Transparent Dynamic Reconfiguration for CORBA

João Paulo A. Almeida[1,2], Maarten Wegdam[1,2], Marten van Sinderen[1], Lambert Nieuwenhuis[1]
almeida@cs.utwente.nl, wegdam@lucent.com, sinderen@cs.utwente.nl, L.J.M.Nieuwenhuis@cs.utwente.nl

[1]Centre for Telematics and Information Technology, University of Twente, PO Box 217, 7500 AE, Enschede, The Netherlands

[2]Lucent Technologies, Bell Labs Twente, Capitool 5, 7521 PL, Enschede, The Netherlands

Abstract

Distributed systems with high availability requirements have to support some form of dynamic reconfiguration. This means that they must provide the ability to be maintained or upgraded without being taken off-line. Building a distributed system that allows dynamic reconfiguration is very intrusive to the overall design of the system, and generally requires special skills from both the client and server side application developers. There is an opportunity to provide support for dynamic reconfiguration at the object middleware level of distributed systems, and create a dynamic reconfiguration transparency to application developers. In this paper, we propose a Dynamic Reconfiguration Service for CORBA that allows the reconfiguration of a running system with maximum transparency for both client and server side developers. We describe the architecture, a prototype implementation, and some preliminary test results.

Keywords: dynamic reconfiguration, distributed systems, middleware, CORBA, on-line upgrade

1 Introduction

Distributed computing systems are being used for many years in various large-scale commercial and industrial environments. Such systems are also deployed in mission-critical and highly available applications, e.g., for telecommunications switches and e-commerce solutions. Consequently, long downtimes for these applications are usually unacceptable due to economical or safety reasons. In many cases, the availability of a distributed computing system is determined by its downtime due to various types of maintenance. In practice, a reconfiguration implies that the system needs to be taken offline and restarted after installation of new software components. The downtime due to maintenance can be avoided by using *dynamic reconfiguration* [1, 3, 4, 5, 7, 9, 10, 11, 13, 20, 25], i.e., the system can be maintained or upgraded without being taken off-line.

The aim of *dynamic reconfiguration* is to allow a system to evolve at run-time [9], as opposed to design-time, while introducing little (or ideally no) impact on the system's execution. In this way, the system does not have to be taken off-line to accommodate changes. We distinguish two types of changes, related to the moment they are envisioned [11]: *programmed* changes are foreseen and anticipated by the system designer, while *evolutionary* changes are unanticipated and become necessary over the execution lifetime of an application.

In case of a dynamic reconfiguration, certain entities of the distributed system are affected, while other entities remain functioning. Entities can be objects, groups of objects, components, groups of components, sub-systems, bindings and groups of bindings. Operations on entities can be *replacement, migration, addition,* and *removal.* The possible changes applied to a system depend on the granularity of the reconfigurable entities and the operations that can manipulate such entities in the affected part of a system.

New generations of distributed applications often consist of co-operating objects, and make use of object-middleware technology, such as CORBA [14], Java RMI and DCOM. Object-middleware facilitates the development of distributed applications by providing distribution transparencies to the application designers. Object-middleware offers a widely accepted approach for the provisioning of flexible computing environments. As such, there are many systems that would profit from dynamic reconfiguration facilities for object-middleware, such as, e.g., critical and/or long-running systems. The development of such systems would be facilitated through the inclusion of (transparent) reconfiguration support in the middleware platform. Although we focus on CORBA in this paper, our approach and architecture is also suitable for other object middleware technologies.

This paper is further structured as follows. Section 2 presents terminology, definitions and concepts used in our discussion of dynamic reconfiguration, as well as requirements for a dynamic reconfiguration service for object-middleware; Section 3 describes our dynamic reconfiguration approach; Section 4 presents the architecture of a Dynamic Reconfiguration Service for CORBA; Section 5 describes the implementation based on portable interceptors; Section 0 presents an evaluation of our work, using the results of the experiments conducted with the prototype, and compares our approach to related work found in the literature. Finally, Section 7 presents conclusions and future work.

2 Dynamic reconfiguration

This section further introduces the concept of dynamic reconfiguration and presents a general dynamic reconfiguration model, which has been adopted in this paper. We then briefly discuss the problem of consistency preservation during reconfiguration. Next, dynamic reconfiguration is placed in an object middleware context, followed by a list of requirements that we have considered in the design of a Dynamic Reconfiguration Service for CORBA-based applications.

2.1 A model of dynamic reconfiguration

The purpose of dynamic reconfiguration is to allow a system to evolve incrementally from its current configuration to another configuration without being taken off-line. Dynamic reconfiguration should introduce as little impact as possible (ideally no impact at all) on the system execution.

In this paper, a system configuration is defined as a structure of software entities. Dynamic reconfiguration entails operations for the *replacement*, *migration*, *addition* and *removal* of these entities. Replacement means that an entity is replaced by another entity, where the new entity may run in another execution environment and have both functional and quality-of-service (QoS) properties that may differ from the old entity. Migration means that an entity is moved from one to another location, which may also imply a change in execution environment.

Figure 1 depicts the dynamic reconfiguration model based on [9, 10], which has been adopted in this paper.

In this model, the *reconfiguration design activities* comprise the specification of the changes and the specification of constraints that need to be preserved during reconfiguration. Changes are specified in terms of the above-mentioned *entities* and *operations* on these entities. *Reconfiguration constraints* are predicates on the reconfiguration process that restrict its execution, for example *"the reconfiguration process must last less that 10s"* or *"entity A should be available during the whole process"*.

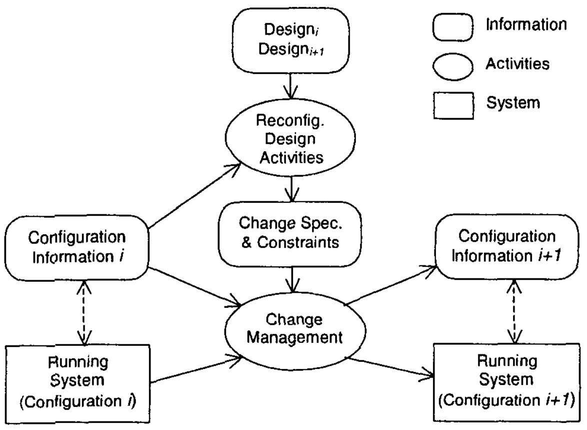
Design$_i$
Design$_{i+1}$
Information
Activities
System
Reconfig. Design Activities
Configuration Information i
Change Spec. & Constraints
Configuration Information $i+1$
Change Management
Running System (Configuration i)
Running System (Configuration $i+1$)

Figure 1 – Dynamic Reconfiguration model

The *change management activities* control the reconfiguration process, i.e., the transfer from the *current configuration* (system S_i) to a *resulting configuration* (system S_{i+1}), and use and produce *configuration information*. The configuration information defines the relationship between the system's entities.

Change Management requires functionality [9, 11, 13] providing guarantees that (i) specified changes are eventually applied to the system, (ii) a (useful) correct system is obtained, and, (iii) reconfiguration constraints are satisfied. Performing reconfiguration on a running system is an intrusive process [11]. Reconfiguration may imply, for example, interference with ongoing interactions between entities. One of the main issues of dynamic reconfiguration is consistency preservation.

2.2 Consistency preservation

Change management functionality must assure that system parts that interact with entities under reconfiguration do not fail because of reconfiguration, i.e., system consistency needs to be preserved. A system must be left in a "correct" state after reconfiguration. In order to support the notion of correctness of a distributed system, three aspects of consistency preservation requirements are identified [11]. A system is said to be correct if:

1. The system satisfies its *structural integrity* requirements,
2. The entities in the system are in *mutually consistent states*, and
3. The *application state invariants* hold.

A system S_{i+1} is said to be a *correct incremental evolution* of a system S_i, if S_{i+1} is correct and the behavior of the affected entities complies with the behavior expected by the unaffected system parts in case the reconfiguration had not taken place. With the term *affected entities* we denote the entities that are replaced, removed or migrated as a result of the reconfiguration process. Each aspect of the correctness notion is explained in the sequel (for a more elaborate discussion, see [1, 2])

2.2.1 Structural integrity. Structural integrity requirements constrain the structure of a system in terms of the relationships between entities and the ways in which these entities may be put together. Consider for example a CORBA system with various client objects that invoke an operation of a server object. If we replace the server object, then the following two conditions on the structural integrity of the system apply: (i) the client objects should be able to interact with the new object, i.e., in CORBA terms, the clients or client ORBs should obtain the object reference of the new object, and (ii) the new version of the server object must satisfy the interface definition of the original object.

2.2.2 Mutually consistent states. Entities are said to be in mutually consistent states, if each interaction between them, on completion, results in a transition between well defined and consistent states for the parts involved [11]. We assume here that *interactions* are the only means by which entities can affect each other's state.

For example, consider again a CORBA environment where an object A invokes an operation on an object B. A and B are said to be in mutually consistent states if A and B have the same assumptions on the result of the interactions between them. To be more specific, either both of them perceive that an invocation has occurred successfully, or both of them perceive that the invocation has failed. Suppose the change manager decides to replace B by B' after A initiated an operation invocation on B. For the resulting system to be in a consistent state, either (i) the invocation has to be aborted, A is informed and synchronization is maintained; or (ii) B receives the request, finishes processing it and sends the response, and then is replaced by B'; or, (iii) B is replaced by B', and B' has to honor the invocation, by processing the request and sending a response to A. In case none of these alternatives occur, A might be kept waiting for a response forever.

In order to guarantee that mutual consistency is preserved after reconfiguration, most approaches prescribe that reconfiguration can only start when the system is in the *reconfiguration-safe state* (or shortly *safe state*). If a system is in the safe state, each of its affected entities has a self-contained and stable state, and none of them is involved in interactions. Figure 2 shows a classification of reconfiguration approaches according to their choices on the preservation of mutual consistency.

We have studied mechanisms that drive the system under reconfiguration to a safe state, while avoiding interactions to be aborted. These mechanisms are designed to assure that interactions in progress are eventually completed, either before reconfiguration has started or after reconfiguration has finished. We propose a specific mechanism, which is discussed in Section 3.

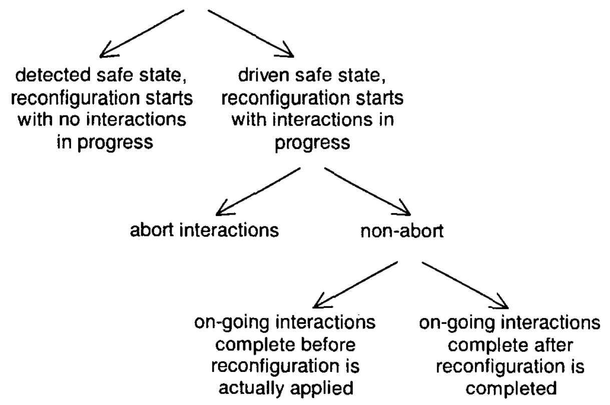
detected safe state, reconfiguration starts with no interactions in progress
driven safe state, reconfiguration starts with interactions in progress
abort interactions
non-abort
on-going interactions complete before reconfiguration is actually applied
on-going interactions complete after reconfiguration is completed

Figure 2 – Reconfiguration approaches and preservation of mutual consistency

2.2.3 Application-state invariants. *Application-state invariants* are predicates involving the state of (a subset of) the entities in a system. The preservation of safety and liveness properties of a system depends on the satisfaction of these invariants [11].

For example, consider an object that generates unique identifiers. An application-state invariant could be "all identifiers generated by the object are unique within the lifetime of the system". In order to preserve this invariant, the new version of the object must be initialized in a state that prevents it from generating identifiers that have been already used by the original object.

2.3 Dynamic reconfiguration support from object middleware

Object middleware is gaining wide acceptance as a generic software infrastructure for distributed computing systems. A growing number of applications are designed and implemented as a set of collaborating objects using object middleware, e.g., CORBA, as a software infrastructure that facilitates distribution transparencies. Most current approaches to dynamic reconfiguration attempt to consider distributed systems in general, and therefore do not exploit the particular characteristics of object middleware.

Object middleware offers interaction support to application objects, which may be deployed in different computer nodes. Middleware platforms are designed to provide several transparencies for the application designer, facilitating distributed application development. For example, a client/server programmer does not have to be concerned with network types, transport mechanisms, byte ordering, server locations, object activation, servant implementations, or target operating systems. The object middleware makes this all transparent. It provides a uniform interaction pattern, independent of the underlying node and network technologies.

Embedding reconfiguration functionality in an object middleware platform is a promising way to leverage this functionality with maximum transparency. We are particularly interested in a CORBA-based solution to dynamic reconfiguration. In a CORBA setting, application objects have either a client or target object role in an instance of interaction: a client object can use the service of a target object by issuing a request on the interface of the target object (a target object, in turn, may issue nested requests on other objects in order to process a pending request, thus playing the role of client in another instance of interaction). We consider the case where the entities that are the subject of dynamic reconfiguration, are target objects. Our objective is to develop a CORBA-based solution that is totally transparent to the clients of reconfigured objects.

2.4 Requirements for a Dynamic Reconfiguration Service

The following requirements have been considered in the design of the *Dynamic Reconfiguration Service* (DRS) for object middleware based systems.

- Correctness
  The service should provide facilities to allow a reconfiguration designer to obtain a correct incremental evolution of a system, as defined in Section 2.2.
- General suitability
  The DRS should be suitable for a broad set of applications and reconfigurations on these applications. It should be possible to reconfigure applications built from components-off-the-self, applications with multi-threaded and single-threaded execution models, re-entrant objects, stateless objects, stateful objects, etc. The service should not only allow the reconfiguration of one object, but also the reconfiguration of several objects atomically from the perspective of the unaffected system parts.
- Minimal impact on execution
  Dynamic reconfiguration is based on the idea that parts of the system remain available during a reconfiguration. Although disruption is unavoidable, the impact of the disruption should be minimized, as well as the duration of the effects of this disruption. The DRS should introduce minimal overhead during normal operation, and scale with respect to the number of clients.
- Maximum transparency
  The DRS should provide a *dynamic reconfiguration transparency*, which allows application developers not to be burdened with, or have expertise about, dynamic reconfiguration. For the client application developer it should be totally transparent. For the server side developer this is not a realistic requirement, but it should be as transparent as possible.
- Minimal impact on CORBA
  The design of the DRS should be CORBA compliant by using existing hooks to extend the functionality of a CORBA ORB. A design that requires major changes to existing ORBs is not likely to be very successful.

3 Dynamic Reconfiguration Approach

This section describes our proposed approach to the reconfiguration of systems based on object-middleware by addressing each of the aspects of correctness identified in Section 2.

3.1 Structural integrity

In the CORBA object model, *referential integrity* and *interface compatibility* are the main issues to be dealt with in order to preserve structural integrity.

Referential integrity becomes an issue whenever an object reference changes. An object reference is defined as a value that denotes a particular object, and is used by the middleware infrastructure to locate the object. Object references acquired by clients prior to reconfiguration may be invalidated due to reconfiguration. For example, in CORBA platforms, migration invalidates the IP address and port number contained in the IIOP profile of an IOR. If a reference points to an object that no longer exists, the established logical binding between a client and a target object is broken. In order to re-establish the binding after reconfiguration, we provide a logically central point of contact for clients to find the objects with invalidated object references.

In the CORBA object model, interfaces satisfy the Liskov substitution principle [14]. This means that if interface B is derived from interface A, then references to an object that supports interface A can be used to denote an object that supports interface B. To avoid that object replacements violate the object model, a new object must satisfy the old interface. This can be done either by implementing the old interface or by implementing an interface derived from it, e.g., by inheritance. If all clients of a reconfigurable object are also reconfigurable objects, it is possible to promote arbitrary changes to the interface by upgrading both clients and target objects atomically.

3.2 Mutual consistency

We propose an approach to drive the system to the safe state that *uses information obtained from the middleware platform at run-time* and *freezes system interactions on-demand*. This approach follows three stages:

1. Drive the system to the safe state by delaying interactions that would prevent the system from reaching the safe state;
2. Detect that the safe state has been reached; and
3. Apply reconfiguration;

In this approach, the system is said to be in the reconfiguration-safe state when each affected object (i) is not currently involved in interactions and (ii) will not be involved in interactions until after reconfiguration. This means that when the system is in a reconfiguration-safe state none of the affected objects are serving requests or waiting for outgoing requests to complete.

We distinguish objects in general as *active* and *reactive*. Reactive objects are objects that only initiate requests that are causally related to incoming requests. Active objects may initiate requests that do not depend on incoming requests, e.g., they may initiate requests as a result of the elapsing of a time-out.

An active object should have capabilities for going to a reactive state, in which it refrains from initiating requests that are not causally related to an incoming request. The implementation of the operation for forcing reactive behavior is a responsibility of the object developer. Once the set of affected system objects is defined, all active objects in the set are requested to exhibit reactive behavior.

3.2.1 Reaching the safe state. We guarantee the reachability of the safe state by interfering with the activities of the system. In a system under reconfiguration, we distinguish three sets of requests: (i) *requests that would prevent the affected objects from reaching the reconfiguration-safe state* (blocking set), (ii) *requests necessary for the system to reach the reconfiguration-safe state* ('laissez-passer' set) and (iii) *requests that do not involve any affected system object.*

In our approach, the middleware platform is responsible for selectively queuing requests that belong to the blocking set and for allowing requests in the 'laissez-passer' set to complete. This is done transparently for the application objects.

In the simple case of replacing *a single non re-entrant*[1] *object*, all requests issued to this object are queued by the middleware platform before they reach the object. In this way, new requests are prevented from being served before the reconfiguration, and the object gets the change to finish handling ongoing requests. When all ongoing requests have been treated, the system is in the safe state. Since all requests are guaranteed to finish within bounded time, the safe state is reachable within bounded time.

In the more complex cases of reconfiguring *multiple (re-entrant) objects simultaneously,* selective queuing of requests directed to affected objects is necessary. Requests issued by an affected object should get 'laissez-passer' status, since its requests have to be executed for the safe state to be reached. This implies that requests in invocation paths that contain at least one affected object should also be included in the 'laissez-passer' set. In particular, re-entrant requests initiated by affected objects are also included in the 'laissez-passer' set. All objects that could otherwise issue new 'laissez-passer' requests are set to exhibit reactive behavior, so that no new 'laissez-passer' requests are generated. At some point, the existing requests are treated, all affected objects are idle, and the system reaches the safe state.

In order to identify requests that belong to the 'laissez-passer' set, we use the propagation of implicit parameters along invocation paths. Every reconfigurable object in an invocation path adds its own identification to the request as an implicit parameter. Given a request and the set of affected objects, it is possible to determine if the request belongs to the 'laissez-passer' set by inspecting its implicit parameters. If at least one of the affected objects has been included in the request's implicit parameters, the request belongs to the 'laissez-passer' set.

3.2.2 Applying reconfiguration. When all affected objects inform the reconfiguration manager that they are idle, the reconfiguration process can proceed. The affected objects' state can be inspected and used to derive the state of the objects being introduced. The change designer may provide functions for state translation. Once new objects or new versions of objects have been installed, their state is properly modified. Queued requests and further new requests are redirected to the new version of an object.

3.3 State introspection

In [11] a scheme is proposed in which invalidated application invariants can be identified and re-established by the change designer with little assistance from the application developer. This scheme consists of requiring objects to provide general-purpose state access-methods that can be invoked by a third party to query or adjust the state of objects. These methods would be used to inspect and modify a selected subset of an object's internal state at runtime. In this scheme, the application designer decides on the particular subset of the objects' state that is exposed by these access methods. In general, objects should provide methods to inspect and modify state variables that control synchronization and computational behavior of the object.

One might argue that this scheme breaks encapsulation, since it allows external access to an object's internal state. Nevertheless, this form of introspection is unavoidable in certain cases, depending on the scope of reconfigurations considered.

[1] an object is denominated reentrant if it plays the role of server as a consequence of issuing a request to another object

4 Architectural Overview

This section describes our architecture for a Dynamic Reconfiguration Service. This architecture extends CORBA with a new common object service, and uses the approach described in Section 3.

4.1 Overview

The Dynamic Reconfiguration Service consists of a *Reconfiguration Manager*, a *Location Agent* and *Reconfiguration Agents*, see Figure 3.

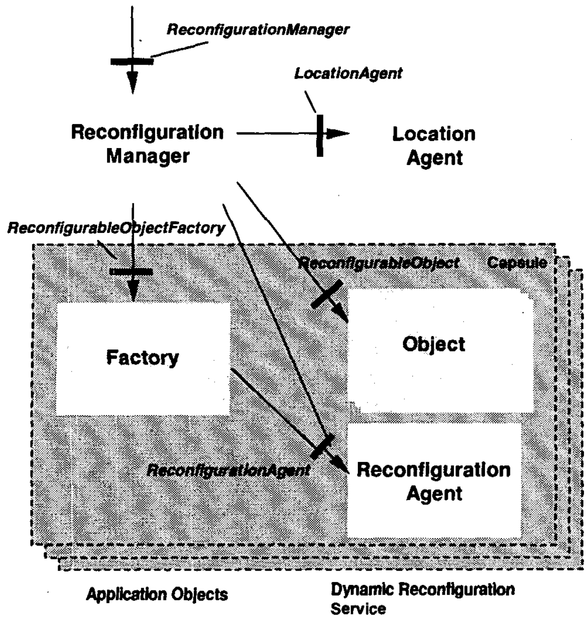

Figure 3 – Architectural Overview

The *Reconfiguration Manager* is the central component of the Dynamic Reconfiguration Service that interacts with the other components of the service. The reconfiguration designer accesses the service of the Dynamic Reconfiguration Service through the **ReconfigurationManager** interface, being able to create, replace, migrate and remove objects.

The Reconfiguration Manager delegates object creation and removal to Reconfigurable Object Factories, registers, re-registers and de-registers objects through interaction with the Location Agent and co-ordinates the Reconfiguration Agents to drive the system to a reconfiguration-safe state.

The *Location Agent* provides a registry for the location of reconfigurable objects. It translates a location-independent object reference to an object reference with the current location of a reconfigurable object. The Location Agent is typically co-located with an implementation repository [6], and uses the standardized CORBA request forwarding mechanism [14].

A *Reconfiguration Agent* is present in every capsule [8] where reconfigurable objects may be located. A Reconfiguration Agent is responsible for restricting the behavior of an affected object during reconfiguration through filtering of requests.

Reconfigurable Object Factories implement the Factory design pattern, creating *Reconfigurable Objects* on behalf of the Reconfiguration Manager. Factories shield the Dynamic Reconfiguration Service from the specific support to object deployment offered by different languages, operating systems or virtual machines, such as e.g., DLLs, the Java class loader and interpreted languages. Reconfigurable Object Factories and Reconfigurable Objects are application specific and are supplied by the application developer.

4.2 Reconfigurable Object Creation

Figure 4 shows the creation of an object in the architecture. The Reconfiguration Manager delegates the creation to a local Reconfigurable Object Factory (2), which creates the object (3) and registers it with the Reconfiguration Agent responsible for the capsule where the object lives (4). In the sequence, the Reconfiguration Manager registers the recently created object with the Location Agent (5), and returns the object reference to the client that requested the object creation (6).

Both the Reconfiguration Manager and the local factories implement the **GenericFactory** interface (as of Fault-Tolerant CORBA [15]).

4.3 Reconfigurable Object Removal

The Reconfiguration Manager delegates object removal to the Reconfigurable Object Factory responsible for the object being removed, and de-registers the object with the Location Agent.

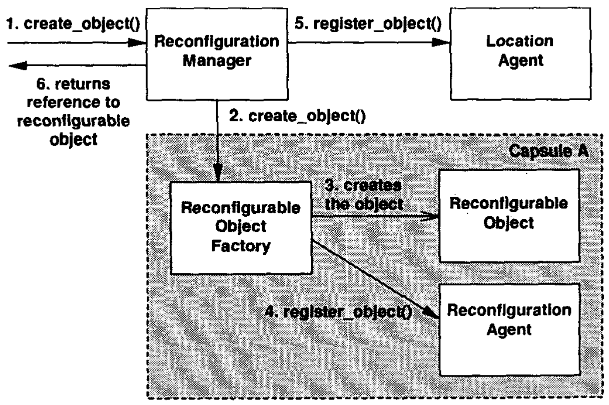

Figure 4 – Object Creation

4.4 Reconfigurable Object Replacement

Figure 5 shows the replacement of an object in the architecture. Firstly, the Reconfiguration Manager delegates the creation of the new version of the object to a local Reconfigurable Object Factory (2), as shown in the example above. In the sequence, the Reconfiguration Manager notifies the affected reconfigurable object and its Reconfiguration Agent of the start of the reconfiguration (5, 6). The Reconfiguration Agent restricts the behavior of the affected object, and notifies the Reconfiguration Manager when the object is ready for reconfiguration (7). The state-transfer is conducted (8, 9), the new location of the object is registered with the Location Agent (10), and the previous version of the object removed through interaction with the local factory (11). Please note that in this figure we abstract from optional state translation.

In the case of replacement of several objects simultaneously, the safe-state is reached when all reconfigurable objects affected notify the Reconfiguration Manager. As a multiple-object replacement is considered a single atomic action from the perspective of the clients of the affected objects, the Location Agent updates their location atomically.

Reconfigurable objects should implement the **ReconfigurableObject** interface, which consists of state-access operations and a 'passivate' operation to be invoked by the reconfiguration manager to notify of the beginning of the reconfiguration. In response to this operation, the object should exhibit a re-active behavior, as described in Section 3.2.

The service is completely transparent for client applications, which will manipulate object references and issue requests to reconfigurable objects in the ways prescribed in the CORBA object model. A client application issues requests that are handled by a client-side ORB. The client-side ORB is responsible for sending requests to the server-side ORB which, under normal operation, delivers the request to the target object. During reconfiguration, requests may be queued by the middleware. In this case, the server-side ORB informs the client-side ORB of the reconfiguration. At the end of the reconfiguration, the Reconfiguration Manager notifies the client-side ORB, which re-issues the request with the new target registered in the Location Agent.

One might believe that the selective queuing of requests interferes with ordering guarantees provided by the middleware infrastructure. Nevertheless, in the CORBA object model, the order in which a client issues requests does not imply the order in which a server processes the requests. In addition, the order in which replies reach a client does not imply the order in which the server processed the requests.

4.5 Reconfigurable Object Migration

Object migration is treated as an object replacement where the factory of the new version of the object is located in the destination capsule.

5 Implementation

The implementation described in this section is based on the use of portable interceptors [16] to extend the functionality of the ORB. Portable interceptors allows the extension of the ORB through a limited request reflection mechanism in an ORB-independent manner. It allows a service to reify requests in specific interception points.

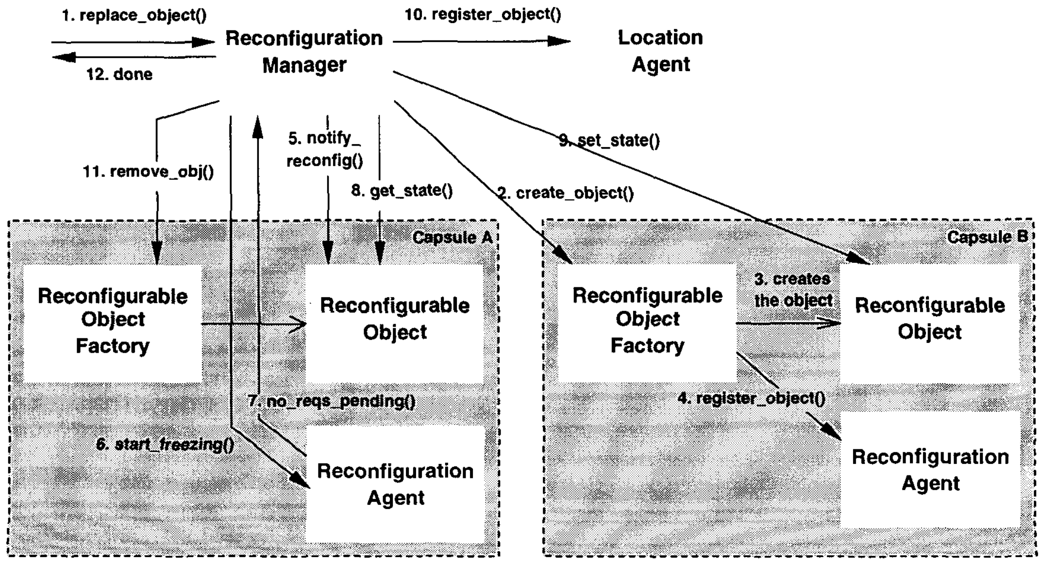

Figure 5 – Object Replacement

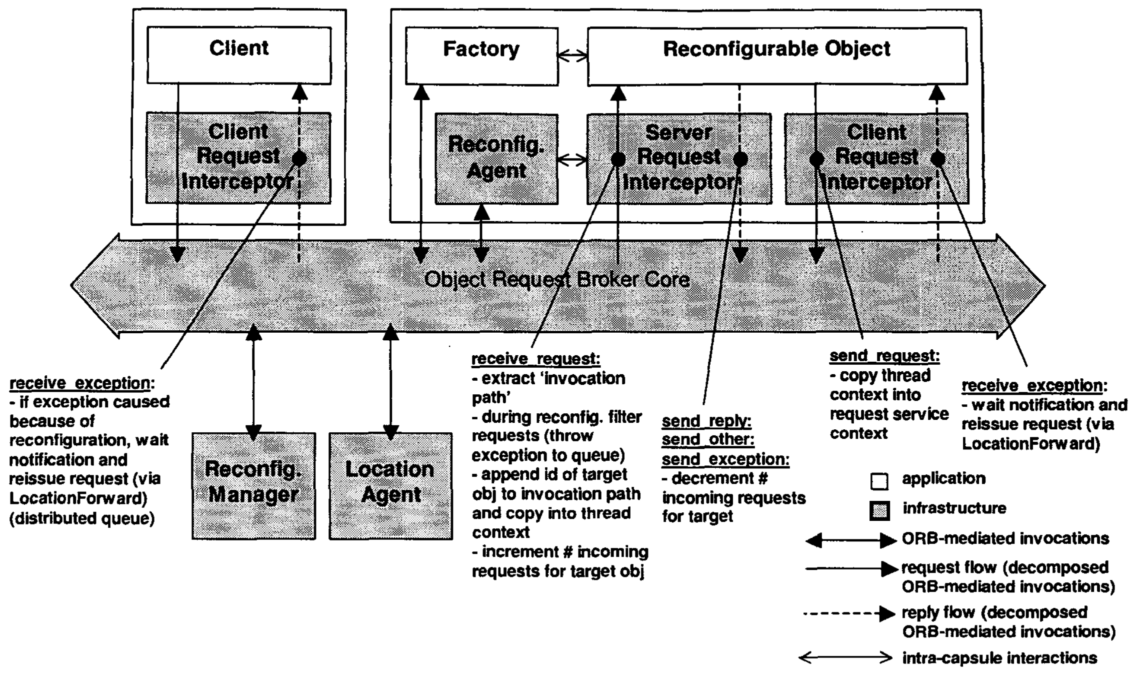
Client
Client Request Interceptor
Factory
Reconfigurable Object
Reconfig. Agent
Server Request Interceptor
Client Request Interceptor
Object Request Broker Core
receive_exception:
- if exception caused because of reconfiguration, wait notification and reissue request (via LocationForward) (distributed queue)
Reconfig. Manager
Location Agent
receive_request:
- extract 'invocation path'
- during reconfig. filter requests (throw exception to queue)
- append id of target obj to invocation path and copy into thread context
- increment # incoming requests for target obj
send_reply:
send_other:
send_exception:
- decrement # incoming requests for target
send_request:
- copy thread context into request service context
receive_exception:
- wait notification and reissue request (via LocationForward)
application
infrastructure
ORB-mediated invocations
request flow (decomposed ORB-mediated invocations)
reply flow (decomposed ORB-mediated invocations)
intra-capsule interactions

Figure 6 – Elements of the implementation and request reification points

Figure 6 depicts an overview of the implementation with a brief description of the actions undertaken in client- and server-side request interception points.

Before a reconfigurable object receives a request, the request is reified in the **receive_request** interception point, and the service context propagated with the request is extracted. A service context is an implicit parameter used by CORBA services to propagate information along with a request. For the DRS, it contains the list of reconfigurable objects that depend on the execution of the request to become idle. The list of reconfigurable objects is appended with the identification of the request's target object and the appended list is copied into the **ReconfigurationCurrent** local object. The **ReconfigurationCurrent** object provides access to an implicit per-thread context, and in this way the thread is associated with the reconfigurable object. When a nested outgoing invocation is initiated, the list is copied from the **ReconfigurationCurrent** object into the request service context.

For outgoing invocations that do not depend on incoming invocations, the application calls a method in the **ReconfigurationCurrent** object in order to associate the current thread with the originating reconfigurable object. The application may spawn a new thread as part of the processing of a request, in which case the application is responsible for transferring information from the **ReconfigurationCurrent** object of the thread treating the request to the **ReconfigurationCurrent** object of spawned threads.

During the first stage of the reconfiguration process, server request interceptors inspect the propagated service context. If any of the affected objects is listed in the service context, the request should be allowed to complete, so that all affected objects can progress to the idle state. If no affected objects are listed, an exception is raised. This exception is intercepted in the client-side client request interceptors, which re-issues the request transparently when the reconfiguration is over. The client application is not at any moment aware of the reconfiguration, potentially observing an increase in the response time of invocations delayed.

6 Evaluation

A prototype of the Dynamic Reconfiguration Service has been implemented to validate the architecture and the mechanisms proposed. The prototype has been developed in Java, using ORBacus 4.0.4 [19].

The prototype has been successfully tested for applications with multiple multithreaded objects, including nested and re-entrant invocations. Furthermore, we have conducted some performance tests on the prototype. In the following sections we present the results obtained from these tests: an estimation of the overhead introduced by the Dynamic Reconfiguration Service during normal operation; and an estimation of the impact of reconfiguration on execution.

We complete the evaluation by comparing our design with related work.

6.1 Overhead during normal operation

In order to assess the overhead of the reconfiguration service during normal operation, i.e., with no on-going reconfiguration, we have set-up a performance test with a client and a server object, in different hosts of a local area network. Since a large part of the overhead introduced by the dynamic reconfiguration service is incurred by the implementation of portable interceptors, we have considered three test cases:

1. client and server with *no portable interceptors*,
2. client and server with *minimal portable interceptors*, i.e., interceptors with placeholders for interception points, but no code, and
3. client and server with *the dynamic reconfiguration service portable interceptors*.

We measure the overhead during normal operation by measuring the response time R observed at the client. In this estimation, we simplify R to consist of the delay introduced by the middleware platform to mediate the invocation $\Delta_{middleware}$ added with the delay introduced by the execution of the application code, $\Delta_{application}$. For our tests, the server object provided an operation with no application code, thus $\Delta_{application}=0$, and we have:

$$R = \Delta_{middleware} + \Delta_{application} = \Delta_{middleware}$$

For test case 1, $\Delta_{middleware}$ is the delay introduced by the plain middleware platform (i.e., the middleware platform without extensions), $\Delta_{plainorb}$:

$$\Delta_{middleware} = \Delta_{plainorb}$$

For test case 2, $\Delta_{middleware}$ can be seen as composed of the delay introduced by the plain middleware platform and the delay introduced by the implementation of portable interceptors, $\Delta_{interceptors}$:

$$\Delta_{middleware} = \Delta_{plainorb} + \Delta_{interceptors}$$

For test case 3, $\Delta_{middleware}$ can be seen as composed of the delay introduced by the plain middleware platform, the delay introduced by the implementation of portable interceptors and the delay introduced by the dynamic reconfiguration service portable interceptors, Δ_{drs}.

$$\Delta_{middleware} = \Delta_{plainorb} + \Delta_{interceptors} + \Delta_{drs}$$

Four batches of 10^4 invocations have been executed for each of these three distinct cases, with different sizes of parameters and result values. The results obtained are summarized in Table 1 (values averaged from 4 x 10^4 invocations). Figure 7 shows a graphical representation of the results.

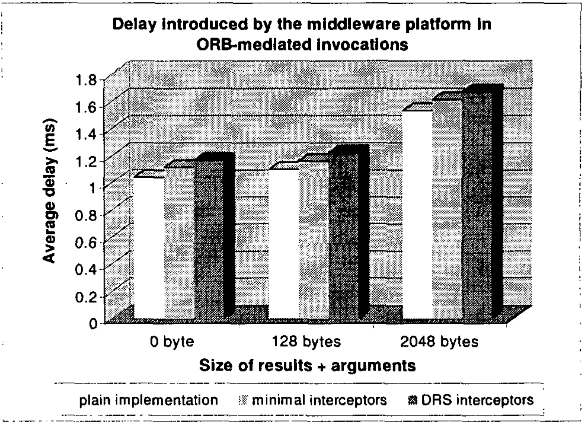

Figure 7 – Normal and increased response times

Summarizing, the increase in $\Delta_{middleware}$ incurred by the introduction of the dynamic reconfiguration service lies in the range 0.13 ± 0.01 ms. In the worst case, with no parameters and no result value, the dynamic reconfiguration service causes an increase of less than 12.5% in the delay introduced in normal ORB-mediated invocations. Typically, the servant will take some time to process the request, lowering the relative increase in invocation response time considerably. Therefore, for most application scenarios, we consider the overhead of the reconfiguration service acceptable. In these experiments we have not considered the overhead incurred by the use of the Location Agent, since this overhead is limited to the first operation invocation and after a reconfiguration.

From the results obtained, we can also conclude that more than half of the delay added by the dynamic reconfiguration service is caused by the implementation of portable interceptors. Since the implementation of portable interceptors is ORB dependent, these experiments should be repeated for other ORB implementations.

Table 1 – Delay introduced by the middleware platform in invocation mediation

		Minimal portable interceptors		Dynamic reconfiguration service			
Size of arguments + result value	$\Delta_{plainorb}$ (ms)	$\Delta_{interceptors}$ (ms)	Increase from $\Delta_{plainorb}$	Δ_{drs} (ms)	Increase from $\Delta_{plainorb}$ $+\Delta_{interceptors}$	$\Delta_{interceptors}$ + Δ_{drs} (ms)	Increase from $\Delta_{plainorb}$
0 bytes	1.0388	0.0771	7.4%	0.0518	4.6%	0.1289	12.4%
128 bytes	1.0999	0.0625	5.7%	0.0641	5.4%	0.1266	11.5%
2 Kbytes	1.5305	0.0834	5.5%	0.0555	3.4%	0.1389	9.1%

6.2 Impact on execution during reconfiguration

Clients of an affected object observe an increase in the response time of operations invoked during a replacement. This increase only applies for invocations that reach the target object *after the beginning of the reconfiguration* and *before the end of the reconfiguration*.

The increase in response time during reconfiguration is highly dependent on the application. It is upper-bounded by the duration of the longest pending invocation in the set of affected objects at the moment the reconfiguration starts plus a fixed delay introduced by the reconfiguration service for co-ordination overhead. For active objects, the amount of time taken for the object to exhibit a reactive behavior should also be considered in the calculation of the upper bound of the increase in response time.

The fixed delay introduced by the reconfiguration service can be seen as a lower bound for the increase in response time during reconfiguration. According to an experiment conducted with the replacement of one single object, this delay is approximately 530 ms. From this value, 320 ms are related to marshalling and de-marshalling of service contexts. We see opportunities for optimizations that should reduce these values.

The experiments should be repeated for different ORB implementations to reach more conclusive results. Further tests should consider the effects of reconfiguration on the performance of the new object right after the reconfiguration, as queued requests are directed to it.

6.3 Related work

In Bidan et. al.'s approach [3], the implementation of a reconfiguration service in CORBA is considered. As is the case for our approach, a reconfigurable entity is a CORBA object. In this approach, the reconfiguration infrastructure maintains a representation of the configuration of the system, through a directed graph of objects connected through links. Objects *A* and *B* are said to be linked if *A* can invoke an operation on target object *B*. In the approach, all client applications and target objects must implement a passivate operation to block the initiation of requests in a specific outgoing link. The algorithm guarantees the reachability of an idle state by sending passivate messages to all the clients of an object and then to the object itself.

Unlike our approach, Bidan et. al.'s approach is not suitable for a system with re-entrant invocations. Therefore, in this approach, an object that has initiated an invocation cannot play the role of server for some consequent invocation.

Furthermore, the approach does not permit multiple object replacements simultaneously. This limits the application of the approach when a group of objects has to be substituted atomically. It is common to have sets of related objects, where a change to one object may require changes to other objects that depend on its behavior or other characteristics [18].

Compared to our DRS, the functionality for dynamic reconfiguration is much more at the application layer, requiring not only server objects but also client applications to incorporate support for reconfiguration. Our approach exploits the particular characteristics of object-middleware to provide more transparency for the application designer, facilitating distributed application development.

In [20], another approach to dynamic reconfiguration of CORBA-based systems is presented. The approach does not address consistency issues explicitly and is heavily based on the use of an interpreted language for object implementation.

Our approach only interferes with those parts of the system that actually interact with the set of affected objects during reconfiguration, contrary to approaches [3, 9, 11, 25] that block all potential system activities that may prevent the system from reaching the safe state. For a more extensive comparison of our approach with other non-middleware-specific dynamic reconfiguration approaches [9, 11, 25], see [1].

This paper focuses on reconfiguration of non-redundant objects. Approaches for redundant objects promote object replacements by temporarily executing both old and new versions of an object simultaneously. These approaches, such as the one adopted by [11], are limited to object replacements where the new version of an object has the same externally visible semantic properties when compared to the old object.

7 Conclusions

In this paper, we have presented an approach to dynamic reconfiguration of object middleware based systems that allows modifying a running system with maximum transparency for both client and server side developers. The proposed approach can be used in systems with large and changing numbers of clients and objects and is suitable for a broad set of applications and reconfigurations.

The approach has been used in the design of a Dynamic Reconfiguration Service for CORBA, which has been validated through the implementation of a prototype. Some preliminary test results have been presented to assess the overhead introduced by the DRS during normal operation. These results indicate that the overhead is quite minimal, and is acceptable for most application domains. We also presented preliminary measurements on the impact on the execution of a system during reconfiguration, which is also acceptable. Our approach only interferes with those parts of the system that actually interact with the set of affected objects during reconfiguration, allowing the rest of the system to execute normally.

The prototype uses portable interceptors (a standardized ORB extension mechanism) to realize request reflection and instrument the middleware platform to obtain configuration information at runtime. This avoids requiring the application developer or integrator to provide extensive descriptions of the system and its objects. By using portable interceptors, we are able to freeze system interactions on demand.

We have submitted the approach presented in this paper in Lucent Technologies' response [23] to the Request For Information (RFI) on Online Upgrades [17] issued by the OMG, hoping that this approach becomes incorporated in a forthcoming CORBA standard.

References

[1] J. P. A. Almeida, M. Wegdam, L. Ferreira Pires, M. van Sinderen. An approach to dynamic reconfiguration of distributed systems based on object-middleware, to appear in *Proceedings of the19th Brazilian Symposium on Computer Networks (SBRC 2001)*, Santa Catarina, Brazil, May 2001.

[2] J. P. A. Almeida, Dynamic Reconfiguration of Object-Middleware-based Distributed Systems. M. Sc. thesis, University of Twente, The Netherlands, June 2001.

[3] C. Bidan, V. Issarny, T. Saridakis, A. Zarras. A dynamic reconfiguration service for CORBA, in *Proc. IEEE International Conference on Configurable Distributed Systems*, May 1998.

[4] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and Practice, *IEE Software Engineering Journal*, vol 8, no 2, March 1993.

[5] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of the 12th Brazilian Symposium on Computer Networks,* 1994.

[6] M. Henning. Binding, migration, and scalability in CORBA. *Communications of the ACM* 41(10), October 1998.

[7] C. Hofmeister, E. White, J. Purtilo. Surgeon: a package for dynamically reconfigurable distributed applications, in *Proceedings of the IEEE International Conference on Configurable Distributed Systems*, March 1992.

[8] ITU-T / ISO. *Open Distributed Processing Reference Model. Part 1 – Overview*, ITU-T X.901 | ISO/IEC10746-1.

[9] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering* 11(4), pp. 424-436, April 1985.

[10] J. Kramer and J. Magee. The evolving philosophers' problem: dynamic change management. *IEEE Transactions on Software Engineering* 16(11), pp. 1293-1306, November 1990.

[11] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems.* Ph.D. thesis, Imperial College, London, March 1999.

[12] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury and V. Kalogeraki. Eternal: Fault Tolerant and Live Upgrades for Distributed Object Systems, in *Proceedings of the IEEE Information Survivability Conference*, Hilton Head, SC, January 2000, pp. 184-196.

[13] P. Oreizy, N. Medvidovic, R. Taylor. Architecture-based runtime software evolution, in *Proceedings of the International Conference on Software Engineering*, April 1998.

[14] Object Management Group, *The Common Object Request Broker: Architecture and specification*, Revision 2.4.1, formal/00-11-07, November 2000.

[15] Object Management Group. *Fault tolerant CORBA specification, V1.0*, ptc/00-04-04, April 2000.

[16] Object Management Group. *Interceptors FTF published draft of CORBA core and services chapters*, ptc/00-03-03, March 2000.

[17] Object Management Group. *Online updates RFI*, orbos/00-09-15, September 2000.

[18] Object Management Group. *Online updates RFP draft*, ab/00-03-06, March 2000.

[19] Object Oriented Concepts, Inc. http://www.ooc.com.

[20] N. L. R. Rodriguez and R. Ierusamlimschy. Dynamic Reconfiguration of CORBA-based applications, in *Proceeding of the SOFSEM'99: 26th Conference on Current Trends in Theory and Practice of Informatics,* LNCS 1725, Springer-Verlag, Berlin, pp. 95-111, 1999.

[21] D.C. Schmidt and S. Vinoski. Object interconnections. Object adapters: concepts and terminology. *SIGS C++ Report*, October 1997.

[22] C. Szyperski. *Component software – Beyond object-oriented programming*, ACM Press, New York, 1997.

[23] M. Wegdam and J. P. A. Almeida. *Lucent response to OMG ORBOS RFI on online updates*, orbos/01-01-01, January 2001.

[24] M. Wegdam, D.-J. Plas, A. van Halteren, B. Nieuwenhuis. Using message reflection in a management architecture for CORBA, *In Proceedings of the 11th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2000)*, Austin, Texas, USA, December 2000.

[25] M. A. Wermelinger. *Specification of software architecture reconfiguration.* Ph.D. thesis, Universidade Nova de Lisboa, September 1999.