Balancing Active Objects on a Peer-to-Peer Infrastructure

Javier Bustos[1,2], Denis Caromel[2], Alexandre di Costanzo[2], Mario Leyton[2], and Jose M. Piquer[1]

[1] Computer Science Department, University of Chile. Blanco Encalada 2120, Santiago, Chile.
{jbustos,jpiquer}@dcc.uchile.cl
[2] INRIA Sophia-Antipolis, CNRS, I3S, UNSA. 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.
First.Last@sophia.inria.fr

Abstract. We present a contribution on dynamic load balancing for distributed and parallel object-oriented applications. We specially target on peer to peer computing 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 our algorithm has 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 enables, 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 know as load balancing. Moreover, when the redistribution decisions are taking on runtime, it is called *dynamic load balancing*.

There are many definitions for *Peer to Peer* (P2P), many of them are similar to other distributed infrastructures, such as Grid, client/server, etc. Two of the best definitions are provided by Oram in [11], and Schollmeier in [13]. From these definitions and more generally, P2P focus on sharing resources, decentralization, instability, and fault tolerance.

We present an active object load balancing algorithm based on well known algorithms [14] 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 contribution 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 the exploit of P2P to improve the balance 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 particular example on [10]).

While many dynamic load balancing algorithms has been presented and studied in depth [16, 15, 14], a previous work [4] shows that, for intensive-communicated parallel applications, new constraints (like bandwidth) appear, and most of those algorithms become not applicable.

This work is organized as follow. 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 that are automatically stored in a queue of pending requests. If the queue is empty, the active objects waits until the arriving 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* for results of remote methods calls and synchronization is handled by a mechanism known as *wait-by-necessity* [5]. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee.

An another communicating way is the *group communication* model. Group communication enables to trigger method calls on a distributed group of active objects of the same compatible type, with a dynamic generation of groups of results. It has been shown in [2] that this group communication mechanism, plus a few synchronization operations (WaitAll, WaitOne, etc.), provides quite similar patterns for collective operations such as those available in e.g. MPI, but in a language centric approach.

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), its futures, its passive (mandatory non-shared) objects may migrate from JVMs to JVMs through the *migrateTo(...)* primitive. The migration may be initiated from outside through any public method but it is the responsibility of the active object to execute the migration, it is 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 needs 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 particularity of the infrastructure is the peers high volatility due to principally those peers are users' computers. That's why it aims to maintain a created JVMs network alive while there are available peers, this is called *self-organizing*. On condition that it is not possible to have exterior entities, as such centralized servers, which maintain peer databases. All peers should be enabled to stay in the infrastructure by their own means. There is a solution, which is widely used in data P2P networks; this consists of maintain for each peers a list of their neighbors.

In the same way, 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 has just joined the network, it knows only peers from a list of potential members of the network. Because not all peers are always available, knowing a started 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 Acquaintance* (NOA). This is the minimum size of the list of acquaintances of all peers. Peers keep frequently their lists up-to-date that are why a new parameter must be introduced: *Time to Update* (TTU). This is the frequency, which peers must check theirs own acquaintances' lists to remove unavailable peers and in the case of the longer of the lists is less than the NOA, discovering new peers. To discovering new acquaintances, peers send exploring messages through the infrastructure. Then, peers verify the others availabilities by sending a *heartbeat* to theirs 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. The message is forwarded to each acquaintance and if the message has already been received, it is not forwarded further. 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's BFS got a lot of justified critics [12] for scaling, bandwidth using, etc. Thanks to ProActive asynchronous method with automatic futures, which provides an advance to the basic BFS. Before forwarding the message, peers, which are free for

accepting computation, wait 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 nodes asked reach zero. The message contains the initial number of nodes asked and it is decrease each time a peer shares its node. For peers, which are occupied, the message is forwarded as normal BFS.

We made a permanent infrastructure with INRIA lab desktop computers and we have been experiment about massive parallel applications for one year. In our experiments, the network size of 250 machines with 100 Mb/s Ethernet connections the message traffic has not yet posed significant problem. In the context of the second Grid Plugtests [6], we are trying to beat the world record for finding all solutions of the n-Queens problem with $n = 25$.

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, 14]) focus on fully dedicated processors with homogeneous networks, using a threshold monitoring strategy and reacting to load imbalances.

Nevertheless, on P2P networks heterogeneity and resource sharing (like processor time) are key aspects. Therefore, most of these algorithms become inapplicable. 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 our algorithm on a reduction in the *wait-by-necessity* time delay.

A key assumption in our model is the utilization of asynchronous calls on parallel applications (like figure 1 (a) and (c)).

Figure 2 presents active object services and their CPU usage during periods of time `t`. Message calls are represented by arrows and active object services by grey areas, there can be a delay between the call and the service.

We consider the behaviour shown in figure 2 (left) similar to the one in figure 1 (b): 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

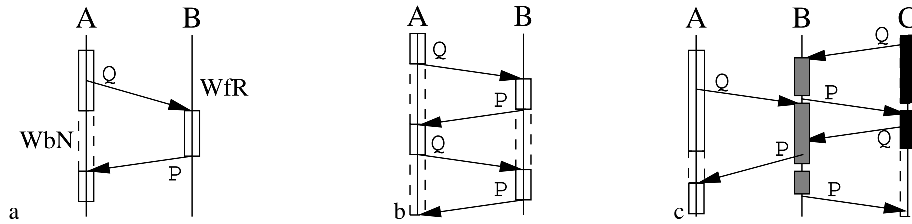

Fig. 1. Different behaviours for active objects request (Q) and reply (P): (a) B starts in wait-for-request (WfR) and A made a wait-by-necessity (WfN). (b) Bad utilization of the active object pattern: asynchronous calls become synchronous. (c) C has a long waiting time because B delayed the answer.

object as a central server. Migrating the active object to a faster machine may reduce the application response time but will not correct the application design problem.

We focus on the behaviour presented by figure 2(right). This figure shows an *overloaded* CPU (cpu2 has no idle time) and another CPU (cpu1) doing *wait-by-necessity*.

The active object on cpu1 is delayed because the active object on cpu2 has not enough free processor time to serve its request. Migrating the active object from cpu2 to a machine with available processor resources speeds up the global parallel application. This happens, because cpu1 *wait-by-necessity* time will shorten, and cpu2 will decrease its load.

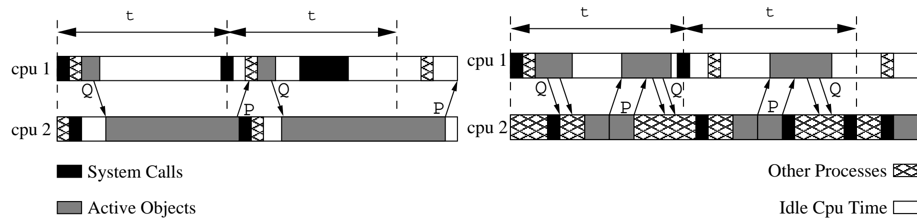

Fig. 2. CPU overloading: (left) A long time running service. (right) Shared CPU among different user processes.

4.2 Active Objects Balance Algorithm

Suppose function called load(A,t) exists, which gives us the usage percentage of processor P 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 original 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[3], it asks to 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, it speeds up the application performance. However, it works only for homogeneous networks.

In order to adapt this algorithm to heterogeneous networks, we introduce a function called `rank(A)`, which gives us 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 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 our algorithm implementation, we studied the network and selected a processor `B`[4] as *reference* in terms of processing capacities. Then, we modified our algorithm to:

Every `t` units of time

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

If p is the probability of having a computer on an underloaded state, and the acquaintances subset size is n, then using this approach (all computers are independents)

[3] On a previous work [4] we showed that overloaded initiated algorithms have the best reaction time on load balancing

[4] Choosing the correct processor B requires further research, but for now the median is a reasonable approach

the probability of having at least k responses is

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

Therefore, a good election of the parameter n permits a reduction on the bandwidth used by the algorithm with a minimal addition on reaction time. For instance, if $p = 0.8$ and $n = 3$ one has a response probability of 0.99, and using $p = 0.6$ and $n = 6$ one has the same response probability.

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)`-ϵ (where $\epsilon > 0$ is to avoid discarding similar, but unequal, processors), if true, then it 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 Time

A main load balancing algorithm problem is *migration time*, defined as the time interval since the processor requests an object migration, until the objects arrives at the new processor [5]. 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 to minimizing *migration time* using distributed memory [7] (not yet implemented in Java), or migrating idle objects [9] (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.

Moreover, migrating the active object with the shortest service queue length (see section 2) will minimize the amount of data which has to traverse the network, minimizing the migration time.

5 Experimentation

Our P2P load balancer was deployed on a set of 25 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 operative system and connected by a 100 Mbps Ethernet network. Functions `load()` (resp. `rank()`) of previous section are 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, we experimentally defined the algorithm parameters as:

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

– OT = 0.8
– UT = 0.3

Having, in normal conditions, 80% of desktop computers in underloaded state.

Due the fact that 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:

– $rank(P) = \log_{10} speed(P)$
– having $\epsilon = 0.5$

On implementation time, a new constraint came to light: in our algorithms, 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 avoiding network collisions, we experimentally defined update time as:

$$if\ load = 0\ t_{update} = 30$$

$$else,\ \tilde{t}\ follows\ a\ normal\ distribution$$

$$t_{update} = 5 + \frac{30 + 10\,\tilde{t}}{1 + load}[sec]$$

This formula has a constant component (migration time) and a dynamic component which decrease the update time while the load increase, looking for a minimization on reaction time.

We tested the impact of our load balancing algorithm over a real 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 neighbors for their last iteration. We divided a 3600x3600 matrix in 36 workers all equivalents, and each worker communicates with its direct acquaintances. We measured the execution time of 1000 sequential calculus of Jacobi matrices.

Looking for lower bounds on this parallel calculus, we calculate the mean time of Jacobi calculus for 2,3 and 4 workers by machine, using the computers with higher *rank*. Horizontal lines on figure 3 are the mean values of this experience. Therefore, we tested our algorithm using the central server algorithm, having all workers randomly distributed among 16 (of 25) machines and using as reference a cpu clock of $3000MHz$.

Using the information of the *non-balanced* experience, we test the number of actives objects as a load index, and `UT=2,OT=4`. Mean values from this experience are represented by the symbol `x` on figure 3.

We implemented the central server and P2P algorithms from section 4.1. In figure 3, measured values for the first one are represented by white circles and for P2P by boxes. Having, in normal conditions, 80% of computers in underloaded state, and looking for a reduction in the number of messages among acquaintances, we used a subset of 3 neighbors, which give us a probability greater than 0.99 of having a acquaintance response in case of overloading.

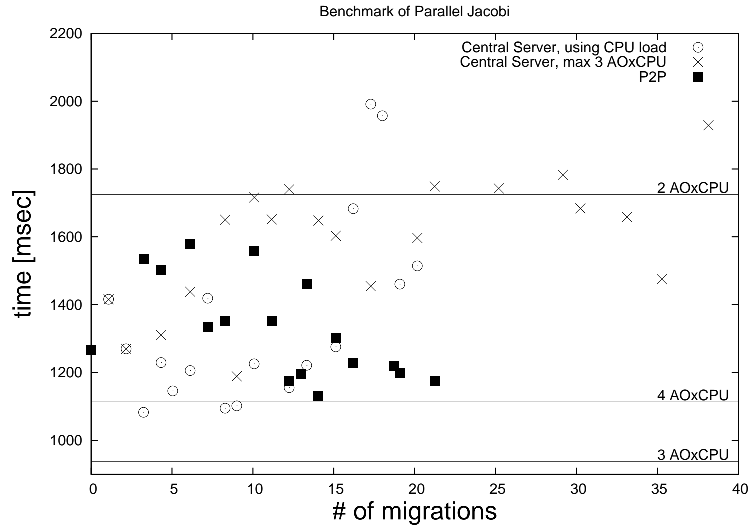

Fig. 3. Impact of load balancing algorithms over Jacobi calculus

While using the static information seems to be a good idea, the heterogeneity of the P2P network produces the worse scenario: large number of migrations and bad migration decisions, therefore poor performance on Jacobi calculus. Using ready queue as load index produces a better performance than the previous case, while the central server oriented algorithm produces low mean times for low rate of migrations (a initial distribution near to the optimal), P2P oriented algorithm presents almost the same performance independently of number of migrations. Moreover, considering the addition of migration time on Jacobi calculus performance, our balance algorithm produces the best migration decisions only using a minimal subset of its neighbors. For this reason, it presents scalability conditions for large networks.

6 Related Work

X do the same but now it is not applicable, Y did the same but for homogeneous networks and Z do the same without worried of the reaction time.

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, a order relation to improve the balance is defined. A case study presents the impact of our approach over a parallel application, showing the best performance on migration decision and scalability for our P2P based algorithm. As future work, for the P2P

infrastructure we will prospect solutions with not fixed TTL to avoid network flooding due to BFS algorithm. It is the continued goal of this work to search for the best load index and parameters for our algorithm, looking for the best performance in bandwidth, reaction time and parallel application speed.

References

1. Oasis Group at INRIA Sohpia-Antipolis. "Proactive, the java library for parallel, distribuited, concurrent computing with security and mobility". http://www-sop.inria.fr/oasis/proactive/, 2002.
2. Laurent Baduel, Françoise Baude, and Denis 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. Francoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssier e. Communicating mobile active objects in java. In *Proceedings of HPCN Europe 2000*, volume 1823 of *LNCS*, pages 633–643. Springer, May 2000.
4. Javier Bustos, Denis Caromel, and Jose Piquer. Information collection policies: Towards load balancing of communication-intensive parallel applications. http://www.dcc.uchile.cl/˜jbustos/Pub/intensive.pdf, 2005.
5. Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
6. ETSI and INRIA. http://www.etsi.org/plugtests/GRID.htm.
7. Matteo Frigo, Charles E. Leiserson, and Keith 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. Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 8, 1994.
10. Fabrice Huet, Denis Caromel, and Henri Bal. A high performance java middleware with a real application. In *Proc. of High Performance Computing, Networking and Storage (SC2004), Pittsburgh, USA*, 2004.
11. Andy Oram. *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Sebastopol, CA, 2001.
12. Jordan Ritter. Why Gnutella can't scale. No, really., 2001. http://www.darkridge.com/ jpr5/doc/gnutella.html.
13. Rudiger 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.
14. Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.
15. 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.
16. M. J. Zaki, Wei Li, and S. Parthasarathy. Customized dynamic load balancing for a network of workstations. In *Proceedings of the High Performance Distributed Computing (HPDC '96)*, page 282. IEEE Computer Society, 1996.