University of Leeds

**SCHOOL OF COMPUTER STUDIES**

**RESEARCH REPORT SERIES**

**Report 97.17**

# Simulation and Prototyping of a Coherent Distributed Dynamic Load Balancing Algorithm

by

**Mourad Kara**

May 1997

**ABSTRACT**

This paper presents the performance evaluation of a coherent distributed dynamic load balancing algorithm called the DGP (Decentralised Global Plans) algorithm. The performance evaluation is underpinned by a research methodology integrating a simulation-based performance evaluation with a prototype-based performance evaluation. The DGP algorithm addresses the issue of coherence and coordination in distributed dynamic load balancing algorithms. The DGP algorithm is based on a strategy called Global Plans (GP). This new approach first explicitly specifies a global goal and then design a strategy around this global goal such that each scheduler (1) takes into account local decisions made by other schedulers; (2) takes into account the effect of its local decisions on the overall system and (3) ensures load balancing.

A simulation-based performance evaluation of DGP with two other well-known dynamic load balancing algorithms published in the literature shows that DGP performs consistently better. More significantly, the results indicate that the global plan approach provides a better framework for the design of distributed dynamic load balancing algorithms. A DGP-based prototype has been developed using a distributed programming model which supports dynamic configuration. The overall prototype has been implemented and tested on a network of Sun workstations. The results from the prototype-based performance evaluation are consistent over a range of experiments and clearly demonstrate the benefits gained from load balancing. An evaluation of both simulation and prototyping results indicated that both set of results were consistent providing a high level of confidence in the results and their applicability.



**Keywords:** dynamic load balancing algorithms, coherence and coordination, research methodology, simulation, prototyping


## 1 Introduction

Over the past decade, dynamic load sharing and balancing algorithms in distributed computing systems have been an active area of research [12, 41, 4, 38]. Dynamic load balancing algorithms have been based on the following observation; in a network, it is likely to find idle hosts while there are jobs queueing for execution in other hosts. Hence, it would be advantageous to move jobs from heavily loaded hosts to idle or lightly hosts. An early study by Livny and Melman in 1982 has been undertaken to confirm this hypothesis [32]. Their work has shown that there is indeed a high probability of finding within a distributed system some hosts idle while others have jobs queueing for execution. More recently, Rommel investigated the probability of load balancing success in distributed systems [40]. Subsequently, these findings suggested that load balancing is likely to benefit job response times. Several studies have also been performed on network of workstations to assess, in practice, the potential of idle workstations [36, 31, 35, 47], and some these studies found that even during daytime only 30% of the total network-wide computation capacity is used.

Distributed-controlled dynamic load balancing algorithms are known to have several advantages over centralised algorithms such as scalability, and fault tolerance [48]. Distributed implies that the control is decentralised and that a copy of the algorithm (called a scheduler) is replicated on each host of the computer network. Hence, a Distributed Dynamic Load Balancing Algorithm (DDLBA) is composed of a set of cooperating schedulers. However, distributed control also contributes to *the lack of global goals and lack of coherence*. Most of distributed dynamic load balancing algorithms (DDLBAs) use strategies that take good local scheduling decisions and the effect on the global system may not be positive. For example most load balancing algorithms select the least loaded host for remote execution. The only measure of a good decision is the impact on the local host, regardless of the consequences of its actions on the global system. The selection of the least loaded host [50, 43] or random host [12, 34, 8] for remote execution may lead to job thrashing or host overloading, requiring extraneous work to counter the limited scope for decision making.

The DGP algorithm (Decentralised Global Plans) addresses the issue of coherence and coordination in distributed dynamic load balancing algorithms [18]. The DGP algorithm is based on a strategy called Global Plans (GP), and aims at maintaining all computational loads of a distributed system within a band called $\Delta$. The rationale for the design of DGP is to allow each scheduler to consider the actions of its peer schedulers. With this level of coordination, the schedulers can act more as **a coherent team**. This new approach first explicitly specifies a global goal and then design a strategy around this global goal such that each scheduler (1) takes into account local decisions made by other schedulers; (2) takes into account the effect of its local decisions on the overall system and (3) ensures load balancing.

Most research studies in performance evaluation of dynamic load balancing algorithms have taken the form of either simulation or prototyping (or to a lesser extent mathematical modelling with basic assumptions, e.g. [34]). Examples of simulation-based research are numerous ([43, 12, 7, 27]) and prototyping too ([3, 48, 31, 42, 51]). However, very little has been done in combining the simulation and prototyping data analysis.

This paper presents the performance evaluation of the DGP algorithm. The approach taken for the performance evaluation of DGP is to explicitely specify the research methodology through combining modelling/simulation and prototyping under one framework. This implies an attempt to reason about the results from both simulation and implementation and validate.

The simulation-based performance performance evaluation of the DGP algorithm is presented by comparing DGP with two well known algorithms from the literature Disted [50], and algorithm #2 [43] (called DJS2 in this paper). In general, the results of the experiments were consistent over the range of experiments, and it was found that DGP performed better than DJS2 and Disted in improving response time as well as load balancing.

A prototype-based performance evaluation of the DGP is presented. The design paradigm adopted is the dynamic configuration model [25]. The performance experiments had to be conducted with clearly defined constraints for using a network of workstations: i.e. that not the whole network would be available for experiments, and we also wanted some flexibility in starting/stopping new schedulers without disrupting the overall prototype. This has been possible by using a dynamic configuration model as the programming paradigm for the experiments. An evaluation of both simulation and prototyping results indicated that both set of results were consistent providing a high level of confidence in the results and their applicability.

Next Section defines the research methodology adopted for the performance evaluation of the DGP algorithm. Sections 4 and 5 address the modelling & simulation-based , and the prototype-based performance evaluation respectively. Section 6 provides the comparative analysis of the two approaches and section 7 provides the conclusions, and future directions.

## 2 The research Methodology

As mentioned above, research studies and papers in performance evaluation of distributed dynamic load balancing algorithms contain very little in terms of validating simulation with prototyping. One of the reasons we felt it important to add a separate section on the research methodology is to express this concern and explicitly outline the methodology we adopted. This performance methodology provides a higher level of confidence by undertaking similar experiments (through the same objectives), and also undertaking a corresponding level of data reasoning from simulation and prototyping. This section outlines the research methodology used in the performance evaluation of the DGP algorithm. The methodology can be illustrated through the figure (Figure 1) below:

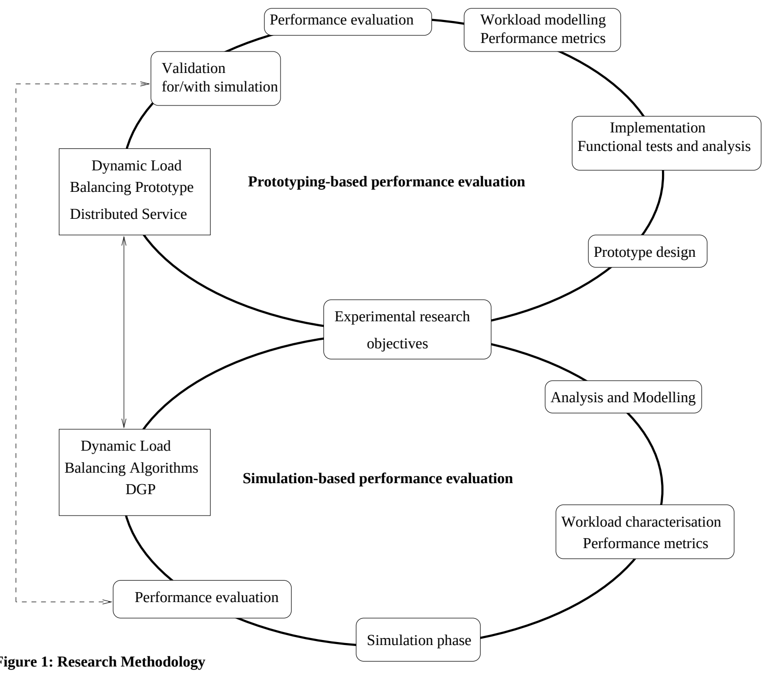



**Figure 1: Research Methodology**

There are two main coupled design cycles in the figure above. The bottom cycle covers analysis of the algorithms in a controlled environment usually using performance tools such as simulators. The top cycle covers prototyping of these algorithms (once tuned in the simulation domain). The feedback loop is a validation process from prototyping to simulation. The experimental research objectives phase is identical for the simulation and prototyping and is the starting point. The next two subsections briefly outline the entries in the figure.

## 2.1 Simulation-based performance evaluation

*Experimental research objectives(empirical).* This is the phase in which the objectives of the research are translated into a set of research experiments.

*Analysis and modelling.* This phase provides an analysis of the problem to be tackled, and extract only those features that are essential to the reasoning of the algorithm (modelling). This phase also defines the system model assumptions and characteristics in which the algorithm evolves.

*Workload characterisation.* This phase aims to formulate and quantify the input to the algorithm and model. Usually, performance metrics are measured by varying different scenarios of workloads.

*Performance metrics* are the key metrics identified by the experimental design phase for the data analysis and performance evaluation.

*Simulation phase.* In this phase, the model, its workload, performance metrics and environment (scenarios, configuration) are put together usually through a discrete-event simulation technique. important components of this phase are calibration, verification, validation and conditions for simulation completion as well as error analysis.

*Performance evaluation.* This phase undertakes data collection and analysis of the results obtained from simulation. It discusses and evaluates the results obtained. This phase should check with the "validation for/with simulation" phase of prototyping.

## 2.2 Prototyping-based performance evaluation

*Prototype design (distributed services).* In this phase, the prototype design, application programming interfaces (API), application-level protocols between the schedulers, programming platforms, as well as the network are developed and agreed upon.

*Implementation.* This phase takes the design and makes the selection for the tools used for the implementation (e.g. Sun lightweight processes, RPCs, sockets, fifo pipes etc..)

*Functionality tests and analysis.* These are basic tests to evaluate the functionality of the prototype. Scripts and basic processes can be created to load the service, and also error management etc..

*Workload modelling.* This is a complex phase, and implies building generic processes from which the distributed workload can be quantified and generated.

*Performance metrics.* This are the output parameters that are collected and used to assess the implementation. In prototyping, workload modelling and measurements are more complex to define, and measure.

*Performance Evaluation.* Identical phase to simulation.

*Validation for/with simulation.* The results obtained from simulation and prototyping should not necessarily be the same, (in fact unlikely), but that a proper analysis as to the difference between the results obtained and possible extrapolation should be done. What is important is to be able to reason about the results (data analysis) and obtain similar conclusions, and if this is not the case, then we should be able to identify the reasons for it. (proper analysis as to the difference between the results obtained).

# 3 The DGP Algorithm Modelling and Simulation

## 3.1 Coherence and coordination

Coherence and coordination have been widely applied to the analysis and design of distributed problem solving [6, 11, 10], however, there is still some confusion as to the difference between these two abstractions. Distributed Problem solving considers how the work of solving a particular problem can be divided among a number of agents that cooperate at the level of dividing and sharing the knowledge about the problem. A dynamic load balancing algorithm can be modeled as a set of self-interested agents (i.e. schedulers) with the ability to communicate when it is to their mutual advantage.

Coherence refers to the properties of the total distributed problem solving system, i.e. how well the system behaves as a unit, along some dimensions of evaluation such as performance and efficiency [10]. The degree of coordination refers to interaction properties among a set of agents performing some collective activity. The degree of coordination exhibited among a collection of agents is the extent to which they can avoid extraneous work activity (e.g indirect activity of synchronising and aligning their tasks) in achieving their primary ends. This extraneous work is called *articulation work* . Effective coordination implies some degree of mutual predictability and lack of conflict. Coordination does not necessarily imply cooperation since antagonist agents can be coordinated such as in legal proceedings. Good local decisions do not necessarily contribute to good global goal behaviour because good local decisions may have unfortunate global effects. The primary difficulty in establishing coherence and coordination is the attempt to achieve them without centralised control [6].

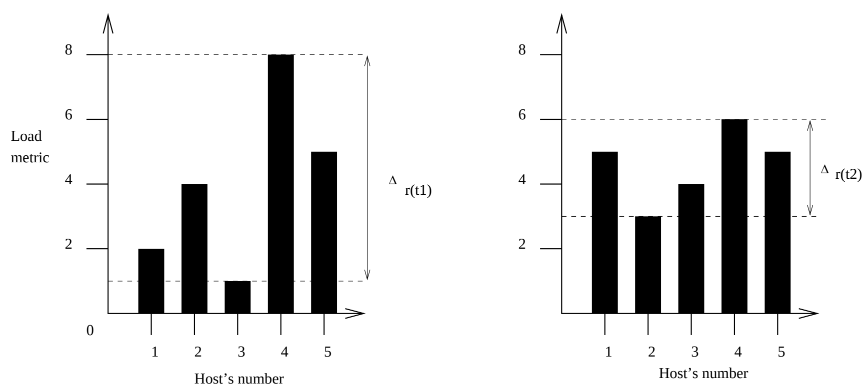
Host overloading occurs when several hosts target the least loaded host, which in turn becomes heavily loaded [50]. This process may restart again by targeting the new least loaded host. Most hosts spend their time in redistributing jobs and little on executing local jobs. Hence several jobs are continuously moving around the network before execution. This leads to job thrashing where jobs infinitely move around the network [12]. For example, Eager *et al.* [12] have shown that the uncontrolled random strategy is unstable, and that whatever the average load, it is guaranteed that eventually the system will reach a state in which the hosts are spending all their time to transferring tasks and none of their time to processing them. To prevent instability, Eager *et al.* provided a limit to the number of transfers of a host Tf. Zhou [50] tackled these issues by sending at most one task per period of time, whereas Stankovic [45] provides a mechanism by which if a host selects the least loaded, it won't select it for a period of time, even if the same host is found to be the least loaded again. Another mechanism developed by Stankovic and Mirchandaney to improve stability is priority encoded associative memory and was applied on algorithms based on [44] and also algorithms based on Stochastic Learning Automata [33].

## 3.2 The Global Plan

The Global Plan (GP) strategy is applied to a network of hosts. The rationale for this strategy is to develop *a global goal* that achieves load balancing and that is shared by all hosts of the network. Each host considers the actions of its peer hosts in the network. With this level of coordination, the hosts can act as a more coherent team. This new approach first explicitly specifies *a global goal* and then design a *strategy* around this global goal such that each host (1) takes into account local decisions made by other schedulers; (2) takes into account the effect of its local decisions on the overall system and (3) ensures load balancing. The global goal is defined as being to maintain the different loads of the network within a band, the smaller the band the more balanced is the network; and the larger the band the less balanced is the network. Under this assumption, the goal to be achieved by GP is as follows:

*Global goal:* A network of hosts is *balanced* if the difference in load between the highest and lowest loaded node is less than $\Delta$, (where $\Delta$ is constant, and has the same unit as the load of the hosts). Furthermore, $\Delta_{r(t)}$ is defined as the minimum band that contains all loads at a time t. In other words, a system is balanced at a time $t$ if $\Delta_{r(t)} \leq \Delta$.

To illustrate this definition, assume two snapshots of a network with five hosts were taken at time $t_1$ and $t_2$. Figures 2 (a) and (b) represent a network of five hosts at two different times, $t_1$ and $t_2$ respectively. The figures show the individual loads of the hosts. For example, at $t_1$, host 1 has a load of 2 and host 4 has a load of 8. Assuming that $\Delta = 3$, figure 2(a) shows an unbalanced state, because $\Delta_{r(t_1)} = 7$ and therefore $\Delta_{r(t_1)} > \Delta$. Figure 2(b) depicts a balanced system since all loads belong to the band $\Delta$ or since $\Delta_{r(t_2)} \leq \Delta = 3$.




(a) unbalanced system

(b) balanced system

Figure 2: System states

The Global Plan strategy aims to satisfy the global goal discussed above. The idea is to ensure that all loads are within $\Delta$ to maintain the loads within the network balanced. Another factor in the design of GP is to avoid targeting least loaded hosts, as discussed in previous sections, as this leads to instability and host overloading. The GP strategy has four elements :

1. A set of input loads $X = (x_1 \; .... \; x_n)$, where $x_i$ represents the load of the host i, and n the number of hosts of the system.

2. A parameter $\Delta$, $\Delta > 1$. $\Delta = 0, 1$ results in an infinite loop. $\Delta$ is the unique internal parameter of the GP strategy, and GP's algorithm attempts to maintain all the loads within this band.

3. A set of output loads $Y = (y_1 \dots y_n)$ such that

$$\max_{i=1}^{n} y_i - \min_{i=1}^{n} y_i < \Delta$$

   Where $y_i$ is the new load of the host i after the execution of the table of global allocations. the vector Y is a transformation of X by the GP strategy. Y contains the new loads of the system after the execution of the table of allocations.

4. A table of global allocations that is represented by a set of triples T = $\{(p, q, r)$, where is p in units of load, and q,r are hosts $\}$. The triple $(p, q, r)$ results in an allocation of p units of load from the component q to the component r.

The GP strategy outputs a set of loads $y_i$ (i= 1, n) that range within an interval $\Delta$, given an input set of loads $x_i$. The allocations are performed by *incremental* allocations from components with maximum loads to components with minimum loads. if there are several components with the same maximum or minimum load, then the one with the lowest index is selected. The algorithm for the GP strategy is given below. It consists of a loop exiting when all the loads are within $\Delta$. For each iteration, the allocations are recorded in the table T:

## The Global Plan (GP) strategy ($\Delta$)

**X : input set of loads (loaded from the load information process).**
**Y : output set of loads.**
**T : table of allocations (initialised to an empty set).**
$y_k$ **corresponds to the load of host k,** $1 \leq k \leq n$ **.**
**l represents the host with the lowest load,** $1 \leq l \leq n$**.**
**h represents the host with the highest load,** $1 \leq h \leq n$**.**
**n : Number of hosts in network.**
**Y = X;**
**Compute l, h;**
**while** ($y_h$ - $y_l \geq \Delta$ )
  {
  **(1).Search in T for an entry that matches the triple (p, h, l), for any p.**
  **if (search is successful)**
      **then Increment triple by 1 : (p+1, h, l)**
      **else Insert new entry (1, h, l)**
  **(2). Decrement** $y_h$**; Increment** $y_l$**;**
  **(3). Compute l, h;**
  }

### 3.3 A Distributed algorithm

Typically, a load balancing algorithm has three types of policies:

1. **A load information policy.** This determines the type of information that makes a load index, and the way such information is communicated to other schedulers (for example, broadcasting, focussed addressing, polling).
2. **A transfer policy.** This determines whether a job should be executed locally or remotely based on the local load conditions.
3. **A placement policy.** This determines the method to find a suitable host for remote execution.

The Global Plan (GP) strategy combines the transfer and placement policies. The dynamic load balancing algorithm discussed in this paper is based on the GP strategy, is called Decentralised Global Plans (DGP), and is composed of a network of cooperating schedulers. These latter operate asynchronously, and maintain their own table of allocations. Each cooperating scheduler is made up of the three communicating processes described above and operates as follows. The load information process is based on the periodic broadcasting policy. Periodically (with a period $I_p$), each scheduler broadcasts its local load to the other schedulers. The transfer and placement processes are implemented by the Global Plans (GP) strategy. The GP strategy is executed periodically with a period $S_p$. The load metric used in DGP is based on resource (CPU) queue length. Zhou [50] has shown that algorithms using load index based on resource(s) queue length performs better than those based on CPU utilisation. Kunz [29] also investigated various load indices and reached a similar conclusion to Zhou's.

The application of the GP strategy to a distributed algorithm leads to several considerations. The algorithm based on the GP strategy is called Decentralised GP (DGP) because of the distributed nature of the environment in which GP is applied. Each host of the network system has only control over its own resources. The distributed nature of the network implies that no single host has a true image of the overall state (i.e loads) of the system. Load information are sent to the other hosts and hence is dated information. The schedulers are asynchronous, and maintain their own table of load information.

After the execution of GP, each scheduler extracts only those allocations that are related to itself from the table of allocations. Recall that each scheduler has its own table of allocations. This means that for a host *i*, it executes all entries that include $i$ as a source (*, $i$ , *), if the algorithm is source initiated. For a server initiated policy, a host $i$ would search for all entries that include $i$ as a sink (*, *, i).

Due to the distributed nature of the DGP algorithm (as with all DDLBA), there are two main sources of errors: *asynchrony, and dated information.* In [21], we have assessed the effect of dated information on response time and job movement by simulating an ideal algorithm (PRF_DGP) which runs the DGP algorithm with an accurate global view of the system. A comparison with

DGP has indicated that the dated information has a low impact so long as $I_p$ remains low (between 1 and 5 seconds under moderate load).

It is clear from the formulation of the global goal and the incremental iterations of GP that each scheduler responds to the three key points put forward to ensure coherent cooperation:

- *DGP takes into account local decisions by other hosts:* This is due to the fact that allocations are taken from the table of allocations, and this latter has been produced by taking into account of all other local decisions from the other hosts.
- *DGP takes into account the effect of its local decisions on the overall system:* Again this is due to the fact that a global scheduling took place and distribution of the allocations is carried out.
- *DGP ensures load balancing:* This is due to the global goal. To quantify this statement, a performance metric is devised, and used for performance evaluation, that assesses the fluctuation of the system load. This metric gives an idea to the extent of achieving the load balancing objective.

DGP is a dynamic load balancing algorithm (versus load sharing). It is worth pointing out the distinction with load sharing. These two common objectives of distributed scheduling (load sharing and load balancing) tend to be confused. A load balancing algorithm attempts to maintain the loads of the hosts balanced by redistributing some jobs to the lightly loaded hosts. A load sharing algorithm differs in that it does not go as far as balancing the system but just make sure that there is no idle host while a job is queueing in another host.

The DGP algorithm is based on non-preemptive scheduling. Preemptive scheduling in distributed systems, known also as *process migration* [9] involves a considerable amount of computation when transferring a process and all its environment (such as virtual address space, cached writable files) from one host to another. Furthermore, it is difficult to implement on Unix systems, and it is even more complex for jobs that generate several processes, because inter or intra communications between processes have to be addressed explicitly.

## 3.4 Other dynamic load balancing algorithms

This section presents the two other algorithms used for evaluation purposes (Disted and DJS2). They share the same information policy as DGP, i.e. periodic broadcasting (with a period $I_p$), and their transfer and placement policies are also executed periodically (with a period $S_p$). Since DGP is an algorithm for transfer and placement policies, it is appropriate to to apply the same information policy for the three algorithms and differ the transfer and placement policies. These two algorithms are also known to perform well compared to other algorithms [50, 43]. Therefore, results will reveal performance differences in transfer and placement strategies.

*Disted* is a threshold based algorithm [50]; it checks its local load against a threshold Tr, and if the local load is greater, then it sends one job to the least loaded host. We have implemented a variant whereby instead of sending one job, a parameter called $M_{jm}$ (for Maximum Job Movement) is used. This parameter sets the number of jobs to be sent (i.e. $M_{jm} = 1$ is the default).

*DJS2* has two parameters *bias1* and *bias2* [43]. If the local load is greater than *bias1*, then send one job to the least loaded host, and if the local load is greater than *bias2*, then send two jobs to the least loaded host.

For the purpose of comparison, another algorithm was implemented, NoLB (No Load Balancing). This is an algorithm based on an n*(M/M/1) model (Kendall's notation [13]). Each processor works in an independent manner and communication between processors is not permitted. This algorithm is implemented for comparison purposes and should reflect the lower bound on performance.

### 3.5 Workload modelling

In this study, the workload is represented by two dimensions: intensity ($I_{wkl}$), and pattern ($P_{wkl}$). The intensity is the average percentage of CPU used on all hosts of the network, and the pattern is defined by the standard deviation of the amount of CPU used on all hosts of the network. In [21] experiments were carried out to assess the sensitivity of $P_{wkl}$ over a range of $I_{wkl}$, and it was found that the effect of $P_{wkl}$ was greater for larger $I_{wkl}$. Using Mirchandaney definition [34], a workload is of type 2 if $P_{wkl} \neq 0$. The workload table for $I_{wkl}$ used in the sets of experiments is shown in table 1, and was modified from Zhou [50].

The default workload used for all experiments in this paper is moderate ($I_{wkl} = 0.70$, $P_{wkl} = 0.28$). The pattern of the workload was selected not to be too high to overemphasise the benefit of load balancing while not being realistic. On the other hand, $P_{wkl}$ was not chosen too low as to hide the effect of load balancing. The default number of hosts is set to 10, the workload corresponds to 1 idle, 1 lightly loaded, 2 moderately loaded and 6 highly loaded hosts (1I, 1L, 2M, 6H). This led to a pattern of 0.28. A detailed analysis of the workload can be found in [21].

| Workload | Range |
|---|---|
| Idle (I) | 0-10% |
| Light (L) | 15-45% |
| Moderate (M) | 50-70% |
| Heavy (H) | 75-99% |

Table 1: Workload characterisation.

### 3.6 Performance indices

The aim of a load balancing algorithm is to provide a better response time by balancing the computational load, thus reducing the likelihood of jobs waiting on a host while there are other hosts idle. There are two separate issues: *load balancing*, and *response time.* Most research *implicitly* assumes that by means of good load balancing, it is likely that good response time will be obtained [14, 37]. We define performance indices that distinguish between the objective of balancing the load of the system and the objective of improving response time.

*Load balancing.* Assuming that the sole objective of a load balancing algorithm is to balance the computational load of a distributed system, then it is necessary to define a metric that measures the quality of load fluctuations between hosts. A good indicator should reflect the fluctuations between the different loads of the hosts. One such indicator is the standard deviation of the loads of the hosts taken at a time t ($\sigma_{ldb}(t)$). Load balancing may be identified by a two dimensional metric: the intensity of load balancing ($I_{ldb}$), and the pattern of load balancing ($P_{ldb}$). $I_{ldb}$ is the mean of $\sigma_{ldb}(t)$ over time (i.e an experimental session), while $P_{ldb}$ is the standard deviation of $\sigma_{ldb}(t)$ over time. $I_{ldb}$ indicates the fluctuations of all hosts over a period of time, and $P_{ldb}$ measures the steadiness of these fluctuations.

*Response time.* The response time $R_t$ is represented by the mean job response times. A job response time is the sum of its waiting time in the distributed system and its execution time.

*Overheads.* There are two types of overhead involved with running load balancing algorithms: the execution overhead, and the communication overhead. The execution overhead includes the time required for the execution of the algorithm. This overhead is known to count for a small percentage of the overall overhead [48]. Communication overhead is concerned with message transfers between hosts. Another type of cost is the Percentage of Wrong job Movements ($P_{wm}$) This is the ratio between the number of wrong job assignments over the number of job transfers. A job is wrongly transferred if the host to which it is sent is already overloaded at the time of receipt. $P_{wm}$ provides a metric on wrong decisions in the scheduling process. Table 2 provides a summary of the performance indices that are used for the experiments.

| Indices | Definitions |
|---|---|
| $I_{ldb}$ | intensity of load balancing |
| $P_{jm}$ | percentage of job movements |
| $P_{ldb}$ | pattern of load balancing |
| $P_{wm}$ | percentage of wrong job movements |
| $R_t$ | response time |

Table 2: Performance indices

### 3.7 System modelling

The model used for this study is an extension and a modification of the m*(M/M/1) distributed system model proposed by Krueger, Livny and Melman [32, 28]. Jobs arrive independently at each host and join an external queue. The arrival rate of each stream of tasks follows a Poisson distribution. The service time demand of the tasks has a negative exponential distribution. This description (based on queueing theory) has been used as the common approach in most simulations for scheduling in distributed systems [7, 12, 37, 43].

The network model used for this study displays some independency of the topology and is based on a mathematical model (similar to Stankovic's [43] that computes communication at the software level. It is defined as the size of the message to be sent (in Kbytes) times the average delay per Kbytes, this latter being a parameter of the simulation. For the distributed system model which is based on diskless workstations with file servers, there is not much difference between Zhou's communication overheads [50] and Stankovic's network overheads [43] since the job transfers and load information transfers involve the same message sizes respectively. Therefore, whether the overheads are fixed or based on a mathematical formula with fixed size messages, the result is the same as far as overhead assumptions are concerned. The choice made was of Stankovic's approach to distinguish communication protocol overheads from the network transmission overhead. In the communication model, each host has two queues, an input and an output queue (FCFS). The input messages are received from the network and depending on whether they are load information or job messages, they are sent to the scheduler or to the external queue respectively. The output queue submits messages to the network.

### 3.8 Simulation tools

Simdsa is a simulation software that was developed for the analysis of the dynamic load balancing algorithms [19]. Simdsa aims at simulating the environment for testing distributed scheduling algorithms by taking into account the criteria and model outlined in the previous section. Simdsa was designed using the discrete-event simulation approach [1] and was implemented in the C language under the Unix operating system on Sun4 workstations. Simdsa offers a menu-driven program, as well as a library of C functions for simulation programming. Two other software tools were built to support Simdsa: *expgen*, and *simdbm*. Expgen (for automated experiment generator) is a translator which enables visual specifications of sets of experiments to be converted into a program written in Simdsa. Simdbm (for simulation database manager) is a network-based simulation manager that manages, schedules and checkpoints [31] several executions of simdsa on a network of workstations.

There are several ways to determine default values assigned to parameters used for simulations (calibration phase), such as experimental determination, assumptions based on other research work, or educated guesses based on heuristic or mathematical calculations. Some values assigned to parameters in this simulation are determined by experiments, such as the algorithm's internal parameters. Overheads for message packing and unpacking have a value of 20ms. Zhou [48] conducted experiments on the effect of message overheads for similar environments to the one described in the previous section, and he concluded that load balancing algorithms are quite insensitive to the overhead assumptions so long as these are not too high. The 20ms is based on measurements done by Zhou on a load balancer prototype and are accepted as being realistic. Algorithm overheads are assumed to be 50ms, a figure based on investigations by Stankovic [43] on the effect of algorithm overhead on performance. The network average delay was determined from experiments by Theimer and Lantz on local area networks as 12.5 Kbytes per second [30, 46]. They provided this figure based on practical calculations of efficient broadcast and multicast communications. They assumed that at most 1% of the bandwidth would be used by the scheduling algorithms. In

Stankovic's experiments, the average delay of 4 seconds seemed unrealistic with current hardware performance. The table (Table 3) below indicates default values of simulation parameters.

| Parameters | Values |
|---|---|
| Number of hosts | 10 |
| Scheduling Period | 2s |
| Information period | 2s |
| Network delay | 12.5 Kbytes/s |
| Message overhead | 20 ms |
| Algorithm overhead | 50ms |

Table 3: Default parameter values of simulation

There are several ways to terminate experiments using simulation such as specifying the the timelength, the sample size, or the maximum error level attained. The latter is the most appropriate because results found are within equivalent boundaries, providing consistency in the interpretation of the results. Steady-state simulation of queueing processes are known to require several runs (or replications) to reach an acceptable level of confidence [39]. In this simulation, each experiment is executed for several runs. The precision error level is fixed at 5% at a 90% confidence interval. Each run is executed until all the performance indices reach a maximum error level of 5%. Then, independent runs are executed until the averages of the performance indices over these runs reach a maximum error level of 5%. This method of multiple runs is called *the method of replications* [24].

### 3.9 Experiments

This section presents in detail the experiments performed on the three algorithms using the simulation approach. The comparison of the three algorithms is carried out using three sets of experiments:

1. Determination of the algorithms' internal parameters.
2. Effect of the scheduling period ($S_p \approx I_p$) on performance.
3. Effect of scalability on performance, (i.e Varying the number of hosts).

*Experiment 1.* The first experiment finds the optimum value for their internal parameters (given a workload [1]). These values are then retained and used for rest of the experiments. The values tested range from 0 up to a value where no further improvements by these algorithms over NoLB is noticed.

*Experiment 2.* The aim of this experiment is to determine the effect of the scheduling period on the load balancing decisions, and hence performance. The scheduling period is the frequency

[1] *though in [21], we found these internal parameters were fairly robust across a range of workload couples in a diskless environment.*

of activation of a scheduler on a host (transfer and placement components). The selected three algorithms are run under different values of the scheduling period (in seconds), ranging from 0 up to a value where no further improvements by these algorithms over the NoLB algorithm is noticed.

*Experiment 3.* This experiment has been carried out to support the default number of hosts selected (10) rather than address explicitly the issue of scalability. The algorithms are tested with a number of hosts varying from 1 to 40 hosts. It is important to emphasize that *the same workload* (0.70, 0.28) is roughly maintained for the different number of hosts selected (for number of hosts $\geq 2$). This is important to maintain consistency in performance experiments.

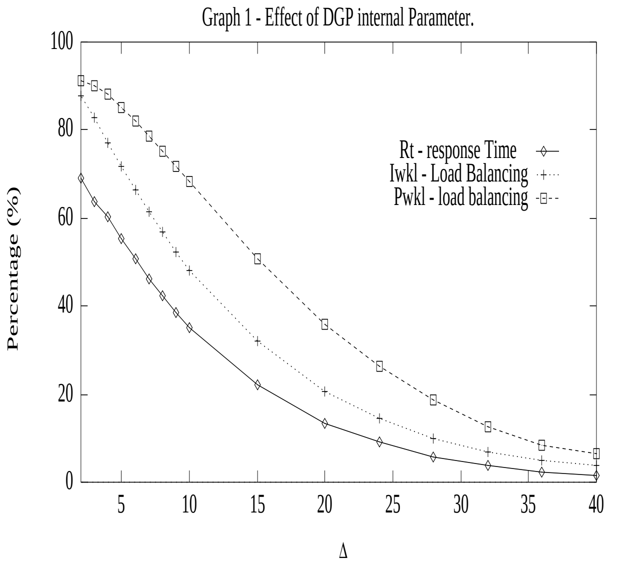
For all three experiments, the NoLB algorithm is run. The performance indices of the distributed dynamic load balancing algorithms presented in the results (graphs) are represented in terms of the *relative performance* over the NoLB algorithm. For example, the relative performance (percentage improvement) of the intensity of load balancing ($I_{ldb}$) of DGP over NoLB is

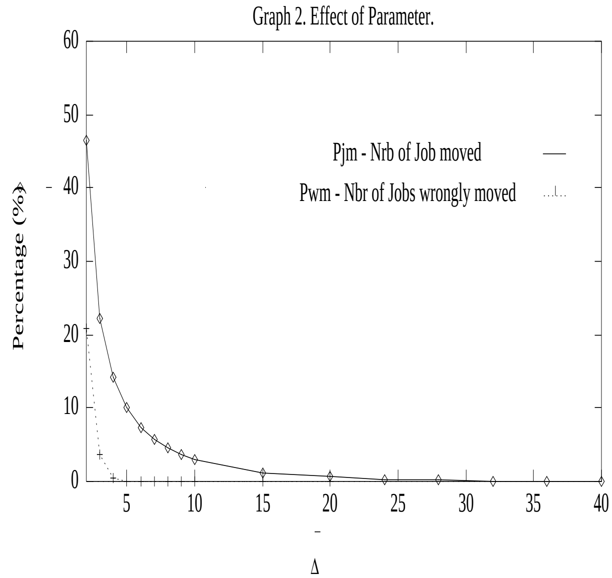
$$\frac{I_{ldb}(DGP) - I_{ldb}(NoLB)}{I_{ldb}(NoLB)}$$

## 3.10 Results

*Experiment 1.* Graphs 1 and 2 represent respectively the results for the DGP algorithm. $\Delta$ has been varied from its minimal value 2 up to a value after which little or no further improvement is obtained. All the curves in these graphs exhibits the same shape, they decrease steadily as $\Delta$ increases. The best improvements in $R_t$, $I_{ldb}$, and $P_{ldb}$ are achieved for $\Delta$ at its minimum ($\Delta$=2). For this same value, the highest cost in terms of $P_{jm}$, and $P_{wm}$ is also obtained. The decline in the cost (graph 2) is sharper than for the performance indices (graph 1). In graph 1, the curves of $P_{ldb}$ is always above $I_{ldb}$ curves.

The DGP algorithm periodically attempts to maintain the load of all hosts within $\Delta$ by moving jobs around the network following the GP strategy. When reducing $\Delta$, it is expected to improve load balancing. This experiment confirmed this fact, and indicated that this improvement implies improvement in response time $R_t$. Furthermore, these improvements follow the same shape indicating a strong impact of load balancing on response time. On the other hand, when decreasing $\Delta$, a larger number of movements of jobs is required to maintain all the loads within $\Delta$, and a larger number of wrong job transfers follows too due to the outdated load information and the asynchrony of the schedulers. At the other extreme, for $\Delta$ very large, the band is so large that prior to load balancing, most of the loads are already within $\Delta$. Hence, the DGP has not much improvement to perform, and so converges to NoLB's behaviour.




The experiment 1 enabled the determination of the best values of DGP internal parameters. Similar experiments were undertaken for Disted and DJS2 algorithms. Best values for DJS2 were biais1 = 2 and Biais2 = 4, and for Disted $M_{jm} = 2$. In all cases, it was found that internal parameters have an impact on performance and costs. However, compared to disk based systems [43] these parameters seemed less sensitive. This is mainly related to the low cost of transferring jobs in a diskless environment.

Other interesting results may be drawn for all tests in this experiment:

- $R_t$, $I_{ldb}$, and $P_{ldb}$ curves have the same shape.
- $P_{ldb}$, and $I_{ldb} \geq R_t$
- Sensitivity of internal parameters in diskless environment.

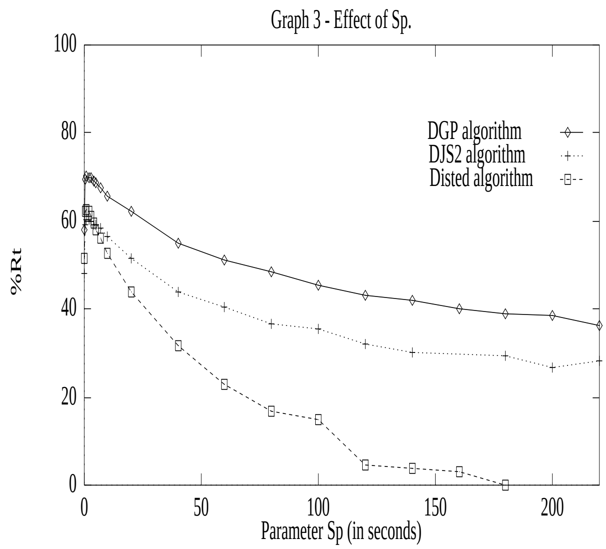
From these results, two conclusions can be drawn:

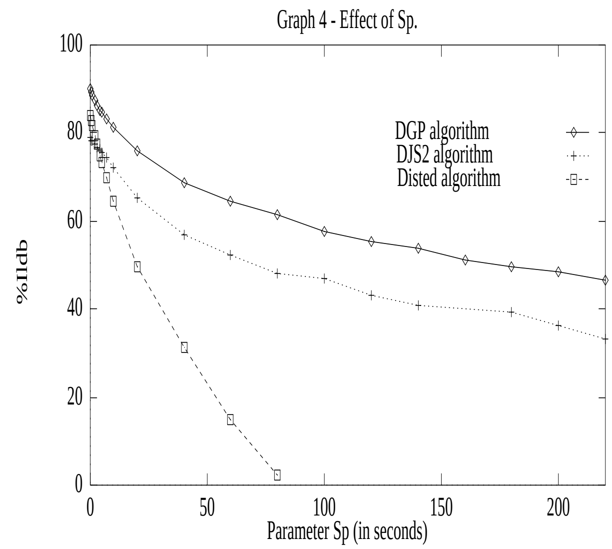
1. *Load balancing and response time relationships.* It can be seen from the graphs that improvements in $I_{ldb}$, $P_{ldb}$ lead to improvements in $R_t$. More importantly, their curves exhibit the same pattern underlining the strong impact of load balancing on response time $R_t$.

2. *Explicit Performance indices.* This experiment also indicated that the work on performance indices and the distinction between response time and load balancing carried out in section 5.2 was vital to identify the objectives and the derived objectives of the algorithms. Furthermore, it can be seen by the fact that $P_{ldb}$, and $I_{ldb} \geq R_t$, that the algorithms first balance the system and then improve response time.

*Experiment 2.* Graphs 3 and 4 depict the results for the performance indices $R_t$ and $I_{ldb}$ respectively. The rest of the performance indices are depicted in graphs A.1 to A.3 in Appendix

A. The experiment has been run for $S_p = 0$ up to $S_p = 200s$ where no further improvement of the performance indices over the NoLB algorithm was noticed. The $R_t$ exhibits an independent curve with two separate intervals $0 < S_p < 0.5$, and $S_p \geq 0.5$. In the first interval, the $R_t$ increases sharply with $S_p$. In the second interval, a slight decrease in response time is noticed. The remaining performance indices roughly follow the same shape and decrease slowly.

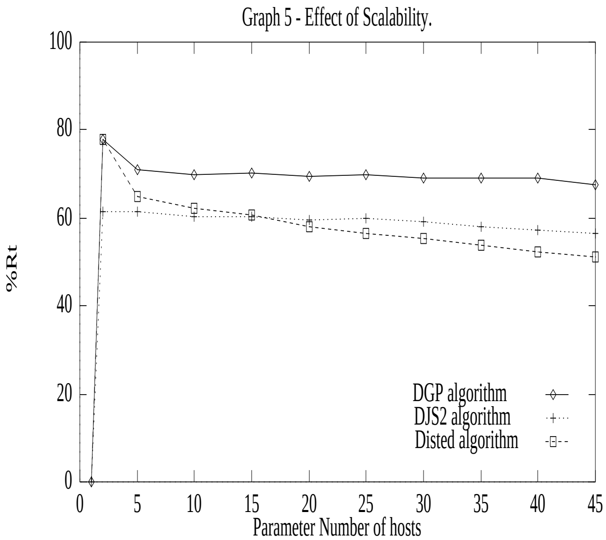


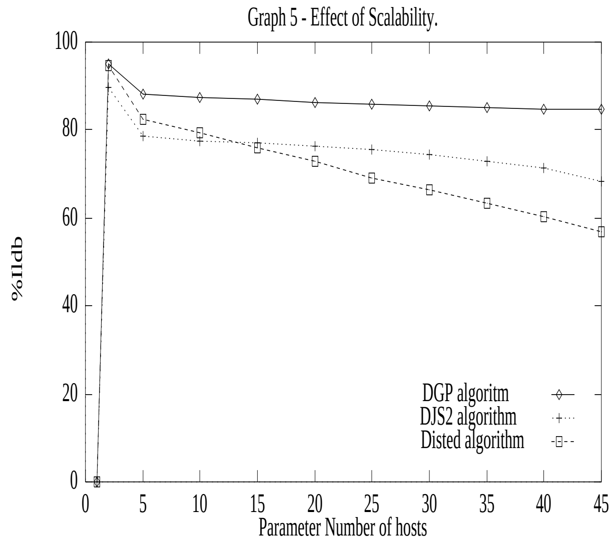




It can be seen from graph 3 that DGP provides a better improvement in response time and load balancing than Disted and DJS2 for the whole experiment. The $P_{jm}$ and $P_{wm}$ seem reasonably low for DGP because it has no upper limit parameters on the number of job movements. Disted has a lower number of job movements because of its parameter $M_{jm}$, on the other hand this prevents from doing good allocations when needed. The point of DGP is that it has the freedom and effectively it performs the allocation needed to balance the system, even though there is the potential of two sources of errors (dated information and asynchrony).

There is a clear trade-off between the frequency of $S_p$ and the overhead incurred in the execution of the scheduler. A very small $S_p$ incurs large overheads due to the frequency of execution of the schedulers. On the other hand, jobs are more likely to be executed by the load balancing scheduler than the local scheduler. When $S_p$ is high, then the algorithms follow the behaviour of the NoLB algorithm. The best interval found for $S_p$ is between 1 and 7 seconds. This is because the average service times of the hosts are 5s, 6s and 7s. Hence there is no point in invoking the scheduler more frequently than required. The $S_p$ should not be too small as the scheduler would find no job to schedule, and only incur more overheads.

*Experiment 3.* Graphs 5 and 6 depict the results for the performance indices $R_t$ and $I_{ldb}$ respectively. The rest of the performance indices are depicted in graphs A.4 to A.6 in Appendix A. It can be seen that for a number of hosts beyond 5, the performance indices exhibit similar behaviour. Optimal performance is reached for a number of hosts equal to two because it is very unlikely that a wrong job transfer occurs. Though, to be realistic, it is clear that interpretations of these results should consider systems of 5 or 10 hosts upwards.




When increasing the system size, the impact of the source of errors (asynchrony of schedulers and outdated information) on scheduling decision making increases dramatically with the size of the system. All the benefits expected to be gained from having a large system are consumed by the high number of wrong transfers, the broadcasting of messages (due to the information policy), and the processing of a large number of messages. Since these three algorithms differ in their transfer and placement policy, the problems highlighted of outdated information, the number of messages to process, and the large number of decisions and allocations to be made are common to all of them. The DJS2 allows at most two jobs to be sent per scheduling period which is less than that required for a large number of hosts. Hence a slowly growing number of job movements is carried out for an increasing number of hosts, while DGP has the freedom to send as many jobs as the GP dictates. On the other hand, GP exhibits a slightly higher rate of $P_{wm}$. The strength of DGP over DJS2 and Disted is that it spreads the load evenly over the number of hosts, making use of the large number of hosts available, whereas DJS2 and Disted select the least loaded host regardless of the number of hosts available.

Barak and Kornatzky [2]. have defined a scalable algorithm as one being able to maintain performance improvements as the number of hosts increases. the DGP algorithm provides a more steady and levelled set of curves ($R_t$, $I_l db$, and $P_{ldb}$ (see Graph 5, 6, A.4) than Disted and DJS2, hence providing better scalability.

## 4 Prototype-based Performance Evaluation

This section presents the design of a run time prototype for the DGP algorithm. The experiments were undertaken in the University of Leeds network and hence the experiments had to operate with clearly defined constraints. These implied that only a subset of the network could be available and we also wanted some flexibility in starting/stopping new schedulers without disrupting the overall network and the prototype. This has been possible by using a dynamic configuration model [25] as the programming paradigm for undertaking performance experiments.

## 4.1 Programming paradigm

Writing distributed software is know to be a difficult task because of the level of complexity encountered in issues such as concurrency, synchronisation, and communications. These problems are even more complex due to the lack of support and software tools. The current option is to write distributed software using low level interprocess communication primitives provided by network operating systems such as Unix. This is for example achieved through the BSD sockets interface, enabling to specify the Transport protocol (TCP or UDP) over the IP network. The disadvantages of programming with these primitives are numerous. A good illustration would be the analogy of programming with the assembly language. These primitives are not simple to use, since the programmer has to go through all the low level details of establishing communications. Furthermore, they are not portable.

For these reasons, we have opted for the design a software model which provides a higher level of abstraction, and hence, hides underlying operating systems, and the hardware of the underlying network system including their system calls and communication primitives. The advantage of this model is that it enables programmers to develop dynamic load balancing systems across a number of different hosts, operating systems and networks. One further crucial issue supported by this model is dynamic reconfiguration at runtime. In effect, it is expected that the dynamic load balancing system may not run on the entire network, and hence reconfiguration at runtime may be required to remove a scheduler from a host for maintenance or upgrade or other reasons. Dynamic configuration would enable the dynamic load balancing system to continue while changes are carried out on part of the network. The subset of the network on which the dynamic load balancing system is running is called *a domain* [22].

This distributed programming model has two levels, and a runtime system. The two levels are *the scheduler* and *the configuration level*, and the run time system is the configuration manager. The distinction of these two levels is necessary for the support dynamic configuration [25].

*The scheduler level* enables the specifications of scheduler ports and their associated intercommunication primitives. The main objective of this level is to provide means for the programmer for scheduler's intercommunication without using direct port naming. This is a key property in dynamic configuration called *interconnection independence* [26], because it is essential to hide all intercommunication details that take place outside the schedulers. The configuration level in conjunction with the configuration manager handle such connections.

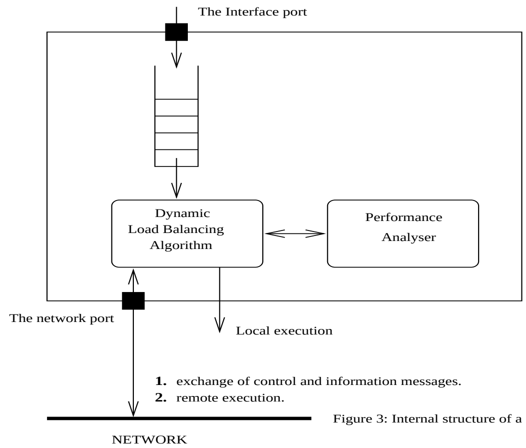


Figure 3: Internal structure of a scheduler and external links.

A scheduler is depicted in figure 3. It has two processes, and a queue to store incoming jobs. The performance analyser collects relevant performance indices and workload traces for data analysis. The dynamic load balancing algorithm process runs the DGP algorithm presented in Section 3. There are two services at the scheduler level: port specifications, and their associated intercommunication primitives (See table 4 below). The port specification is accomplished by invoking the functions: itf_port(), and ntw_port(). Itf_port() creates the interface port which is the input port that receives jobs from the local host. This function is mainly used by the configuration level operation inter_sched(). Ntw_port() creates the second port from which messages are sent and received to and from schedulers through the network. These functions provides two options: blocking and non blocking communications.

| Definitions of Functions | Application Programming Interface |
|---|---|
| Port specifications : Interface port<br>Port specifications : Network port | int itf_port(option);<br>int ntw_port(option);<br>int option; |
| Intercommunications to interface port<br>(Send and Receive) | snd_itf(job)<br>rcv_itf(job);<br>job_t *job; |
| Intercommunications to network<br>(Send and Receive) | snd_ntw(job)<br>rcv_ntw(hid,msg);<br>int hid;<br>msg_t *msg; |

Table 4: API for Scheduler Level

*The configuration level* enables the manipulation of schedulers at runtime for the construction or reconfiguration of dynamic load balancing systems in a domain. There are three functions provided at this level: creation of a scheduler, removal of a scheduler, and the interconnection to a scheduler. Several applications may be connected to a scheduler by sending jobs to its input port through *the* *interconnection function.* Table 5 provides the Application Programming Interface (API) for the configuration level.

| Definitions of Functions | Application Programming Interface |
|---|---|
| Creation of a scheduler | add_sched(scheduler_type, hostname)<br>sched_t scheduler_type;<br>char *hostname; |
| Deletion of a scheduler | rm_sched(hostname)<br>char *hostname; |
| Interconnection to a scheduler | inter_sched(mode, path, argv)<br>char *path, **argv;<br>mode_t mode; |

Table 5: API for Configuration Level

*The configuration manager* is the runtime support system for the configuration and scheduler levels. It holds information on the state of the current domain, the schedulers and their attributes. All transactions within a domain are recorded in a database. The API to this database consists of a library of C functions called *the config library* [22]. The configuration manager also ensures the compatibility of scheduler's type within a domain. For example, if the dynamic load balancing system has a central control, then there must be only one master scheduler, several slave schedulers and no cooperating schedulers. If the dynamic load balancing system is decentralised (like for DGP), then there are only cooperating schedulers, and no master nor slave schedulers.

### 4.2 Implementation

The prototype is implemented in the C language and Unix system calls. This includes the configuration and the scheduler level functions, the configuration database, and its interface library of functions. The scheduler is implemented by two independent communicating processes, one for the algorithm, and one for the performance analyser. Communication between the two processes is supported by Unix socket interprocess communication facilities. The queue is implemented by a fifo interprocess communication facility [15]. Communication between peer schedulers is also supported by the BSD Unix socket interface to the TCP/IP protocol stack over Ethernet. Normal and abnormal terminations of schedulers are handled by signals. Normal termination of schedulers (by the function rm_sched()) operates with software signals. The DGP algorithm is implemented using lightweight processes (or threads in Unix [5]). These threads share the same address space. The load information, transfer and placement processes of DGP are implemented as threads. A detailed account of the implementation with lightweight processes is presented in [23]. The dynamic load balancing prototype has been developed using the model' APIs described in Section 4.1. It runs on an Ethernet based network of Sun4 workstations. It has been operational for a domain of ten hosts (Sun4 workstations). Interactive commands were also developed for changes to the configuration level. These include **addsched, rmsched, lookdom** (for displaying the current domain). Other general commands such as **setup** (for initial setup of a whole domain), and **shutdown** (for deleting a whole domain) were also built.

### 4.3 Workload modelling

One of the most difficult tasks of this prototype-based performance evaluation was the design of a workload model. A networked application (an Office Information System, OIS) that generates a large number of small jobs around the network of workstations was selected as the kernel for the workload model. The OIS application had a number of processes running on each workstation and generated forms (files) that were triggered by events (e.g. input forms or timing point). These forms were parsed and processed by a Form Descriptor Interpreter (FDI), and the forms (called Forms Descriptors FD) were written in a pseudo-language (Jackson's like notation [16, 17]). The figure 4 depicts the workload model used for a workstation.

There is a need to have a precise measure of the input and output data to the prototype to ensure consistency in repeating similar experiments. For this reason, an automated generator of input forms was built. This generator of input forms is a simulation program called *SimOIS* (for SIMulation for Office Information Systems) that generates input forms at statistical rates (Poisson distribution). SimOIS is based on a discrete-event scheme [1]. Measurements of the output of the system are handled by the performance analyser process of the scheduler (see figure 3). A complete study of the input forms had to be done to calibrate and validate the parameters selected for the development of SimOIS [20].

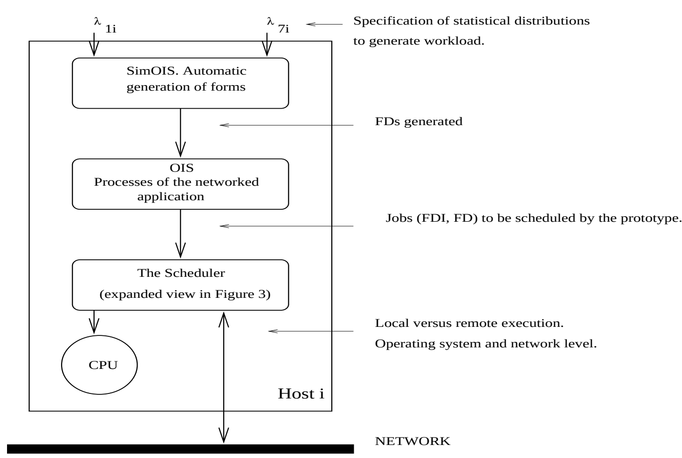



Figure 4 : Implementation of Host i with SimOIS

Each type of input forms has an initial time, an end of generation time, and precedence constraints. These constraints may relate to a timing point, or to the arrival of another type of form before activation. The arrival rate ($\lambda$) is the number of forms generated per unit of time, and $\lambda_{ij}$ is the arrival rate of forms generated for the form type i at the host j. For the OIS , there are 7 input forms.

Evaluating a host's workload involves the determination of the appropriate values for $\lambda_{ij}$ and the remaining parameters of each type of form. It is important to link these $\lambda$s and the intensity of the host's workload $I_{wkl}$ (see section 3.6). Then, by tuning $\lambda$s, it is possible to obtain different workload intensities for one host and therefore for a network of hosts. The workload types of a host (Idle, Light, Moderate, and Heavy, as depicted in Table 1) are determined by tuning $\lambda_{1i}$ to $\lambda_{7i}$. Several experiments were run to determine the relationships between the host's workload in the parameters $\lambda_{1i}$, $\lambda_{7i}$ [20].

The performance experiments were run on a network of Sun4 workstations, using a domain of six workstations. The workload used consists of one idle, one light loaded, one moderate loaded and three heavy loaded (ILMHHH). This workload was selected to be heterogeneous. It has been shown that the more unbalanced the workload is, the greater the benefit from load balancing [21]. Hence the unbalance of the workload was selected not to be too high to overemphasize the benefit from load balancing while not being realistic. On the other hand, the unbalance was not chosen too low as to hide the effect of load balancing.

### 4.4 Results

This section introduces the experiments performed on the overall implementation in order to measure its performance. Two sets of experiments are used: determination of the GP strategy internal parameter ($\Delta$), and the effect of the scheduling period on the performance. The experimental results were averaged over 3 runs, and computed at a 90% confidence interval. In order to ensure consistency in the experiment results, these were run late at night to ensure that the hosts used were free from jobs other than those generated by the OIS, the configuration manager runtime system, the scheduler threads, and SimOIS. The default values used in the experiments are given in table 6. The default workload consists of 1I, 1L, 1M, 3H in a network of 6 hosts, ($I_{wkl} = 0.58$, $P_{wkl} = 0.30$).

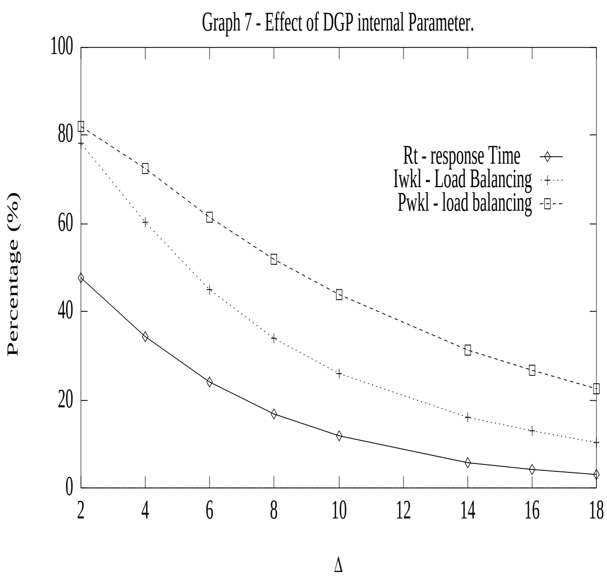
| Parameters | Values |
|---|---|
| Number of hosts | 6 |
| Scheduling period | 1 second |
| Information period | 1 second |
| $\Delta$ | 2 |

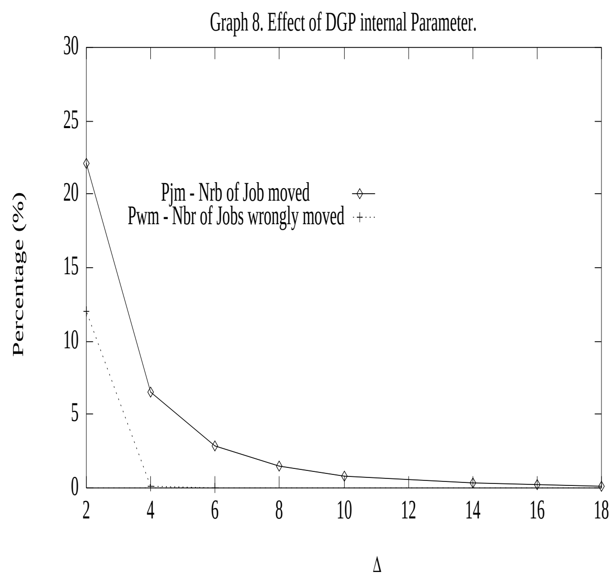
Table 6: Parameter values.

As with the simulation-based experiments, the NoLB algorithm is run. The performance indices of the DGP algorithm presented in the results (graphs) are represented in terms of the *relative performance* over the NoLB algorithm. For example, the relative performance (percentage improvement) of the intensity of load balancing ($I_{ldb}$) of DGP over NoLB is

$$\frac{I_{ldb}(DGP) - I_{ldb}(NoLB)}{I_{ldb}(NoLB)}$$

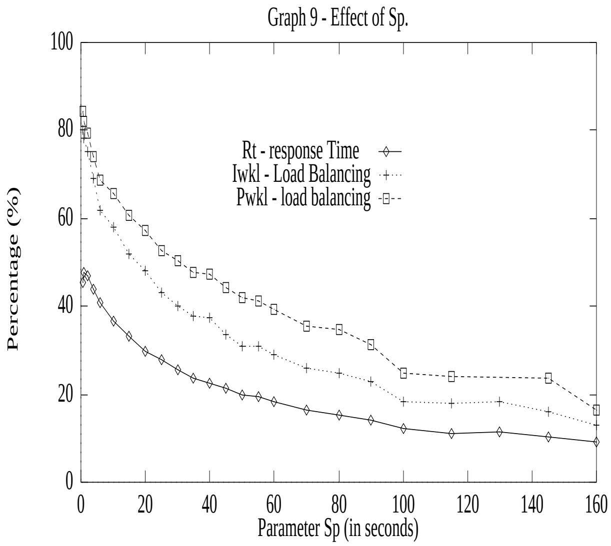
*Experiment 1.* In this experiment, the parameter $\Delta$ is varied from its minimum value up to a maximum where little or no further improvement over the NoLB algorithm is observed. The template used for representing the results consists of two graphs. The first contains the response time $R_t$, and the load balancing indices $I_{ldb}$ and $P_{ldb}$, while the second graph provides the costs in terms of $P_{jm}$ and $P_{wm}$. The same template is used for the second experiment. Graphs 7 and 8 respectively indicate the results of the first experiment. In Graph 7, the three curves exhibit the same shape, they decrease steadily as $\Delta$ increases. The best improvement for the three indices are achieved for $\Delta$ at its minimum ($\Delta$=2). For this same value, the highest cost in terms of $P_{jm}$ and $P_{wm}$ is also obtained. The DGP algorithm periodically attempts to maintain the load of all hosts within $\Delta$ by evenly moving jobs around the network. When reducing $\Delta$, it is expected to improve load balancing. This experiment confirmed this fact, and indicated that this improvement implies an improvement in response time $R_t$. Furthermore, these improvements follow the same pattern indicating a strong impact of load balancing on response time. On the other hand, when decreasing $\Delta$, a larger number of movements of jobs is required to maintain all loads within $\Delta$, and a larger number of wrong job transfers follows due to the outdated load information, and the asynchrony of schedulers. At the other extreme, for $\Delta$ very large, the band is so large, that prior to load balancing, most of the loads are already within $\Delta$. Hence the DGP algorithm has not much improvement to perform, and so converges to NoLB's behaviour.

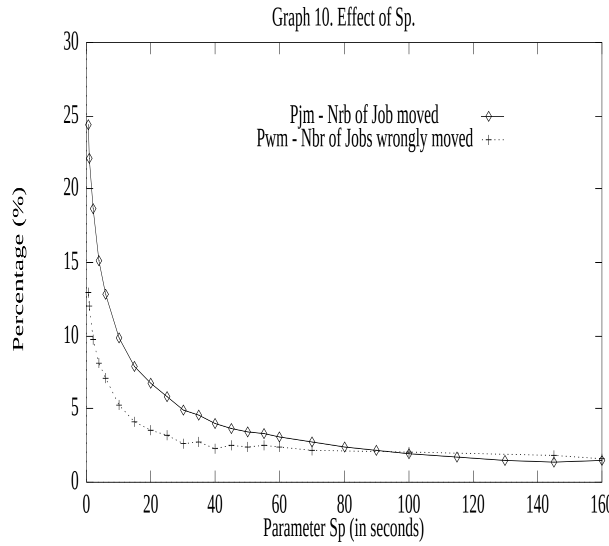



*Experiment 2.* In this second experiment, $\Delta$ is equal to 2, because it is the best value for DGP under the default workload. This experiment consists of varying the scheduling period $S_p$. Graphs 9 and 10 respectively indicate the results. First, the same remarks concerning the relationship of load balancing and response time found in experiment 1 is noticed in this experiment. For a small $S_p$, the best improvement is obtained. This is due to the high frequency of execution of the algorithm. However, the $S_p$ should not be selected too small, as to incur large overheads due to the frequency of the schedulers and eventually leading to unnecessary job movements. The $S_p$

starting at 1s is reasonable since the granularity of the computation are jobs (FDI, FD), i.e coarse grain. On the other hand, when the $S_p$ is high, the jobs are more likely to be executed by the local scheduler than the load balancing scheduler, and hence the DGP follows the behaviour of the NoLB algorithm. For example, suppose that an acceptable level of response time improvement is a minimum of 40%, then the best interval for $S_p$ is between 1s and 8s.




## 5 Evaluation and Validation Analysis

It can be seen that under similar workload to the one used in the simulation, the prototype performance can reach up to 50% improvement in response time (this is compared to the 70% reached by DGP through simulation, see graphs 1 and 7). Table 7 below depicts the results for DGP algorithm with simulation and prototyping under their default settings (table 3 and 6) respectively. The decrease in the performance was found to be sharper in prototyping compared with that of the simulation because of several factors. In the simulation, all jobs are migratable, whereas in the implementation, there are several processes which are part of the implementation of the OIS that cannot be migrated. Also, an interpreter (FDI) had to be resident on each host to translate FDs into processes. Furthermore, the OIS generates on average jobs that require very similar execution time (around 1.5s). Jobs requiring high execution time are likely to benefit more from load balancing because their transfer through the network should be relatively small compared to the overall execution time. This is the issue of granularity (computation versus communication). In the simulation, job execution times have an average of 5, 6, or 7 seconds The jobs generated by the OIS are I/O intensive (due to the nature of a information system application which involves many file operations).

| Experiments | Workload ($I_{wkl}$,$P_{wkl}$) | $R_t$ | $I_{ldb}$ | $P_{ldb}$ | $P_{jm}$ | $P_{wm}$ |
|---|---|---|---|---|---|---|
| Simulation | (0.7,0.3) | 69.1% | 87.7% | 91.3% | 46.5% | 20.9% |
| Prototyping | (0.6,0.3) | 47.7% | 78.2% | 82.0% | 22.1% | 12.0% |

Table 7: Simulation and prototyping results under default settings.

Since the prototype-based experiments were run late at night (and under a controlled environment), the real network delay is less than the one used for the simulation assumptions (a throughput 12.5 Kbytes per second). However, the I/O intensive jobs compensate for the low network delay. Both with simulation and prototyping, the response time and the load balancing indices followed the same shape. This confirmed the conclusions reached by the simulation that $R_t$, $I_{ldb}$, and $P_{ldb}$ are closely dependent and have the same behaviour. In addition, the relation $P_{ldb}$, and $I_{ldb} \geq R_t$ holds for both across the range of experiments, indicating that a load balancing algorithm (such as DGP) first improves load balancing, then response time.

## 6 Conclusions and Future Directions

In this paper, we have described a distributed-controlled algorithm (DGP) for dynamic load balancing that addresses the issues of coherence and coordination for distributed systems. The experimental results obtained indicate that a DGP distributed algorithm delivers better performance results when *a global plan* is embedded with it. We also believe that the issue of coherence and coordination will become more important with larger systems. The GP strategy builds on its awareness of local and global interests of the schedulers.

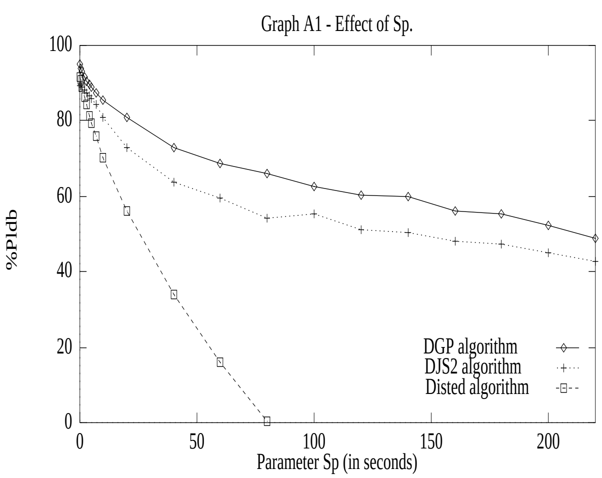
This paper advocated the research approach taken; to combine the contribution of simulation and prototyping. Reaching similar conclusions through the methodology outlined in Section 2 provides a higher degree of confidence in the conclusions obtained. The next step is to develop steps to incorporate calibration for simulation by prototyping as well as trace-driven simulation. Another important element to be added to the methodology is a rigorous framework for validating simulation with prototyping.

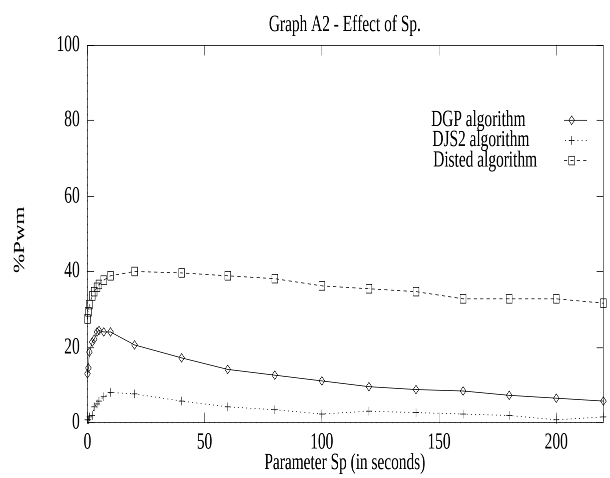
We presented a characterisation of workloads and performance metrics for measuring explicitly load balancing ($I_{ldb}$,$P_{jm}$,$P_{ldb}$,$P_{wm}$). The results consistently show, through simulation and prototyping, that load balancing and response time are closely dependent and that there is a greater improvement in load balancing ($I_{ldb}$,$P_{ldb}$) than response time ($R_t$).

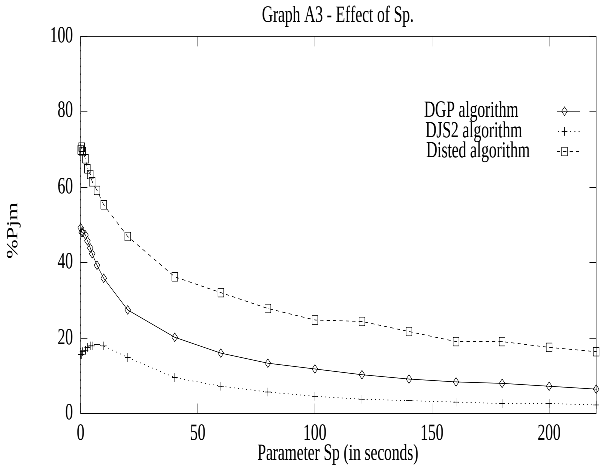
There are several directions to this work. Scalability studies on performance of dynamic load balancing algorithms have indicated that these algorithms, in general, do not scale very well [49]. One approach to tackle the problem of scalability proposed by zhou [49] is to investigate *intercluster load balancing*. The idea is to perform load balancing within each cluster of hosts, and load balancing among the clusters using long term information.

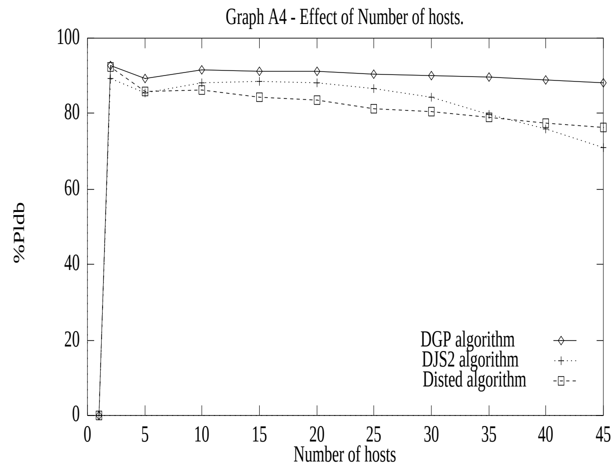
The distributed programming model presented in the previous sections assumed the existence of one domain within the network. The creation of several domains, may facilitate intercluster load balancing. Our approach into intercluster load balancing would imply that the unit of transaction of the system changes, and it is no longer a job or a process (depending on the granularity of the algorithm) but a host that may be exchanged between domains. These transactions operate at a meta level and are possible due to the dynamic configuration support of the model. Assume for example that a domain has 30 hosts with an average workload of 60% of CPU utilisation, and another domain has 25 hosts with an average workload of 20% of CPU utilisation. There is much reconfiguration to be done, whereby some hosts of the first domain are moved to the second domain and vice versa (a dynamic redistribution). However, much work remains to be done in this area, to determine whether it is worthwhile exchanging jobs and/or hosts between clusters, and indeed whether clustering itself can provide a better performance than a one network-wide algorithm.

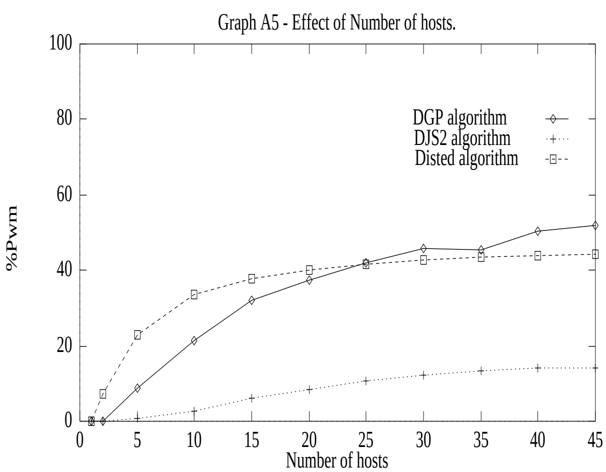
## Appendix A

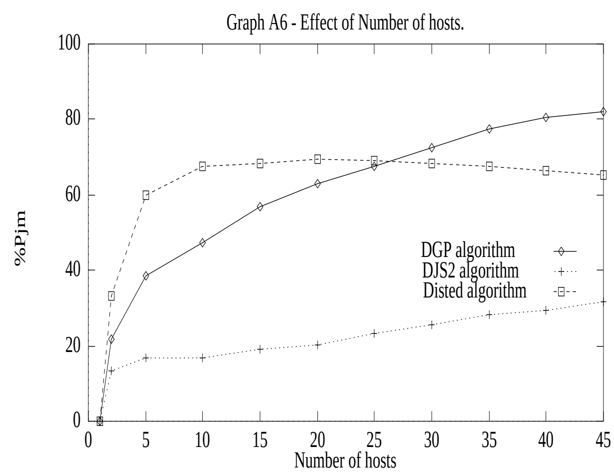



Graph A3 - Effect of Sp.
%Pjm
Parameter Sp (in seconds)
DGP algorithm
DJS2 algorithm
Disted algorithm
0
20
40
60
80
100
0
50
100
150
200


Graph A4 - Effect of Number of hosts.
%Pldb
Number of hosts
DGP algorithm
DJS2 algorithm
Disted algorithm
0
20
40
60
80
100
0
5
10
15
20
25
30
35
40
45


Graph A5 - Effect of Number of hosts.
%Pwm
Number of hosts
DGP algorithm
DJS2 algorithm
Disted algorithm
0
20
40
60
80
100
0
5
10
15
20
25
30
35
40
45


Graph A6 - Effect of Number of hosts.
%Pjm
Number of hosts
DGP algorithm
DJS2 algorithm
Disted algorithm
0
20
40
60
80
100
0
5
10
15
20
25
30
35
40
45

## References


[1] J Banks and J S Carson. *Discrete-Event System simulation.* Prentice-Hall International, 1984.

[2] A Barak and V Kornatzky. Design principles of operating systems for large scale multicomputers. *Lecture Notes on Computer Science*, 309:104–123, September 1987.

[3] A Barak and A Shiloh. A distributed load-balancing policy for a multicomputer. *Software-Practice and Experience*, 15 no 9:901–913, September 1985.

[4] G Bernard, D Steve, and M Simatic. A survey of load sharing in distributed systems. *Distributed Systems Engineering Journal*, December 1993.

[5] A D Birrell. An introduction to programming with threads. *Technical Report*, 35, January 1989.

[6] A H Bond and L Gasser. *Readings in Distributed Artificial Intelligence.* Morgan Kaufman Publisher, 1988.

[7] T L Casavant. Analysis of three dynamic load-balancing strategies with varying global information requirements. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 185–192, 1987.

[8] S Chowdury. The greedy load sharing algorithm. *Journal of Parallel and Distributed Computing*, 9:93–99, 1990.

[9] F Douglis and J Ousterhout. Process migration in the sprite operating system. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 18 – 25, September 1987.

[10] E H Durfee and V R Lesser. Using partial global plans to coordinate distributed problem solver. *Proceedings of the 1987 International Joint Conference on Artificial Intelligence*, pages 875–883, 1987.

[11] E H Durfee, V R Lesser, and D D Corkill. Coherent cooperation among communicating problem solvers. *IEEE Transactions on Computers*, 36:1275–1291, 1987.

[12] D L Eager, E D Lazowska, and J Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE 12 no 5:662 675, May 1986.

[13] E Gelenbe and G Pujolle. *Introduction to Queueing Networks.* John Wiley & Sons, 1987.

[14] A Hac and X Jin. Dynamic load balancing in a distributed system using a decentralized algorithm. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 170–177, September 1987.

[15] K Haviland and B Salama. *Unix System Programming.* Addison-Wesley, 1987.

[16] M A Jackson. *Principles of Program Design.* Academic Press, 1975.

[17] M A Jackson. *System Development.* Prentice-Hall International, 1983.

[18] M Kara. A global plan policy for coherent cooperation in distributed dynamic load balancing algorithms. *Distributed Systems Engineering Journal*, 2 (4), December 1995.

[19] M Kara. Sim/dsa/dbm and expgen: An integrated set of tools for simulation on a network of workstations. *Technical Manual (Version 1.03)*, May 1991.

[20] M Kara. Using dynamic load balancing in distributed information systems. *Research Report Series, School of Computer Studies, The University of Leeds*, Report 94.18, May 1994.

[21] M Kara. Phd thesis: Development of distributed scheduling systems with an application to office automation. *School of Computer Studies, The University of Leeds*, September 1991.

[22] M Kara, P H Jesty, and T G Gough. Flexis-i : A dynamic configuration model for distributed scheduling systems. *Proceedings of the First IFIP International Workshop on Configurable Distributed Systems*, March 1992.

[23] M Kara, P H Jesty, and T G Gough. A blackboard-based framework for the development of cooperating schedulers. *Proceedings of the IEEE International Conference on Software Engineering and Computer Systems*, May 1992.

[24] W D Kelton and A M Law. An analytical evaluation of alternative strategies in steady-state simulations. *Operational Research*, 32:169–184, 1984.

[25] J Kramer. Configuration programming: A framework for the development of distributable systems. *Proceedings of the International Conference on Computer Systems and Software Engineering*, pages 374–384, May 1990.

[26] J Kramer and J Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, pages SE–11 no 4, April 1985.

[27] O Kremien, J Kramer, and J Magee. Scalable, adaptive load sharing for distributed systems. *IEEE Parallel and Distributed technology*, 1 (3), August 1993.

[28] P Krueger and M Livny. The diverse objectives of distributed scheduling policies. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 242 – 249, September 1987.

[29] T Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17 no 7:725–730, July 1991.

[30] K A Lantz, W I Nowicki, and M M Theimer. An empirical study of distributed application performance. *IEEE Transactions Software Engineering*, SE-11 no 10:1162 – 1175, October 1985.

[31] M J Litzkow, M Livny, and M W Mutka. Condor - a hunter of idle workstations. *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 104 – 111, June 1988.

[32] M Livny and M Melman. Load balancing in homogeneous broadcast distributed systems. *Proceedings of the Computer Network Performance Symposium*, pages 47 – 55, April 1982.

[33] R Mirchandaney and J A Stankovic. Using stochastic learning automata for job scheduling in distributed processing systems. *Journal of Parallel and Distributed Computing*, 3:527 – 552, 1986.

[34] R Mirchandaney, D Towsley, and J A Stankovic. Adaptive load sharing in heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 9:331–346, 1990.

[35] M W Mutka. Estimating capacity for sharing a privately owned workstation environment. *IEEE Transactions on Software Engineering*, 18 (4), April 1992.

[36] M W Mutka and M Livny. Scheduling remote processing capacity in a workstation-processor bank network. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 2 – 9, September 1987.

[37] L M Ni, C Xu, and T B Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, SE-1 no 10:1153 – 1161, October 1985.

[38] M Nuttall and M Sloman. Workload characteristics for process migration and load balancing. *Proceedings of the IEEE International Conference on Distributed Computing Systems, ICDCS'97*, May 1997.

[39] K Pawlikowski. Steady-state simulation of queueing processes: A survey of problems and solutions. *ACM Computing Surveys*, 22 no 2:123–170, June 1990.

[40] C G Rommel. The probability of load balancing success in a homogeneous network. *IEEE Transactions on Software Engineering*, 17 no 9:922–933, September 1991.

[41] B Shirazi and A R Hurson. Special issue on scheduling and load balancing. *Journal of Parallel and Distributed Computing*, 16 no 4, December 1992.

[42] S Shou. Utopia: A load sharing facility for large heterogeneous distributed computers. *Software: Practice and Experience*, December 1992.

[43] J A Stankovic. A perspective on distributed computer systems. *IEEE Transactions on Computer*, C-33:1102–1115, December 1984.

[44] J A Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions Software Engineering*, SE-11 no 10:1141 – 1152, October 1985.

[45] J A Stankovic and I S Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. *Proceedings of the Fourth International Conference on Distributed Computing Systems*, pages 49 – 59, May 1984.

[46] M M Theimer and K A Lantz. Finding idle machines in a workstation-based distributed system. *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 112 – 122, June 1988.

[47] L H Turcotte. Survey of software environments for exploiting networked computing resources. *Engineering Research Center for Computational Field Simulation*, MSU-EIRS-ERC-93-2, February 1993.

[48] S Zhou. Performance studies of dynamic load balancing in distributed systems. *Report No UCB/CSD 87/376*, 1987.

[49] S Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14 No. 9:1327–1341, September 1988.

[50] S Zhou and D Ferrari. A measurement study of load balancing. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 490 – 497, September 1987.

[51] W Zhu, C F Steketee, and B Muilwijk. Load balancing and workstation autonomy on amoeba. *Australian Computer Science Communcations*, 17(1), 1995.