# Adaptive Placement of Method Executions within a Customizable Distributed Object-Based Runtime System

Design, Implementation, and Performance

Michel Banâtre, Yasmina Belhamissi, Valérie Issarny,
Isabelle Puaut, Jean-Paul Routeau
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE

*Fifteenth International Conference on Distributed Computing Systems*
*Vancouver, Canada, June 1995.*

## Abstract

*This paper presents the design and implementation of a mechanism aimed at enhancing the performance of distributed object-based applications. This goal is achieved by means of a new algorithm implementing placement of method executions that adapts to processors' load and to objects' characteristics, the latter allowing to approximate the cost of methods' remote execution. The behavior of the proposed placement algorithm is examined by providing performance measures obtained from its integration within a customizable distributed object-based runtime system. In particular, the cost of method executions using our algorithm is compared with the cost resulting from the standard placement technique that consists of executing any method on the storing node of its embedding object.*

## 1 Introduction

Definition of object-oriented languages with support for distributed programming has given rise to several proposals since the last fifteen years. First proposals introduced distributed object-based programming systems [1]. However, the efficiency of this approach has been too poor to be unanimously accepted by the distributed systems community. One way to improve the performance of distributed object-based applications is to integrate a load balancing strategy within the underlying runtime system.

Existing load balancing strategies fall into two categories depending on whether they are static or dynamic. Solutions of the former category are adequate for parallel programming environments where applications can be statically assigned a set of processors. They are not satisfactory for a distributed environment whose resources are shared among independent applications. Dynamic load balancing algorithms can be classified according to their information, transfer, and negotiation policies [2]. The *information policy* determines how workstations communicate with one another to exchange state information. The *transfer policy* states when to attempt a job transfer and which job to transfer. The *negotiation* (or *location*) *policy* selects the nodes to or from which jobs will be transferred. To our knowledge, existing load balancing algorithms for distributed systems define a negotiation policy that is independent of the computation model. The selection of a node is solely based upon the same load state measure, queue length [3, 4, 5, 6, 7].

In this paper, we introduce a load balancing algorithm for a distributed object-based runtime system whose negotiation policy is based upon both the system's node load and the objects' characteristics. The proposed algorithm results from a study undertaken in our group that was aimed at defining transparent policies for improving performance of distributed object-based applications [8]. The transfer policy is invoked on each method call. The proposed negotiation policy then implements an adaptive initial-placement strategy for method executions, that is to say, executions do not migrate. Finally, the information policy relies on controlled exchange of load information that minimizes the number of messages sent for exchanging load information. In order to evaluate the behavior of the adaptive (initial-)placement policy, it has been integrated within the customizable distributed object-based runtime system Isatis.

This paper details the adaptive placement policy and the controlled exchange algorithm, together with their integration within the Isatis system. Section 2 first defines the adaptive policy. Section 3 then describes its integration within Isatis; both the base architecture of Isatis and its enrichment for the integration of the placement policy are described. Section 4 studies the performance of the placement strategy on the Unix implementation of Isatis. Finally, we draw some conclusions including comparison with related work in section 5.

## 2 Adaptive Placement of Method Executions

Our approach for increasing the efficiency of distributed object-based programs relies on the definition of a policy for initial-placement of method executions on the system's nodes, assuming a network of homogeneous workstations. This section describes the proposed initial-placement policy, as well as a new strategy for the exchange of load information between nodes that limits the number of message exchanges.

### 2.1 The adaptive placement policy

The time taken by the execution of a method invocation is given by adding the three following times: $(i)$ the time taken for locating the storing node of the called object, $(ii)$ the time taken for transferring the request to the storing node, and $(iii)$ the time taken for executing the method, either on the object's storing node (*local* execution) or on another node (*remote* execution).

Our goal in the design of a new policy for initial-placement of method executions was to provide a transparent facility that minimizes $(iii)$. In a message-passing framework in which objects are not replicated[1] and communicate by synchronous method calls, the value of $(iii)$ is given by the addition of the four following components:

1. Time required for transferring the call request to the node selected for the method's execution. It includes the time taken for moving the called object to the selected node if remote;
2. Waiting time within the chosen processor's ready queue before the method is started;
3. Method execution time, which is further split in:
   - 3.1. Actual execution time (cpu);
   - 3.2. Waiting time within the processor's ready queue, which is attributable to processor sharing;
   - 3.3. Waiting time resulting from interaction with other objects, that is to say, (synchronous) method calls performed by the method;
4. Time taken for returning the result to the caller.

The processor to be used for executing a method should be chosen so as to minimize the time resulting from the combination of these four components. Considering subcomponents of 3, sub-component 3.1 does not vary since a network of homogeneous workstations is considered. Hence, sub-component 3.1 is discarded. The evaluation of sub-component 3.2 depends upon the method's actual execution time (sub-component 3.1), which is not known before the method's execution. The value of 3.1 could be approximated by recording the average execution times of methods within the objects' states. However, as we have not yet foreseen if the resulting overhead would be negligible, we chose in a first approach to ignore the value of sub-component 3.2. An extension of the placement strategy dealing with 3.2 is given in section 4.3. The evaluation of sub-component 3.3 requires to cope with object interactions. This sub-component is discarded in a first step; it is dealt with as an extension of the placement strategy in section 4.3. The remaining components lead to the definition of two sets of decision criteria:

- $T_{pro_i}$, which is the decision criterion related to the nodes' load (component 2). $T_{pro_i}$ is the waiting time within the processor queue of node $i$ before the method execution starts.
- $T_{trans_i}$ and $T_{res_i}$, which are the decision criteria related to the called method (components 1 and 4). $T_{trans_i}$ is the time taken for transferring the call request on node $i$ in a message, which is given by the input parameter size and the object's size –if the object needs to be moved to $i$. $T_{res_i}$ is the time taken for transferring the result message from $i$ to the caller's node.

where $i$ may be any of the system's nodes on which the considered application executes.

At a first glance, the node $i$ that must be chosen for executing a method is the one that minimizes the addition of $T_{pro_i}$, $T_{trans_i}$, and $T_{res_i}$. However, the soundness of an addition-based approach has to be demonstrated since we consider multiple, independent criteria that possibly conflict. For that purpose, we have used the multi-criteria evaluation theory (e.g., see [9]) that serves solving decision problems where multiple criteria, often conflicting, have to be taken into account. Using the multi-criteria theory, it has been shown in [8] that the decision placement for a method $m$ of an object $o$ could be taken according to the evaluation of the sum of $T_{pro_i}$, $T_{trans_i}$, and $T_{res_i}$, leading to the following definition:

$$Wait(o,\ m,\ i) = T_{pro_i} + T_{trans_i} + T_{res_i}.$$

Given the $Wait$ function, the adaptive placement algorithm is straightforward; it consists of the two following steps:

**Step 1:** Computation of the set *eligible* of eligible nodes for executing method $m$, that is nodes $\{i_1, ..., i_n\}$ such that $\forall j \in \{i_1, ..., i_n\}$:

$$Wait(o,\ m,\ j) \leq Min(Wait(o,\ m,\ k),\ k \in hosts) + threshold,$$

where *hosts* identifies the nodes hosting the considered application, and *threshold* is a system-defined parameter.

[1] Although object replication is supported by the Isatis object-based system within which the proposed placement policy was integrated, it is not considered in the remainder of the paper for the sake of simplicity.

**Step 2:** Selection of the node that will execute $m$ among components of *eligible*. This selection is computed randomly by default. However, it may be computed with respect to additional preference criteria specified by the programmer; for instance, this can be the preference for the node hosting the highest number of objects invoked by $m$.

### 2.2 Efficient computation of the system's global load

The quality of the adaptive placement algorithm depends upon the relevance of the global load information that is used. Best results are obtained when the information is close to the system's actual load and when the overhead of the algorithm handling the exchange of load messages is kept as small as possible. The algorithm that we use is called *controlled exchange* and is based on a *periodical* exchange of load information (e.g., see [10]). The periodical exchange approach is recognized as an adequate basis with respect to the global load's relevance. However, it is less satisfying from the perspective of the number of exchanged messages since it is based on message broadcasting.

Basically, the controlled exchange algorithm enhances the periodical approach as follows. At the end of every period, a node $i$ sends its new local load value $T_{pro_i}$ to a node $j$ only if this value makes $i$ *approximately eligible* for executing a method of an object located on $j$. In this definition, the eligibility notion refers to the one used by the adaptive placement algorithm introduced previously. Furthermore, eligibility is approximated with respect to the storing node $j$, that is to say, if $i$'s load is such that a local execution on $j$ will not be preferred.

**Remote detection of non-eligibility.** Given the adaptive placement algorithm, a node $i$ is not eligible for executing a method $m$ of an object $o$ located on node $j$ if:

$$Wait(o, m, i) > Wait(o, m, j) + threshold.$$

This leads to define the function $nonEligible(o, m, j)$ that evaluates the set of nodes that are not eligible for executing a method $m$ of an object $o$ located on node $j$. $nonEligible(o, m, j)$ is equal to:

$$\{\ k \mid k \in hosts \wedge Wait(o, m, k) > Wait(o, m, j) + threshold\ \}.$$

The above function is defined with respect to a particular method of a given object. It must be generalized in order to detect nodes that are not eligible for executing any method of any object located on $j$. The proposed generalization relies on the introduction of a reference method $m_r$ whose features are such that the difference between the evaluation of $Wait$ for $m_r$ on $j$ and the one of $Wait$ for $m_r$ on a remote node $i$ is minimal. The value of $Wait$ being given by the addition of $T_{pro_i}$, $T_{trans_i}$, and $T_{res_i}$, a minimal difference is obtained by setting:

$T_{trans_i}$ = the cost of sending the smallest invocation message between $i$ and $j = mc$

where, we assume here that the cost of sending a given message between any two pairs of nodes is always the same. This leads to identify node $i$ as being non-eligible for executing a method of an object located on $j$ if: $T_{pro_i} - T_{pro_j} + threshold < T_{trans_i}$, that is, if $T_{pro_i} - T_{pro_j} + threshold < mc$. This now allows us to define the controlled exchange algorithm.

**The controlled exchange algorithm.** The controlled exchange algorithm consists of maintaining the array $load_i$ on every system's node $i$ where $load_i(j)$, $j \neq i$, gives the last load value received from node $j$ by $i$, and $load_i(i)$ gives the local load value of $i$.

The update of $load_i$ relies upon the following execution on every node $i$. Periodically, node $i$ computes its local load $T_{pro_i}$, updates $load_i(i)$. It sends the value of $T_{pro_i}$ to node $j$, $j \neq i$, if $i$ is approximately eligible for executing a method on an object located on $j$, that is to say, if $eligible_i(j) = (load_i(j) - load_i(i) + threshold > mc)$ evaluates to true.

It should be noticed here that a mutual non-eligibility problem can occur if both $eligible_i(j)$ and $eligible_j(i)$ evaluate to false; load messages are then no longer exchanged between $i$ and $j$. In order to avoid this problem, any node $i$ that detects the transition from $eligible_i(j)$ to $\neg eligible_i(j)$ for any node $j$, notifies the transition occurrence to $j$. Then, if $i$ is notified of the dual transition from $j$, one of the two nodes is selected for sending its load information on a non-controlled, periodical basis.

## 3 Integrating Adaptive Placement within the Isatis System

In this section, we focus on the integration of the proposed adaptive placement strategy within the customizable distributed object-based runtime system Isatis, developed at IRISA within the Gothic INRIA/BULL project.

### 3.1 The Isatis system

The Isatis system is aimed at supporting the execution of various distributed object-based languages above different base operating systems. At that time, our prototype supports the execution of the Arche concurrent object-oriented language [11] and of the C++ language extended with built-in classes for distribution management. Furthermore, the prototype supports interoperability between nodes executing either Unix or Mach.

#### 3.1.1 The Isatis model

The Isatis model is defined by its object and execution models.

**The object model.** An Isatis object follows the usual definition: an object is made of a state and a set of methods that provide the only means to manipulate the object's state. Fine grain (less than one hundred bytes) objects to coarse grain (more than one thousand bytes) objects are equally supported by the system. Objects can communicate through either synchronous or asynchronous method calls. Objects are passive by default but active objects can be easily implemented by using asynchronous method calls. Creation as well as destruction of objects is explicit by default, but the Isatis system can be customized so as to implement the garbage collection technique suited to a given programming language. In addition, although not addressed in the remainder of this paper, an object can be explicitly declared as being persistent, in which case its lifetime is independent of the one of its creator.

**The execution model.** The base abstraction of the Isatis execution model is the notion of protection delegate [12]. A protection delegate is a distributed set of multi-threaded tasks. The number of tasks per protection delegate is fixed at creation-time. A protection delegate corresponds roughly to an application.

Communication between objects belonging to the same protection delegate is achieved in two different ways depending on whether they belong to the same task or not. In the former case, communication is achieved *via* a simple procedure call if the method call is synchronous while it leads to the creation of a thread handling the call if the method call is asynchronous. On the other hand, communication between objects belonging to distinct tasks is carried out as for method calls involving objects belonging to different protection delegates, by means of a Remote Procedure Call (RPC).

#### 3.1.2 Isatis objects

Isatis objects are implemented in the C++ language. Customization of the objects' behavior according to the needs of either a specific language or a specific base operating system then relies upon specialization of Isatis classes. We identify two kinds of Isatis objects, those implementing system-controlled functionalities and those implementing application objects. There is at most one instance of each system-controlled object per task of each protection delegate. On the other hand, the number of application objects within protection delegates is application-specific.

**System-controlled objects.** System-controlled objects fall into three main classes that respectively correspond to the management of tasks, multi-threading, and interprocess communication. An object managing a task carries the information needed to implement identification of: (i) tasks (either belonging to the same protection delegate or not), and (ii) objects of the embedding protection delegate (both system-controlled and application objects).

**Application objects.** An application object translates into three kinds of Isatis objects: one corresponds to the object's naming, and the two others to the object's representation which may be either remote or local.

Naming objects implement references to application objects. They are introduced in order to support mobility of actual objects.

A local representation object corresponds to the actual object. Remote representation objects are in charge of implementing remote procedure calls to the corresponding actual object, by calling inter-process communication system-controlled objects.

**Implementing application objects.** Application objects of Isatis introduced above rely on the definition of the four following C++ classes, called Isatis application classes in the remainder of this paper:

- *t_naming* that serves as a basis for implementing naming objects,
- *t_generic_object* that is inherited from *t_remote*, used for implementing remote representation objects, and *t_actual*, used for implementing actual objects.

The three kinds of Isatis objects implementing a given application object are obtained by defining classes inheriting respectively from classes *t_naming*, *t_remote*, and *t_actual*. Subclasses of *t_naming* and *t_remote* add method implementations for each method declared by the interface of the target application object. Methods declared within a subclass of *t_naming* are simple; they consist of calling the corresponding method on the referenced instance of the subclass of *t_generic_object*. Methods declared within a subclass of *t_remote* establish a connection with the task hosting the actual object, carry out an RPC for the method's execution, and release the connection. Finally, subclasses of *t_actual* provide the actual implementation of application objects.

### 3.2 Implementing adaptive placement

Integrating adaptive placement of method executions within Isatis leads to two kinds of extensions: system objects carrying load information must be defined, and application classes must be refined so as to implement placement decision.

**Implementing controlled exchange.** Two classes implementing system-controlled objects used for adaptive placement are defined:

- *load*, which is the representation of the array $load_i$ of section 2.2, defines data-objects carrying global load information; and
- *loadExchange* that defines service-objects sending load information to instances of *load* located within peer tasks. It implements the controlled exchange algorithm introduced in section 2.2.

As for other system-controlled objects, there is a single instance of both *load* and *loadExchange* per task of each protection delegate. These objects are further made known to application objects through a specialization of system-controlled objects managing tasks (see § 3.1.2); these objects now contain a reference to the local instance of *loadExchange*, and references to all the instances of *load* created within peer tasks.

In a Unix framework, which is the platform used for performance measures given in section 4, the local load related to a task is obtained from the waiting time in the ready queue of the multi-threading library plus the waiting time in the Unix ready queue.

**Implementing adaptive placement decision.** A given application object being actually represented within a single task at a time, the integration of the adaptive placement facility requires only to refine the behavior of objects that are an instance of a subclass of *t_actual*. The extension consists of enriching the code of each method with a call to a function evaluating the placement decision and with the corresponding placement of the method's execution. The placement decision function is implemented by a user-defined class, called *placement*, computing the adaptive placement decision presented in section 2.1. This class declares a reference to the system-controlled object managing the embedding task, that interfaces with the system and hence allows to access system-controlled information including load information.

The implementation of a method $m$ of an application object $a$ for which adaptive placement of method executions takes place, has the following structure:

```
Selection of the node N for the execution of m
if N is remote then
    Move a within the task located on N
    Transfer of the call request to N
    Gets the modified copy back
else local execution of m
```

The implementation of the *placement* class subsumes in particular computation of criteria $T_{pro_i}$, $T_{trans_i}$, and $T_{res_i}$. The computation of $T_{pro_i}$ is dealt with through instances of *load* and *loadExchange* introduced in the previous paragraph. Assuming the invocation of a method $m$ of $a$, criterion $T_{trans_i}$ is evaluated from the size of $a$ and $m$'s input parameter, while the evaluation of $T_{res_i}$ depends upon the size of $m$'s result. The sizes of $m$'s input and output parameters are given by the signature of $m$. The duration of a message exchange for a given message size is obtained from the system-controlled object managing the embedding task, that stores statistics about transmission delays obtained by experimentation.

# 4 Analysis of the Behavior of the Initial-Placement Algorithm

This section studies the behavior of the adaptive placement strategy for the Unix implementation of Isatis. Further evaluation of the algorithms through simulation can be found in [8].

## 4.1 Experimental setup

We ran our experiments on top of a 10Mb Ethernet network of SUN SPARCstations 2 running SunOS 4.1. In this configuration, it takes 2197 $\mu s$ for a C program to wait for an echo of a TCP packet from a remote machine. The code for placement of method executions was entirely written in the Arche language. Machines share a common file system through NFS [13]. Transport objects are implemented using Unix sockets (stream mode, TCP transport protocol).

All the performance measurements were obtained by running the applications in multi-user mode. During the experiments, the network was shared with non-Isatis applications, and was not heavily loaded. The machines used for running the applications were equally loaded by a small number of Unix processes.

## 4.2 Behavior of the algorithm for independent processes

We concentrate here on the behavior of the placement algorithm in the situation where the adaptive strategy can be the best exploited, that is to say, where the application is composed of independent processes equally distributed on the system's nodes.

### 4.2.1 The application

The application is simple and made up of *NumWorkers worker* objects per node. Each worker object issues *NumComp* method calls on *computation* objects of ten bytes each. A method call on a computation object lasts *MethDuration*. Method executions on worker and computation objects are placed using the adaptive initial-placement strategy.

The nodes' loads are made vary by launching on each node a variable number of *loader* objects whose methods always execute locally (they do not inherit from *placement*). A load difference between nodes is enforced through the creation of *NumLoaders* objects on *NumLightly* lightly loaded nodes, and *LoadBalance**NumLoaders* on the remaining nodes (heavily loaded nodes). The system's parameters used for the experiments are given in table 1.

### 4.2.2 Results

Performance of the adaptive placement algorithm, labeled *Adaptive* within figures, is examined through a comparison with the simple default initial-placement algorithm named *Storage*, that always executes a method of an object $o$ on $o$'s storing node. The behavior of the adaptive placement algorithm is shown in two system configurations setting respectively parameter *NumLightly* to 3 and 1. The first configuration consists of a single highly loaded node, the remaining nodes being equally weakly loaded, while the

| Name | Description | Value |
|---|---|---|
| NumNodes | Number of nodes in the system | 4 |
| LoadPeriod | Period for the controlled exchanged algorithm | 500 ms |
| Threshold | Threshold used by the proposed initial-placement algorithm | 20 ms |
| NumWorkers | Number of worker objects per node | 5 |
| NumComp | Number of computations issued by worker objects | 10 |
| MethDuration | Duration of the execution of a method on a computation object | 800 ms |
| NumLoaders | Number of loader objects on each least loaded node | [1..20] |
| LoadBalance | Balance of the nodes' loads (ratio between the number of loaders on a highly loaded node and the one of a lightly loaded node) | 5 |
| NumLightly | Number of lightly loaded nodes | 3 or 1 |

Table 1: Parameters used in the experiment

second configuration consists of the symmetrical configuration with three highly loaded nodes and a single lightly loaded node.

Figure 1 depicts the response time of the application (the time taken for executing worker objects) in these two system configurations according to the nodes' loads (parameter *NumLoaders*).

A similar behavior is observed for the two system configurations, the response time of the application in the second configuration being always higher than in the first one due to the presence of more loader objects. At a first glance, compared with the default placement policy, the adaptive placement algorithm lowers the response time of the selected application, and the performance gain increases with the nodes' loads. The gain is nul when the nodes' loads are low ($NumLoaders < 4$), and divides the application's response time by a factor ranging from 3 to 10 when $NumLoaders \geq 5$. Let us compare with more details the storage and adaptive placement strategies for the different areas of the curves.

For $NumLoaders < 4$, the response time of the application is almost the same for adaptive and storage placement strategies. This is explained by the fact that the load difference between any two pairs of nodes is less than the overhead of remote execution due to the communication cost, and hence both the adaptive and default placement strategies select local executions. In fact, the application's response time when using the adaptive placement strategy is higher than when using the default placement policy since, in the latter situation, no placement method is executed for choosing the best node for methods' executions. However, the overhead due to the execution of the placement algorithm is too small to be noticed here;

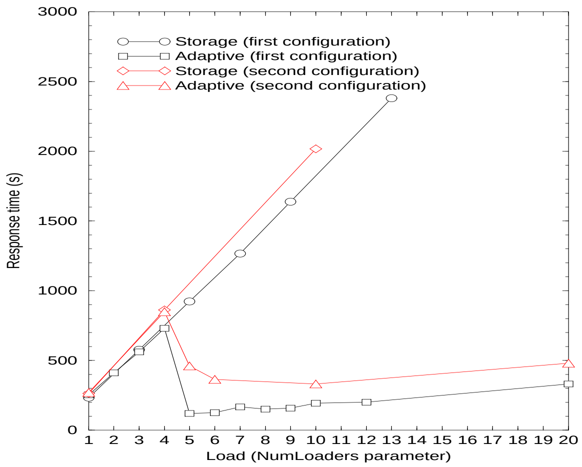


Figure 1: Influence of the nodes' loads

this is due to the duration of method executions chosen (800 ms), which is (hopefully) much more higher than the duration of the placement method.

For $4 \leq NumLoaders \leq 6$, the application's response time falls abruptly. The load difference between the most and least loaded nodes is then high enough to compensate the communication cost of a remote execution, and thus leads to select remote nodes for method executions. More precisely, for both system configurations, the adaptive algorithm running on the most loaded node(s) begins to select remote nodes for the execution of methods when $NumLoaders = 4$, the percentage of remote executions being equal to 100% when $NumLoaders > 6$. It should be noticed that the least loaded node did not become overloaded by the received computations because the difference between the loads of the most and least loaded nodes was high enough. Otherwise, the highly loaded node would have stopped sending computations to the other nodes.

Finally, for $NumLoaders > 6$, the application's response time increases linearly with the nodes' loads, due to processor sharing with an increasing number of loader objects.

### 4.3 Extensions

The following paragraphs briefly discuss two extensions of the proposed initial-placement algorithm consisting respectively of approximating more finely the cost of remote executions by taking into account the duration of method executions, and considering interactions between objects.

**Considering method duration in the placement decision.** As stated in section 2.1, the proposed algorithm for initial-placement of method executions makes the placement decision independently of the method durations (sub-component 3.1 of methods response time is discarded). This leads to underestimate the benefit of remote executions. In this paragraph, we examine the influence of not considering this component on the placement decision. For that purpose, we extend the proposed initial-placement strategy so as to consider the execution time of each method in the evaluation of *Wait*, the execution time being fixed to *MethDuration*. Figure 2 depicts the application's response time for both the base and modified initial-placement strategies.

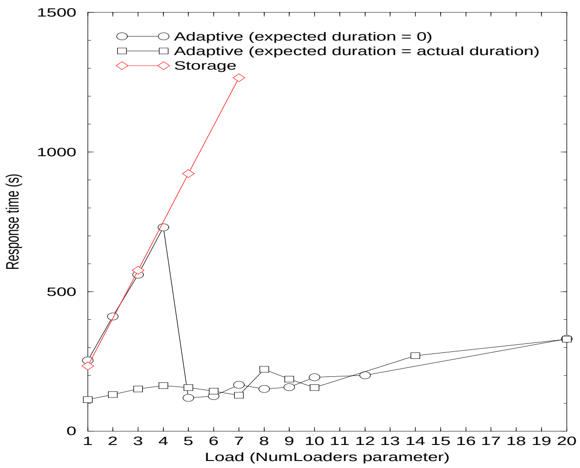


Figure 2: Influence of the knowledge of method duration on the application's response time

The figure shows that compared with the base initial-placement algorithm, the modified adaptive algorithm, having the knowledge of the exact duration of methods before their execution, lowers the application's response time for small load values (*NumLoaders* < *4*). Since the modified placement policy knows in advance the methods' durations, it approximates the actual response time of a method execution with more details than the base placement algorithm, and hence selects earlier remote executions.

Taking into account the methods' durations in the placement algorithm requires having either a system-wide or per-object estimation of the methods expected duration. We are currently trying to find the best trade-off between the soundness of the stored information and the runtime overhead for evaluating it.

**Dealing with object interactions.** Up to this point, performance measures have been given for an application with a low degree of object interactions. In the presence of a high number of inter-object communications, a mechanism implementing an adequate placement of objects in addition to placement of method executions is required. Ideally, the strategy for object placement must be designed so as to store communicating objects on the same node. The design of such a strategy has been undertaken in our group in the framework of the study aimed at defining mechanisms increasing performance of distributed object-based applications [8]. The proposed strategy, that is complementary to the adaptive placement policy, consists of initially placing any object on the node of its creator and then to migrate the object according to its interactions with other objects. We are currently working on the integration of the strategy within Isatis.

## 5 Conclusion

This paper has presented the design and implementation of a mechanism aimed at enhancing the performance of distributed object-based applications. This goal is achieved by means of a novel algorithm for transparent, initial-placement of method executions. The resulting placement policy is qualified as adaptive since it takes into account the processors' load as well as the objects' features (object size and parameter size), for computing the best placement of method executions. Compared to existing load balancing strategies for distributed system, our approach differs in that we take into account the objects' characteristics in addition to the load of the system's nodes, in the placement decision. Considering existing distributed object-based runtime systems, placement of method executions relies on a simple strategy that is one of the following [14, 15, 16]: execution on the storing node of the called object, execution on the calling node, or execution on a node explicitly mentioned by the programmer. Obviously, the two first (transparent) policies are not satisfying since they do not exploit the processors' load. On the other hand, the approach that consists in leaving the programmer specifying the node on which a method should be executed allows him/her to choose the most appropriate node with respect to the nodes' load and the resulting communication costs. However, this leads to add to the source of complexity of distributed programming.

The load information used by the policy being crucial for the relevance of the placement decision, we have designed an algorithm computing the system's global load that is based on the periodical exchange approach used for load balancing within distributed systems (e.g., see [10]). The proposed algorithm enhances the base periodical exchange algorithm by reducing the number of generated message exchanges. Related proposals [17, 18, 19] usually introduce a load threshold up to which a node stops being eligible for handling remote executions, and hence stops sending load messages until its load decreases under the fixed threshold. The novelty of the controlled exchange algorithm lies on the definition of a sending condition per node pairs, this condition being further based upon the system's dynamic characteristics.

The implementation of the proposed adaptive policy has been detailed in the framework of the Isatis customizable distributed object-based runtime system. Performance measures of Isatis integrating the adaptive placement policy have been given and compared with the one resulting from the integration of the simple alternative initial-placement policy consisting of executing any method on the node storing the called object. Results show that the response times of applications using this simple initial-placement can be divided by up to a factor of ten when using the proposed placement strategy. This figure is obtained when there is a large difference between the loads of the most and least loaded nodes. A small overhead (negligible if the method duration is important) is paid when the nodes are too lightly loaded and/or equally loaded. Comparison of the proposed initial-placement strategy with a policy taking into account the methods expected duration showed that this knowledge can further lower applications' response times if the methods' expected duration is close to their actual duration. We are currently investigating the integration of such a knowledge in the proposed placement strategy. A study on the addition of a migration policy for dealing with a higher degree of interactions between objects in also under way.

**Acknowledgments.** We wish to thank Manuel Chevalier for implementing and measuring the performance of the algorithms for adaptive placement and controlled exchange. We also would like to acknowledge S. Weather and the anonymous referees for their valuable comments on an earlier version of this paper. Finally, we are grateful to C. Moullec for his helpful suggestions regarding evaluation of the placement strategy.

## References

[1] R. S. Chin and S. S. Chanson, "Distributed object-based programming systems," *ACM Computing Surveys*, vol. 23, no. 1, pp. 91–124, 1991.

[2] K. Benmohammed-Mahieddine, *An Evaluation of Load Balancing Algorithms for Distributed Systems.* PhD thesis, Leeds University, UK, 1992.

[3] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Evaluation*, vol. 6, no. 1, pp. 53–68, 1986.

[4] O. Kremien and J. Kramer, "Methodical analysis of adaptive load sharing algorithm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 6, pp. 747–760, 1992.

[5] K. Benmohammed-Mahieddine, P. M. Dew, and M. Kara, "A periodically symmetrically-initiated load balancing algorithm for distributed systems," in *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pp. 616–623, 1994.

[6] C. J. Hou, K. G. Shin, and T. K. Tsukada, "Transparent load sharing in distributed systems: Decentralized design alternatives based on the condor package," in *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, pp. 202–211, 1994.

[7] P. Krueger and G. Shivaratri, "Adaptive loation policies for global scheduling," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 432–444, 1994.

[8] Y. Belhamissi, *Placement des calculs et des données dans un système distribué à objets.* Thèse de doctorat, Université de Rennes I, Rennes, France, 1994.

[9] P. C. Fishburn, "A survey of multi-attributes/multi-criteria evaluation theories," in *LNCS 155*, pp. 181–224, Springer-Verlag, 1978.

[10] A. Barak and A. Shiloh, "A distributed load balancing facility for a multicomputer," *Software Practice and Experience*, vol. 15, no. 9, pp. 901–913, 1985.

[11] M. Benveniste and V. Issarny, "Concurrent Programming Notations in the Object-Oriented Language Arche," Research Report 1822, INRIA, Rennes, France, 1992.

[12] M. Banâtre, Y. Belhamissi, V. Issarny, I. Puaut, and J. P. Routeau, "Arche: A framework for parallel object-oriented programming above a distributed architecture," in *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pp. 510–517, 1994.

[13] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network file system," in *Proc. of Usenix 1985 Summer Conference*, (Portland), pp. 119–130, June 1985.

[14] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanrayanan, R. J. LeBlanc, M. Ahamad, U. Ramachandra, W. F. Appelbe, J. M. Bernabéu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenoh, "The design and implementation of the Clouds distributed operating system," *Computing Systems*, vol. 4, no. 3, pp. 243–275, 1990.

[15] Odjik, "The DOOM system and its applications: A survey of Esprit 415 subproject A, Philips research laboratory," in *Proceedings of PARLE*, pp. 461–479, 1987.

[16] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul, "Emerald: A general purpose programming language," *Software Practice and Experience*, vol. 21, no. 1, pp. 91–118, 1991.

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

[18] S. Zhou, "A measurement study of load balancing performance," in *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pp. 490–497, 1987.

[19] R. Alonso and L. L. Cova, "Sharing jobs among independently owned processors," in *Proceedings of the Eigth International Conference on Distributed Computing Systems*, pp. 282–287, 1988.