to clear the socket. This restart cannot be done using a *fork* or *execute* system call, since interrupt processing, which is vital to the correct function of *searchd* would be blocked. Therefore, the process will exit with an error code and the restart is done at the shell level.

The *gsearch* shell script functions in the same manner as the search script described above, except that it invokes the global search daemon gsearchd. The shell scripts run in *suid* mode and are owned by *root* to allow modification of system files.

Operation of the searchd daemon

A simplified state diagram of the *searchd* daemon is shown in Figure 6. When started, the daemon initializes a broadcast socket using *initsocket*. This procedure creates a broadcast datagram socket for the requested service, standard or global search. Then, it sets the appropriate socket options and constructs an Internet address for the network to be used. It returns a pointer to the newly created socket, which is used by the following communication routines.

Next, the alarm timer is initialized to a predefined interval and the *start_search* procedure is called. This procedure will also be called every time the alarm time expires. The following loop is executed for a given amount of time or forever. Inside the loop, *receive_packet* is called, followed by *update search*. Finally, the alarm time is restarted.

The *start_search* routine counts the number of consecutive iterations in which no packets have been received. If a certain predefined threshold value is exceeded, it can be concluded that the receiving socket has locked up. In that case, the program exits with an error code, thus enabling the *search* shell script to restart the daemon. Otherwise, the local load data is collected and a datagram packet assembled. In case the local load is within the search window, and no other packet has been received in the meantime, a broadcast of the packet is initiated. Finally, the alarm timer is restarted.

The *broadcast* procedure initiates the broadcast of a search packet on the network. Its counterpart is the routine *receive_packet*, which listens on the network until it gets a valid packet. In that case, the alarm timer is temporarily disabled to allow undisturbed processing. Next, the received packet is timestamped and the address of the sending host is returned. The alarm timer is used to interrupt the *receive_packet* procedure by the *searchd* program, if no packet has been received during the given time interval.

If a valid packet has been received, *update_search* is called. This procedure compares the iteration number of the new packet with the current one. If the new number is smaller, the packet is from a previous iteration and was not delivered in time. In that case, it is ignored. Otherwise, the current search number is updated. This also has the effect of automatically synchronizing daemons which have been restarted to the current iteration number. The new packet is subsequently used to update the current load status.

Remote execution

In the general case, result return and job migration require that input file (and executable files if necessary) be sent over the network from the source processor to the destination processor, and that output files be sent back. In an NFS environment, it is not necessary to explicitly send files from the source to the destination because all processors and clients have access to the same secondary storage. Job migration simply requires sending the command from the source to the destination, and returning results and error messages.

The job migration decision is made as follows. First

the current load of the processor is compared to the threshold which would be used for a maximum search. If the load is above that value, the global minimum location and its timestamp are read from a local file. If the timestamp indicates the minimum is current, the job is executed on the processor with the minimum load using a remote shell. This causes any error messages associated with the remote execution to be sent back directly. The remote shell uses a reliable communication mechanism (TCP/IP protocol), so execution of the job is guaranteed. The remote execution handler, *x*, uses the information generated by the search process and the current load on the host it is running on to decide if the job given in the command line argument is to be executed remotely or locally. The minimally loaded machine on the network is determined from the information in the *lbstat* file. In case of a hierarchical search, the *glbstat* file can be used. The local load is obtained from the kernel.

Load balancing is only initiated if the current local load is larger than a certain local minimum and the current minimal load on the network. It is initiated by obtaining the current working directory and using the *rsh* system command to establish communication with the remote host. First, the working directory on that host is set to the working directory of the local host. Then, the command is executed. The input, output, and error message channels are linked to the remote host such that job execution appears transparent to the user.

A sample *x* command looks like this:

```
x ditroff -me -Tps filename >~ /printfile
```

It should be noted that the only difference to the usual command invocation is the leading *x* which indicates load-balanced execution. Even shell metacharacters such as {~ * >} and the like are allowed. The above command translates into the following line:

```
rsh minloadhost ' cd workdir; ditroff -me -Tps filename >
                                        homedir/printfile '
```

The above line will perform the transparent remote execution. It was deliberately decided to use the remote-shell facility instead of a remote procedure call initiating a job via the *system* or *popen* commands. The reason is that the *rsh* call allows shell metacharacters, multiple command execution, and provides linked input, output and error channels. Therefore, command execution is transparent to the user. This is achieved at a slight performance expense due to a somewhat longer startup time. However, since jobs to be load-balanced are usually long, this extra time should be negligible in most cases.

The above described mode of execution depends on the existence of NFS or a similar remote file sharing facility. The program was conceptually designed for a system mainly consisting of discless workstations and file servers. In such an environment, a network file system is usually a part of the operating system and transparent to user applications. However, if the remote execution facility is to be used in an environment where NFS is not used, explicit copying of files has to be done. This generates a much larger overhead and network traffic and should be avoided when possible. For such applications, a software module can be provided which simulates an NFS environment.

Operating/file system modifications

The following updates were made to the *services* database in the *Yellow Pages*.

```
search      720/udp     search-service      # experimental
gsearch     721/udp     global search       # experimental
```

The *rc.local* script was updated to include the following lines which start the search daemons at boot time.

```
If [ -f /usr/local/search -a -f /usr/local/searchd ]; then
        /usr/local/search & (echo -n ' search')
                                        > /dev/console
fi
if [ -f /usr/local/gsearch -a -f /usr/local/gsearchd ]; then
        /usr/local/gsearch & (echo -n ' gsearch')
> /dev/console
fi
```

The search daemons create several files in the */usr/spool/rwho* directory. This directory was chosen since it is private to each host and its usage does not interfere with normal system operation. Its usage eliminates further system modification since no special directory has to be created on each host's partition. The files created are listed in the following.

```
/usr/spool/rwho/lbstat      (load information for search)
/usr/spool/rwho/glbstat     (load information, global)
```

PERFORMANCE OF GAMMON

As mentioned before, the implemented version of the window-broadcast search strategy uses only one fixed window. Table 1 shows the simulation results of such a truncated search. This table shows that, under this condition, the absolute minimum is located 70–78% of the time, and there are 18.6–23.1% of the unresolved cases in which all search parameters are in the interval $(w_1, U]$ and no workload information is broadcast. Unresolved searches are not critical here because they reflect a condition in which no processors are lightly loaded enough to accept additional jobs. For cases that are resolved, those that do not find the absolute minimum only have a one to two percent difference from the absolute minimum. Since workload information is heuristic in nature, small errors in identifying processors for load balancing are not critical. An important tradeoff achieved is that reasonable results are obtained at substantially lower overhead.

The overall performance of the search algorithm was tested on two differently configured systems: one with one server (Aquinas) and two clients, and another with one server (Dwarfs) and ten clients. On Aquinas, a complete search was resolved in 50–80 ms, and on Dwarfs, it was resolved in 150–180 ms. The performance on Dwarfs can be considered the worst case since the number of clients and the physical net length approach the allowable limits. The results for both Aquinas and

Table 1. Results of simulation for the truncated search

Number of processors	% of searches unresolved	% resolved searches finding minimum	Average rank of result	% difference of result from minimum
3	20.0	78.00	0.194	2.790
4	18.6	76.04	0.224	2.192
5	20.1	73.97	0.268	2.086
6	21.8	73.66	0.252	1.665
7	21.4	72.26	0.279	1.506
8	20.5	70.57	0.292	1.495
9	23.1	72.82	0.256	1.151
10	21.8	74.04	0.256	0.959

Table 2. Comparison of cumulative overhead for status distribution in one hour

Deamon	Cumulative system time (seconds)	Cumulative user time (seconds)	Total number of broadcasts
search	0.1094	0.0137	65
broadcast	0.2593	0.0316	181
rwho	0.2633	0.0613	181

Dwarfs were consistent with the simulation results in which the absolute minimum was located 70% of the time for all resolved searches.

Another obvious technique for distributing status information would be to broadcast it periodically, as is done with the *rwhod* daemon in Unix. To determine the savings of using the proposed method as opposed to using the technique of the *rwhod* daemon, the resource utilization of both was measured for the period of one hour. The overhead is summarized in Table 2. For comparison, a daemon that only broadcasts the load average value is studied (as opposed to the *rwho* daemon which broadcasts other information as well). Recall that the daemons perform a search every minute. When a search using GAMMON is performed, it is possible for daemons at two computers to start a broadcast simultaneously; hence the total number of broadcasts is slightly above 60 (first value in the last column of Table 2). When the broadcast of *rwhod* daemons are used, each computer initiates a broadcast every minute independent of other computers. These results indicate that both *rwhod* and the simple broadcast daemon introduce considerably more communication overhead than the search daemon we proposed.

To compare the overall improvement in performance due to the GAMMON load-balancing strategy as compared to that without load balancing, the server Aquinas with its two clients, Calvin and Hobbes were used. First, workload was generated by processes that either initiated a CPU-bound job or slept for the amount of time the job consumed during its last execution with equal probability. As these processes were executed, the load and the amount of time consumed by executing processes was tabulated. Also, a history of the initiation of jobs was created. During the second portion of the test, the global scheduling strategy was enabled, and the jobs were initiated according to the history. Again the load was monitored, and the execution time tabulated.

Table 3. Performance of the global scheduling strategy

Name	Job time w/out global scheduling	Job time with global scheduling	% difference
Aquinas (Sun 3/260 server)	18 963	18 540	2.23
Calvin (Sun 3/50 client)	12 143	10 430	14.10
Hobbes (Sun 3/50 client)	7 319	5 912	19.22
Totals	38 425	34 882	9.2

The results as summarized in Table 3 show a small improvement for the server, and a much larger improvement for the clients. This is expected as the server is about 2.7 times faster than the clients, and has direct access to the secondary storage through the VME bus rather than via Ethernet. As a result, the clients will benefit more by sending jobs to the server.

CONCLUSIONS

This paper has presented an efficient technique for a distributed extremum search and a load-balancing protocol using this technique. The search technique can be implemented on *existing* CSMA/CD networks at the applications level. This is important because it is typically not possible to make hardware modifications to existing networks. The maximum or the minimum of a set of numbers can be identified in a small bounded number of

broadcasts on the average. Since the search technique has a constant average behaviour, the GAMMON strategy using this efficient search technique is feasible. Performance of GAMMON was found to be favourable at low-to-moderate traffic intensities, and load-balancing intervals (the total time to perform max/min identification, job migration, and result transfer) that are small relative to the average service time.

Implementing GAMMON on a network of Sun workstations reveals a number of disparities between the assumptions made in the theoretical model and the physical limitations of distributed computing systems. By relaxing a number of assumptions without degrading the performance, a load-balancing system has been developed that is comparable in performance to the theoretical model and allows improvements in overall as well as in individual processor performance.

ACKNOWLEDGEMENT

This research was supported in part by the National Aeronautics and Space Administration Contract NCC2-4881 and the National Science Foundation Grants MIP 85-19649 and MIP 89-10584.

REFERENCES

1 **Baumgartner, K M and Wah, B W** 'Load balancing protocols on a local computer system with a multi-access bus' *Proc. Int. Conf. Parallel Processing* Pennsylvania State University, USA (August 1987) pp 851–858

2 **Baumgartner, K M** *Resource allocation on distributed computer systems* PhD Thesis, School of Electrical Engineering, Purdue University, USA (May 1988)

3 **Chou, T C K and Abraham, J A** 'Load balancing in distributed systems *IEEE Trans. Softw. Eng.* Vol 8, (July 1982) pp 401–412

4 **Chow, Y C and Kohler, W** 'Models for dynamic load balancing in a heterogeneous multiple processor system' *IEEE Trans. Computers* Vol 28, (May 1979) pp 334–361

5 **Chu, W W, Holloway, L J, Lan, M T and Efe, N K** 'Task allocation in distributed data processing' *IEEE Comput.* (November 1980) pp 57–68

6 **Eager, D L, Lazowska, E D and Zahorjan, J** 'Adaptive load sharing in homogeneous distributed systems' *IEEE Trans. Softw. Eng.* Vol 15, (May 1986) pp 662–675

7 **Hwang, K, Croft, W J, Goble, G H, Wah, B W, Briggs, F A, Simmons, W R and Coates, C L** 'A UNIX-based local computer network with load balancing' *IEEE Computer* Vol 15 No 4, (April 1982) pp 55–66

8 **Juang, J Y and Wah, B W** 'Optimal scheduling algorithms for multistage resource sharing interconnection networks' *Proc. Comput. Softw. Appl. Conf.* IEEE, UK (November 1984) pp 217–225

9 **Juang, J Y and Wah, B W** 'Unified window protocols for contention resolution in local multiaccess networks' *Proc. INFOCOM* IEEE, UK (April 1984) pp 97–104

10 **Juang, J Y and Wah, B W** 'Global state identification for load balancing in a computer system with multiple contention buses' *Proc. Comput. Softw. Appl. Conf.* IEEE, UK (October 1986) pp 36–42

11 **Kratzer, A and Hammerstrom, D** 'A study of load leveling' *Proc. COMPCON* IEEE, UK (Fall 1980) pp 647–654

12 **Law, A M and Kelton, D W** *Simulation modeling and analysis* McGraw-Hill, New York, NY, USA (1982)

13 **Livny, M and Melman, M** 'Load balancing in homogeneous broadcast distributed systems' *Proc. Modeling and Performance Evaluation of Computer Systems* ACM SIGMETRICS, USA (1982) pp 47–55

14 **Metcalfe, R and Boggs, D** 'Ethernet: distributed packet switching for local computer networks' *Commun. ACM* Vol 19 No 7 (1976) pp 395–404

15 **Ni, L M and Hwang, K** 'Optimal load balancing strategies for a multiple processor system' *Proc. 10th Int. Conf. Parallel Processing* IEEE, UK (August 1981) pp 352–357

16 **Tanenbaum, A S** *Computer networks* Prentice Hall, UK (1981)

17 **Trivedi, K S** *Probability and statistics with reliability, queuing, and computer science applications* Prentice-Hall Inc., Englewood Cliffs, NJ, USA (1982)

18 **Wah, B W and Juang, J Y** 'An efficient protocol for load balancing on CSMA/CD networks' *Proc. 8th Conf. Local Computer Networks* IEEE, UK (October 1983) pp 55–61

19 **Wah, B W and Juang, J Y** 'Resource scheduling for local computer systems with a multiaccess network' *IEEE Trans. Computers* Vol 34 No 12 (December 1985) pp 1144–1157

20 **Wah, B W and Juang, J Y** *An efficient contention resolution protocol for local multiaccess networks* Patent Number 4630264, Filed: September 21, 1984, Granted: (December 16 1986)

21 **Wang, Y T and Morris, J T** 'Load sharing in distributed systems' *Trans. Computers* Vol 34 IEEE, UK (March 1985) pp 204–217