IEEE Transactions on Parallel and Distributed Systems, Vol 3, No. 6 November 1992.

Methodical Analysis of Adaptive Load Sharing Algorithms

O. Kremien, J. Kramer

Department of Computing,

Imperial College of Science, Technology and Medicine

180 Queen's Gate, London SW7 2BZ, U.K.

email: ok@uk.ac.ic.doc, jk@uk.ac.ic.doc

Index Terms: adaptability, algorithm evaluation, efficiency, hit-ratio, load sharing, performance, scalability, stability.

ABSTRACT

This paper presents a method for qualitative and quantitative analysis of load sharing algorithms, using a number of well known examples as illustration. Algorithm design choices are considered with respect to the main activities of information dissemination and allocation decision making. We argue that nodes must be capable of making local decisions, and for this efficient state dissemination techniques are necessary. Activities related to remote execution should be bounded and restricted to a small proportion of the activity in the system. The quantitative analysis provides both performance and efficiency measures, including consideration of the load and delay characteristics of the environment. To assess stability, which is also a precondition for scalability,we introduce and measure load sharing *hit-ratio*, the ratio of remote execution requests concluded successfully. Using our analysis method, we are able to suggest improvements to some published algorithms.

1. INTRODUCTION

Distributed systems offer the potential for improved performance and resource sharing. In such systems it is possible for some nodes to be heavily loaded while others are lightly loaded or even idle, resulting in poor overall system performance [21]. Improving performance is a difficult goal. Eager et al. [11] have noted that within the myriad of load distribution algorithms proposed in the literature lie two distinct strategies for improving performance: load balancing algorithms, which strive to equalise the workload among nodes, and load sharing policies which simply attempt to assure that no node is idle while other software components wait for service. Krueger and Livny [18] show that load balancing strategies put much higher resource requirements on the system than load sharing strategies, and that these resource requirements may outweigh their potential benefits. Furthermore, scalability may require that a load distribution policy is forced to limit negotiation to a subset of nodes. We therefore concentrate on load sharing.

The purpose of load sharing is to improve performance (e.g. reduce average response-time) by smoothing out transient peak overload periods on some of the nodes. This can be achieved through dynamic initial placement and by redistributing the workload among the nodes after start-up.

Our model of the *physical* distributed system assumes multiple independent processing nodes interconnected by a communication network. Our model of the *logical* distributed system is that of multiple independent applications, which may comprise single or multiple software component processes, running on the same physical system. Requests for application creation, which translate into the creation of one or more component processes, arrive randomly at nodes. In this environment, load sharing is the problem of allocation: of mapping and remapping the logical system to the physical system.

A flexible load sharing algorithm is required to be general, adaptable, stable, scalable, transparent to the application, fault tolerant and induce minimum overhead on the system [11, 28].

For instance, scalability implies that an algorithm should be independent of physical system characteristics such as system size and physical topology, and exhibit minimum sensitivity to physical resource characteristics such as communication bandwidth and processor speed. Whenever any of these characteristics is scaled up, speed ratios and consequent delays in the system may change. We require that the algorithm should still apply after such changes. Since some of the algorithms studied in this paper were not designed to be independent of system size, emphasis will be put on the effect of delays. Mirchandaney, Towsley and Stankovic [23]

studied the problem of communication delays and out-of-date state information and its impact on simple threshold load sharing algorithms. Unfortunately, probing overhead was still assumed to be negligible. We believe that non-negligible delay should be assumed for all message exchanges [6, 13] in order to permit a thorough study of the effects of delays.

Algorithm stability, which is a precondition to scalability, is an indication of the ability of the algorithm to avoid poor allocation decisions. To assess stability we identify and measure load sharing *hit-ratio,* the ratio of remote execution requests concluded successfully. Another measure of stability is percentage of remote execution in the system. Activities related to remote execution should be bounded and restricted to a small proportion of the activity in the system.

The properties listed above are interdependent. For example, lengthy delays in processing and communication can affect the algorithm overhead significantly, result in instability and indicate that the algorithm is not scalable. How should a load sharing algorithm be designed and evaluated, especially when design requires that some characteristics are sacrificed in favour of others? How can we aid the design and analysis process? This paper presents a method which supports both qualitative and quantitative evaluation and aids the designer of an algorithm to make trade-offs for a particular environment. Use of the evaluation method is demonstrated showing the combined effect of delays and the selection of specific design choices. Furthermore, the method is used to suggest improvements to some published algorithms.

Section 2 presents the qualitative analysis approach, stressing those issues left for the designer to resolve. This analysis method is illustrated with an example from the literature. For quantitative analysis, Section 3 presents an evaluation method which allows for objective comparison of different approaches to load sharing. Section 4 illustrates use of the analysis method on some published load sharing algorithms and enables us to suggest improvements. Conclusions are provided in Section 5.

2. QUALITATIVE ANALYSIS - ALGORITHM CHARACTERISTICS & DESIGN CHOICES

As mentioned, a flexible load sharing algorithm is required to be general, adaptable, stable, scalable, transparent to the application, fault tolerant and induce minimum overhead on the system. We now turn our attention to a qualitative analysis of load sharing policies. We discuss a number of algorithm characteristics based on earlier classifications [29, 9] which have been extended to include other important characteristics. In particular dynamic (adaptive) distributed cooperative algorithms are identified as the most promising [21, 25, 2, 8, 31 and others]. This choice is highlighted in Fig. 1 using Casavant's Classification [9]. Design choices and open issues are discussed.

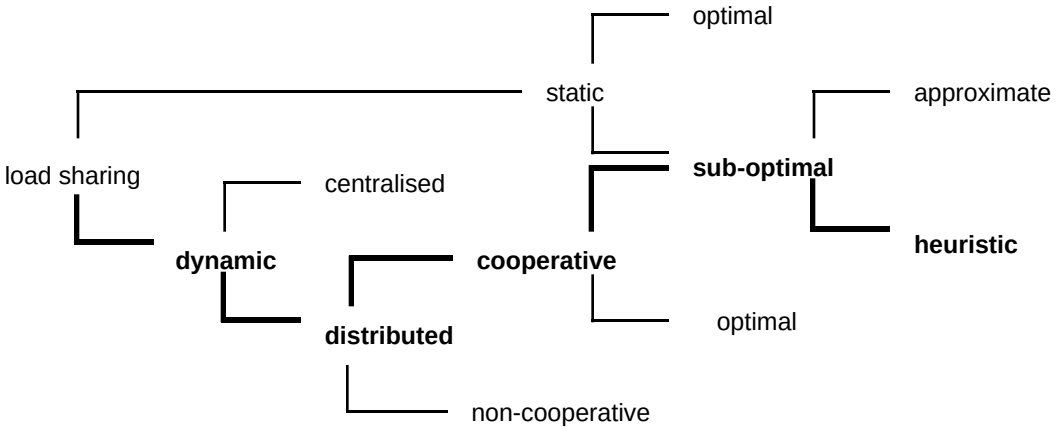


Figure 1: Casavant's Classification

Algorithm Design: General Properties

The responsibility for load sharing can be centralised or distributed. It can reside on a single node or be distributed among the nodes. To be fault tolerant, algorithms should avoid having a single point of failure. Furthermore, for low overhead, they should employ simple techniques for fault-tolerance such as periodic information exchange. Researchers generally agree that symmetrically distributed load sharing algorithms are preferable for fault-tolerance and low overhead purposes [25, 2, 31, 32, 28].

However there is less agreement with regard to scalability. The problem may lie with the definition of scalability. Some include independence of system size in their definition [3], while others [31, 32, 28] define it as the ability to handle a predefined maximum number of nodes. For the latter, a centralised algorithm may be employed. The functional capacity of any centralised server is bounded and cannot grow when the system in which it is embedded is enlarged. Consequently, as the system grows beyond a certain size, a centralised server is bound to become saturated [3]. To be independent of system size, an algorithm should be symmetrically distributed, taking advantage of distribution by maintaining only a partial view of the system at each node. The algorithm must then make the most from the partial information available. We use scalability to infer such independence of system size.

Load sharing involves software component allocation. Should a component be permitted to be allocated (migrate) once or many times? If the latter is the case, is it bounded? If a component is allowed to migrate indefinitely, a system could be unstable and thrash. One can additionally consider migration of more than one component at a time. Although this permits overloaded nodes to be rapidly offloaded, one has to avoid "flooding" other nodes, i.e. not all the components selected for remote execution are sent to the same target node. This calls for cooperation and negotiation.

Load sharing can be supported at different stages in the component life-cycle (before or during execution) and at different levels of an application (the whole application or components of a distributed application). It is clear that the more flexibility that is allowed, the higher is the price to pay for the service provision. Nevertheless, to exploit the potential flexibility in a distributed programming environment, we believe that load sharing should take place at the configuration level, in terms of software components and their interconnections [22]. With respect to when allocation should take place, non-preemptive migration before a component/application is started (i.e. initial allocation/placement) is the simplest to support. The alternative, preemptive migration of applications after start-up, requires mechanisms for checkpointing, transfer of code and context, and restart [24]. To allow such migration at the component level additionally requires a guarantee of no loss of messages in transit [15]. Although migration after start-up provides greater flexibility, it is not clear that the benefit is worth the support costs which are much higher [12,19], especially in a distributed programming environment. We therefore believe that much of the benefit can be realised from good initial placement decisions, preferably at the component level.

An Example

For didactic purposes, to describe and illustrate our method, we select an example of a simple distributed adaptive load sharing algorithm for dynamic initial placement of applications by Zhou [32], termed ZhouD (DISTED- distributed). Each node periodically broadcasts its local load to all other nodes, unless it has not changed since the last update. A data structure at each node holds load state information regarding all nodes in the system. It is updated whenever a state update message is received. It represents the node's view of the system state. The decision making component is activated upon application arrival, and uses a threshold transfer policy. When the local load is at or below some set threshold, T_l, all applications are processed locally. Otherwise, a remote server is selected using the following placement policy. The local version of the state vector is searched for the least loaded node. If this load is lower than that of the local node by Δ or more, the application is sent to that node and no acknowledgement is expected. Otherwise, the application is processed locally. To ensure that remote execution is worthwhile, only applications with expected execution time above T_{cpu} (drawn from command type) are eligible for remote execution.

As can be seen in ZhouD adaptive algorithms for load sharing comprise two main activities - *information dissemination* and *decision-making (control).* We discuss each in turn.

Information Dissemination

Information holding, exchange and update strategies play an important role in maintaining the local view of the system state at a node. Due to communication delays and state distribution, a complete and consistent view of the entire system, or even of a subset , may never be available at a node of the system.

With the goal of minimising overhead, a state metric should be selected that concisely represents the load on the most congested resources. It should also indicate the capacity of the node in terms of those resources. The state metric chosen should have an absolute well understood meaning. Since decisions are to be taken based on this state information, the requirement is that it should be a true reflection of load. For instance, we try to reduce the likelihood that a component sent to a remote node should find it or cause it to become overloaded.

Ferrari and Zhou [14, 30] have studied the problem of load metric selection and found a theoretical foundation for all load indices based on queue lengths. This foundation provides a justification for these indices, specifies the assumptions under which they are valid, and gives an indication of the limits of their applicability. Queue length is the state metric used by the algorithms studied in this paper.

Decisions made by a node are based on both local (such as local load state, communication rates with other nodes) and non-local dynamic state information held in a state vector. A node may also choose to keep some static information regarding other nodes in the system (e.g. node identifiers). For all these types of information, we have to decide whether to hold state information regarding all nodes in the system, or only a subset of the system. If, for scalability purposes, subsets are used, then criteria must be specified for selecting the nodes to be included in the subsets. In ZhouD, state information is held regarding all other nodes in the system. Therefore, the algorithm is dependent on system size .

There are few choices for load state update strategy [20]. A node may choose to request information when the need arises, disseminate information to advertise its state without an explicit request from another site, or use a predictive analysis technique to update the contents of the state vector. The objective is to select a strategy which minimises the overhead imposed and provides the most accurate system state information when this is needed. Strategies based on information request or predictive analysis may result in a long delay, especially when some resources are overloaded. With state dissemination the node may have the information available when it is needed, and be able to make local decisions as desired for scalability purposes. Birman and Joseph [5] have argued that “decisions should be local whenever possible since an

agent that must interact with others before making a decision would be delayed until they respond". Zhou concludes that algorithms using periodic and event-driven information dissemination policies provide comparable performance [31, 32]. Periodic state update is used by ZhouD. However, if dissemination costs are high, periodic state update can result in a high percent of wasted load exchanges. In addition, it can result in outdated information if the update rate is too slow. This can be mitigated to some extent when periodicity is dynamically tuned as suggested by Zhou. With event-driven state update, a load message is sent only after a significant change has occurred. If n-state[1] representation of the load is used, with a method to filter out insignificant transient changes, state update is not expected to occur frequently. One can still prevent overuse at high event rate by means of a lower bound periodicity, i.e. prevent state exchange before a certain period has elapsed since the last state exchange. To take account of failures, a combination of event-driven and slow periodic update may be used.

A decision needs to be made as to whether to send only direct information (information regarding only the node itself) in a state-update message, or to include also indirect state-information (information regarding other nodes in the system) [25, 2]. If the latter is selected, which other nodes should be included? How are different estimates weighted? How are outdated estimates filtered out? The only up-to-date state information that a node holds is its own. An earlier version of MOSIX [2] used indirect information. Experience with this mechanism has shown that it resulted in misleading (outdated) information, and the new version uses only direct information [4]. In ZhouD only direct state information is used. One can still decide whether to update all nodes in the system, as is done in ZhouD, or only a subset. Use of broadcast [21, 32] or multicast [28] results in intolerable overheads. If subsets of maximal predefined size are selected, one needs criteria to choose the nodes to be included.

Decision Making

A load sharing policy tries to reduce the mean response time through remote initial placement and migration after start-up. Any load sharing policy should also strive to minimise remote execution activities in the system. When the system is heavily congested, much higher delays than the average may be expected which can severely degrade performance. As concluded by Zhou [31], only a small percentage of the applications need to be remotely executed in order to achieve effective load sharing .

With symmetrically distributed load sharing, the algorithm is distributed (replicated) across all nodes in the system. When should remote execution be considered, i.e. what is the *transfer policy* [11] ? In ZhouD a threshold transfer policy is used. Which components are eligible for

[1]Mapping a metric of a large number of possible values into a small set of n values.

remote execution? Clearly only those components which can execute on any other node in the system with exactly the same results are eligible. In [6, 30] it is shown that even when 50-70% of all components are immobile (i.e. must execute locally), performance improvements due to load sharing can still be realised. Filtering techniques are suggested to set the eligibility of components for remote execution [27]. In ZhouD, eligibility is also based on estimation of cpu requirements. Since this is not always possible, we believe that whenever a component is mobile, filtering should simply be based on current load conditions. The objective is always to minimise the number of software components moved so that performance reduction is still achieved. When a component is found to be eligible for remote execution, what *location policy* [11] is used to determine to which node a component selected for transfer should be sent? ZhouD selects the least loaded node provided that it is less loaded by at least Δ, according to state information available at each node.

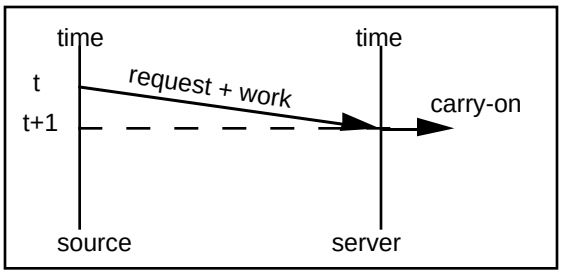


Figure 2: Source initiated: Single request

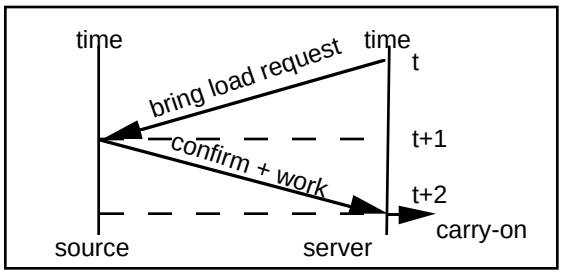
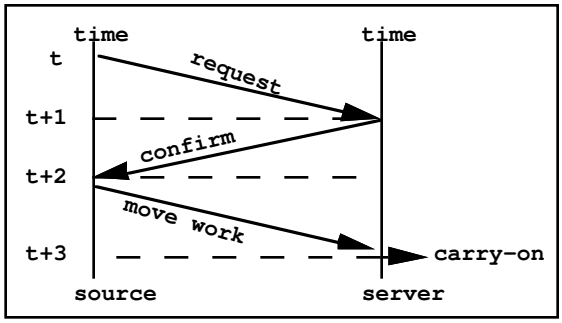


Figure 3: Server initiated Request-reply



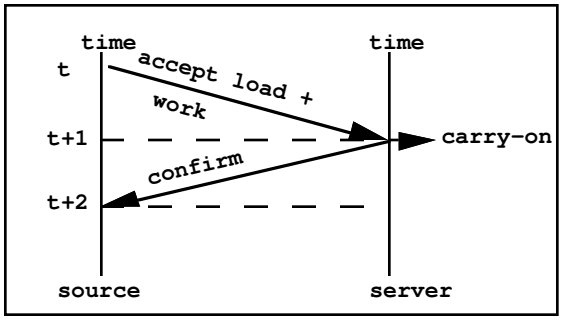


Figure 4: Source initiated negotiation, without (left) and with (right) work attached.

Who should initiate remote placement or migration? It can be initiated by the source of work (overloaded node), or the server of work (underloaded node). A source initiated algorithm can be limited to a single request (Fig. 2) or involve a request and a response (Fig. 4), whereas a server initiated algorithm involves both a request and a response (Fig. 3).

Source-initiated algorithms place much of the overhead on the overloaded nodes, whereas server-initiated algorithms place much of the overhead on the underloaded nodes. The latter

have the potential of being more stable, since the algorithm is initiated by the underloaded node which can more easily prevent overloading by controlling the amount of work requested. Once a node is no longer underloaded, it will stop requesting work. Furthermore, the algorithm can be 'turned off' under heavy system loads, since there are no requests for load transfer if there are no underloaded nodes. Server-initiated algorithms also degrade more gracefully since a failed server will cease to request work. A disadvantage of server initiated algorithms is that the server is not always aware of event occurrence at the source which may prompt remote execution.

One should also decide whether to employ a single request protocol or negotiation. Single request protocols, as used in ZhouD, are fast but not robust. They can lead to instability and performance degradation resulting from incorrect decisions. It is possible for a burst of work (flooding) to be imposed on a node which was mistakenly considered to be underloaded. Negotiation enables poor decisions to be tolerated by inhibiting their consequence: a bad decision can be reversed. With negotiation, work in transit can be taken into account by the server. The disadvantage of negotiation, when it is not possible to attach the work to the request, is that compared with a single request protocol, the number of phases involved is larger and will result in a longer delay. With negotiation, one has to define the *acceptance policy* to determine which components selected for transfer should be accepted by the selected destination.

As a consequence it seems that a combination of negotiation initiated by the server when possible, and by the source when it is the only one aware of an event occurrence, can give the best results in terms of performance and stability but at a slightly higher cost than single request.

There are a number of possibilities for specifying *when decision-making should be activated*: periodic decision making [25], event based decision making, as in ZhouD (upon application arrival), or a combination of these choices.

With periodic decision-making it is hard to select a sensible time period, since it is very dependent on the loads on the system, and needs to be tuned to reflect the dynamics of the system. We advocate the use of a combination of the two methods with events as the basic method and a lower bound periodicity selected to prevent overuse and overload. This helps to prevent an algorithm from being invoked too frequently when event rate is high, and keeps it from ‘stealing’ more than an allowed percentage of cpu time. It also permits its operation at the highest allowed rate in overload conditions.

Qualitative Analysis Summary

The DISTED algorithm suggested by Zhou is general. The algorithm is not scalable since every node holds information regarding all other nodes, and it updates and is updated by all other nodes. (Note that the algorithm was not designed to be independent of system size and we emphasise this point for didactic purposes only.) Also, the algorithm does not try to limit remote execution activity for scalability and stability purposes. This may cause further performance degradation. The algorithm is source initiated, using a request only protocol which does not allow a destination node to reject additional load when it is unable to accommodate it. Each of the issues raised can lead to instability. Thus, under certain circumstances, we may expect to have a high percentage of fruitless decisions which cannot be refused (i.e. moved applications arriving at overloaded nodes producing high percentage remote execution and low hit ratio). ZhouD is summarised in Table 1.

	ZhouD
decision making invocation	event driven (application arrival)
transfer policy	local information only (threshold)
location policy	least loaded
acceptance policy	single request - no rejection allowed
information policy	periodic state dissemination

Table 1: ZhouD - design choices

When designing a load sharing algorithm, there are numerous design choices and open issues left for the designer to resolve. These issues have to be examined carefully with respect to the required properties and applications environment. In this section we have presented these issues together with a discussion of some of the implications of design decisions. For a more quantitative evaluation, measures of performance and efficiency are necessary. These are discussed next.

3. QUANTITATIVE ANALYSIS - ALGORITHM EVALUATION

For an objective evaluation of different approaches to load sharing, one needs a quantitative analysis technique for these algorithms together with a set of characteristics which capture the essence of their behaviour. Casavant and Kuhl [7] have characterised the structure and behaviour of distributed decision-making policies, with particular attention to load sharing policies. The terms *performance* and *efficiency* are advocated.

In a general sense, performance is an absolute measure which is described in terms of response-time, utilisation or any other objective function specified. The efficiency of a

particular solution is a relative term concerned with the cost or penalty paid for the level of performance attained. Efficiency is normally characterised by resource demands in terms of time and space. We too adopt these terms for our analysis of load sharing algorithms, but extend their meanings as follows.

Casavant defines performance in terms of *optimality* (how close to an optimal solution) and *stability*. A stable system is defined as a system in which bounded input produces bounded output. This is necessary but not sufficient. We believe that activities related to remote execution should be restricted to a small proportion of the activity in the system (application arrival). An algorithm should make local decisions, minimise the number of incorrect decisions and not allow them to proceed. Stability, overhead and scalability are thus related to the price paid for the level of performance attained. As a consequence measures of stability, overhead and scalability should be part of the efficiency definition. In particular, a new measure of stability and scalability should be added to the efficiency measures. This is **hit ratio**, the ratio of successful decisions. A successful decision is one where, when a node is requested to perform an operation, it is capable of doing so. An unsuccessful decision will result in worthless information exchange or even fruitless component movement to an overloaded node. This should be minimised.

Performance Evaluation

Application performance is a function of the cpu requirements of the components comprising the application, the number of remote messages exchanged by these components, and the current load on required resources. Our objective function is to improve performance which is simply defined as minimisation of the system average response time. For this, the number of application remote messages exchanged has to be kept as low as possible. For each remote message exchanged, we model the cost incurred by adding a delay to the time consumed by the message originator. Application performance (response time) is modeled as described in Figure 5.

Given an application A, comprising n components: A(1)..A(n), we define:

response_time(A(i))=wait_time(A(i))+cpu_required(A(i))+itc_cost(A(i))+system_cpu(A(i))

where:

wait_time(A(i))=start_time(A(i))-arrival_time(A(i))

itc_cost(A(i))=number_of_remote_messages(A(i)) × MESSAGE_DELAY

system_cpu(A(i))=cpu 'stolen' for system activities during the component's execution

and

$$\text{response\_time(application A)}=\underset{i=1..n}{MAX}\ \text{response\_time(A(i))}$$

Given a system with a load-sharing algorithm LS and m applications running, we have:

$$\text{performance(LS)}=\text{response\_time(LS)}=\frac{1}{m}\sum_{k=1}^{m}\text{response\_time(application A)}$$

Figure 5: system performance (response time)

In addition to the absolute metric of performance, relative ones are used to give guidance as to the likelihood of improved performance achieved by adding the load sharing facility to the system. The performance of an algorithm under study is compared to the performance of baseline algorithms. Three baselines are used: **no cooperation** among nodes (no load sharing); **random** - where a component to be moved is sent to a randomly selected node hoping to find it less loaded, and there is no state information exchange. The random algorithm uses a threshold transfer policy, as in most of the algorithms studied in this paper; **ideal** - where nodes have accurate system state information when decisions are to be taken (making the unrealistic assumptions that there are no delays incurred by communication, and the cost of algorithm invocation is zero). Note that both no cooperation and ideal are independent of delay assumptions. In general, we would wish the evaluation of an algorithm to show that its performance approaches ideal and is considerably better than no cooperation and random. If an algorithm is not significantly better (say at least 10%) than random, then the latter, which is the simplest load sharing algorithm that can be devised, should be used instead or a better solution should be sought. Whenever possible, it is valuable to devise an analytical model (e.g. queueing model). This can approximate the behaviour of an algorithm, even under some simplifying assumptions, or serve as a bounding case.

We define the **distance** of a load sharing (LS) algorithm from a baseline as a percentage:

$$\text{distance}(\textbf{LS},\textbf{baseline})=100\times\frac{\text{response\_time}(\textbf{baseline})-\text{response\_time}(\textbf{LS})}{\text{response\_time}(\textbf{baseline})}$$

A positive distance indicates the improvement achieved under the assumptions made. A negative distance signals a degradation in performance. Algorithms being compared must use

the same models for hardware configuration, network and cpu delays, and operate under the same loads. Before proceeding any further, we discuss the main costs involved.

Cost Modelling

Associated with each activity, i, related to load sharing is a basic execution time cost B(i). A cost has to be assumed for initial placement, and a much higher one for migration after start-up. Similarly, there are also cpu costs incurred every time an observation of the local state is taken, and whenever any preprocessing is performed (e.g average over a sliding window). RATE(i) represents the rate, per unit of time, at which an activity is invoked. This cost is a function of the hardware configuration (system size, cpu speed, etc.), and also reflects the sophistication of the activity. Some of these activities may involve message exchange and a component waiting for another to respond (e.g. initial placement which involves negotiation). The following simplified cost function C(i) is used in our model:

$$C(i)=B(i)+(\text{MESSAGE-DELAY} \times \text{CYCLES}(i))$$

CYCLES(i) is the average number of message exchange cycles during an invocation of activity i and MESSAGE-DELAY is a given parameter. When there is no message exchange involved we have C(i)=B(i).

Load sharing activities 'steal' cpu time otherwise available for application processing. As a result they directly effect application performance. However, we have tried to keep the cost function as simple as possible, while still capturing the main contributors to delays. For objective evaluation of different algorithms we need to be able to state the overhead per unit of time for each contributor to the total overhead (algorithm invocation, state collection etc.). The overhead per unit of time for activity (i) is given by:

$$\text{overhead}(i) = C(i) \times \text{RATE}(i)$$

The aggregate overhead per unit of time is given by:

$$\text{aggregate overhead} = \sum_{i=1}^{n} C(i) \times \text{RATE}(i)$$

Related to the costs mentioned above is the cost of information transfer. The model should be flexible enough to permit different hardware configurations, such as the availability of communication co-processors or their absence and use of diskless nodes as well as those equipped with local disks. Diskless nodes, like the assumption of dedicated file servers, indicate that the costs of accessing program and data files are roughly the same for all of the nodes [32]. The other extreme, of stations equipped with local disks or shared file servers (i.e.

nodes providing both file service and application processing activities), imply that the costs of accessing remote programs are much higher than local ones [25].

After using the performance measures to confirm that one or more approaches to load sharing will improve the overall performance of the system, the following efficiency measures can be used to check which of the specific requirements are satisfied. The efficiency measures also allow for trade-offs to be made when comparing different load sharing algorithms with comparable performance.

Efficiency Evaluation

Efficiency measures *quantify* the overhead or cost associated with the attainment of a specific level of performance in terms of the following:

- memory requirements for algorithm constructs,
- state dissemination cost – rate of load sharing state messages exchanged per node,
- % remote execution (initial placement or migration after start-up),
- run-time cost (in terms of cpu time) measured as the fraction of time spent running the load sharing component.

Finally, as mentioned before, we use

- **hit ratio**, the ratio of successful decisions such that the node requested to accept additional work is capable of doing so. This measure provides guidance as to the *quality* of the decisions made.

We have adopted the term 'hit' from virtual storage, where it means that the page is available in memory, whereas a 'miss' and the consequent page fault results in a much more costly operation. In the load sharing context, a hit means that remote execution can proceed, and a miss means additional overhead before remote execution can progress, or wasted resources in case it is too late to stop it. Both can be costly.

Evaluation of General Algorithm Properties

These performance and efficiency measures support evaluation with respect to the required general algorithm properties described in Section 2.

Overhead - The efficiency measures listed above permit assessment of the run time cost induced by an algorithm, in terms of memory, cpu and communication bandwidth. To be able to make trade-offs, we calculate, for example, the cost per unit of time of running an algorithm.

This measure can be used to test the sensitivity to different values of the basic costs involved {B(1)...B(n)}, design choices made and communication delays. One can also check for which values the overhead exceeds a predefined percentage of cpu permitted to be 'stolen' from the application components. The other efficiency measures can be used in a similar way.

Adaptability - To show adaptability the algorithm under study (LS) should be tested under a wide range of load patterns, but with the same overall load. We expect the performance measures, under all these different loads, to be within the allowed bounds:

$$\text{distance(LS, no cooperation)} \geq T_1, \quad \text{distance(LS, random)} \geq T_2,$$

where T_1, T_2 are predefined minimums (such as T_1=30%, T_2=10%). For instance, for ZhouD we obtain the performance results in Table 2.

	response-time	distance
ZhouD	1.76±0.035	not relevant
no-cooperation	3.81±0.08	53%
random	1.85±0.04	5%
ideal	1.53±0.03	-13%

Table 2: ZhouD - Performance Measures (Even load with 10ms delay)

Stability - This can be evaluated as

%remote execution, which should be bounded and restricted, say < 10% , and

hit-ratio, say > 0.90

For stability purposes remote execution decisions should be initiated at a low rate (to limit %remote execution decisions) and concluded successfully (high hit ratio). This agrees with Zhou's conclusion that only a small percentage of the components need to be executed remotely in order to achieve effective load sharing [31]. For ZhouD, the efficiency results are given in Table 3. Therefore, we have efficiency which is better than that of random, but still needs to be improved.

	%remote exec.	hit-ratio	overhead
ZhouD	21.29	0.77	0.006
random	32.6	0.52	0.002

Table 3: ZhouD - Efficiency Measures (Even load with 10ms delay)

Scalability - This is measured by scaling up some resources while others remain untouched. To show scalability, the algorithm should be tested for different system sizes (with the same relative load per node), node ordering, processor speeds and network delays (effected by communication bandwidth and resource congestion). Our goal is to have performance measures within allowed bounds. Insensitivity to scale is indicated if:

$$\frac{\text{performance(LS(scalability-parameter=A))}}{\text{performance(LS(scalability-parameter=B))}} = 1 \pm \alpha$$

where $\alpha << 1$, and A>>B. Scalability parameters can be system-size, node ordering, communication bandwidth, etc. The expected variation is represented by α. We expect the variation resulting from simulation to be negligible. Stability is a precondition for scalability.

Note that in order to show that an algorithm does not hold a specific property, it is enough to show that the relevant conditions are not satisfied for one particular case. Unfortunately, to show that a property does hold, we have to show that the relevant conditions are satisfied for *all* possible cases of loads and system sizes, which is, of course, impossible. Hence we merely ask for reasonable assurance over a wide range of cases.

We believe that these definitions of performance and efficiency capture the most significant characteristics of load sharing algorithm behaviour, and provide guidelines where trade-offs are required. They also equip us with the means of showing that required properties hold in practice, by showing that the conditions are satisfied for selected cases of load, system-size, processor speeds and node ordering.

4. MORE EXAMPLES

This section illustrates the evaluation method using a number of well known adaptive load sharing algorithms for dynamic initial placement of applications. All algorithms are fully distributed and designed to be adaptive, fault-tolerant and incur low overhead, but differ in some of the other design choices made. They all use the same load state measure, queue length, and the primary goal is to minimise the average response-time in the system. The algorithms selected for analysis are well known and widely referenced. They include early works by Stankovic [25] and Eager [11] and more recent ones by Zhou [32]. In particular, Stankovic's work was one of the first studies of adaptive load-sharing algorithms, offering new insights into the problem and suggesting novel solutions. Many important issues like processing and communication overheads and their effect on performance and stability were first addressed in this paper. The algorithm published in [25] was selected as it sets an example for good scientific practice by publishing all input data thereby enabling other researchers to test and

replicate the results. (Suggestions for improvements were published by the same author in [26]). *Note that, to some extent, the assessment is thus unfair as the algorithms were not originally designed to meet all requirements,* and that there exist more recent algorithms that can potentially produce better performance results. Nevertheless they do provide good examples for describing and illustrating our assessment method.

A simulation tool is used for performance and efficiency evaluation [16]. The efficiency and performance measures described in the previous section are available in the simulator to permit objective evaluation and comparison of load sharing algorithms. The simulation model and associated tool were validated against some analytic models (queueing theory models). In addition, the results of published load-sharing algorithms were reproduced.

Assumptions

The following assumptions are used throughout our analysis.

Applications are processed in first-come-first-served order and are run to completion. A system is first analysed subject to an even load on its nodes (Table 4). The simulated system comprises 5 node-types. Node types aid in the description of systems comprising a large number of nodes. Each node type is characterised by the number of nodes of this type, cpu demands, arrival rates and application size distribution functions. For a larger system all we need to do is multiply the #nodes as appropriate.

	#nodes of this type	**CPU requested**	**arrival rate**	**resulting intensity per node**
node-type 1	**1**	exponential(0.5s)	poisson(0.153)	76.5%
node-type 2	**1**	exponential(0.7s)	poisson(0.125)	87.5%
node-type 3	**1**	exponential(0.6s)	poisson(0.143)	85.8%
node-type 4	**1**	exponential(0.5s)	poisson(0.143)	71.5%
node-type 5	**1**	exponential(0.7s)	poisson(0.125)	87.5%
overall load				81.76%

Table 4: Even load (71.5-87.5%) case

We also study the case of uneven or extreme load imposed on the system (Table 5). Both cases have the same overall load.

	#nodes of this type	**CPU requested**	**arrival rate**	**resulting intensity per node**
node-type 1	**4**	exponential(0.7s)	poisson(0.144)	99.3%
node-type 2	**1**	exponential(0.8s)	poisson(1.075)	11.6%
overall load				81.76%

Table 5: Uneven load (11.6-99.3%) case

Each node in the system has a communication co-processor, and is also equipped with a hard disk. Delay in the network is modelled as a function of information size - the size of the information to be sent divided by packet size (1K bits) times the average delay per packet. Application size is taken from the distribution function described in Table 6.

percentile	size	percentile	size	percentile	size	percentile	size
10	1000	60	16000	85	22000	98	38000
20	12000	70	18000	90	30000	99	44000
40	14000	80	20000	95	34000	99.5	50000

Table 6: Application size distribution

Packet delay (preparation (packaging), transmission and reception (unpackaging)) is assumed to be 10ms, and the basic cost per invocation for these very simple algorithms is assumed to be 10ms. These algorithms involve internal load state observation. The cost associated with such observation or pre-processing of state information is assumed to be 10ms. Note that when conducting such a study, the ratio between the requested service time and various delays is more important than the absolute numbers [13]. Also, load assumptions are drawn from exponential distributions which are not always realistic [32]. Nevertheless, from this study we are able to draw some important conclusions.

In his study of load-sharing algorithms, Zhou assumes that the work to be executed is attached to the request message, and can be executed immediately upon arrival [32]. We make the same assumption.

Description of the Algorithms Under Study

With the algorithm proposed by Stankovic [25], termed ST, each node periodically calculates an estimate of the number of applications at each node in the system, and sends this information to the nodes to which it is directly connected (direct neighbours). State information is updated as follows whenever a state update message is received. Load estimate of direct neighbours is updated according to the latest information received. For non-direct neighbours, an average over the estimates in all incoming messages is taken. This state vector represents the node's view of the system state. The decision making (control) function is activated periodically, and also whenever the local scheduler is about to select an application for execution (following a completion event). For this algorithm, applications will not be remotely executed when the system is very lightly or very heavily loaded. For moderate loads in the system, each node compares its own load to its estimate of the load of the least loaded node. The difference between the loads of these two is then compared to a bias. If the difference is

less than the bias, the application is executed locally; otherwise, one application is moved to the least loaded node for remote execution, using a single-request protocol.

The third algorithm studied is the THRHLD algorithm, which is taken from Zhou's study [32]. It implements the Threshold algorithm suggested by Eager, Lazowska, and Zahorjan [11]. In our study, this algorithm is termed EagerT. In EagerT, each server keeps track of a single-valued load metric for itself only (again, queue length). The transfer policy is the same as that of ZhouD (except that Eager's original algorithm does use cpu based eligibility). The location policy employed selects a random node which is requested to accept additional load. A request-reply protocol is employed. The potential server performs the same threshold decision, in order to decide whether to accept the additional load or not (acceptance policy). This can be repeated up to a specified limit.

Conclusions from Earlier Studies

Some important results drawn from these studies are that extremely simple load-sharing algorithms, i.e. those that collect small amounts of state information and that use this information in simple ways, yield dramatic performance improvements relative to the no cooperation case; that special concern should be given to the effect of occasional poor decisions that will inevitably be made and may lead to instability; and that the overhead resulting from applying these policies and the state information method employed may negate the benefits of an improved load-sharing algorithm. An interesting result from Zhou's study [Zhou88] is that the impact of immobile jobs on load-sharing is found to be less serious than the immobility factor might suggest: most of the performance gains are still retained even when up to 50% of the jobs eligible for remote execution are immobile (must execute locally). Zhou also found that the performance of all hosts, even those originally with lighter loads, improve under effective load-sharing. This is somewhat counter intuitive, but very encouraging: by cooperating with each other, no node seem to lose**.** Zhou also claims that a low percentage of remote execution (initial placement or migration after start-up) is enough to realise the benefits of load-sharing.

Qualitative Analysis

Stankovic's algorithm ST is general and simple but computation time, communication and memory overhead are system size dependent. Like ZhouD, it does not try to limit remote execution activity for scalability and stability purposes[2]. Stankovic uses a request-only protocol

[2]We recognise that neither of these algorithms was designed to be independent of system size. This criticism is intended only to illustrate the assessment method.

which does not allow the destination to reject additional load when it is unable to accommodate it. Each can lead to poor performance and efficiency.

Unlike the other algorithms, ST uses (unreliable) indirect state information, on a comparative basis, when decisions are to be taken. It also uses periodic decision making. Both can result in instability. The other algorithms base their transfer policies on local and accurate information only (state information related to the node itself). This is expected to contribute to their stability.

ST and ZhouD are capable of making local decisions. By doing so they obviate the need to go through a, possibly lengthy, negotiation process before a decision is taken. If the state information available locally is of high quality (it represents the real state of the nodes), this can contribute to stability and scalability since nodes become less dependent on current delays in the system.

ZhouD can be viewed as a slight improvement of ST. The transfer policy is based only on local accurate information. Also, only direct state information is used. But the algorithm still uses a request only form of protocol. We anticipate high percentages of fruitless decisions which cannot be reversed (migrated applications arriving at overloaded nodes). Each of the issues raised can lead to instability. ZhouD is expected to result in high %movement and low hit ratio. Worse performance and efficiency are expected from ST.

With EagerT, computation time is not dependent on the system size. The request-reply protocol employed allows for rejection of additional work. On the other hand, for each decision to be taken, a node goes through a multi-phase probing process. In the case of long delays resulting from congested or slower resources, this can degrade performance and efficiency. Also, by employing a random-based location policy, Eager limits applicability of his algorithm. In his study, Zhou employs Eager's algorithm for non-homogeneous loads. We do the same.

Note that the state metric used by all algorithms (queue length) does not have an absolute well understood meaning. This and other information policy related issues are not discussed further in this paper.

The main design choices relevant to our study are summarised in Table 7. We now turn our attention to a quantitative analysis of the above algorithms. Whenever possible, algorithms use the same parameter settings as recommended in the original papers.

	ST	ZhouD	EagerT
decision making invocation	basic method: periodic and also event driven (application completion)	event driven (application arrival)	event driven (application arrival)
transfer policy	local and indirect non-local information on a comparative basis	local information only (threshold)	local information only (threshold)
location policy	least loaded	least loaded	random
acceptance policy	single request - no rejection allowed	single request - no rejection allowed	request-reply to allow rejection (threshold)
information policy	periodic state dissemination	periodic state dissemination	probing (request-reply) upon application arrival

Table 7: Design choices summary

Quantitative Analysis

In this section we first discuss the selection of algorithm parameters before proceeding with the comparative analysis.

ZhouD - Threshold Setting

Before applying the evaluation method described above we run Zhou's algorithm was run for a few cases. Only the load threshold and cpu threshold parameters of the transfer policy were allowed to vary. Estimation of the cpu requirements in an application is not always possible. Furthermore, it appears that comparable performance can be achieved by setting the cpu threshold to zero and selecting a load threshold of a higher value since both tend to restrict remote execution activities.

	Response-time	%remote exec.
Tq=1,Tcpu=500ms	1.92±0.04	20.10
Tq=1, Tcpu=0ms	1.79±0.03	40.1
Tq=2, Tcpu=0ms	1.76±0.03	30.7
Tq=3, Tcpu=0ms	1.81±0.05	20.5
Tq=4, Tcpu=0ms	1.87±0.05	15.2

Table 8: Threshold setting

The performance achieved (Table 8) confirms our hypothesis regarding eligibility. We get comparable results for $T_{l=1}$,$T_{cpu=500ms}$ and for $T_{l=3}$, $T_{cpu=0}$.

Note that even when parameters are tuned for one set of conditions, they may not be the best under different conditions. Another possibility is to dynamically change parameter settings to reflect the dynamics of the system. This may not be a good approach because of the demands in storage and computation time that may be imposed on the system. We tune some of the parameters for the basic case (10ms delay) when even load is imposed on the system. Note that

different parameter settings could be attained under different delay or load intensity assumptions. In any case, our main goal is to show the significance of some design choices and their effect on the general behaviour of an algorithm rather than the absolute performance results themselves.

Both ZhouD and EagerT result in good performance under T_l=3. For fair comparison we select the same T_l threshold value (queue length) to be used by both, with T_{cpu} set to zero. Information dissemination period is 500ms and Δ is set to 1 (as in Zhou's study).

EagerT - Probe Limit

	response-time	%remote exec.	hit-ratio	overhead
probe limit=1	1.78±0.035	16.4	0.60	0.006
probe limit=2	1.85±0.036	18.5	0.44	0.008
probe limit=5	1.81±0.036	22.1	0.36	0.010

Table 9: Probe limit setting

For EagerT, we use a probe limit of 1. Even under these favourable conditions (delay of 10ms) repeated probing does not seem to improve performance and at the same time it is harmful to efficiency (Table 9).

Stankovic's Algorithm - Periodic Decision Making

For the ST algorithm, we use a bias of 2 (as recommended in his study) and the time between invocations is set to 800ms.

ST uses periodic decision making which is hard to tune. We run the algorithm with more frequent decision making (every 500ms instead of 800ms). By invoking the algorithm more often, performance is still in the same range, although slightly degraded (Table 10). The degradation in efficiency is more significant. It is clear that periodic decision making is hard to tune when there is no prior knowledge and can itself be a source of instability.

	response-time	%remote exec.	hit-ratio	overhead
decision making=800ms	1.86±0.037	36.37	0.77	0.028
decision making=500ms	1.90±0.038	43.9	0.59	0.036

Table 10: ST with Frequent Invocations

Analysis under Favourable Conditions (10ms delay)

The results achieved under the assumptions made are compared to the three baselines: **no cooperation**, **random** and **ideal**. All algorithms give a significant improvement in

performance over no cooperation (>50%, Table 11) but not all significantly improve random (>10%) even under these favourable conditions. As expected, these algorithms suffer from poor efficiency. Results were achieved with a confidence interval of 90% and 1-3% accuracy (ususally 2%).

	response-time	%remote exec.	hit-ratio	overhead
STankovic	1.86±0.037	36.37	0.77	0.028
ZhouD	1.76±0.035	21.29	0.77	0.006
EagerT	1.78±0.035	16.4	0.60	0.063
no-cooperation	3.81±0.08	not relevant	not relevant	not relevant
random	1.85±0.04	29.5	0.56	0.007
ideal	1.53±0.03	not relevant	not relevant	not relevant

Table 11: Favourable conditions case (10ms state processing)

Analysis with Congested Resources / Different Service Speed Ratios (Less Favourable Conditions)

Poor efficiency and its implications on performance becomes more evident when conditions are less favourable. We first run the random algorithm, to see the implications of delays on its performance and some of the efficiency measures (Table 12). From the results achieved, we can see that even a simple random algorithm, based on a threshold transfer policy can achieve significant performance reductions without any state information exchange. We now run the algorithms studied with longer network-delays (25ms, 100ms and 150ms) resulting from congested resources. Response-time results are compared to those achieved for the same delay assumption. All other cost assumptions are left unchanged.

	response-time	**%remote exec.**	**hit-ratio**	**overhead**	degradation vs. 10ms
random-10ms	1.85±0.04	29.5	0.56	0.007	not relevant
random-25ms	1.96±0.04	29.5	0.55	0.012	-6%
random-100ms	2.37±0.05	27.4	0.58	0.053	-28%
random-150ms	2.43±0.05	25.3	0.62	0.056	-31%

Table 12: RANDOM under Different Delays

	response-time	**%remote exec.**	**hit-ratio**	**overhead**	degradation vs. 10ms	distance from random
ST - 10ms	1.86±0.037	36.37	0.62	0.028	not relevant	0%
ST - 25ms	1.98±0.040	38.44	0.56	0.038	-6%	0%
ST - 100ms	2.46±0.050	37.05	0.47	0.089	-32%	-3.7%
ST - 150ms	2.75±0.028	35.60	0.44	0.132	-48%	-13%

Table 13: ST under Different Delays

We can see from Table 13 that under the assumptions made ST does not improve performance if compared to random. Under a different set of assumptions, ST may exhibit better performance than random. We actually had to use a higher cost for state processing than that used for the other algorithms, but to be able to make an unbiased judgement we use the same overhead assumption. ST exhibits poor efficiency. On top of the other reasons for instability, we also have the use of indirect information on a comparative basis.

We now try ZhouD for various delays. With ST and ZhouD (Table 14), performance degradation is mainly the result of excessive fruitless migration, which now costs more. If compared to ST, ZhouD exhibits an improvement of both performance and efficiency. Unlike ST, all other algorithms base their transfer policies on local and accurate information only (state information related to the node itself). This contributes to their stability, although with insufficient effect.

	response-time	**%remote exec.**	**hit-ratio**	**overhead**	degradation vs. 10ms	distance from random
ZhouD - 10ms	1.76±0.035	21.29	0.77	0.006	not relevant	5%
ZhouD - 25ms	1.81±0.036	21.89	0.73	0.010	-3%	8%
ZhouD - 100ms	2.18±0.042	23.00	0.62	0.035	-19.3%	8%
ZhouD - 150ms	2.29±0.046	21.80	0.57	0.054	-30%	6%

Table 14: ZhouD under Different Delays

	response-time	**%remote exec.**	**hit-ratio**	**overhead**	degradation vs. 10ms	distance from random
EagerT - 10ms	1.78±0.036	16.4	0.60	0.063	not relevant	2.7%
EagerT - 25ms	1.97±0.039	17.5	0.54	0.0032	-9%	0%
EagerT - 100ms	2.26±0.043	16.3	0.54	0.040	-21%	4.6%
EagerT - 150ms	2.49±0.050	16.9	0.54	0.068	-38.3%	-2.5%

Table 15: EagerT under Different Delays

With EagerT, remote information is required by the location policy (Table 15). Performance degradation results mainly from the negotiation used. EagerT with probe limit set to 1 can be viewed as a modified random algorithm which allows for rejection. Compared to Random, this is expected to improve results but is perhaps balanced by added state observation costs, which seem to degrade performance when there are longer delays. EagerT exhibits very low hit ratio which is a result of the random location policy used. As the variation in loads on different nodes grows, this is expected to degrade even more.

When we assume short delays (10/25ms), the random algorithm gives performance which is very close to that of the load sharing algorithms studied, in spite of its poor efficiency. This

could be the result of unrealistic assumptions. In any case, we would like to see less activity related to remote execution and better decisions.

Modified Algorithm

We modify the algorithms to try and handle some of the sources of instability mentioned above. This is done by selecting the "best" of each world. Our aim is to show the significance of some design choices made rather than suggest a specific algorithm. In our modified algorithm, we use Zhou's information (state dissemination) and location policies together with the transfer and acceptance policies suggested by Eager. Nodes are capable of making local decisions. The request-reply protocol allows extra work to be rejected and work in transit to be taken into account. Thus, the efficiency measures of the modified algorithm are expected to improve. Note however that the information policy is left unchanged and all nodes are still updated implying that the algorithm is dependent on system size and again not scalable. The main design choices are made as described in Table 16.

	Modified
decision making invocation	event driven (application arrival)
transfer policy	local information only (threshold)
location policy	least loaded
acceptance policy	request-reply to allow rejection (threshold)
information policy	state dissemination

Table 16: Modified - design choices

The modified algorithm was run with the same load threshold, update period and Δ parameters (T_l=3, P=500ms, Δ=1), and produced the results given in Table 17. Note that for the basic case (10ms) performance is slightly degraded (still in the same range), but there is a dramatic improvement in the efficiency measures (see Table 17 and Figure 6) which convinces us that this is a better approach. The modified algorithm is more stable. This becomes evident when looking at the results for longer delays (100ms, 150ms). The degradation in response time, when compared to the basic case (10ms), is much slower than it was for any of the other algorithms. Thus, the modified version is less sensitive to the assumptions made. The importance of using a request-reply protocol allowing for rejection of work and of making local decisions should not be underestimated. Through state dissemination a node can have the information available when needed. Both have significant contribution to stability and scalability.

	response-time	%remote exec.	hit-ratio	overhead	degradation vs. 10ms	distance from random
Modified-10ms	1.90±0.038	14.16	0.82	0.015	not relevant	-2.7%
Modified -25ms	1.95±0.039	14.42	0.83	0.018	-3%	0%
Modified -100ms	2.08±0.040	12.15	0.85	0.044	-9.4%	12.3%
Modified-150ms	2.12±0.040	10.05	0.85	0.062	-11.5%	12.8%

Table 17: Modified algorithm (Eager+Zhou)

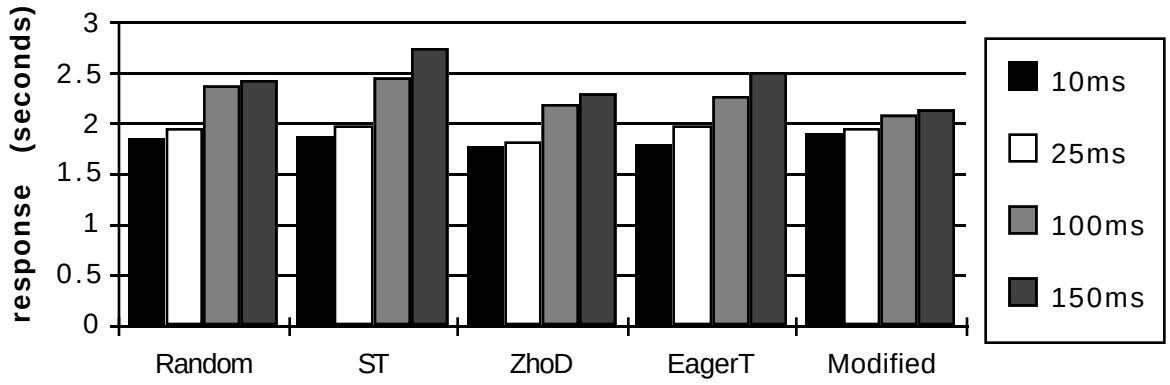


Figure 6: Performance (response-time) under various network delays

Uneven Load

We also indicate the effect of different load intensities imposed on individual nodes with the same overall load for the two cases. Table 18 summarises the performance results for 10ms and 150ms delay assumptions. These results are compared to those achieved under even load (Fig. 7). All algorithms adapt to changing load intensities with some (ZhouD and modified) exhibiting smaller variations under these conditions.

	response-time-10ms delay	response-time-150ms delay
STankovic	2.13±0.040	2.98±0.061
ZhouD	1.83±0.037	2.62±0.052
EagerT	2.33±0.047	2.96±0.087
Modified	2.06±0.062	2.39±0.072

Table 18: Uneven Load (11.6-99.3%)

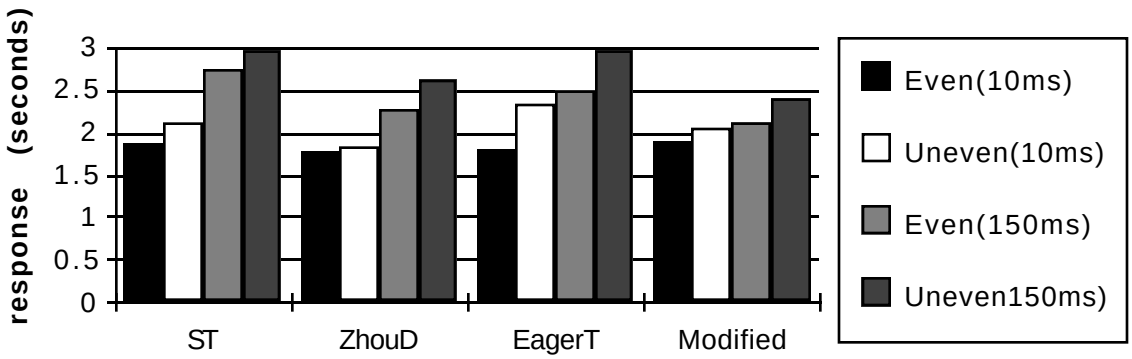


Figure 7: Performance (response-time) under different load intensities and delay assumptions

We now turn our attention to the effect of system size on those of the algorithms which were designed to be independent of system size.

System Size

It is clear that ZhouD, ST and the Modified algorithms are not scalable. They all hold state information related to all the nodes in the system. Some use broadcast for state dissemination and others use a scheme which is topology dependent. In particular, algorithms which use a single request protocol (ST and ZhouD) experience noticeable degradation of performance and efficiency as the system size grows, since more remote execution will be tried.

EagerT is independent of system size and under the same delay assumption we get comparable results under different system sizes (Table 19). Note however that the negotiation before a decision is made implies sensitivity to resource characteristics (Fig. 7) which violates the scalability requirement.

system size	response-time	%remote exec.
5 nodes	2.26±0.045	16.3
10 nodes	2.18±0.043	18.1
15 nodes	2.17±0.022	19.2

Table 19: EagerT with 100ms delay

We have assumed the availability of a communication co-processor and therefore the effect of broadcast is to overload the communication channels (which can be seen in the state dissemination rate per node) but it is not accounted for in the performance measure.

5. CONCLUSIONS

Several algorithms from the literature were selected and evaluated using the suggested analysis method. We have shown how these measures can be used to critically examine load sharing algorithms and suggest improvements.

Stability and scalability related issues were studied more carefully. The algorithms were analysed under favourable and less favourable conditions. Activities related to remote execution should be restricted to a small proportion of the activity in the system (application arrival), and an algorithm should minimise the number of incorrect decisions and prevent them from proceeding. To this end, a request-reply rather than single-request protocol ought to be used. Eligibility based on command type or any other cpu requirement estimation, as suggested by some other researchers [6, 27, 32], is not generally feasible. Eligibility restricts remote execution related activities and consequently contributes to stability and scalability. The same effect can be achieved by using a more restrictive transfer policy.

Decision-making issues such as periodicity and thresholds were also examined. By adjusting them, it has been shown that efficiency measures can be improved. Also, nodes should be capable of making local decisions, the quality of which depends largely on the quality of the available information. To achieve further improvements, the information dissemination policy should therefore also be evaluated. The state metric, which is also a source of instability, should have an absolute well understood meaning. The load metric can be preprocessed before transmission to filter out transient, insignificant state changes. This can be achieved by taking the average over a window or mapping the load metric into a small set of n values (n-state representation). If we have two algorithms giving performance results within the same range, we would prefer an algorithm with a low %remote execution and high hit-ratio, since it is less sensitive to assumptions made.

Recent work on a Flexible Load Sharing algorithm (FLS) [17] uses both of the above load state metric techniques, i.e. calculate the average utilisation U (also a function of queue length) over a window, and map it into one of three possible load states: $\begin{cases} O & \text{- Overloaded if } U>T_o \\ U & \text{- Underloaded if } U<T_u \\ M & \text{- Medium load otherwise.} \end{cases}$

The new state metric has a well understood meaning and, as suggested by Alonso, uses two thresholds T_o and T_u to allow sharing even when nodes are slightly used [1]. Event driven update is used as the basic information policy, together with slow periodic update to account for failures. This is in contrast to Alonso who still uses Eager's random location policy. These modifications result in significantly improved efficiency measures. For scalability purposes, the system is partitioned into overlapping domains (a domain is simply a group of nodes). Domain membership changes dynamically as nodes are independently and mutually included or

excluded from their respective domains. Biased random selection is used to retain nodes of interest in its domain (an Overloaded node retains an Underloaded one) while replacing others by random selection from the rest of the system. Further work is still required to assess and confirm the current promising results in practice.

REFERENCES

[1] R. Alonso, L. L. Cova, "Sharing Jobs Among Independently Owned Processors", *Proc. of the 8th Int. Conf on DCS*, pp. 282-287, 1988 IEEE.

[2] A. Barak, A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer", *Software-Practice and Experience*, 15(9), pp. 901-913 1985.

[3] A. Barak, Y. Kornatzky, "Design Principles of Operating Systems for Large Scale Multicomputers", *Proc. of the International Workshop on Experience with Distributed Systems* ,Kaiserslautem, FRG September 1987, LNCS 309, Springer-Verlag, pp. 104-123.

[4] A. Barak, "The Evolution of the MOSIX Multi-Computer UNIX System",Technical Report 89-17, Hebrew university of Jerusalem, September 1989

[5] P. Birman, T. A. Joseph, "Exploiting Virtual Synchrony in Distributed Systems", *Proc. ACM-SIGOPS 11th Symp. Operating Systems Principles*, pp. 123-138, Nov. 1987.

[6] L. F. Cabrera, "The Influence of Workload on Load Balancing Strategies", *in Proc. summer USENIX Conf.*, pp. 446-458, June 1986.

[7] T. L. Casavant, J. G. Kuhl, "A Formal Model of Distributed Decision-Making and its Application to Distributed Load-Balancing", *Proc. of the 6th Int. Conf on DCS*, pp.232-239, 1986 IEEE.

[8] T. L. Casavant, J. G. Kuhl, "Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements", *Proc. of the 7th Int. Conf on DCS*, pp.185-191, 1987 IEEE

[9] T. L. Casavant, J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Trans. on Soft. Eng.*, vol SE-14, No.2, pp. 141-154, February 1988.

[10] T. L. Casavant, J. G. Kuhl, "Effects of response and stability on scheduling in distributed systems", *IEEE Trans. on Soft. Eng.*, Vol. SE-14, no. 11, pp.1578-1588, November 1988.

[11] D. L. Eager, E. D. Lazowska, J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Engineering*, 12(5), pp. 662-675, May 1986.

[12] D. L. Eager, E. D. Lazowska, J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing", *Proc. ACM Sigmetrics*, pp. 63-72, May 1988.

[13] K. Efe, B. Groselj, "Minimizing Control Overheads in Adaptive Load Sharing", *Proc. of the 9th Int. Conf on DCS*, pp. 307-315, 1989 IEEE.

[14] D. Ferrari, S. Zhou, "A Load Index for Dynamic Load Balancing", *Proc. Fall Joint Computer Conf.*, Dallas, Texas, ACM-IEEE, pp. 684-690, Nov. 1986.

[15] J. Kramer, J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", *IEEE Transactions on Software Engineering*, 16(11), pp.1293-1306, Nov. 1990.

[16] O. Kremien, J. Kramer, J. Magee, “Rapid Assessment of Decentralized Algorithms”, *Proceedings of the 7th Int. Conf on Computer Systems and Software Engineering (CompEuro 90)* , Israel, May 1990, pp. 329-335.

[17] O. Kremien, J. Kramer, "Flexible Load Sharing in Configurable Distributed Systems", *Proc. of IEE Int. Workshop. on Configurable Distributed Systems*, London, U.K., March 1992, pp. 224-236.

[18] P. Krueger, M. Livny, "The Diverse Objectives of Distributed Scheduling Policies", *Proceedings of the IEEE Int. Conf on DCS*, IEEE, pp. 242-249, 1987.

[19] P. Krueger, M. Livny, "A Comparison of Preemptive and Non-Preemptive Load Distributing", *Proceedings of the 8th IEEE Int. Conf on DCS*, IEEE, pp. 123-130, 1988.

[20] A. Kumar, M. Singhal, T. L. Ming,"A Model for Distributed Decision Making: An Expert System for Load Balancing in Distributed Systems”, *Proc. of the 11th Symp. on Operating Systems*, IEEE, pp. 507-513, 1987.

[21] M. Livny, M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems", *Proc. of the Conf. on Performance* ACM, pp. 47-55, 1982.

[22] J. Magee, J. Kramer, M. Sloman, "Constructing Distributed Systems in Conic" *IEEE Transactions on Software Engineering*, 15(6), pp. 663-725, June 1989.

[23] R. Mirchandaney, D. Towsley, J. A Stankovic, "Analysis of the Effects of Delays on Load Sharing", *IEEE Transactions on Computers*,C-38(11), pp. 1513-1525, Nov. 1989.

[24] J. M. Smith, "A Survey of Process Migration Mechanisms", *ACM Operating Systems Review*, Vol. 22, No. 3, pp. 28-40, July 1988.

[25] J. A. Stankovic, "Simulations of Three adaptive,Decentralized Controlled, Job Scheduling Algorithms", *Computer Networks* 8, pp.199-217, August 1984.

[26] J. A. Stankovic, "Stability and Distributed Scheduling Algorithms", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, pp. 1141-1152, October 1985.

[27] A. Svensson, "History, an Intelligent Load Sharing Filter", *Proc. of the 10th Int. Conf on DCS*, IEEE, pp. 546-553, 1990.

[28] M. M. Theimer, K. A. Lantz, "Finding Idle Machines in a Workstation-Based Distributed System", *IEEE Transactions on Software Engineering*, 15(11), pp. 1444-1458, November 1989.

[29] Y.-T Wang, R. J. T. Morris, "Load Sharing in Distributed Systems", *IEEE Transactions on Computers* c-34(3), pp. 204-217, March 1985.

[30] S. Zhou, “An Experimental Assessment of Resource Queue Lengths as Load Indices”, *Proc. USENIX Winter Conference*, Washington D.C., pp. 73-82, Jan. 1987.

[31] S. Zhou, D. Ferrari, “A Measurement Study of Load Balancing performance”, *Proc. of the 7th Int. Conf on DCS*, IEEE, pp. 490-497, 1987.

[32] S. Zhou, “A Trace-Driven Simulation Study of Dynamic Load Balancing”, *IEEE Transactions on Software Engineering*, 14(9), pp. 1327-1341, September 1988.