Isatis: A Customizable Distributed Object-Based Runtime System

Michel Banâtre, Yasmina Belhamissi, Valérie Issarny, Isabelle Puaut, Jean-Paul Routeau

IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE

French-Japanese Workshop on Object-Based Parallel and Distributed Computation
Lecture Notes in Computer Science.
Tokyo, Japan, June 1995.

Abstract. This paper discusses the design and implementation of a customizable distributed object-based runtime system. Our main goal in the system's design was to provide a distributed object-based system supporting execution of various (concurrent) object-oriented languages, and that can be easily enriched with mechanisms for object management so as to fit the applications' needs. Enrichment of the runtime system with a particular mechanism is illustrated through the integration of a facility that is aimed at enhancing the applications' performance. This goal is achieved by means of a load balancing strategy that implements initial-placement of method executions and migration of data objects according to both the processors' load and the objects' features.

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 [19], that is to say a concurrent object-based language together with its dedicated distributed object-based runtime system (e.g., Argus [34], Emerald [39], POOL/DOOM [2, 36]). However, this approach has not had the expected success. Although it has demonstrated that object-object oriented languages provide the adequate basis for the development of distributed applications, the efficiency of the approach has been too poor to be unanimously accepted by the distributed systems community. This inefficiency mainly came from the mismatch between the abstractions provided by operating systems and the ones offered by programming languages.

In order to increase efficiency of distributed object-based programming languages, a new approach has later been undertaken. It consists of defining generic distributed object-based runtime systems that support the execution of various programming languages. The runtime system may then be customized so as to execute efficiently applications written in a given language by implementing the best mapping between the language's abstractions and the ones of the operating system. From that perspective, micro-kernel architectures such as Amoeba [41], Chorus [40] and Mach [1] offer an adequate basis; they provide a basic set of abstractions that can be used easily for building more elaborated abstractions. Using micro-kernel architectures for building generic distributed object-based runtime systems has been undertaken by at least two projects: COOL-v2 [32] built on top of Chorus and GUIDE-v2 [18] built on top of Mach. Although the resulting systems have shown promising results from the standpoint of performance, the use of a specific underlying operating system is restrictive from the standpoint of portability. Furthermore these systems lack of flexibility in that the system's customization is addressed only at the language-level.

Due to the evolution in the fields of networking and processor technology, distributed architectures now support the execution of different kinds of applications (e.g., scientific, multimedia) having different requirements. This raises the need for application-specific operating systems (e.g., see [4]), that is to say, systems that can be customized at the application-level. The construction of application-specific operating systems brings up several issues including the ease of customization specification by the application programmers, and the provision of adequate base operating systems so as to easily support safe and efficient customization. Much work has been undertaken both in the operating system and the programming language communities in order to address these needs. This has led to complementary solutions that can be summarized as follows:

- Specialization of the system's code through partial evaluation by exploiting the applications' invariants (e.g., see [37]), which eases the process of customization specification by providing an automatic solution.
- Definition of new operating system kernels so as to support efficient and safe customization (e.g., see [13, 22]).
- System structuring in terms of high-level abstractions so as to ease the customization process both from the perspective of its specification and of its integration. Most of the existing solutions in that framework exploit the object paradigm. An example of such is the Choice operating system that is structured in terms of objects, the inheritance mechanism being used for customization [17]. Another example is the Apertos operating system that defines a reflective architecture by means of the meta-object notion [42]. This system introduces a meta-level that defines control and policy, which can be customized by modifying the meta-level components [33].

In the framework of the INRIA Solidor research group, we are investigating solutions for efficient execution of concurrent object-oriented applications through customization. Our approach follows the Choice approach. We have designed and implemented a minimal distributed object-based runtime system, called Isatis. Isatis is defined in terms of C++ objects and can be easily customized through the inheritance mechanism. We are now concentrating on the provision of solutions that allow *safe* customization of Isatis for *efficient* execution of applications that can be written in different object-based languages and that can execute on different operating systems. Our approach for safe customization relies on the exploitation of type conformance [15]. We further address enhancement of the applications' performance by designing new policies for distributed object management. This paper focuses on this last aspect by providing an introduction to the Isatis system and discussing its customization with the integration of a load balancing strategy.

This paper is structured as follows. Section 2 first introduces the Isatis architecture, discussing its object and execution models. Section 3 then illustrates the integration of an elaborated load balancing strategy within Isatis; the proposed strategy implements novel policies for transparent, initial-placement of method executions and migration according to both processors' loads and objects' features. Finally, we draw some conclusions in section 4.

2 The Isatis System

In this section, we provide an overview of the Isatis distributed object-based runtime system. We first give the background to the Isatis design. We then define the Isatis model, and introduce objects managed by the Isatis system. Finally, we examine the usage of the Isatis system from the perspective of both different object-oriented languages and different underlying operating systems.

2.1 Background

The design of the Isatis system was primarily motivated by our previous experience with the design and implementation of two distributed object-based programming systems: Gothic [6] and Arche [8]. Although this work led to promising results from the standpoint of specific object management mechanisms (e.g., fault tolerance mechanism for Gothic [7], and garbage collection for Arche [38]), the proposed distributed object-based systems were not entirely satisfactory for several reasons. Among them, let us mention the systems' poor performance and lack of portability. We thus decided to adopt a generic approach in order to solve these shortcomings but also to be able to implement more easily our research results in the field of distributed object-oriented programming, at the level of both programming languages and object management mechanisms.

Due to the wide range of research results in the field of distributed object-based systems, the approach undertaken for the design and implementation of Isatis' base features first consisted of exploiting existing proposals. The resulting process of selection led us in particular to examine work on the treatment of inter-process communications and multi-threading. From the standpoint of inter-process communication management, our goal was to support various communication methods as for instance achieved in the Network objects [14] and Concert/C [5] proposals. Due to the object-oriented framework, we based our solution on the one of Network objects that consists of defining transport objects managing connections between address spaces. Our implementation of multi-threading is based on the the Rex LWP library developed at Imperial College (London, UK) [20]. The main reason that led to this choice is that Rex LWP was the studied package embedding the least amount of hardware-dependent code.

2.2 The Isatis Model

The Isatis model is defined by its object model –i.e., the characterization of the objects it manages– and its execution model –i.e., the characterization of its execution units.

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 a page) objects to coarse grain (segment) 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 the asynchronous method call mechanism.

Creation as well as destruction of objects is explicit by default. Although implicit destruction of objects is desirable from the perspective of reliability, its implementation requires the integration of an additional mechanism within Isatis. This obviously follows from the fact that Isatis supports programming models offering passive objects as well as models defining active objects. Garbage collection techniques being different for these two kinds of models (e.g., see [30]), their implementation depends upon the programming language to execute and hence requires system customization.

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. Object mobility over the network is supported through a migration mechanism. However, the base Isatis system does not implement any migration policy.

The execution model. The base abstraction of the Isatis execution model is the notion of protection delegate [8]. A protection delegate is a distributed set of multi-threaded protection domains (e.g., multi-threaded processes using the Unix terminology, tasks using the Mach terminology), the number of protection domains being fixed at creation-time. A protection delegate corresponds roughly to an application and may be related to the notion of job offered in the GUIDE-v2 model [18].

Isatis objects belonging to the same application are mapped within the same protection delegate. Communication between objects belonging to the same protection delegate is achieved in two different ways depending on whether they belong to the same protection domain 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 protection domains is carried out as for method calls involving objects belonging to different protection delegates, by means of a Remote Procedure Call (RPC).

2.3 Isatis Objects

Isatis objects are implemented in the C++ language. Customization of the objects' behavior according to the needs of either a particular application, language, or operating system then relies upon specialization of Isatis classes by system programmers by means of the C++ inheritance mechanism. 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 protection domain 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 categories that respectively correspond to the management of protection domains, multi-threading, and inter-process communication. An object managing a protection domain carries the information needed to implement system mechanisms, this includes the identification of peer protection domains and the description of the embedded application-objects.

From the standpoint of multi-threading, we have extended the Rex LWP library developed at Imperial College (London, UK) [20]. Resulting objects are those implementing threads and synchronization.

Finally, objects implementing inter-process communications follow the Network objects proposal [14] that deals with method invocations over a network. The base principle of the undertaken solution consists of implementing transport objects that generate and manage connections between protection domains (where involved domains may either belong to the same protection delegate or not). Various specializations of transport objects may coexist within the system so as to be able to deal with different communication methods. In our current prototype, we have implemented two kinds of transport objects: one that directly uses TCP and one that is implemented on top of the PVM system [24]. Connection objects are further introduced and correspond to connections that are established between domains.

Application objects. An application object translates into a set of Isatis objects: one corresponds to the object's naming, and the other to the object's representation.

Naming objects are the Isatis objects implementing references to representation objects. They are introduced in order to support mobility of actual objects. A naming object embeds the last known information about the actual object's location, that is, the identifier of the protection delegate p embedding the actual object, the identifier of the hosting protection domain d of p, and the object's address within d. A naming object further embeds a reference to either the corresponding actual object if locally present or to the local proxy otherwise. As for the location information, the reference is the last known one and hence can be erroneous in the presence of object migration. Since the base Isatis system does not implement any migration policy, information carried out by naming objects is necessarily accurate.

A given application object translates into a local and a distributed representations. The local representation corresponds to the object's actual implementation while the distributed representation enriches this base representation with distribution management. An object's local representation is implemented by means of a single Isatis object. On the other hand, the distributed representation of an application object $\mathcal{O}$ is implemented by the following Isatis objects:

- $\mathcal{O}$'s distribution manager that implements in particular policies for initial-placement of method executions and for object migration, its base behavior is described hereafter;
- $\mathcal{O}$'s proxies that manage remote references to $\mathcal{O}$, they basically implement an RPC to $\mathcal{O}$; and
- $\mathcal{O}$'s server stubs defined for both $\mathcal{O}$'s local representation and $\mathcal{O}$'s distribution manager.

Isatis objects introduced above rely on the definition of the four following C++ classes; these classes together with their descendants are called Isatis application classes in the remainder of this paper:

- `t_identification` that serves as a basis for implementing naming objects,
- `t_generic_object` that is inherited from:
  - `t_remote`, used for implementing remote referencing, that is to say, proxies and stubs
  - `t_actual`, used for implementing representation objects, that is to say, local representations and distribution managers.

The Isatis objects implementing a given application object are obtained by defining classes inheriting from `t_identification`, `t_remote`, and `t_actual`. Subclasses of `t_identification` and `t_remote` add method implementations for each method declared by the interface of the target application object while subclasses of `t_actual` provide the object's actual implementation. By default, the subclass of `t_actual` implementing the distribution manager of an object $\mathcal{O}$ is defined by implementing each method `m` of $\mathcal{O}$ according to the algorithm given hereafter.

(1) `placement`: Selection of the node N for `m` execution
(2) if N is remote then
(3)   `migration`:
(4)   Transfer of $\mathcal{O}$ within the protection domain located on N
(5)   Transfer of the call request to N
(6) else local execution of `m`

Customization of Isatis from the standpoint of distribution management is in particular achieved by implementing the functions `placement` and `migration` (lines 1,3) as virtual methods. By default, the `placement` method selects the local node, and the `migration` method transfers the object's local representation for the duration of the method execution. The algorithm's structure allows a wide range of customization of object distribution management. This includes the integration of strategies for load balancing and object replication. Section 3 discusses customization of this base algorithm so as to implement a load balancing strategy according to the nodes' load and objects' features.

2.4 Using the Isatis System

We have stated that one of our main design goals for the Isatis system was to support the execution of different object-oriented languages above different base operating systems. In the following, we discuss how this goal is achieved.

Execution above different operating systems. Executing Isatis above different operating systems follows from Isatis' structure. The structure allows (system-)programmers to refine the system's classes according to the target operating system. This applies in particular to classes implementing thread management and inter-process communication. At that time, the Isatis prototype executes above nodes executing either Unix or Mach.

System classes may also be specialized so as to implement stronger semantics. For instance, the semantics of the base RPC mechanism of Isatis is the weakest one, that is, synchronous or asynchronous, maybe, single destination semantics. Depending on application requirements, more elaborated RPC semantics may have to be offered (e.g., see [26, 28]). In the framework of Isatis, integrating method calls based on RPCs with stronger semantics amounts to implement subclasses of the base RPC classes of Isatis.

Executing different object-oriented languages. Executing a given object-oriented language above Isatis requires the translation of each user-defined class in terms of the corresponding Isatis application classes. It follows that a modification of the compiler is required as in GUIDE-v2 [18]. However, given the implementation of Isatis objects described in the previous subsection, this step is quite straightforward.

At that time, our prototype supports the execution of applications written in an extended version of the C++ language as well as applications written in the Arche concurrent object-oriented language [12]. The first compiler of the Arche language generated C code to be executed on a dedicated distributed object-based runtime system [8]. Due to the features of the Isatis system, we have modified the compiler so as to generate C++ code. To give an idea of the manpower needed to execute a given language above Isatis, modification of the Arche compiler, within Isatis took us approximately 0.5 man-month.

3 Enhancing Isatis Performance through Load Balancing

Efficient execution of distributed object-based applications relies in particular on an adequate placement of objects on the system's nodes. Ideally, such a placement must be made so as to optimize the two conflicting criteria that are the number of message exchanges and the processors' loads. In general, processor allocation policies offered by existing distributed object-based systems such as Argus [34], Clouds [21], Doom [36], Emerald [39] and Guide [31], are simple. The node to be used for executing a method is selected among one of the following: object's storing node, calling node, and node explicitly mentioned by the programmer. A study has been undertaken in our group in order to integrate a transparent load balancing facility within the Arche distributed object-based system [8], dedicated to the execution of Arche applications. Results of this research are reported in [11]. Briefly stated, they consist of a migration policy that computes the objects' best storing nodes with respect to both communication costs and memory usage, and a policy, qualified as adaptive, for best initial-placements of method executions with respect to both processors' loads and objects' features. Compared to the Isatis system, the Arche system differs in several points. In particular, Arche protection domains belonging to the same protection delegate communicate through distributed shared memory, and Arche objects are (implicitly) persistent.

In the framework of the Isatis system, we have undertaken the design of a load balancing strategy based on the proposal made for the Arche system. The next subsection provides an overview of the resulting strategy. It is followed by a discussion on its integration within Isatis in subsection 3.2. Subsection 3.3 then addresses the gain due to the integration of the load balancing strategy within Isatis.

3.1 Overview of the Load Balancing Strategy

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 load balancing strategy was to provide a transparent facility that minimizes these three components, assuming a network of homogeneous workstations. This led us to define :

- An initial-placement policy for minimizing (iii), and
- A migration policy so as to co-locate communicating objects on the same node, and hence minimizing (i–ii).

In order to minimize the costs due the implementation of the load balancing strategy, the initial-placement and migration policies are coupled. However, for the sake of clarity, they are introduced separately in the following.

Adaptive initial-placement of method executions. In a message-passing framework in which objects are not replicated and communicate by synchronous method calls, the time taken by a method execution once the invocation request is received on the storing node of the called object, 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. Minimizing the value of sub-component 3.3 amounts to have communicating objects located on the same node, this is addressed through the migration policy, complementary to the initial-placement policy. Let i be any of the system's nodes on which the considered application executes, the remaining components (1,2,4) lead to the definition of two sets of decision criteria:

· T_{pro_i}, which is the decision criterion related to the nodes' loads (component 2).
· 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. T_{res_i} is the time taken for transferring the result message from i to the caller's node.

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 [23]) 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 [11] that the decision placement for a method $\mathtt{m}$ of an object $\mathcal{O}$ could be taken according to the evaluation of the function $Wait$ defined as follows.

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

Given the function $Wait$, the algorithm computing adaptive initial-placement for the execution of a method $\mathtt{m}$ of the object $\mathcal{O}$ is straightforward; it consists of the two following steps:

- **Step 1.** Computation of the set *eligible* of eligible nodes for executing $\mathtt{m}$, that is to say, nodes $\{i_1, ..., i_n\}$ such that $\forall j \in \{i_1, ..., i_n\}$:
  - $j \in hosts(\mathcal{O})$ where $hosts(\mathcal{O})$ identifies nodes on which execute protection domains of the protection delegate embedding $\mathcal{O}$, and
  - $Wait(\mathcal{O}, \mathtt{m}, j) \leq Min(Wait(\mathcal{O}, \mathtt{m}, k), k \in hosts(\mathcal{O})) + threshold$, where *threshold* is a system-defined parameter that sets the range of selection.
- **Step 2.** Selection of the node that will execute $\mathtt{m}$ 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 actual objects invoked by $\mathtt{m}$.

Efficient computation of the system's global load. The quality of the adaptive initial-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 exchange of load messages is kept as minimal as possible. The algorithm that we use is called *controlled exchange* [9] 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. Periodically, a node i sends its new local load value 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 initial-placement algorithm. Furthermore, eligibility is approximated with respect to node j, that is, if i's load is such that a local execution on j will not be preferred.

More formally, given the periodical reception of a load message from node k, node i sends a load message to node k if the following function:

$$eligible_i(k) = load_i(k) - load_i(i) + threshold > mc$$

evaluates to true, where *load* is the global load vector maintained on each node and *mc* is the cost of sending the smallest invocation message between any two pairs of nodes.

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

Object migration. In the presence of a high number of inter-object communications, a mechanism implementing an adequate placement of objects in addition to the adaptive initial-placement of method executions is required. Ideally, an object placement policy must be designed so as to reduce the applications' communication costs, which is mainly achieved by storing communicating objects on the same node. The design of such a policy has been undertaken in our group for the Arche distributed object-based system [11], and is aimed at the migration of persistent object. The proposed migration policy, that is complementary to the initial-placement policy, executes in parallel with the application. It consists of initially placing any object on the node of its creator and then migrating the object according to its interactions with other objects.

In the framework of Isatis, we have devised an alternative migration policy aimed at parallel object-based applications that is suitable for volatile objects. In order to minimize the execution overhead due to the migration policy, our solution consists of coupling the migration decision for an object together with the adaptive initial-placement decision, introduced previously. The migration of an object is examined only when one of its method is invoked. Furthermore, on the invocation of a method of an object $\mathcal{O}$ the only node considered for migration of $\mathcal{O}$ is the node selected for the method's execution. Migration of $\mathcal{O}$ then depends on its interactions with objects located on the remote node considered for migration compared to its interactions with objects located on the other nodes, including its current node. An alternative solution for selecting the node on which an object $\mathcal{O}$ is to be migrated would have been to take into account all the nodes hosting objects communicating with $\mathcal{O}$. We did not retain this solution because it increases the migration's communication cost.

The algorithm for implementing object migration is simple. Once a remote node is selected for a method execution, it is checked whether the embedding object $\mathcal{O}$ should be migrated on this node according to the number of interactions that took place between $\mathcal{O}$ and (objects on) this node compared to the interactions of $\mathcal{O}$ with the other nodes. More precisely, given the function $\mathcal{C}(N, \mathcal{O})$ that returns the number of interactions of $\mathcal{O}$ with objects located on node N, and assuming that N_i is the node selected for the method's execution by the adaptive placement policy[1], the migration algorithm is defined as follows:

[1] It can be noticed here that the information keeping count of object interactions

- If N_i is remote and is such that $\forall N_j \in hosts(\mathcal{O})$: $\mathcal{C}(N_i, \mathcal{O}) \geq \mathcal{C}(N_j, \mathcal{O}) + threshold$,
  - then $\mathcal{O}$ is migrated on N_i
  - else a copy of $\mathcal{O}$ is transferred on N_i for the duration of the method's execution.

3.2 Integrating the Load Balancing Strategy within Isatis

The integration of the load balancing strategy discussed above within Isatis leads to the following extensions:

- Introduction of system objects implementing controlled-exchange of load information,
- Refinement of the application classes so as to implement the proposed placement and migration policies.

It should be noticed here that, at that time, the proposed enhancement of Isatis has been made only for Arche applications; it follows that most of the implementation is written in the Arche language.

Implementing the controlled-exchange algorithm. The implementation of the controlled-exchange algorithm relies on two system-controlled objects defined by classes `load` and `loadExchange`.

The class `load` defines data-objects carrying global load information, that is, the load of processors hosting peer protection domains. In a Unix framework, which is the platform used for performance measures sketched in subsection 3.3, the local load related to a protection domain is obtained from the number of threads in the ready queue of the protection domain plus the number of processes of the processor's ready queue.

The class `loadExchange` defines service-objects sending their local load information to instances of `load` located within peer protection domains. It implements the controlled-exchange algorithm.

As for other system-controlled objects, there is a single instance of both `load` and `loadExchange` per protection domain of each protection delegate. These objects are further made known to application objects through a specialization of system-controlled objects managing protection domains (see § 2.3); each of these objects now embeds a reference to the local instance of `loadExchange`, and references to all the instances of `load` created within peer protection domains.

---

could be taken into account in the step (2) of the algorithm implementing the adaptive placement decision so as to select for execution the node that minimizes the communication cost of the method's execution.

Implementing adaptive initial-placement and migration. The integration of the proposed initial-placement and migration policies amounts to modify the behavior of the objects' distribution manager introduced in subsection 2.3: virtual methods `placement` and `migration` are redefined so as to respectively implement the adaptive initial-placement decision, and migration policy. In particular, the migration action now consists of transferring either the object's local representation in the absence of migration but in the presence of remote execution, or both the object's local representation and distribution manager in the presence of migration. We get the following structure for the redefinitions of `placement` and `migration`:

```
placement:
    Selection of the node N for m execution according to the adaptive
    initial-placement policy;

migration :
    migrate := decision of migrating O on N;
    if migrate
        then transfer on N of
            both O's distribution manager and O's local representation
        else transfer on N of O's local representation
    fi;
    Transfer of the call request on O's local representation to N;
    if not migration then gets the modified copy back of O's local representation fi
```

Let us notice here that the number of message exchanges due to a remote execution of `m` and to a migration of $\mathcal{O}$ depends on the sizes of the called object and of the input parameters.

The proposed placement and migration policies are implemented by an Arche class, called `Placement`, that can be inherited by any application class. This class redefines the virtual methods `placement` and `migration` of `t_actual` as sketched above. It further declares a reference to the built-in Arche class `domain` whose instance implements a system-controlled object managing the embedding protection domain (see § 2.3). Such an object allows to access system-controlled information including load information as well as the description of embedded application objects. The redefinition of `placement` subsumes computation of criteria T_{pro_i}, T_{trans_i}, and T_{res_i} so as to implement adaptive initial-placement decision. The criterion T_{pro_i} is computed by means of instances of system-controlled classes `load` and `loadExchange`. Assuming the invocation of a method `m` of $\mathcal{O}$, let us now examine the computation of method-related criteria T_{trans_i} and T_{res_i}. Criterion T_{trans_i} is evaluated from the size of $\mathcal{O}$ and `m`'s input parameters, and from the copying information for $\mathcal{O}$, which values are obtained from the local instance of `domain` that interfaces with the system. Finally, the evaluation of T_{res_i} depends upon the size of `m`'s result, which is obtained from the signature of `m`[2].

[2] Let us notice here that this applies in the case of Arche applications due to the fact

3.3 Assessment of the Load Balancing Strategy

In [9], performance measures of Isatis implementing the proposed adaptive initial-placement policy and the default migration policy have been compared with the ones of the base Isatis system that executes systematically a method on the storing node of its embedding object. The application considered for performance measures was a simple parallel application made of independent computations. Results show that the response times of applications using the default placement policy can be divided by up to a factor of ten when using the adaptive initial-placement policy. 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.

So as to measure the benefits due to the migration policy, we have been running a simple parallel application made of communicating objects. As for results discussed above, the application's performance is significantly increased when the application is run on a system with unequally loaded nodes. Furthermore, the coarser grain the objects are, the higher is the gain due to our load balancing strategy.

For a complete evaluation of the load balancing strategy, we now need to run real applications. We are currently implementing well-known scientific applications in the Arche language towards this goal. However, the results we have obtained so far suggest some remarks from the standpoint of the customization process. In particular, since the load balancing strategy is mainly beneficial for coarse-grain objects, customization of application objects with the proposed load balancing strategy should be considered accordingly by the application programmers. Better performance could further be obtained by supporting dynamic customization according to the environment's features through the adaptive programming technique [25]. This would allow to execute the load balancing strategy only when the load of the system's nodes is unbalanced. However, mechanisms implementing adaptive programming technique being not mature at that time, we prefer to concentrate on static customization.

Although our first results concerning the load balancing strategy have to be complemented with performance measures of real applications, some enhancements of the load balancing strategy can already be envisioned.

As stated in subsection 3.1, the adaptive initial-placement algorithm 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 [9], comparison of the base adaptive initial-placement policy with a policy taking into account the methods' expected duration shows that this knowledge can further lower applications' response times if the methods' expected duration is close to their actual duration. We are currently inves-

that objects are passed by reference, a more complex computation would be required should objects be passed by value, due to subtyping.

tigating the integration of such a knowledge in the adaptive initial-placement algorithm.

Finally, it should be noticed that the proposed load balancing strategy implements initial-placement and migration among nodes hosting the application which are fixed at creation time. This strategy could be complemented by an underlying one that would allow dynamic selection of nodes hosting the application; an example of such a strategy can be found in [27].

4 Conclusion

Despite its advantages for distributed programming, the object paradigm has had a moderate success in the distributed systems community due to the poor performance exhibited by distributed object-based applications. The inefficiency of distributed object-based applications being mainly due to the mismatch between the abstractions of object-based programming languages and the ones of traditional operating systems, customizable distributed object-based runtime systems such as Cool-v2 [32] have been introduced. Such systems allow to define the best mapping between language and system abstractions, and their performance tends to make the object-based distributed programming model as good a candidate as other distributed programming models for the development of distributed applications. However, so as to make execution of applications as efficient as possible, customizable distributed object-based runtime systems must support customization at the application-level, hence allowing the development of application-specific systems (e.g., see [4]). Furthermore, mechanisms for distributed object management aimed at improving performance, like for instance load balancing, have to be devised.

Summarizing the Isatis experience. This paper has introduced the Isatis customizable distributed object-based runtime system developed at Irisa within the Inria Solidor research group. The Isatis system is aimed at allowing the execution of distributed object-based applications written in different object-based languages, above various operating systems. It further supports customization at the application-level. The system's customizability is mainly achieved through its implementation in terms of C++ classes: Isatis customization amounts to implement subclasses of Isatis' base classes. At that time, our prototype supports the execution of applications written in either the Arche concurrent object-oriented language [12] or an extended version of C++. Furthermore, Isatis is currently implemented on top of Sun Sparc stations running SunOS, as well as on PCs running Mach.

Customization of the Isatis system has been illustrated through the integration of a load balancing strategy aimed at enhancing the applications' performance. The proposed strategy implements adaptive initial-placement of method execution and object migration. The placement policy is called adaptive as node selection for method execution is computed according to both processors' loads and objects' characteristics. The migration policy is coupled with the adaptive placement policy. On the invocation of a method of an object $\mathcal{O}$, the only node considered for migration of $\mathcal{O}$ is the node selected for the method's execution by the adaptive placement policy; migration of $\mathcal{O}$ then depends on its interactions with objects located on this node compared to its interactions with objects located on the other nodes. Integration of the load balancing strategy within Isatis is straightforward: it relies on a class implementing the placement and migration policies that can be inherited by any application class, an on the implementation of two system-level classes computing the system's global load. Performance of the Isatis system customized with the proposed load balancing strategy has been compared with the base Isatis system that does not support object migration and that implements the simple alternative initial-placement policy consisting of executing any method on the storing node of the called object. Results show that the response times of applications executing above the base Isatis system can be divided by up to a factor of ten when using the load balancing strategy.

In addition to the Isatis customization detailed in this paper, we have studied customization of the Isatis system so as to support different failure semantics for remote method calls. In particular, we have examined the implementation of the *at-most-once* failure semantics, based on the proposal of [29]. Briefly sketched, the resulting customization leads to refine transport objects so as to manage stable data structures keeping track of active remote calls, and to implement algorithms for orphan detection, orphan destruction, and detection of abnormal terminations.

Although not yet examined, another Isatis customization that could be considered is the introduction of the garbage collector dedicated to the Arche language and presented in [38]. The proposed garbage collector is comprised of a collection of local garbage collectors, loosely coupled to a global garbage collector. Each local collector identifies and reclaims the objects that can be detected to be garbage on a local basis, while it communicates with the global collector to get/provide information related to objects that possibly reference/are referenced by remote objects. The integration of such an algorithm within Isatis is close to the integration of the algorithm performing controlled exchange of load information. The local collectors together with the global collector would be implemented by means of system-controlled objects, exchanging information about object referencing.

As mentioned in the introduction, there has been much work in the field of system customization. Proposals aimed at defining new operating system kernels (e.g., see [13, 22]) should be seen as complementary to our work: Isatis could be ported above such kernels. Considering solutions oriented towards system structuring, the Isatis structuring is close to the one of the Choice operating system [17]. However, compared to this proposal, our work contributes from the standpoint of the design and integration of new policies for distributed object management.

Current and future work. In this paper, we have focused on the use of the customization approach at the application-level from the standpoint of performance enhancement. We are currently investigating complementary issues relevant to the customization process. These are mainly aimed at supporting safe system customization [15]. From the perspective of safe customization, we are examining how to enrich the type conformance relation so as to guarantee that a customized system object exhibits a behavior that is compatible with the one of the object it replaces. Briefly, our approach consists of extending the behavioral subtyping relation [3, 35] so as to integrate system-level specification within types. We are further designing new kernel mechanisms that ensure safe and efficient system customization when a type-unsafe programming language is used for implementing customized system functions [16]. Basically, the proposed mechanisms enforce isolation of the applications software components from the software components implementing the customized functions.

Acknowledgments: The authors would like to acknowledge Manuel Chevalier, Eric-Olivier Lebigot and Joel-Yann Fourré for participating to the implementation of the load balancing strategy.

References

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the Usenix 1986 summer Conference*, pages 93–112, 1986.
2. P. America. Pool-T: a parallel object-oriented language. In *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.
3. P. America. A Behavioural Approach to Subtyping in Object-Oriented Programming Languages. Technical report, Philips Research Laboratory, Eindhoven, The Netherland, 1989.
4. T. E. Anderson. The Case for Application-Specific Operating Systems. Technical Report UCB/CSD 93/738, Computer Science Division, University of California Berkeley, Berkeley, California, USA, 1992.
5. J. S. Auerbach, A. S. Gopal, M. T. Kennedy, and J. R. Russel. Concert/C: Supporting distributed programming with language extensions and portable multiprotocol runtime. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pages 152–159, 1994.
6. J. P. Banâtre and M. Banâtre, editors. *Les systèmes distribués : l'expérience du système Gothic*. InterEditions, 1991.
7. J.P. Banâtre, M. Banâtre, and G. Muller. Ensuring data security and integrity with a fast stable storage. In *Proceedings of the fourth International Conference on Data Engineering*, pages 285–293, 1988.
8. 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*, pages 510–517, 1994.
9. M. Banâtre, Y. Belhamissi, V. Issarny, I. Puaut, and J. P. Routeau. Adaptive placement of method executions within a customizable distributed object-based system: Design, implementation and performance. In *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*, pages 279–287, 1995.

10. A. Barak and A. Shiloh. A distributed load balancing facility for a multicomputer. *Software Practice and Experience*, 15(9):901–913, 1985.
11. Y. Belhamissi. *Placement des calculs et des données dans un système distribué à objets*. Thèse de doctorat, Université de Rennes I, Rennes, France, 1994.
12. M. Benveniste and V. Issarny. Concurrent Programming Notations in the Object-Oriented Language Arche. Research Report 1822, INRIA, Rennes, France, 1992.
13. B.N. Bershad, S. Savage, P. Pardyak, E. Gunder, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.
14. A. Birrel, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230, 1993.
15. C. Bryce, V. Issarny, G. Muller, and I. Puaut. Towards safe and efficient customization in distributed systems. In *Proceedings of the Sixth SIGOPS European Workshop: Matching Operating Systems to Application Needs*, Schloss Dagstuhl, Germany, 1994.
16. C. Bryce and G. Muller. Matching micro-kernels to modern applications using fine-grained memory protection. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, 1995.
17. R. H. Campbell, N. Islam, R. Johnson, P. Kougiouris, and P. Madany. Choices: Framework and refinement. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pages 9–15, 1991.
18. P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte, J. Mossière, and X. Rousset de Pina. Persistent shared object support in the Guide system: Evaluation and related work. In *Proceedings of the Ninth ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 129–144, 1994.
19. R. S. Chin and S. S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, 1991.
20. S. Crane. The Rex Lightweight Process Library. Technical report, Imperial College, London, UK, 1994.
21. 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*, 4(3):243–275, 1990.
22. D. R. Engler, M. Franz Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.
23. P. C. Fishburn. A survey of multi-attributes/multi-criteria evaluation theories. In *Lecture Notes in Computer Science 155*, pages 181–224. Springer-Verlag, 1978.
24. G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, 1992.
25. M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17(9):911–921, 1991.
26. G. Hamilton, M. L. Powell, and J. G. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 69–79, 1993.

27. 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*, pages 202–211, 1994.
28. Y. M. Huang and C. V. Ravishankar. Designing an agent synthesis system for cross-RPC communication. *IEEE Transactions on Software Engineering*, 20(3):188–198, 1994.
29. V. Issarny, G. Muller, and I. Puaut. Efficient treatment of failures in RPC systems. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, pages 170–180, 1994.
30. D. Kafura, M. Mukherji, and D. W. Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE Transactions on Parallel and Distributed Systems*, 1995.
31. S. Krakowiak, M. Meysembourg, H. Nguyen van, M. Riveil, and C. Roisin. Design and implementation of an object-oriented strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3:11–22, 1990.
32. R. Lea, C. Jacquemot, and E. Pillevesse. COOL: System support for distributed object-oriented programming. *Communications of the ACM*, 36(9), 1993.
33. R. Lea, Y. Yokote, and J. Itoh. Adaptive operating system design using reflection. In *Proceedings of the French-Japanese Workshop on Object-Based Parallel and Distributed Computation*, 1995.
34. B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, 1988.
35. B. Liskov and J. Wing. A new definition of the subtype relation. In *Proceedings of ECOOP'93*, pages 118–141, 1993.
36. Odjik. The DOOM system and its applications: A survey of Esprit 415 subproject A, Philips research laboratory. In *Proceedings of PARLE - Parallel Architectures and Languages Europe*, pages 461–479, 1987.
37. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995. 314-329.
38. I. Puaut. A distributed garbage collector for active objects. In *Proceedings of the Ninth ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1994.
39. 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*, 21(1):91–118, 1991.
40. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.
41. A. S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, S. J. Jansen, and G. Van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33:46–63, 1990.
42. Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992.