A Dynamic Reconfiguration Service for CORBA

Christophe Bidan, Valérie Issarny, Titos Saridakis, Apostolos Zarras
IRISA / INRIA
Campus de Beaulieu, 35042 Rennes Cédex, FRANCE
{issarny, bidan, saridaki, zarras}@irisa.fr

Abstract

Providing software qualities such as availability, adaptability and maintenability to long-running distributed applications, forms a major challenge for the configuration management of a software system. Modifications of system's structure are expected to happen on-the-fly, to cause minimum execution disruption, and to be effected in a way that preserves a consistent state of the participating entities. This paper presents a novel algorithm for performing consistent dynamic reconfiguration of CORBA applications, where consistency refers to RPC integrity. The novelty of the algorithm is that it passivates the links affected by the reconfiguration, which causes the node activities that use them to block, but does not result in blocking the entire node. The consequent execution disruption is minimal, a fact that is practically verified by a performance evaluation done in a number of different reconfiguration scenarios.

Keywords: *Consistency, CORBA application, Dynamic Reconfiguration, Execution disruption, RPC integrity.*

1. Introduction

The need for a *dynamic reconfiguration service* (DRS) to support software evolution is an urgent demand in the domain of long-running distributed applications. In general, dynamic reconfiguration refers to the changes of system's logical and physical structure (e.g. insertion, removal or replacement of physical nodes, logical components and interaction links). These changes can be roughly classified into two categories: `i)` programmed reconfiguration (or *ad hoc* changes) that is part of the system design and is scheduled to happen when certain conditions are satisfied during application execution (e.g. embedded failure recovery), and `ii)` evolutionary reconfiguration (or *post hoc* changes) that refers to structure modifications caused by events independent from the application specifications (e.g. maintenance actions, components upgrade, etc). Despite the differences in the nature of these reconfiguration classes, the primitive operations that should be provided by the reconfiguration service are the same in both cases: creation and removal of components, creation and removal of links, and state transfer among components. In addition, two requirements are placed on the use of these primitives to perform a reconfiguration: to preserve some well-defined consistency constraints and at the same time to introduce a minimal disruption to the length of time of the system's operation.

In order to accommodate an application independent dynamic reconfiguration service, the functional concerns of the application components should be separated from the structural configuration concerns, and only the latter should be considered in reconfigurations. This allows to build a system level service that supports configuration changes independently from application specifications, and to incorporate it into programming environments aimed at developing distributed applications. CORBA compliant programming environments allow a clean separation of application functional and structural concerns, where an application consists of a number of objects that communicate over an ORB, which offers RPC capabilities [11]. A number of execution-support facilities (e.g. object life-cycle control, persistent objects, transactional facilities, etc) can be integrated into an ORB as Common Object Services (COS), and can be directly used by application objects [10]. This technology favors the construction of a reconfiguration mechanism on top of CORBA services like the LifeCycle COS, and its incorporation in a CORBA compliant platform.

In this paper we address the issue of dynamic reconfiguration in the CORBA framework. Building on the semantics of the LifeCycle CORBA service, we propose a novel algorithm for dynamic reconfiguration that introduces minimal disruption on system's execution by passivating only the links attached to the target of the change. In a multi-threaded execution environment, passivating links results in suspending the activities of neighbor objects that use these links, while the rest of the activities remain unaffected. Changes are initiated and coordinated by a Dynamic Reconfiguration Manager (DRM) which interacts with the

reconfiguration activities of application objects. The overall schema guarantees local node consistency [8] which is interpreted as RPC integrity in the CORBA framework, i.e. all RPCs initiated by activities affected by the reconfiguration will be completed before the changes are effected.

The remainder of the paper is organized as follows: in the next section we present our motivations and design decisions by juxtaposing the reconfiguration requirements of typical CORBA applications with existing reconfiguration algorithms. From this discussion it follows that a novel algorithm is needed to guarantee RPC integrity while introducing minimal disturbance to application execution. The prominent aspects of the algorithm, its implementation and an evaluation of its performance are presented in §3. The concluding section summarizes the paper, stresses our contribution, discusses the limitations of the proposed algorithm, and refers to current status and future work issues.

2. Motivations

The need to support the structural changes of distributed applications has been recognized by the OMG, who has proposed the LifeCycle COS that consists of a set of facilities for controlling objects life-cycles (see chapter 6 in [10]). However, there is a significant difference between controlling object life-cycles and managing configuration changes: the former deals with the primitives that a CORBA platform should possess in order to support the creation, transfer and removal of objects, while the latter is occupied with how to use these primitives and coordinate the resulting actions in order to effect the configuration changes efficiently and without affecting some well-defined form of consistency in the system. Our goal is to extend the LifeCycle facilities to support dynamic reconfiguration of a CORBA application. This can be achieved by the construction of a service on top of the LifeCycle COS, which uses the primitives provided by this CORBA service in order to coordinate the reconfiguration actions.

2.1. Design Objectives

Two major requirements on a reconfiguration service are to be efficient and to preserve some well-defined consistency of the system. Efficiency refers to minimal execution disruption, both during normal system execution and during configuration changes. The former aspect of execution disruption is due to the overhead introduced by the mechanism that detects reconfiguration conditions, and can be minimized by a careful implementation. The latter execution disruption aspect depends on the reconfiguration management algorithm, which is the focus of attention in the remainder of the paper. To keep the execution disruption due to configuration changes minimal, the reconfiguration should affect the smallest possible set of system activities. In the case of composite objects consisting of independently executing activities (e.g. multi-threaded objects), only the activities actually affected by the reconfiguration should be blocked, and not entire objects containing affected activities.

The consistency objective is somewhat more delicate to specify. To build an application independent reconfiguration service, the guaranteed consistency should be application independent. We define the term consistency in the CORBA framework as the property of RPC integrity, *i.e. a state of an object is consistent if there are no pending RPCs initiated by that object*. RPC integrity guarantees local node consistency but not application consistency in terms of some system-wide invariant. The latter is application dependent and additional mechanisms should be built on the top of our reconfiguration service to guarantee it. In the remainder of the paper, we concentrate our attention on node consistency issues.

At this point, the problem of a dynamic reconfiguration management integrated with a CORBA compliant platform is well defined. What is needed is an algorithm that satisfies the efficiency and consistency constraints mentioned above, by providing a set of operations that allow the creation, state transfer, and removal of objects and the creation and removal of links. In the remainder of this section we make a brief overview of some representative reconfiguration algorithms proposed in the literature. Beside the fulfillment of the efficiency and consistency requirements, we also examine their independence of some system-specific support and the amount of programmer guidance that they require.

2.2. Consistent Dynamic Reconfiguration

Preserving some form of consistency during reconfiguration actions has been one of the principal goals of algorithms dealing with configuration changes. We have identified two categories of reconfiguration mechanisms, focusing on application consistency and on node consistency respectively. Although application consistency entails application dependence, mechanisms providing such consistency cause low execution disruption. On the other hand, despite the fact of elevated execution disruption, mechanisms providing node consistency require less programmer guidance and less system-specific features.

Among the prevalent work in dynamic reconfiguration preserving application consistency is the one based on the POLYLITH distributed programming environment [5]. Reconfiguration points are explicitly defined in the application source code by the programmer. The execution disruption caused by this mechanism is minimal due to the fact that the application execution is disrupted only when necessary to perform some configuration change. Nevertheless, the process of explicitly identifying the reconfiguration points

is too error prone, especially as the distributed application scales in source code size. Similar approaches presented in [4] and [13] are respectively based on facilities provided by the Chorus operating system, and on PCL (a configuration language for modeling evolving system architectures). These approaches alleviate the programmer from some of the drudgery of identifying reconfiguration points, by automating the corresponding process. They also cause low execution disruption, but they remain application dependent, and in addition they require system-specific features.

An effort towards formalizing reconfiguration rules is presented in [14], where application consistency is expressed in existence and placement constraints and reconfiguration rules are given as an execution condition followed by a sequence of reconfiguration actions. Although the authors do not give any details concerning the execution disruption caused by their implementation using DARWIN and the NIH C++ class library, we can deduce that a mindful implementation should not introduce high execution disruption. In the same spirit, two approaches are based respectively on a Finite State Machine model [9] and on a CSP- or CCS-like process algebra [2]. They both require the formal specifications of the reconfiguration conditions, which can be seen as part of the application specifications. Hence, if the programming environment supports formal specifications, the guidance that is expected from the programmer is not very high and it amounts to defining the synchronization constraints for the reconfigurations. In the absence of a support for formal verification, the cost of validating the reconfiguration constraints becomes almost prohibitive for large scale distributed applications.

The DURRA programming environment supports an event-triggered reconfiguration mechanism [1], which is used mostly for error recovery purposes. The description of the application structure contains more than one alternative configurations, associated with an execution event which serves as the condition that triggers a reconfiguration action. Medium guidance is required, since the programmer has to consider all possible execution events that may trigger a reconfiguration, but there is no need to modify application's source code.

The fact that the above approaches require substantial programmer guidance and depend on application factors, render them unsuitable for the purpose of implementing a dynamic reconfiguration service for CORBA platforms. To obtain application independence, the reconfiguration mechanism should focus on node interactions rather than application specifications. This form of local consistency has been first considered in the CONIC programming environment to support the integrity of transactions [8]. The reconfiguration schema and its variants (e.g. [3] and [12]), have been based on the *quiescent* state of nodes, in which a node can no longer interact with its environment. They propose an order for rendering nodes in a quiescent state, in such a way that no partially completed transactions are blocked by the reconfiguration actions. The variants of the initial algorithm focus either on efficiency aspects by minimizing the set of blocked nodes (e.g. [3]) or by associating locks with object methods, which must be acquired prior to call a method on a server, in order to facilitate node passivation (e.g. [12]).

The algorithms that provide application consistency, introduce low execution disruption but are application dependent and generally need substantial guidance. On the other hand, the algorithms that guarantee node consistency, are application independent but introduce a non-negligible execution disruption since they block more system activities than necessary. From these remarks raises the need for a novel reconfiguration algorithm, which should remain application independent by preserving some form of local consistency while, at the same time, it should minimize the execution disruption that it introduces. The next section presents the design, implementation and performance evaluation of such an algorithm.

3. Dynamic Reconfiguration Service

Our algorithm for the reconfiguration service is designed for a CORBA framework supporting multi-threaded objects, following the thread-per-operation execution model. Since we do not have an implementation of the LifeCycle COS, we first need the standard LifeCycle primitives *create* and *remove* that correspond to the homonym configuration actions. In addition, we need to extend their semantics so as to include basic knowledge about configuration and consistency constraints. Moreover, we need the primitives *link, unlink, transferLink,* and *transferState* to respectively create and destroy a link, transfer the requests pending on a passivated link to another existing link, and to transfer the state from one object to another. In the following subsections, we use these primitives to elaborate on the Dynamic Reconfiguration Service (DRS) properties in terms of the consistency and efficiency constraints.

3.1. Consistency and Efficiency Constraints

The consistency constraint that we wish to guarantee in the CORBA framework, is the one of RPC integrity, i.e. a reconfiguration action should not leave initiated RPCs pending. The consistency constraint (CC) is formally expressed as:

$$CC \equiv \forall l \in \mathcal{L} : (remove(l) \Rightarrow (\forall r \in \mathcal{R}_l : send(r) \Rightarrow result(r)))$$

where $\mathcal{L}$ is the set of system links, $\mathcal{R}_l$ the set of RPCs issued over l, $send(r)$ and $result(r)$ are the predicates that hold if RPC r is initiated and the result is returned respectively, and $remove(l)$ holds if link l is removed. Notice

that CC implicitly assumes RPCs whose synchronization semantics are either synchronous or asynchronous with deferred synchronization on the result reception. In CORBA, the asynchronous RPC semantics is also provided. However, this case can be handled by considering that $result(r)$ holds as soon as the request is received, which is correct with respect to the ORB specification.

The proposed CC differs from the one usually employed in the design of dynamic reconfiguration services [8]: the latter takes into account a sequence of nested RPCs (e.g. transactions) while our CC is given in terms of independent RPC requests. We have decided to design a DRS service based on CC due to the fact that in CORBA, the notion of nested RPCs is not specified; only object-to-object communications are defined. Nested RPCs become meaningful when using additional object services such as the OTS COS for transaction management (see chapter 10 in [10]). For illustration purposes, let us consider a configuration consisting of three objects o_1, o_2, and o_3 connected in sequence, where an RPC from o_1 to o_2 leads to a nested RPC from o_2 to o_3. If the application manager replaces object o_3 by object o_4, then using CC the RPC integrity is guaranteed if there are no pending RPCs from o_2 to o_3 prior to o_3's removal.

Given CC, let us now examine the conditions under which the reconfiguration actions can be effected in order to ensure CC as actions' postcondition. The *create* and *link* actions do not affect CC since they do not affect any initiated RPCs. On the other hand, to satisfy CC when removing a link, one should guarantee that there are no pending RPCs over that link. Correctness of the *remove* action requires to ensure that the resulting object removal does not lead to the deadlock of any pending RPCs neither on the incoming nor on the outgoing links. The *transferState* action can be handled like the *remove* action, in which case the object state is copied only when the original object is not servicing any request. In the same way, the *transferLink* operation can be handled like the *unlink* action, and its precondition is that the original link is passivated and the destination link exists. More formally, we have:

$$
\begin{array}{lcl}
pre_{create} & \equiv & true \\
pre_{link} & \equiv & true \\
passive(l) & \equiv & (\forall r \in \mathcal{R}_l : \\
 & & send(r) \Rightarrow result(r)) \\
pre_{unlink}(l) & \equiv & passive(l) \\
pre_{remove}(o) & \equiv & \forall l \in incoming(o) \cup \\
 & & outgoing(o) : passive(l) \\
pre_{transferLink}(l_1, l_2) & \equiv & passive(l_1) \wedge exist(l_2) \\
pre_{transferState}(o_1, o_2) & \equiv & pre_{remove}(o_1) \wedge exist(o_2)
\end{array}
$$

where $incoming(o)$, and $outgoing(o)$ stand respectively for the incoming and outgoing o's links.

From the standpoint of the efficiency constraint, the DRS must ensure that a minimal set of system activities is disturbed by a reconfiguration action. Optimally, these activities should be only those affected by the configuration changes. Activities affected by a reconfiguration action are given by the preconditions of the reconfiguration actions, which indicate the links that must be passivated. The efficiency constraint (EC) is formally expressed as:

$$
EC \equiv \forall l \in \mathcal{L} : blocked(l) \Rightarrow (unlink(l) \vee (l \in \mathcal{L}_o \wedge remove(o)))
$$

where $blocked(l)$ holds if the entity issuing a request over l is blocked by the DRS for reconfiguration purpose, $unlink(l)$ and $remove(o)$ hold if the corresponding reconfiguration actions are requested, and $\mathcal{L}_o$ is the set of incoming and outgoing o's links.

3.2. Algorithm

Given the DRS specification, the DRS algorithm is almost direct. It should be guaranteed that links that must be passive with respect to the requested reconfiguration action are indeed passive and remain so (CC criterion) for the duration of the action, and that all the other links (i.e. activities that do not issue request over the links that need to be passive) are not affected (EC criterion). The following is the description of the reconfiguration algorithm:

```
Reconfiguration =
var
   % data object that stores the configuration graph
   config: config_desc;
begin case
   create(obj: obj_desc):
      config.addObj(obj); create(obj);
   link(client_obj: obj_desc; server_obj: obj_desc):
      config.addLink(client_obj, server_obj);
   unlink(client_obj: obj_desc; server_obj: obj_desc):
      passivateLink(client_obj, server_obj);
      config.delLink(client_obj, server_obj);
   transferLink(client_obj: obj_desc;
               server_obj1, server_obj2: obj_desc):
      if (config.passive(client_obj, server_obj1)) then
         moveLink(client_obj, server_obj1, server_obj2);
   remove(obj: obj_desc):
      blockObject(obj); config.delObj(obj); remove(obj);
   transferState(obj1, obj2: obj_desc):
      blockObject(obj1); copy(obj2, getState(obj1));
      unblockObject(obj1);
end case
end   % End of Reconfiguration
```

The behavior of the operations on the `config` data object is direct from their name, and the **create**, **delete** and **copy** functions realize the corresponding actions on the objects according to the LifeCycle COS specification. We do not describe the `moveLink` and `unblockObject` operations, their implementation is straightforward from our presentation. The core operations of the algorithm lie in the `passivateLink` and `blockObject` operations. The `passivateLink` operation consists of requesting the

client object to no longer issue a request on the corresponding link (i.e. to block any object activity issuing a request over the link) and to make sure that there is no pending RPC request on this link. The `blockObject` operation functions as follows:

```
blockObject(object: obj_desc) =
  var clients, servers: list of obj_desc;
  begin
    % get the list of object's clients
    clients := config.getClients(object);
    forall obj in clients do
      passivateLink(obj, object);
    end;
    % get the list of object's servers
    servers := config.getServers(object);
    forall obj in servers do
      passivateLink(object, obj);
    end;
  end    % End of blockObject
```

The precondition that must hold for removing an object is that all the incoming and outgoing links of the object are passive, i.e., they are no pending RPC requests on these links and no new RPC request will be issued through these links. Thus, it consists of performing `passivateLink` operations on the object o to be removed, and on the objects for which o is a server so as to respectively passivate outgoing and incoming o's links. However, care should be taken about the order in which the `passivateLink` operations are performed due to nested RPC requests. For illustration purpose, let us consider the configuration made of objects o_1, o_2, and o_3 such that o_1 sends a synchronous RPC request to o_2, and the call treatment within o_2 leads to issue a nested synchronous RPC request to o_3. Let us now assume that the removal of o_2 is requested. The link from o_1 to o_2 must be passivated before the link from o_2 to o_3 so as to not introduce a deadlock. In general, client objects of an object o to be removed are requested first to passivate their link to o, and once they have all acknowledged link passivation, o is requested to passivate its outgoing links. Notice that the proposed ordering applies to applications that have cycles in their configuration graph as long as their execution is cycle-free.

Configurations with a cycle in their execution are handled by using a different reconfiguration algorithm that is not further considered in the remainder of this paper due to space limitations. Briefly stated, the undertaken solution consists of temporarily unblocking activities that are blocked on a passivated link, in a way similar to the algorithm of [3], which leads to a less efficient algorithm that does not satisfy the EC criterion. The resulting non-conformance with the EC requirement in the case of configurations with execution cycles is of a minimal penalty in the CORBA framework. A CORBA application is typically structured as a set of objects engaged in pairwise interactions according to the client/server communication model. An object playing the role of a client issues a request *via* an RPC to an object playing the role of the server. The latter performs the requested services and returns a result to the client. In this framework, we can safely assume that a big majority of applications are based on cycle-free object interactions, i.e. in the case of nested RPCs (RPCs that initiate other RPCs and depend on their successful termination), an object that has initiated a nested RPC cannot play the role of the server for some consequent RPC.

The correctness of the proposed reconfiguration algorithm depends on: (`i`) the satisfaction of the CC and EC criteria, and (`ii`) the algorithm's safety and liveness. The liveness and safety properties are trivial to show due to the passivation of links and the ordering of passivation actions, respectively. It is also trivial to show that CC and EC are guaranteed. By design of the algorithm, a link is passivated before being removed and no request is issued over that link after its removal, hence satisfying CC. In the same way, EC is guaranteed since only activities that are issuing requests over the links that are affected by a reconfiguration are blocked.

3.3. CORBA specification

Figure 1 illustrates the major components and interfaces defined by the DRS. The objects belonging to an application

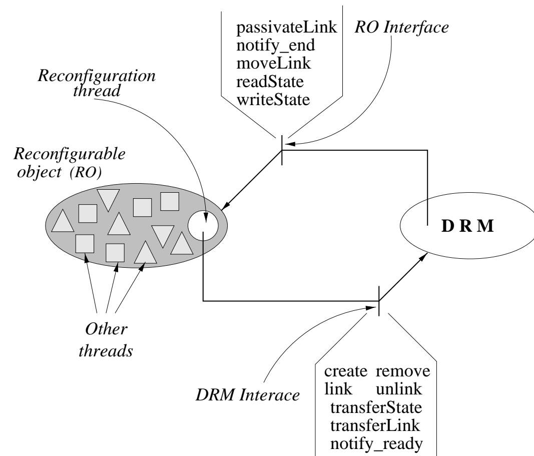

Figure 1. Major components of the DRS

that needs to be dynamically reconfigured must be *reconfigurable*, which allows them to interact with the *Dynamic Reconfiguration Manager* (DRM) that provides the reconfiguration primitives. The IDL interface of the DRM is the following:

```
interface DRM {
  long create(in string object);
  long remove(in string object);
  long link(in string client, in string server);
  long unlink(in string client, in string server);
  long transferState(in string source,
                     in string dest);
  long transferLink(in string server1,
                    in string server2);
  long notify_ready(in string object);
}
```

In addition to the reconfiguration operations introduced so far, it provides the `notify_ready` operation. This is due to the fact that the requests for link passivation that are issued to reconfigurable objects are sent asynchronously[1] for efficiency purpose; `notify_ready` is thus used to detect termination of the requested link passivation actions. We do not detail the implementation of the DRM, it is direct from the algorithm described in the previous subsection. Let us simply remark that the DRM is currently centralized; distribution of the DRM is one of the planned enhancements for the DRS so as to cope efficiently with large scale distributed CORBA applications. For an object to be reconfigurable, it must inherit from the `RO_object` class whose IDL interface is given below:

```
interface RO_Object {
  void passivateLink(in sequence<string> servers);
  void moveLink(in string server1,
                in string server2 );
  void notify_end(in sequence<string> status,
                  in sequence<string> servers);
  void writeState(in RO_state state);
  void readState(out RO_state state);
}
```

The argument of the `passivateLink` operation is the list of server objects with which the links to be passivated serve to communicate. This operation consists of updating data structures so as to block any object thread that will be issuing a request over the passivated links and, concurrently, to notify the action termination using the DRM `notify_ready` operation once all the RPC requests that are being served over the passivated links are terminated. The `notify_end` operation is then called by the DRM to notify termination of the reconfiguration action that led to the request for passivating the links with the server objects. If the first argument of `notify_end` indicates that both the server objects and the links are still present (i.e. the links were passivated to execute the `transferState` action) then the threads blocked on an RPC request to the server objects resume execution. Otherwise, these threads remain blocked. A thread that remains block subsequently to the execution of `notify_end` will be unblocked if the target link of the suspended RPC request is substituted by another one using `moveLink`.

[1]The **CORBA**::**Request**::`send_oneway()` operation is used in order to perform non-blocking RPCs.

In the case where `moveLink` is not called, this means that there is a software fault since the reconfiguration altered the application consistency. At this time, we have not integrated software fault management within the DRS so as to detect the above type of faults. This is an issue for future enhancement of the DRS, which may be handled using timeouts and garbage collection. The `moveLink` operation allows to move the requests that are blocked over a link to another link by updating the server object reference and then resuming the blocked threads. Finally, the `writeState` and `readState` operations respectively modifies the object state and returns the current value of the object state. These two operations are provided as substitution of the **copy** operation provided by the LifeCycle COS since we have no such service available at this time; in particular, `writeState` and `readState` together augment the semantics of **copy** as given in the LifeCycle COS (chapter 6 in [10]).

In addition to the implementation of the operations provided in the reconfigurable object IDL interface, any reconfigurable object redefines the ORB `invoke` operation that is provided to issue RPC requests over the ORB. The redefinition consists of issuing an RPC request only if the target link is valid and is not passivated.

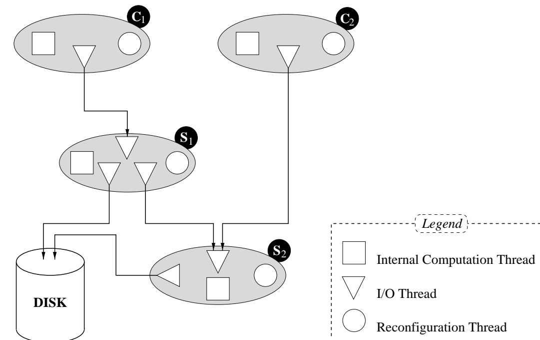

Figure 2. Configuration of the test application

3.4. Implementation and Evaluation

We have implemented the DRS on Orbix2.0 MT (a CORBA compliant, multi-threaded platform), supported by the Solaris 5.5.1 thread API, and we have performed some measurement on Sun Ultra WorkStations running Solaris 5.5.1. Figure 2 illustrates the configuration of the application used for performance evaluation purposes: C_1 is a client object that does a certain amount of internal compu-

tation and then issues synchronous RPCs to S_1, which is a storage server object. Upon receiving a request, S_1 writes some data on the disk, performs some internal computation, and does a nested RPC to S_2, which is another storage server object that replicates the data after executing some internal computation on them. Finally, C_2 is similar to C_1; a client object that does some internal computation and then issues synchronous RPCs to S_2 for writing data on the disk.

Based on this configuration, we measured the execution times of different versions of an application where each of the C_1 and C_2 issues 1000 requests. The parameters varying among the different versions are the size of the data written on disk, and the amount of internal computations performed by each object. We executed five versions of the application with communication intensity ranging from more than 95% to less than 5%, in two sets of measurements. The first set was performed over a cluster of Sun Ultra WorkStations and the goal was to measure the overhead introduced by the reconfiguration mechanism. For doing so, we measured the execution time of the application with and without the DRS. The second set of measurements aimed at computing the actual cost of configuration changes, and it included three independent reconfigurations, performed one at a time: in the first reconfiguration measurement we replaced C_2, in the second we replaced S_1, and in the last one we replaced S_2. We executed this set of measurements on a single Sun Ultra WorkStation, in order to better monitor the reconfiguration process, and to manage isolating the cost of the reconfiguration algorithm from the distribution factors. The requests issued to the DRM in each reconfiguration case are given in the following table:

$C_2 \rightarrow C_2'$	$S_1 \rightarrow S_1'$	$S_2 \rightarrow S_2'$

create(C_2');
unlink(C_2, S_2);
transferState(C_2, C_2');
link(C_2', S_2);
remove(C_2);
	create(S_1');
	unlink(C_1, S_1);
	unlink(S_1, S_2);
	transferState(S_1, S_1');
	transferLink(C_1, S_1, S_1');
	link(S_1', S_2);
	remove(S_1);
		create(S_2');
		unlink(S_1, S_2);
		unlink(C_2, S_2);
		transferState(S_2, S_2');
		transferLink(C_2, S_2, S_2');
		transferLink(S_1, S_2, S_2');
		remove(S_2)

The results presented in table 1(i) refer to the execution disruption caused by the mechanism that detects reconfiguration conditions. This cost is almost stable in terms of absolute time (between 6sec and 8sec) and causes a disruption that ranges from 45% in the communication intensive case to less than 1% in the computation intensive case. We bring to the reader's attention that these measurements are taken on the prototype implementation of the DRS, which does not focus on this aspects of execution disruption. Improving the performance of the mechanism that detects reconfiguration conditions is part of the ongoing work.

On the other hand, the cost of executing a reconfiguration is given in table 1(ii). The first column indicates the execution time of the reconfigurable application without performing any reconfiguration actions, while the next columns show the cost of replacing each of the C_2, S_1, and S_2 respectively. The reader may verify that the cost of the performed reconfigurations always remains below 7%, and becomes almost negligible in the computation intensive case. This fact verifies the efficiency of DRS when it comes to perform configuration changes, which was one of our basic objectives.

Comm.	**Non-Reconf.**	**Reconf.**
>95%	18.31sec	26.59sec
50%	32.00sec	38.54sec
35%	47.27sec	54.12sec
20%	79.07sec	80.11sec
<5%	408.08sec	408.12sec

(i)

Comm.	**base**	$C_2 \rightarrow C_2'$	$S_1 \rightarrow S_1'$	$S_2 \rightarrow S_2'$
>95%	32.12sec	33.77sec	34.03sec	34.29sec
50%	48.82sec	50.61sec	51.00sec	51.47sec
35%	63.70sec	65.25sec	65.82sec	66.67sec
20%	85.02sec	86.25sec	86.97sec	87.57sec
<5%	406.1sec	407.6sec	407.9sec	408.3sec

(ii)

Table 1. Performance results

4. Conclusions

We have presented the conception, implementation design, and performance evaluation of a novel algorithm for dynamic reconfiguration of multi-threaded CORBA applications, which preserves RPC integrity. The algorithm builds on the semantics of the CORBA LifeCycle COS, and can be seen as an extension of it. By design, the execution disruption introduced due to reconfiguration actions is minimal, since the algorithm passivates links, a fact that results in blocking only the threads that use them but not entire objects (i.e. nodes of the configuration graph). The performance evaluation showed a low reconfiguration cost in the order of 0.50sec - 1.80sec, which is almost independent of the communication intensity of the application. On the other hand, the overhead due to the existence of the reconfiguration threads that is introduced to the normal execution of an application was shown to decrease as the communication intensity decreases, ranging from 45% for communication intensive applications to less than 1% for computation intensive applications.

Contribution. From the standpoint of CORBA applications, DRS can be integrated with an ORB execution platform and thereafter used as a COS extension. The facts that it is application independent and also that it is built on top of the LifeCycle COS, advocate to this direction. Hence,

our first contribution is an extension of a CORBA COS for dynamic reconfiguration that preserves RPC integrity. From the point of view of configuration management, the presented algorithm succeeds almost optimum score in all four categories on which we have compared related work in §2.2. It is application independent, it requires minimum programmer guidance, and it introduces minimal execution disruption while preserving a well-defined consistency constraint on objects interactions. Thus, another basic contribution is the successful assembly of the prevalent characteristics of existing dynamic reconfiguration algorithms in a single service for the management of configuration changes of client/server distributed applications. Finally, the generality of the algorithm permits its use for the configuration management of a wide spectrum of distributed systems. The preconditions for its employment, namely RPC communications and multi-threaded objects, can be scarcely interpreted as limitations, especially in the framework of CORBA applications.

Current and Future Work. Building on our previous experience on providing some form of dynamic reconfiguration for CORBA applications [6], we have carefully implemented the DRS described in §3, focusing primarily on efficiency aspects. We are very happy with its performance, considering that it is still in an experimental stage. Currently we are occupied with performing some refinements on the activation of the reconfiguration thread, and with tuning the coordination actions of the DRM in order to further optimize the DRS performance. We also study an extension of the DRS, which will ease the reconfiguration management by allowing a set of related configuration changes to be effected in a single reconfiguration action.

In the near future, we plan to exploit the DRS utility by using it in the customization of execution platforms for applications that require support for configuration changes (e.g. tolerating object failures, guiding server replication for load balancing purposes, etc). For doing so, we intent to integrate the DRS in Aster, a distributed programming environment for distributed system customization, based on the formal specifications of the application's requirements (e.g. see [7]).

References

[1] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. Durra: A Structure Description Language for Developing Distributed Applications. *IEE Software Engineering Journal*, pages 83–94, March 1993.

[2] G. Etzkorn. Change Programming in Distributed Systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 140–151, March 1992.

[3] K. M. Goudarzi and J. Kramer. Maintaining Node Consistency in the Face of Dynamic Change. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 62–69, May 1996.

[4] S. Hauptmann and J. Wasel. On-line Maintenance with On-the-fly Software Replacement. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 70–80, May 1996.

[5] C. R. Hofmeister and J. Purtilo. Dynamic reconfiguration in distributed systems. adapting software modules for replacement. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 101–110, May 1993.

[6] V. Issarny, C. Bidan, and T. Saridakis. Designing an Open-ended Distributed File System in Aster. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, pages 163–168, September 1996. Also available at http://www.irisa.fr/solidor/work/aster.html.

[7] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, May 1998.

[8] J. Kramer and J. Magee. The Evolving Philosophers Problem. *IEEE Transactions on Software Engineering*, 15(1):1293–1306, November 1990.

[9] A. S. Lim. Abstraction and Composition Techniques for Reconfiguration of Large-Scale Complex Applications. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 186–193, May 1996.

[10] OMG Document. CORBAservices: Common Object Services Specification. Technical report, Object Management Group, November 1995.

[11] OMG Document. The Common Object Request Broker: Architecture and Specification (Revision 2.0). Technical report, Object Management Group, July 1995.

[12] I. Oueichek and X. R. de Pina. Dynamic Configuration Management in the Guide Object-Oriented Distributed System. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 28–35, May 1996.

[13] I. Warren and I. Sommerville. A Model for Dynamic Configuration which Preserves Application Integrity. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 81–88, May 1996.

[14] M. Zimmermann and O. Drobnik. Specification and Implementation of Reconfigurable Distributed Applications. In *Proceedings of the 2nd International Conference on Configurable Distributed Systems*, pages 23–34, May 1994.