A survey of load sharing in networks of workstations

To cite this article: G Bernard et al 1993 Distrib. Syst. Engng. 1 75

View the article online for updates and enhancements.

Related content
- A global plan policy for coherent co-operation in distributed dynamic load balancing algorithms
  M Kara
- KTK: kernel support for configurable objects and invocations
  B Mukherjee, D Silva, R Schwan et al.
- Micro-kernel support for migration
  M O'Connor, B Tangney, V Cahill et al.

Recent citations
- A situatedness-based Knowledge Plane for autonomic networking
  Dominique Galti et al
- C.S. Chandrashekaran et al
- Examination of load-balancing methods to improve efficiency of a composite materials manufacturing process simulation under uncertainty using distributed computing
  Feng Zhang et al
A survey of load sharing in networks of workstations

Guy Bernard†, Dominique Steve‡ and Michel Simatic§

Institut National des Télécommunications, 9 rue Charles Fourier, 91011 Evry Cedex, France
O2 Technology 7, rue du Parc de Cigny, 78035 Versailles Cedex, France
Alcatel Alsthom Recherche, Route de Nozay, 91460 Marcoussis Cedex, France

Received 8 December 1992

Abstract. This paper is a survey of existing policies and mechanisms for load sharing in loosely-coupled distributed computing systems, where user machines are personal workstations interconnected by a local area network. We are interested only in centralized operating systems providing mechanisms for remote process communication, thus we do not study distributed operating systems in which load balancing and process migration may be provided by a network-wide virtual memory mechanism. We define load sharing, load balancing, non-pre-emptive migration and pre-emptive migration, and we discuss the goals of load sharing and load balancing strategies related to process scheduling. We argue against the usefulness of load balancing strategies in the context of networks of workstations. A load sharing algorithm is composed of three parts, namely, a location policy, an information policy and a transfer policy. We review the different location policies, information policies and transfer policies that have been proposed in the literature. We discuss their ability to take personal use of workstations into account, and we compare them with respect to the performance that can be really obtained in the context of networks of workstations. We show that only some policies are efficient in such a context. Thereafter, we present the mechanisms proposed for supporting the policies, and discuss them with respect to network interfaces, file system design, machine heterogeneity and program interactivity. The question of pre-emptive versus non-pre-emptive migration is addressed, and we argue that pre-emptive migration does not provide substantial benefits in a network of workstations. Most of the implemented load sharing systems described in the literature are presented, and finally some perspectives in this research area are described.

1. Introduction

A loosely coupled distributed system is made of a set of machines linked by a communication network. There is no physical memory common to the processors. Rather, interprocess communications are done by message exchanges over the network. Such an architecture provides mechanisms for resource sharing (processors, disks, printers) between applications running on different machines. These mechanisms are well understood today. The most important of them, namely the Remote Procedure Call, was formalized and implemented as early as 1981 [56,17] (a recent survey on research work in the area of Remote Procedure Call, and a detailed comparison of eight existing mechanisms, may be found in [77]). However, the Remote Procedure Call mechanism by itself does not define the way it is used to achieve resource sharing. The main problem that the distributed system designers are faced with is that of performance#. Three main classes of policies, built upon the possibility of having access to remote resources from any machine of the network, may be applied in order to enhance the global system performance:

- A file location policy determines the sites on which the files should be placed, and possibly the number of copies that should exist, in order to optimize some criteria. (A comparison of existing file location policies may be found in [33].)
- A task assignment policy aims at solving the problem of assigning tasks to processors when a job to be run consists of a set of communicating tasks. The goal is to transform the logical parallelism of the tasks into a real parallelism over a network of several processors. Most of the time, some a priori knowledge

---

# 'Building a 16-node distributed system that has a total computing power about equal to a single-node system is surprisingly easy.' [76].
of the program behaviour is needed, and assignment algorithms are complex [26]. Thus, task assignment policies are generally static, i.e. not able to take into account rapid changes in the system state.

- A program assignment policy considers application programs as atomic entities, for which is raised the question of the choice of the machine on which each program should run. The goal here is not to execute programs started by the users anywhere in the network. The system. This approach has three consequences: (i) the adaptability to changes in the system state; (iii) migrating programs as atomic entities, for which is raised the question of the choice of the machine on which each program behaviour is needed, and assignment algorithms are complex [26]. Thus, task assignment policies are generally static, i.e. not able to take into account rapid changes in the system state.

- A program assignment policy considers application programs as atomic entities, for which is raised the question of the choice of the machine on which each program should run. The goal here is not to execute programs started by the users anywhere in the network. The system. This approach has three consequences: (i) the adaptability to changes in the system state; (iii) migrating programs as atomic entities, for which is raised the question of the choice of the machine on which each program behaviour is needed, and assignment algorithms are complex [26]. Thus, task assignment policies are generally static, i.e. not able to take into account rapid changes in the system state.

Since the three classes of policies listed above have the same broad goal (improving performance in a distributed system by resource sharing), they are not mutually exclusive. Algorithms involving process migration, file migration and file replication [34,35,36], and algorithms considering assignment of independent programs, each of them being composed of several communicating modules [25] have been considered.

Research work in the area of program assignment in loosely-coupled distributed systems started about ten years ago, both with implementations [70,61,65,40] and with theoretical papers [16]. Since then, research activity in the field has always been intensive. The main reason is the generalization of loosely-coupled distributed systems as computer environments, which has been a driving force for designing resource sharing algorithms in general, and program assignment policies in particular.

Compared to networks of mainframes or networks of multiuser minicomputers, networks of workstations present several characteristics (we will come back to them throughout this paper).

(i) The total computing power is most of the time underutilized.

(ii) Users sometimes need peak of computing power, for which the computing power provided by a single workstation is not sufficient.

(iii) Workstations are frequently diskless, so that system binary files and user data files are stored on a server machine. Workstations have remote access to them.

(iv) Most of local area networks provide a message broadcast capability, which seems an attractive tool for supporting a program assignment policy.

(v) In most computing environments, each workstation is dedicated to an 'owner', i.e. the customary user of the workstation.

With respect to a program assignment policy, these characteristics have the following implications. The second point makes a program assignment policy desirable. The first point makes it possible. The third point makes it cheap, since assigning a program on a remote workstation does not involves an extra overhead for file migration. The fourth point may be used to design simple algorithms, but we will show that broadcast may be expensive. The fifth point raises some problems, since 'owners' do not easily accept suffering large response times on 'their' workstation, under the pretext that another user needs a peak of computing power.

The purpose of this paper is to make a synthesis of research works in the area of program assignment policies in networks of workstations. We restrict the scope of this survey to network operating systems [76], i.e. computer environments where each machine runs its own standard, centralized operating system, augmented with some communication facilities for interprocess communications over a network (a typical example is Unix 4.3bsd). The study of program assignment policies in distributed operating systems, where each processor runs a part of the same network-wide operating system (in which program assignment is only an instance of a more general object migration facility), is out of the scope of this paper.

Other surveys were published a few years ago. The most important are those of Wang and Morris [79], and Zhou and Ferrari [81,82]. However, Wang and Morris considered only theoretical models of algorithms, and the outstanding work of Zhou and Ferrari did not pretend to cover the whole spectrum of program assignment policies. Furthermore, other policies and many implementations have been recently described in the literature, most of them in the context of networks of workstations. Thus, a synthesis as complete as possible is not useless.

The main design choices involved by program assignment policies may be summarized by a set of questions: 'why?', 'how?', 'when?', 'where?' and 'which one?'. The paper is organized in order to give answers to these questions. The goals of program assignment policies are addressed in section 2. Section 3 deals with a classification and a comparison of policies. The underlying mechanisms are described and discussed in section 4. Existing implementations are presented in section 5, and finally section 6 derives the perspectives of evolution in the field.

2. Goals of program assignment policies

In this section we define the terms we use throughout the paper and we list the various objectives of program assignment policies.
2.1. Terminology

There is some anarchy in the literature about the meaning of the words used by the various authors. We define here the terms that we will employ in the following.

In a loosely-coupled distributed system, programs are invoked by users from some terminal, or by the system itself (e.g., periodic electronic mail handling). A program assignment facility may decide to run the program on a machine that is different from the one the program was invoked on. Non-pre-emptive process migration consists in starting the execution of the program on a remote machine, and running the program there until it ends. Pre-emptive process migration permits the execution of a program to be suspended on the current machine, and resumed on another machine. In any case, we use intentionally the term 'process' instead of 'program', to emphasize the dynamic character of the execution (both in time and space). However, it is a whole program that is run by the operating system of the target machine, even though this program may consist in several communicating tasks, each of them being run possibly as a separate process, a thread, or a lightweight process.

2.2. Objectives

The benefits that may be expected from process assignment strategies are the following [43]:

(i) Load sharing: if there are in the network some machines with small load (or even completely inactive), they can be used to relieve more loaded machines of some processes.

(ii) Network communication savings: by putting on the same machine entities that exchange much data, the network load may be lowered (e.g., a process making a lot of disk I/O operations may be run on the machine that manages the disk).

(iii) Availability: spreading programs on several machines gives a better robustness to machine failures.

(iv) Reconfiguration: the possibility of despatching the programs amongst several machines may be used for system reconfiguration when a machine crashes, when recovery occurs, or before a scheduled halt of a machine.

(v) Remote access to a resource not locally available: for instance, a program that requires a floating-point coprocessor may be invoked from a machine without coprocessor.

The last four objectives are easy to understand. However, the first one requires some discussion. Just as several strategies may be designed to allocate the processor to the processes in a single-processor machine, several strategies may be designed to allocate machines to programs in a distributed system. This way, a global scheduling algorithm is superposed on local scheduling algorithms. A parallel may be drawn between the objectives of program assignment strategies in distributed systems, and that of processor allocation algorithms in a centralized operating system [44].

In a centralized operating system, the minimum property expected from a scheduling algorithm is that the processor should not be idle while some processes are waiting for it. In a distributed system, an analogous minimum property is expected: no machine should be idle while processes are waiting for the processor on another machine in the network. Eager, Lazowska and Zahorjan [21] make the distinction between two broad classes of program assignment strategies. The only goal of load sharing algorithms is to provide this minimum property, while load balancing algorithms aim at equilibrating the process load amongst the machines of the network.

In networks of workstations, load balancing is not only unnecessary but undesirable, for two reasons. First, most workstations are under-utilized, thus balancing the load would result in process migrations between lightly loaded machines, and migration overhead would prevail over the small gain in execution time [78]. Second, since workstations are often dedicated to an owner, the user of a lightly loaded workstation would not be happy to suffer from long response times because another user makes intensive computations. However, as long as an owner is not affected significantly, their workstation may receive some load from outside [3]. For these two reasons, load balancing is not used in networks of workstations. The goal of process assignment in such systems is load sharing.

3. Process assignment strategies

A process assignment strategy is composed of an algorithm and of input parameters for this algorithm (a priori information, or observations). The algorithm is executed in order to take a decision: given the current values of input parameters, should a process be migrated to another machine and, if the answer is yes, to which one? More precisely, a process assignment algorithm is built with three components [82]. The information policy specifies the nature and the amount of information used for decision making, and the way this information is distributed. The transfer policy determines the eligibility of a process for remote assignment. The location policy selects a suitable machine which an eligible process should be assigned to. There are strong interactions between three components, thus it is difficult to study each of them separately. In this paper, we will consider information policies and location policies together.

Most process assignment strategies rely on some load index observed locally on each machine, and used as input parameter for the transfer policy and the location policy. In this section, we first classify and evaluate information and location policies, disregarding the nature of the load index involved (if any), then we discuss the transfer policies and the usual load indices, and finally we compare pre-emptive process migration and non-pre-emptive process migration.
3.1. Information and location policies

As mentioned before, process assignment strategies have a broad goal of load sharing or load balancing. However, in order to reach this goal, it is necessary to set more specific objectives for a process assignment algorithm [78]:

- **quality**: at least, the algorithm should be able to find an idle machine on the network, if there are some;
- **efficiency**: the algorithm should not impose an unacceptable overhead on the system, nor disrupt the machines that do not participate in the process assignment strategy;
- **extensibility**: the algorithm should be able to cope with a large number of machines (workstation-based configurations are currently made of several hundreds of machines);
- **robustness**: the process assignment facility should be interrupted as briefly as possible by the failure of one or more machines.

3.2. A taxonomy of algorithms

According to the taxonomy proposed in [15], the first distinction is between static and dynamic algorithms. In static algorithms, information about the total mix of processes in the system is assumed to be known by the time the executable image of a program is linked, and this information is used to assign a processor to the program; each time the program is started, the corresponding process is run on that processor. In dynamic algorithms, no (or little) **a priori** information is required about resource demands of processes, and no assumption is made about what the system state will be at program execution time. When local conditions make a process migration desirable, the location policy selects a suitable machine for receiving the process.

The family of dynamic algorithms may be further refined (see figure 1). Algorithms with **blind location** are those where the choice of an execution site is made without any information about the current conditions of the remote machines. Conversely, for **conditional location**, the choice of a receiver machine is based upon a **global knowledge** or a **partial knowledge** of the system state, according to whether the decision is made with information about all the machines in the network, or about a subset. The global knowledge may be maintained on a single machine (**centralized information**) or on all the machines of the network (**distributed information**). Finally, the migration decision may be taken by a single machine (**centralized decision**) or else may be taken by each machine (**distributed decision**).

Like any taxonomy, this one is not perfect, and it is difficult to find the right place for a few algorithms. However, we consider that it is better than the one proposed in [15], because it can take into consideration recent algorithms that were not published by the time [15] was written.

3.2.1. Static algorithms. The first research works in the area of process assignment dealt with static algorithms. In [51], process execution times are supposed to be deterministic. Probabilistic execution times are introduced in [16]. In [57], process inter-arrival times and service times are exponentially distributed. In these papers, the objective is load balancing, and optimal solutions (minimizing the average response time) are given. In [12], the objective is to balance the idle period durations amongst the machines.

Static algorithms have two drawbacks. First, their execution cost is high, hence they cannot be used to react to fast changes in the system. Second, when the variability of execution times is taken into account, this is done with exponential assumptions (in order to be able to obtain exact results), when in fact observations on real systems (see for instance [46] or [82]) invalidate these assumptions.

Static algorithms may be worthwhile for computing systems that execute periodically a set of programs with well known behaviour (e.g., real-time systems). This is clearly not the case for networks of workstations. Thus, the remainder of this section will be devoted to dynamic algorithms.

Before describing dynamic algorithms, we first set the values of the parameters that we will use for their comparison.

3.2.2. Parameters used for algorithm comparison. In [78], Theimer and Lantz compare a few algorithms on a quantitative basis, with the following parameter values. In order to be efficient, an algorithm should select a receiving machine for a process in less than 100 ms, consume less than 1% of CPU cycles on any machine, and consume less than 1% of network bandwidth. Furthermore, Theimer and Lantz assume that program generation leads to running the process assignment algorithm once per second on average on every machine, and that, for algorithms involving periodic information emission, the interval between two emissions is 10 s.†

† These values were observed on a system composed of 70 Sun-2 and Sun-3 machines linked by an Ethernet at Stanford University.
In this section we extend the work of Theimer and Lantz, by setting some additional parameter values and by comparing a larger number of algorithms. Here we make the additional assumptions:

- The time required for processing a request (message reception, request processing, response emission) is 5 ms. This is a minimal hypothesis: Theimer and Lantz observed a 4 ms delay for an empty request on Sun-3s, which is confirmed by our own measurements [10], and 5 ms or 20 ms for two algorithms that they implemented.

- Two values are selected for the average percentage of idle machines on the network: 33% and 90%. These values appear as a lower and an upper bound for figures reported by several authors (37% in [82], 80% during the busiest times in [73], 33% at the busiest times and 80% most of the time in [78], 90% in [55]). This way, it will be possible to compare the behaviour of algorithms in rather different global load conditions, and to test the robustness of algorithms with respect to load variation during a typical day.

      In [78], Theimer and Lantz stress the cost of broadcast and multicast on a local area network. When available, this feature looks very attractive, since it makes it possible to send information or a request to a set of machines with the same sending cost and the same bandwidth consumption as for point-to-point communication. However, broadcast and multicast have a major drawback: when the received message asks for an answer, all the recipients compute and send their answer at nearly the same time, so that a lot of answer messages arrive simultaneously, and buffer overflow may occur. Theimer and Lantz observed a loss rate above 50% as soon as a few tenths of answers are generated.

      In [68], Simatic evaluated ten dynamic algorithms according to the parameters described above (these algorithms will be described in the following subsections). His results are summarized in Table 1, which should be read as follows. 'Quality' is the probability to find an idle machine on the network if such a machine exists, when the average percentage of idle machines is 33% and 90%. The column 'Efficiency' indicates whether broadcasts are necessary in the absence of failures, and the average number of messages that is necessary to select a remote machine and transfer a process execution to it. 'Extensibility' is the maximum number of machines that algorithms can take into account while remaining in the limits of the constraints described above. The column 'Robustness' gathers the answer to three questions:

      - column 'HM'—can location policy select a halted machine (i.e., a machine that is not up, or that does not participate to the process assignment facility)? If the answer is 'yes', then the transfer phase will fail, and another selection will be necessary.

      - column 'WF'—in the worst case of failure, is the process assignment facility fully available (OK), downgraded (DG), or non-available (NA)?

      - column 'Insertion'—when a machine joins the process assignment facility (e.g., at boot time), is it possible to use this facility immediately, or else is there a learning phase?

Now we review the different classes of dynamic algorithms.

3.2.3. Blind location. No state information about the machines in the network is used.

In RANDOM [81], the selection of a receiving machine is made randomly. This algorithm is efficient, extensible and robust. However, its quality is low, and it may select a halted machine. A variation is suggested in [21]: a loaded receiving machine may forward the incoming process to another machine, and a maximum number of hops prevents instability.

While being very simple, RANDOM provides substantial performance improvement with respect to no process assignment policy, at least with low or moderate global system load. This result is pointed out by probabilistic models [79], probabilistic simulation [21], measurements [73], and trace-driven simulation [82].

† The figure in this last paper is in fact 70%, but the definition of 'idle' workstations in this paper is restrictive, as will be seen in section 3.4. The value of 90% results from our own correction in order to correspond to the classical definition of the idleness of a machine, i.e., no processes are currently running.

§ It is assumed here that a process transfer phase consists in two messages: one for sending information about the process, and one for returning an acknowledgment. The transfer protocol is independent from the location and information policies.
The outstanding work described in the last paper is a detailed comparison of seven algorithms. We will refer to it several times in the following.

In CYCLIC [79], processes are assigned to remote machines in a cyclic way. The only information to store is thus the identification of the last machine that a process was sent to. Simulation showed a small improvement with respect to RANDOM.

### 3.2.4. Partial knowledge

In this class, algorithms use some information about a subset of the machines in the network. This knowledge may be obtained either implicitly (by memorizing the result of a process transfer request), or explicitly (by message exchanges).

LEARNING [72] is a variation of RANDOM. According to the result of a transfer request towards some machine, the probability to select that machine for the next transfer is increased or decreased. LEARNING does not provide better performance than RANDOM when machine loads vary frequently, and unfortunately this is the case for networks of workstations.

In PROBABILISTIC [6,7], every machine maintains a load vector holding the load of a subset of machines. Periodically, the first half of the load vector, including the local machine load, is sent to a randomly selected machine. This one updates its load vector accordingly. This way, information may be spread in the network without broadcast messages. However, the quality of this algorithm is not perfect, its extensibility is low and insertion is deferred.

There were a number of papers about THRESHOLD and LEAST [21,38,82,54]. They both use a partial knowledge obtained by message exchanges. In THRESHOLD, when a process is to be transferred, a randomly selected machine is asked for its load. If the load is less than some threshold $T$, the process transfer occurs. If not, polling is repeated with another machine. If no suitable receiver has been found after $\text{Maxpoll}$ attempts, the process is executed locally.

LEAST is a variation of THRESHOLD, where systematically $\text{Maxpoll}$ machines are probed, and the least loaded machine is selected for receiving the process. THRESHOLD and LEAST show good performance results with respect to their simplicity (see table 1). Furthermore, the load values used by these algorithms are up-to-date, hence a bad location decision (i.e., one based on obsolete load information) is unlikely. The influence of the values of $T$ and $\text{Maxpoll}$ is discussed in [21], [82] and [54]. An adaptation of LEAST for real-time systems is proposed in [64].

There are several variants of THRESHOLD and LEAST:

- RECEPTION [50,20] is equivalent to THRESHOLD, but driven by available machines rather than overloaded ones. When the load of a machine falls under a threshold, this machine tries to find an overloaded machine by random polling. It is shown in [20] that performance is not so good as that of THRESHOLD if the cost of transferring a process having started its execution is larger than the cost of starting a new process, which is the case in most systems.

- [27] proposes an algorithm based a micro-economic approach. The main drawback is that the execution duration of processes has to be known a priori, which is generally not the case for interactive use of workstations.

- RESERVATION [20,82,54] is a variant of RECEPTION applied to non-pre-emptive migration rather than pre-emptive migration. In this algorithm, an underloaded machine gets a reservation for the next process to be started from an overloaded machine. The performance of RESERVATION is not good, because reservations are made on the basis of information that will be obsolete by the time it is honoured.

To summarize, THRESHOLD and LEAST provide good results when system load is homogeneous between machines. This can be easily explained. If system load is homogeneous, a small subset of machines constitutes a representative sample: if no available machine can be found after $\text{Maxpoll}$ trials, it means that the system is globally loaded, thus it is not worthwhile to continue searching an idle machine. However, this is not always the case in networks of workstations, because some workstations may be overloaded whereas others are completely idle at the same time (e.g., a workstation ‘owner’ is currently outside). In heterogeneous load patterns, partial knowledge about global system state is not accurate enough. Algorithms based on global knowledge are better adapted.

### 3.2.5. Centralized information and centralized decision

In the class of algorithms using a global knowledge of system state, information may be concentrated on a single machine, or else distributed. It is the same for decision making, therefore four subclasses may be considered theoretically. In fact, the subclass where the selection of a suitable receiver would be done on a single machine whereas information would be replicated on every machine is of no practical interest. The first subclass (centralized information and centralized decision) is studied in this subsection.

In CENTRAL [37,48,59,82,13,78], when an overloaded machine wishes to transfer a process, it asks a server for an underloaded machine, if there is one. The server machine is informed of the availability of any machine in the system by means of messages sent to it by every machine in the system. CENTRAL provides very good performance results (see table 1).

Once again, several variations have been suggested:

- Every machine periodically sends its load to the server [37,13].

- Every machine sends its load to the server only when the load has changed by a significant amount [78].

- In Butler [59,60] and Sprite [19], load information is not maintained in memory by a server process. Instead, it is read/written in a single shared file managed by the network file system.
Centralized solutions in a distributed system suffer from two potential drawbacks. First, the server may become a bottleneck. This is not the case for CENTRAL (see extensibility in table 1—remember that the figures are obtained assuming less than 1% CPU overhead on any machine, including the server). Second, a server crash makes the facility unavailable, and the time necessary for recovery may be large. This is the case with CENTRAL. In their implementation, Theimer and Lantz measured a delay of 18 s [78]. The following classes of algorithms introduce some degree of distribution in order to solve this problem.

3.2.6. Centralized information and distributed decision. In GLOBAL [29, 81, 39, 82], information gathering is centralized and information use (decision) is distributed. Periodically, the server broadcasts the load vector. This way, an overloaded machine just finds the less loaded machine from its load vector without asking the server. This algorithm is more efficient and extensible than CENTRAL, because it involves a smaller number of messages. Furthermore, robustness is better, since during server recovery the process assignment facility is still available, yet with old load values.

However, the exact behaviour of algorithms cannot be predicted from the results given in table 1 only. In fact, GLOBAL does not perform better than LEAST [82]. The reason is that GLOBAL uses a larger amount of Information, but this information is not up-to-date (a high frequency for gathering/broadcasting load values would result in an unacceptable overhead). On the other hand, LEAST uses information on a subset of machines only, but this information is up-to-date.

3.2.7. Distributed information and distributed decision. In OFFER [25, 52, 73, 82, 67, 23], every machine broadcasts periodically its load value, thus every machine can maintain a global load vector. Extensibility is very bad (see table 1). The poor results of OFFER are confirmed by Zhou [82]: the mean process response time is larger than with GLOBAL. A variation is proposed in [9], but it is not sufficient to overcome the cost of systematic broadcasts.

REQUEST [72, 73, 78, 30, 42] avoids periodic broadcasts. This algorithm is similar to LEAST, except that all the machines in the system are polled, and that polling is done by a single broadcast message. Extensibility is barely passable. Furthermore, buffer overflows may occur when many answers are received simultaneously. Thus, in the variation proposed in [78], only the machines with reasonable load reply to polling messages, and the reply is delayed by a small time increasing with the local load, so that the first replies received are probably the most interesting. With this variation, REQUEST and CENTRAL show comparable performances.

In RADIO [11], information and decision are distributed too, but no broadcasts occur in normal use. The idea is the following. Currently underloaded workstations are linked by a distributed list (the ‘available list’) where each machine knows the identity of its successor and predecessor. Furthermore, every workstation in the network knows the identity of a head of the available list (the ‘manager’). Process transfers are negotiated directly between an overloaded workstation and an underloaded one, or indirectly via the manager that knows an available workstation (its successor in the available list). Broadcasts are necessary only when the manager crashes or when a workstation joins the process assignment facility (at boot time, for instance). The performances of RADIO are intermediary between those of CENTRAL and those of REQUEST (see table 1). The measured recovery time after a crash of the manager is 800 ms, to be compared to 18 s with CENTRAL.

3.2.8. Synthesis. Whereas all the information and location policies described above give significant improvement in process response time with respect to local scheduling alone, some of them are more attractive for networks of workstations. Algorithms unable to find an available workstation in the network (quality less than 1) should be discarded. OFFER is not extensible enough for an average size network. GLOBAL suffers from out-of-date information, so that attempted transfers may be rejected. Theimer and Lantz [78] conclude that for large size networks, CENTRAL is the best choice, whereas REQUEST is adequate for small size networks. This is our conclusion, too. Let us add however that RADIO may be a good alternative for medium size networks.

3.3. Transfer policies

The location policy and the information policy together define the algorithm used to find a suitable workstation for receiving a process, given that an overloaded workstation tries to get rid of a process. The transfer policy determines when a workstation should be declared ‘overloaded’, and whether a process migration is desirable. Two pitfalls must be avoided by transfer policies. The first one is ‘Role Reversal’ [38]: A is more loaded than B, and thus migrates a process to B, and the effect is that B becomes more loaded than A. The second one (‘Migrate for Nothing’) is related to the process execution time, which is a priori unknown: if the process transfer time is larger than the gain in execution time, the net result is negative.

3.3.1. System state. Most transfer policies for load sharing are based on local thresholds. When the local load is above some value $T$, the workstation is said to be ‘overloaded’. In order to avoid the ‘Role Reversal’ phenomenon, several mechanisms have been proposed. The simplest is that the loads of the sending machine and that of the receiving machine should differ by at least some ‘bias’ [71, 82]. Other mechanisms are defined in [63, 71, 66, 53]. Zhou and Ferrari [81, 82] studied the influence of the value $T$ on the performance.

$\dagger$ Residual time, for pre-emptive migration.
of RANDOM and GLOBAL algorithms. An adaptive method for setting $T$ is proposed in [62].

Transfer policies using a double threshold [38, 3] are an extension of the "bias" mechanism. At any time, the load may be in one of three intervals:

(i) \( \text{load} < \text{LOW} \): the machine is 'underloaded'. It may receive foreign processes.

(ii) \( \text{LOW} \leq \text{load} < \text{HIGH} \): the machine is 'normally loaded'. It will not accept new foreign processes.

(iii) \( \text{HIGH} \leq \text{load} \): the machine is 'overloaded'. It will try to send one or more processes to an underloaded machine.

Note that in this scheme, the transfer policy is concerned by HIGH only, LOW being used by the location policy. This points out the trade-offs between location policies, information policies and transfer policies. The double threshold scheme is very flexible (the values of LOW and HIGH need not be the same on all machines). Furthermore, it is well adapted to networks of workstations, because it can conciliate load sharing and personal use of workstations. For instance, if a user is only editing a file, its workstation may receive a compilation process without bothering the user. If the load index reflects the CPU utilization, the workstation will be 'underloaded' in this case. The value of HIGH is generally set statically and rather empirically in existing systems. An exception is the work of Alonso and Cova [3], who measured the average process response time on a network of four Sun-2 workstations under artificially generated load with different threshold values, but clearly more work is needed in this area.

### 3.3.2. Process eligibility

Given that a workstation is 'overloaded', it will try to get rid of a process. However, this has to be done carefully, in order to avoid the 'Migrate for Nothing' pitfall. Data collected on Unix systems [14, 75] show that most processes consume less than 1 s CPU, thus this is a real problem. Clearly, starting processes remotely as soon as a workstation is overloaded is not a good solution. Therefore, filtering techniques have been proposed in order to set the eligibility of processes for migration (the first three concern non-pre-emptive migration):

(i) In manual filtering [37, 1, 59, 73, 4, 78, 30, 60, 11], users invoke some particular command to indicate that a process is a candidate for migration. This method is very simple, but it is not transparent to the user.

(ii) Type filtering [14, 39, 49, 52] is a variation of manual filtering. Eligible processes are put by the user in a batch queue. Migration is thus restricted to non-interactive processes.

(iii) Name filtering [40, 14, 29, 74, 42, 75] is transparent to the user. The system maintains a list a command names that correspond to processes having presumably a long lifetime (such as compilation and text formatting), which are therefore eligible for migration. However, this scheme cannot take into account user written programs (such as simulations), nor the large variability in execution times (a compilation may end prematurely because of errors). The influence of the filtering rate is studied in [75].

(iv) Age filtering [14, 45] has been proposed for pre-emptive migration. The starting observation is that a large amount of long lived process are in fact very long lived processes (for instance, at least 44% of processes whose lifetime is already 1.0 s have in fact a lifetime of at least 2.0 s). This may be used for filtering: only long lived process are eligible for (pre-emptive) migration.

### 3.4. Load indices

When thresholds are involved in location policies or transfer policies, their value is that of some load index. In this section we review the different load indices that have been proposed. The broad objectives of load indices are discussed in [29].

The first requirement for a load index is to reflect processor activity. Most load indices are based on averaged CPU queue length. Better results are obtained by adding IO queue length [28], as is done in the Berkeley Unix load index. Other indices have been proposed ("Normal Response Time" [24, 1], 'idle process' [73]).

However, for networks of personal workstations, ownership should be taken into account too. The first way is to consider that a workstation is unavailable for receiving foreign processes as soon as the owner is logged in [59, 39]. This is very restrictive, since often owners do not log out even when they do not use their machine. The second way is to monitor user activity (keyboard and mouse in [18], keyboard, mouse and average CPU utilization in [49]). This is restrictive too, because when a user is only editing a file, his/her workstation will be declared busy. The third way, which is the most flexible, is to use a threshold transfer policy associated to a classical load index, such as the one of Berkeley Unix [3, 11].

### 3.5. Pre-emptive versus non-pre-emptive migration

In the previous sections, no assumption was made about the time by which a process is migrated from one workstation to another. For non-pre-emptive migration, a process migration may occur only when a process is started. For pre-emptive migration, a process migration may occur at any time.

Pre-emptive migration is far more complex than non-pre-emptive migration. Whether it is worthwhile is questionable. Results reported in [22] (probabilistic model) and in [46] (trace-driven simulation) show that pre-emptive migration may offer limited benefits over non-pre-emptive migration. More precisely, benefits appear when three conditions hold: (i) heterogeneous load; (ii) high global load; (iii) files are mainly local [45]. In networks of workstations, condition (i) is fulfilled, but condition (ii) is not, and condition (iii) does not hold with diskless workstations.

However, a motivation for pre-emptive migration in a network of workstations is personal use. When an owner reclaims his/her machine, what to do with the possible foreign process running there? Simple
solutions, such as killing them [60], or lowering their priority [37], are not satisfactory. Clearly, pre-emptive migration could be the right solution. Some systems support it (see section 5).

In fact, experience with pre-emptive migration leads to mitigated conclusions. [5] stresses implementation difficulties for Charlotte. Conclusions drawn from Sprite [19] indicate that pre-emptive migration can be used as a last resort to guarantee response time to the owner of a workstation, but is unlikely to be useful for load sharing.

4. Mechanisms

In this section we review and discuss the mechanisms which process assignment facilities rely on, and their relationships with process assignment strategies.

(i) Network interface: most local area networks support broadcasting. However, multicasting is not always available in hardware. If multicasting is not directly available, it can be made up with broadcast and software filtering, but this entails an extra overhead on all machines. Algorithms based on multicast are thus penalized.

(ii) File system structure: in most networks of workstations, the file system is distributed, and efficient mechanisms are available for remote file access. Non-pre-emptive process migration is especially attractive for diskless workstations, since running a process remotely rather than locally does not involve extra file access overhead: in both cases access to the necessary files is available via the network. On the other hand, when some executable files are replicated on some local disks, both information policy and location policy should take this information into account. This is still an open problem.

(iii) Taking heterogeneity into account: hardware heterogeneity may be easily taken into account. Resource requirements of programs and resource availability on machines (e.g., floating point coprocessor) may be integrated in information policy and location policy. The problem of different CPU types may be solved easily too: search paths for executable files may be set conditionally to CPU type. Furthermore, it is possible to weight load values by CPU speed in order to achieve some fairness between workstations.

However, operating system heterogeneity is far more difficult to cope with. In particular, pre-emptive migration is still unsolved. For non-pre-emptive migration, a strategy involving a 'service server' is proposed in [80].

(iv) Program interactivity: there are standard solutions for keeping user interactivity with remotely executing programs (the remote shell facility in Berkeley Unix is an example). However, some existing systems restrict process migration to non-interactive programs.

(v) Pre-emptive migration : the mechanisms involved are very complex. It is necessary to detach the process to migrate from its initial environment, to transfer its state and its context, and to attach the process to a new environment on the receiving machine, all this in a reliable and efficient way. Information to gather includes stack, registers, current directory, open file descriptors. In Unix, this information is scattered.

Table 2. Some implementations of load sharing facility.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Algorithm</th>
<th>Load index</th>
<th>Filtering</th>
<th>OS</th>
<th>PM</th>
<th>Inter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Server</td>
<td>[37]</td>
<td>CENTRAL</td>
<td>QCPU</td>
<td>by name</td>
<td>Cedar</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>NEST</td>
<td>[25]</td>
<td>OFFER</td>
<td>NRT</td>
<td>manual</td>
<td>modified</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MOS</td>
<td>[8]</td>
<td>PROBA</td>
<td>?</td>
<td>by name</td>
<td>modified</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Butler</td>
<td>[59]</td>
<td>CENTRAL</td>
<td># users</td>
<td>manual</td>
<td>UNIX</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>[39]</td>
<td>CENTRAL</td>
<td># users +</td>
<td>by type</td>
<td>UNIX</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Condor</td>
<td>[49]</td>
<td>CENTRAL</td>
<td>QCPU +</td>
<td>by type</td>
<td>UNIX</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>RANDOM</td>
<td>[4]</td>
<td>BSD</td>
<td></td>
<td>manual + threshold</td>
<td>modified</td>
<td>yes</td>
<td>?</td>
</tr>
<tr>
<td>OFFER</td>
<td>[52]</td>
<td>BSD</td>
<td></td>
<td>manual</td>
<td>UNIX</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OFFER</td>
<td>[73]</td>
<td>idle process</td>
<td></td>
<td>none</td>
<td>V-kernel</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>REM</td>
<td>[67]</td>
<td>OFFER</td>
<td>BSD</td>
<td>manual</td>
<td>UNIX</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>GATOS</td>
<td>[30]</td>
<td>REQUEST</td>
<td>BSD</td>
<td>manual</td>
<td>UNIX</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>CENTRAL</td>
<td>[13]</td>
<td>OFFER</td>
<td>QCPU</td>
<td>none</td>
<td>modified</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>GAMMON</td>
<td>[9]</td>
<td>OFFER</td>
<td>QCPU</td>
<td>none</td>
<td>UNIX</td>
<td>no</td>
<td>?</td>
</tr>
<tr>
<td>Siddle</td>
<td>[42]</td>
<td>REQUEST</td>
<td>BSD + free memory</td>
<td>manual + threshold</td>
<td>UNIX</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
which makes process migration difficult to implement [2].

There are some implementations above the Unix kernel [49,52], but performance seems poor. Implementations for Unix with modifications to the kernel provide better results [37,8,7,18,41. However, they are far from those obtained in distributed operating systems designed from scratch. Furthermore, in most implementations, system calls related to inter-process communications are not supported. A survey of process migration mechanisms may be found in [69].

5. Implementations

A summary of existing systems is given in table 2. The list is in no way exhaustive. Only systems described with enough details are included.

In the column 'Load index', 'QCPU' represents the length of the CPU queue, 'BSD' is the load index used in Berkeley Unix and 'NRT' stands for Normal Response Time. Column 'PM' and column 'Inter' indicate whether pre-emptive migration and program interactivity are supported.

No performance results are given in table 2, because a few references report results. Furthermore, when figures are provided, they cannot be compared (incompatible definitions for measured times, different hardware configurations). Some other implementations are reported in [32].

6. Perspectives

One can expect that research work in the area of load sharing will go on in the following directions.

(i) Automatic parallelization: automatic program parallelization that can be derived from process assignment facilities is 'large-grained'. When a program is made of several processes without communications between them, running each process on a different machine may provide a substantial speed up. A typical example is the make command in Unix: each of the several modules that will be linked into an executable program may be compiled in parallel, on as many different machines. A 'parallel make' command is available in Sprite [19], Gatos [31], and Isis [41].

(ii) Distributed virtual memory: with distributed virtual memory [47,58], a process may have access to memory pages on remote machines. This feature could provide an elegant solution to pre-emptive migration. Instead of sending the whole memory image when a process is migrated, the receiver could fetch pages on demand.

(iii) Object-oriented distributed systems: in some object-oriented distributed systems, objects may move between machines, and may contain data as well as executable code. Such systems provide a new approach to load sharing, since it is possible to migrate units of code smaller than a process [43].

7. Conclusion

Load sharing in networks of workstations may provide substantial benefits in process response time, even with the simplest policies. Many algorithms have been proposed. The choice should be made according to environment specificities (network interface, file system structure, size of the network). There is no clear winner.

A filtering policy for selecting which processes are candidates for migration is desirable in environments where many processes are short lived. Manual filtering is the most flexible, although not transparent to the user.

Non-pre-emptive migration may be easily implemented. Pre-emptive migration is very complex, and does not provide decisive improvements. However, it may be appropriate for permitting workstation owners to reclaim their machines.

Current load indices are not able to take into account all the parameters that influence process response time. For instance, main memory sizes and local disks are not considered when selecting a receiving machine.

Finally, distributed virtual memory and object-oriented distributed systems open promising perspectives.

References

[10] Bernard G 1989 Performances et systèmes répartis... de la modélisation à la réalisation PhD thesis Thése de Doctorat ès Sciences Informatiques, Université Paris XI
[12] Bonomi F and Kumar A 1988 Adaptive optimal load balancing in a heterogeneous multiserver system with a central job scheduler Proc. 8th Int. Conf. on Distributed
Load sharing in networks of workstations

for the conception and the mise en œuvre d’applications dans les systèmes répartis hétérogènes Phd thesis Thèse de Doctorat, Université Paris VI


[38] Harbas S R 1986 Dynamic process migration: to migrate or not to migrate Technical note csn-42 University of Toronto


[47] Li K 1986 Shared virtual memory in loosely coupled multiprocessors Phd thesis Yale University


[68] Simunic M 1990 Placement de tâches interactives sur un réseau de stations de travail Rapport de DEA Université Paris VI


[73] Stumm M 1988 The design and implementation of a decentralized scheduling facility for a workstation cluster 2nd IEEE Conf. on Computer Workstations (Santa Clara, CA) (Los Alamitos, CA: IEEE Computer Society)


[76] Tanenbaum A S and Van Renesse R 1985 Distributed operating systems ACM Comput. Surveys 17 419–70


[80] Wills C E 1989 A service execution mechanism for a distributed environment Proc. 9th Int. Conf. on Distributed Computing Systems (Newport Beach, CA) (Los Alamitos, CA: IEEE Computer Society)
