# Balancing Active Objects on a Peer to Peer Infrastructure

Javier Bustos-Jimenez, Denis Caromel, Alexandre Di Costanzo, Mario Leyton, Jose M. Piquer

# Balancing Active Objects on a Peer to Peer Infrastructure

Javier Bustos-Jimenez
Computer Science Department, University of Chile. Blanco Encalada 2120, Santiago, Chile.
School of Computer Science Engineering. Diego Portales University. Av. Ejercito 441, Santiago, Chile.
jbustos@dcc.uchile.cl

Denis Caromel Alexandre di Costanzo
Mario Leyton
INRIA Sophia-Antipolis, CNRS, I3S, UNSA.
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.
First.Last@sophia.inria.fr

Jose M. Piquer
Computer Science Department, University of Chile. Blanco Encalada 2120, Santiago, Chile.
jpiquer@dcc.uchile.cl

## Abstract

*We present a contribution on dynamic load balancing for distributed and parallel object-oriented applications. We specially target on peer to peer systems and its capability to distribute parallel computation, which transfer large amount of data (called* intensive-communicated *applications) among large number of processors. We explain the relation between active objects and processors load. Using this relation, and defining an order relation among processors, we describe our active object balance algorithm as a dynamic load balance algorithm, focusing on minimizing the time when active objects are waiting for the completion of remote calls. We benchmark a Jacobi parallel application with several load balancing algorithms. Finally, we study results from these experimentation in order to show that a peer to peer load balancing obtains the best performance in terms of migration decisions and scalability.*

## 1 Introduction

One of the main features of a distributed system is the ability to redistribute tasks among its processors. This requires a redistribution policy to gain in productivity by dispatching the tasks in such a way that the resources are used efficiently, i.e. minimizing the average idle time of the processors and improving applications performance. This technique is known as load balancing. Moreover, when the redistribution decisions are taken at runtime, it is called *dynamic load balancing*.

There are many definitions for *Peer-to-Peer* (P2P), in this work we use the definition from [15] of *Pure P2P*: *each peer can be removed from the network without any loss of network service*.

We present an active object load balancing algorithm based on well known algorithms [16] and adapted for a P2P infrastructure. This algorithm is a dynamic, fully distributed load balancer, which reacts to load perturbations on the processor and the system. Our main contributions are: the relation between processors load and active objects balancing, the use of an order relation (see section 4) to improve the parallel application performance, and exploiting P2P to improve the balance algorithm.

We believe our proposal is useful because the number of messages it uses in a self organized environment like P2P is limited by a constant independently of the number of acquaintances. This allows us to obtain a scalable by definition, and experimentally efficient, load balancing algorithm.

Our algorithm has been implemented within ProActive [1], an open source Java middleware which aims to achieve seamless programming for concurrent, parallel, distributed and mobile computing implementing the active object programming model. Using this model, intensive-communicated parallel applications are developed (see a practical application example on [11]).

While many dynamic load balancing algorithms have been presented and studied in depth [17, 16], previous work [4] shows that, for intensive-communicated parallel appli-

cations, new constraints (like bandwidth) appear, and most of those algorithms become not applicable.

This work is organized as follows. Section 2 presents ProActive as an implementation of *active object programming model*. Section 3 describes our Peer to Peer infrastructure. Section 4 explains the fundamentals of our active objects load balancing algorithm. Section 5 shows implementation issues and benchmarking of our algorithm with a Jacobi parallel application. Finally, conclusions and future work are presented.

## 2 ProActive and the Active Object Programming Model

The ProActive middleware is a 100% Java library, which aims to achieve seamless programming for concurrent, parallel, distributed and mobile computing. As it is built on top of standard Java API, it does not require any modification of the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine.

The base model is a uniform *active object* programming model. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls, automatically stored in a queue of pending requests (called *service queue*). When the queue is empty, the active objects waits until the arrival of a new request, this state is known as *wait-for-request*.

Active objects are remotely accessible via method invocation. Method calls with active objects are asynchronous with automatic synchronization. This is provided by automatic *future objects* as a result of remote methods calls, and synchronization is handled by a mechanism known as *wait-by-necessity* [6].

Another communication mechanism is the *group communication* model. Group communication allows triggering method calls on a distributed group of active objects with compatible type, dynamically generating a group of results [2].

ProActive provides a way to move any active object from any Java Virtual Machine (JVM) to any other one, this is called *migration* mechanism [3]. An active object with its: pending requests (method calls), futures, and passive (mandatory non-shared) objects may migrate from JVMs to JVMs through the *migrateTo(...)* primitive. The migration can be initiated from outside the active object through any public method but it is the responsibility of the active object to execute the migration, this is known as *weak migration*. Automatic and transparent forwarding of requests and replies provide location transparency, as remote references towards active mobile objects remain valid.

## 3 Peer-to-Peer Infrastructure

The goal of the Peer to Peer infrastructure is to use spare CPU cycles from institutions' desktop computers combined with grids and clusters. Managing different sort of resources (grids, clusters and desktop computers) as a single network of resources with a high instability of them requires a fully decentralized and dynamic approach. Therefore, mimicking data P2P networks is a good solution for sharing a dynamic JVM network, where JVMs are the shared resources. Thereby, the ProActive infrastructure is a P2P network, which shares JVMs for computation. This infrastructure is completely self-organized and fully configurable. Main features and technical aspects are explained below.

The main characteristic of the infrastructure is the peers high volatility because those peers are users' desktop computers. This is why the infrastructure aims at maintaining a created JVMs network alive while available peer exists, this is called *self-organizing*. When it is not possible to have exterior entities, such as centralized servers, which maintain peer databases, all peers should be capable of staying in the infrastructure by their own means. A widely used strategy for achieving self organization consists on maintaining, for each peer, a list of its neighbors.

This idea was selected to keep the infrastructure up. All peers have to maintain a list of *acquaintances*. At the beginning, when a fresh peer joins the network, it only knows peers from a list of potential network members. Because not all peers are always available, knowing a fixed number of acquaintances is a problem for peers to stay connected in the infrastructure.

Therefore, the infrastructure uses a specific parameter called *Number of Acquaintances* (NOA): the minimum number of known acquaintances for each peer. Peers update their acquaintance list every *Time to Update* (TTU), checking their own acquaintances' lists to remove unavailable peers, and if the longer of the list is less than NOA, discover new peers. To discover new acquaintances, peers send exploring messages through the infrastructure. Availability is verified by sending a *heartbeat* to the acquaintances, which is sent every TTU.

As previously said, the main goal of this P2P network is to provide an infrastructure for sharing computational nodes (JVMs). The resource query mechanism used is similar to the Gnutella [8] communication system: Breadth-First Search algorithm (BFS). The system is message-based with application-level routing. Messages are forwarded to each acquaintance, and if the message has already been received, it is dropped. The number of hops that a message can take is limited with a *Time-To-Live* (TTL) parameter. Message transport is provided by ProActive group communication in an asynchronous way.

The Gnutella BFS algorithm received many justified crit-

ics [14] on scalability and bandwidth usage. ProActive asynchronous method calls with future objects, provides an enhancement to the basic BFS. Before forwarding a message, computation available peer waits for an acknowledgment from the requester. After an expired timeout or a non-acknowledgment, peers do not forward the message. However, the message is forwarded until the end of TTL or until the number of asked nodes reaches zero. The message contains the initial number of requested nodes, decreasing its value each time a peer shares its node. For peers, which are occupied, the message is forwarded as normal BFS.

A long term infrastructure with INRIA lab desktop computers was deployed, and we have experimented massive parallel applications for one year. In our experiments, with a network of 250 machines connected at 100 Mb/s Ethernet connections the message traffic has not yet posed problems.

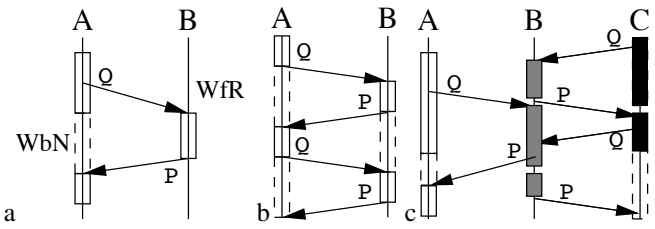


**Figure 1. Different behaviours for active objects request (Q) and reply (P): (a) B starts in wait-for-request (WfR) and A made a wait-by-necessity (WfN). (b) Bad utilization of the active object pattern: asynchronous calls become almost synchronous. (c) C has a long waiting time because B delayed the answer.**

## 4 Active Objects Balance Algorithm

Dynamic load balancing on distributed systems is a well studied issue. Most of the available algorithms (see algorithms compilations on [4, 16]) focus on fully dedicated processors with homogeneous networks, using a threshold monitoring strategy and reacting to load imbalances. On P2P networks, heterogeneity and resource sharing (like processor time) are key aspects and most of these algorithms become inapplicable, producing poor balance decisions to low capacity processors and compensating with extra migrations.

Moreover, due the fact that processors connected to a P2P network share their resources not only with the network but also with the processor owner, new constraints like reaction time against overloading and bandwidth usage become relevant.

In this section, we present an adaptation of a well known load balancing algorithm for P2P active object networks. First we present the relation between active object service and processing time, followed by the algorithm details.

### 4.1 Active Objects and Processing Time

When an active object waits idly (without processing), it can be on a *wait-for-request* or a *wait-by-necessity* state (see figure 1). While the former represents a sub utilization of the active object, the latter means some of its requests are not served as quickly as they should. The longer waiting time is reflected on a longer application execution time, and thus a lesser application performance. Therefore, we focus on a reduction in the *wait-by-necessity* time delay.

Even though the balance algorithms will speed up applications like figure 1 (b), we will not consider this kind of behaviour, because the time spent by message services is so long that the usage of *futures* is pointless. In this sort of application design, asynchronism provided by *futures* will unavoidably become synchronous. This is the same behaviour experienced when using an active object as a central server. Migrating the active object to a faster machine will reduce the application response time but will not correct the application design problem.

Therefore, we focus on the behaviour presented by figure 1 (c), where the active object on `C` is delayed because the active object on `B` has not enough free processor time to serve its request. Migrating the active object from `B` to a machine with available processor resources speeds up the global parallel application. This happens, because `C` *wait-by-necessity* time will shorten, and `B` will decrease its load.

### 4.2 Active Objects Balance Algorithm on a Central Server Approach

Suppose function called `load(A,t)` exists, which gives the usage percentage of processor `A` since `t` units of time. Defining two threshold: `OT` and `UT` (`OT > UT`), we say that a processor `A` is *overloaded* (resp. *underloaded*) if `load(A,t) > OT` (resp. `load(A,t) < UT`).

The load balancing algorithm uses a central server to store system information, processors can register, unregister and query it for balancing. The algorithm is as follows:

Every `t` units of time

1. if a processor `A` is underloaded, it registers on the central server,

2. if a processor `A` was underloaded in `t-1` and now it has left this state, then it unregisters from the central server,

3. if a processor A is overloaded[1], it asks the central server for an underloaded processor, the server randomly choose a candidate from its registers and gives its reference to the overloaded processor.

4. The overloaded processor A migrates an active object to the underloaded one.

This simple algorithm satisfies the requirements of minimizing the reaction time against overloadings and, as we explained on section 4.1, speeds up the application performance. However, it works only for homogeneous networks.

In order to adapt this algorithm to heterogeneous (in processing capacity) computers, we introduce a function called `rank(A)`, which gives the processing speed of A. Note that this function generates a total order relation among processors.

The function `rank` provides a mechanism to avoid processors with low capacity, concentrating the parallel application on the higher capacity processors. It is also possible to provide the server with `rank(A)` at registration time, allowing the server to search for a candidate with similar or higher rank. This would produce the same mechanism, with the drawback of adding the search time to reaction time against overloading. In general, all search mechanism of *the best* unloaded candidate in the server will add a delay into server response, and consequently in reaction time.

Before implementing the algorithm, we studied our network and selected a processor B[2] as *reference* in terms of processing capacities. Then, we modified the previous algorithm to:

Every t units of time

1. If a processor A is overloaded, it asks the central server for an underloaded processor, the server randomly choose a candidate from its registers and gives the reference to the overloaded processor.

2. If A is not overloaded, it checks if `load(A,T) < UT*rank(A)/rank(B)`, if true then it registers on the central server. Else it unregisters from the central server.

3. Overloaded processor A migrates an active object to the underloaded one.

### 4.3 Active Object Balancing using P2P Infrastructure

Looking for a better underloaded processor selection, we adapted the previous algorithm, using a subset of peer acquaintances from the P2P infrastructure (defined on section 3) to coordinate the balance.

Suppose the number of computers on the P2P network is $N$, large enough to suppose them independents on their load. If $p$ is the probability of having a computer on an underloaded state, and the acquaintances subset size is $n << N$, then the probability of having at least $k$ responses is

$$\sum_{i=k}^{n} \binom{n}{k} p^i (1-p)^{n-i}$$

Therefore, having an estimation of $p$, a good selection of the parameter $n$ permits a reduction on the bandwidth used by the algorithm with a minimal addition on reaction time. For instance, using the pairs ($p = 0.8$,$n = 3$) or ($p = 0.6$,$n = 6$), one has a response probability greater than $0.99$.

The algorithm for P2P networks is: Every t units of time

1. If a processor A is overloaded, it sends a balance request and the value of `rank(A)` to a subset $n$ of its acquaintances (using group communication).

2. When a process B receives a balance request, it checks if `load(B,T) < UT` and `rank(B)` $\geq$ `rank(A)` $-\epsilon$ (where $\epsilon > 0$ is to avoid discarding similar, but unequal processors), if true, then B sends a response to A.

3. When A receives the first response (from B), it migrates an active object to B. Further responses for the same balance request can be discarded.

### 4.4 Migration

A main load balancing algorithm problem is the *migration time*, defined as the time interval since the processor requests an object migration, until the objects arrives at the new processor [3]. Migration time is undesirable because the active object is halted while migrating. Therefore, minimizing this time is an important aspect on load balancing.

While several schemes try minimizing the *migration time* using distributed memory [7] (not yet implemented in Java), or migrating idle objects [10] (almost inexistent on intensive-communicated parallel applications), we exploit our P2P architecture to reduce the migration time. Using a group call, the first reply will come from the nearest acquaintance, and thus the active object will spend the minimum time traveling to the closest unloaded processor known by the peer.

The migration time problem is not the only source of difficulty. There is a second one: the *ping pong effect*. This appears when active objects migrate forwards and backwards

[1] On a previous work [4] it was shown that overloaded initiated algorithms have the best reaction time on load balancing

[2] Choosing the correct processor B requires further research, but for now the median has proved reasonable approach.

[3] In ProActive, an object abandons the original processor upon confirmation of arrival at the new processor.

between processors. This trouble is conceptually avoided by our implementation by choosing the migrating active object as the one with *shortest service queue*. During the migration phase, the active object pauses its activity and stops handling requests. For a recently migrated active object, all new requests are waiting in the queue, and will only begin to be treated after the migration has finished. Therefore, a freshly migrated object generally has a longer queue than similar objects on the new processor, thus a low priority for moving.

By experimentation (see section 5), we have observed that these migration problems are not present when using this approach.

## 5 Experimentation

Algorithms were deployed on a set of 25 of INRIA lab desktop computers, having 10 Pentium III 0.5 - 1.0 Ghz, 9 Pentium IV 3.4GHz and 6 Pentium XEON 2.0GHz, all of them using Linux as operating system and connected by a 100 Mbps Ethernet switched network. With this group of machines we used the P2P infrastructure to share JVMs. Using our previous experiences (see section 3), we configured the P2P infrastructure with: `TTU` at 10 minutes, `NOA` at 10 peers and `TTL` at 5 hops. At first only one peer was chosen, and other peers used it to join the infrastructure. Functions `load()` (resp. `rank()`) of section 4.2 and 4.3 were implemented with information available on `/proc/stat` (resp. `/proc/cpuinfo`). Load balancing algorithms were developed using *ProActive* on Java 2 Platform (Standard Edition) version 1.4.2.

In our experience, using our knowledge of the lab networks, we experimentally defined the algorithm parameters as `OT` = 0.8 (to avoid swapping on migration time), and `UT` = 0.3; in order to have, in normal conditions, 80% of desktop computers on underloaded state [4]urther research on how to define this parameters, besided experimentally, is required.

Since the *cpu speed* (in MHz) is a constant property of each processor and it represent its processing capacity, and after a brief analysis of them on our desktop computers, we define the `rank` function as: $rank(P) = \log_{10} speed(P)$, with $\epsilon = 0.5$.

When implements the algorithm, a new constraint came to light: all load status are checked each `t` units of time (called *update time*). If this *update time* is less than migration time, extra migrations which affects the application performance could be produced. After a brief analysis of migration time, and to avoid network implosion, we assume a variable $\tilde{t}$ which follows an uniform distribution and experimentally define the update time as:

$$t_{update} = 5 + 30\,\tilde{t}(1 - load)[sec],\ (load \in [0, 1])$$

[4]F

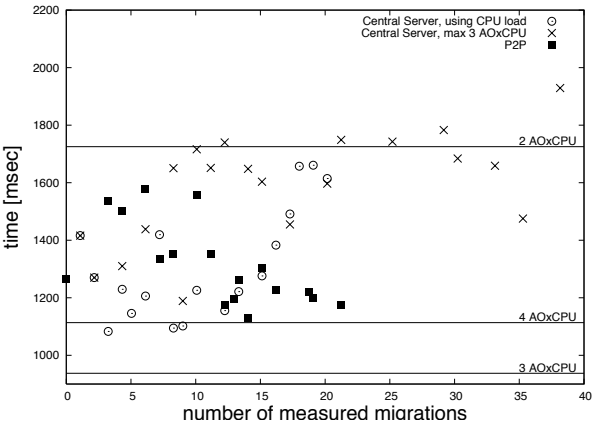


**Figure 2. Impact of load balancing algorithms over Jacobi calculus**

This formula has a constant component (migration time) and a dynamic component which decrease the update time while the load increase, minimizing the overload reaction time.

We tested the impact of our load balancing algorithm over a concrete application: the *Jacobi* matrix calculus. This algorithm performs an iterative computation on a square matrix of real numbers. On each iteration, the value of each point is computed using its value and the value of its matrix neighbors in their last iteration. We divided a 3600x3600 matrix in 36 workers all equivalents, and each worker communicates with its direct matrix neighbors.

We randomly distributed Jacobi workers among 16 (of 25) machines, measuring the execution time of 1000 sequential calculus of Jacobi matrices. First, we used the central server algorithm defined on section 4.2 (having a cpu clock of $3GHz$ as reference) and then using the P2P version defined on section 4.3 (having $n = 3$). Measured values of these experiences can be found in figure 2.

Looking for lower bounds in Jacobi execution time, we measured the mean time of Jacobi calculus for 2, 3 and 4 workers by machine, using the computers with higher *rank* and without load balancing. Horizontal lines on figure 2 are the values of this experience. Applying the information from the *non-balanced* experience, we tested the *number of actives objects* as a load index, defining `UT=3,OT=4` to have around 3 active objects per machine. Measured values for this experience are represented by the `x` symbol in figure 2.

While using the information from the *non-balanced* experience seemed to be a good idea, the heterogeneity of the P2P network produced the worse scenario: large number of migrations and bad migration decisions, therefore poor performance on Jacobi calculus. Using CPU usage as load in-

dex had better performance than the previous case: while the central server oriented algorithm produced low mean times for low rate of migrations (an initial distribution near to the optimal), P2P oriented algorithm presents better performance while the number of migrations increase. Moreover, considering the addition of migration time on Jacobi calculus performance, P2P balance algorithm produces the best migration decisions only using a minimal subset of its neighbors. The use of this minimal subset produces also a minimization in number of messages for balance coordination. This fact and the neighbor approach of our P2P network provide automatically scalability conditions for large networks.

## 6 Related Work

While dynamic load balancing is a well studied issue for distributed systems (see algorithms compilations on [4, 16]), their applications over P2P networks are still on exploratory phase. Most of the research on load balancing for P2P networks are based on a structured approach using a *distributed hash tables* (DHT) [13], where each machine can be represented by several keys, and parallel applications are mapped into this DHT. Load balancing becomes now a search problem on key/data spaces [9]. Our P2P infrastructure is not an another distributed hash table (DHT), because the shared resource is computational node (a JVM) it is not necessary to identify resources as unique as in P2P data. The infrastructure intents to provide an overlay network of JVMs and to supply nodes for applications.

Another approach for load balancing on P2P environments is the use of *agents* which traverse the network equalizing the load among them. Those agents follows a model of an ant colony [12, 5], *carrying* load among computers, and making the system eventually stable. This schemes focus on a load equalization more than a minimization of the *reaction time* against overloading. Therefore, they are not comparable against our scheme.

## 7 Conclusions

We have introduced a P2P dynamic load balancer for active objects, focusing on intensive-communicated parallel applications. We started introducing the P2P infrastructure developed for ProActive and the relation between active objects and CPU load. Then, an order relation to improve the balance was defined. The case study showed that, if the number of migrations increase (this can occurs due an non-optimal distribution or due the dynamic behaviour of the P2P network), the performance (on reaction time and migration decisions) increases for P2P the P2P algorithm and decreases for the central server approach. Also, the load balancing algorithm exploits the P2P architecture to provide scalability conditions for large networks.

As future work, for the P2P infrastructure we will design solutions with dynamic TTL, to avoid network flooding due to BFS algorithm. Concerning load balancing, further parameter (thresholds, update time) fine tuning is required to obtain faster reaction time and lower bandwidth usage, in order to speed up the parallel application.

## References


[1] O. G. at INRIA Sohpia-Antipolis. "Proactive, the java library for parallel, distributed, concurrent computing with security and mobility". http://www-sop.inria.fr/oasis/proactive/, 2002.

[2] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press.

[3] F. Baude, D. Caromel, F. Huet, and J. Vayssiere. Communicating mobile active objects in java. In *Proceedings of HPCN Europe 2000*, volume 1823 of *LNCS*, pages 633–643. Springer, May 2000.

[4] J. Bustos, D. Caromel, and J. Piquer. Information collection policies: Towards load balancing of communication-intensive parallel applications. http://www.dcc.uchile.cl/~jbustos/Pub/intensive.pdf, 2005. Preprint.

[5] J. Cao. Self-organizing agents for grid load balancing. In *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 388–395, 2004.

[6] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[8] Gnutella. http://www.gnutella.com.

[9] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. *Lecture Notes in Computer Science*, 2735:68–79, January 2003.

[10] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 8, 1994.

[11] F. Huet, D. Caromel, and H. Bal. A high performance java middleware with a real application. In *Proc. of High Performance Computing, Networking and Storage (SC2004), Pittsburgh, USA*, 2004.

[12] A. Montresor, H. Meling, and O. Babaoglu. Messor: Load-Balancing through a Swarm of Autonomous Agents. In G. Moro and M. Koubarakis, editors, *Proceedings of the 1st Workshop on Agent and Peer-to-Peer Systems (AP2PC'02)*, number 2530 in Lecture Notes in Artificial Intelligence, pages 125–137, Bologna, Italy, 2003. Springer-Verlag.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[14] J. Ritter. Why Gnutella can't scale. No, really., 2001. http://www.darkridge.com/~jpr5/doc/gnutella.html.

[15] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In IEEE, editor, *2001 International Conference on Peer-to-Peer Computing (P2P2001)*, Department of Computer and Information Science Linko pings Universitet, Sweden, august 2001.

[16] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.

[17] M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Trans. Softw. Eng.*, 15(11):1444–1458, 1989.