Dynamic Reconfiguration of Object-Middleware-based Distributed Systems

João Paulo Andrade Almeida

Thesis for a Master of Science degree in Telematics from the University of Twente, Enschede, The Netherlands

Graduation Committee:

prof. dr. ir. L. J. M. Nieuwenhuis

drs. M. Wegdam

Dr. L. Ferreira Pires

dr. ir. M. J. van Sinderen

Enschede, The Netherlands

June, 2001

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.

This thesis addresses the dynamic reconfiguration of distributed applications that run on top of an object-middleware infrastructure. In this context, a system configuration is defined as a structure of software entities at application-level. Dynamic reconfiguration entails operations for the replacement, migration, creation and removal of these entities at run-time.

This thesis proposes an approach to dynamic reconfiguration for applications built using object-middleware and realizes this approach with an architecture and design of middleware support for reconfiguring CORBA-based distributed systems at run-time.

Table of contents

Preface

This thesis describes the results of a Master of Science assignment at Lucent Technologies. This assignment has been carried out from September 2000 to June 2001 at the Bell Labs Advanced Technologies location in Twente, in co-operation with the Telematics Systems & Services (TSS) Group of the Centre for Telematics and Information Technology (CTIT) of the University of Twente.

Parts of the work presented in this thesis have been described in two scientific papers [1, 2] and in Lucent Technologies' response [32] to a Request For Information (RFI) on Online Upgrades issued by the Object Management Group (OMG).

I would like to thank my supervisors Maarten Wegdam (Lucent Technologies), Luís Ferreira Pires, Marten van Sinderen and Bart Nieuwenhuis (University of Twente). They have given most useful support for the development of this thesis and have co-authored the two papers that have been written in the development of this work.

I would also like to thank the colleagues at Lucent and at the university for the pleasant working environment, and the many friends that have made this two-year stay in Enschede so *gezellig*.

Enschede, 5th June 2001.

João Paulo Andrade Almeida.

1 Introduction

This chapter presents the motivation, the objectives, and the structure of this thesis. It identifies the relevance of dynamic reconfiguration in current distributed computing systems and draws attention to the support of dynamic reconfiguration in object-middleware.

This chapter is further structured as follows: Section 1.1 gives motivation for dynamic reconfiguration of object-middleware-based systems, Section 1.2 states the objectives of this thesis and Section 1.3 outlines the structure of this thesis by presenting an overview of the chapters.

1.1 Motivation

Distributed computing systems have been in widespread use for several years in commercial, industrial and research environments. These systems are currently deployed in mission-critical and long-running applications, such as telecommunication switches and e-commerce solutions.

The reliance on distributed computing systems imposes restrictions on the possibility of restarting them or taking them off-line. It is usually not acceptable, e.g., for economical or safety reasons, to cause major interruptions in the service these systems provide. These systems have high availability, adaptability and maintainability requirements, and, in order to remain useful, they have to cope with advances in technology, modifications of their operating environment and ever-changing human needs [13].

The aim of dynamic reconfiguration [3, 4, 5, 7, 9, 10, 13, 15, 27, 34] is to allow a system to evolve incrementally from a configuration to another at run-time [9], as opposed to design-time, while introducing little (or ideally no) impact on the system's execution. In this way, systems do not have to be taken off-line, rebooted or restarted to accommodate changes.

New generation distributed applications often consist of co-operating objects that run on a variety of hardware and operating systems and make use of object-middleware technology, such as CORBA [16], Java RMI [29] and DCOM [12].

Object-middleware facilitates the development of distributed applications by providing distribution transparencies to the application developers, and shielding the developer from the heterogeneity of operating systems and communications systems.

Object-middleware offers a widely accepted approach for the provision of flexible, distributed computing environments. As such, there are many systems that would profit from dynamic reconfiguration facilities for object-middleware. The development of these systems would be facilitated through the inclusion of (transparent) reconfiguration support in the middleware platform. Embedding reconfiguration functionality in a middleware platform is a promising way to leverage this functionality with maximum transparency to the application developer.

1.2 Objectives

The problem of reconfiguration may involve all levels (from hardware to the application) and all phases of software development (requirements to implementation) [34].

This thesis addresses the reconfiguration of distributed applications that run on top of an object-middleware infrastructure. In this context, a system configuration is defined as a structure of software entities at application-level. Dynamic reconfiguration entails operations for the replacement, migration, creation and removal of these entities at run-time.

More specifically, this thesis focuses on the support of dynamic reconfiguration in middleware. Its main objectives are (1) to propose an approach to dynamic reconfiguration for applications built using object-middleware and (2) to realize this approach with an architecture and design of middleware support for reconfiguring CORBA-based distributed systems at run-time. Dynamic reconfiguration of the middleware infrastructure itself is outside the scope of this thesis.

The dynamic reconfiguration approach should be applicable to a broad class of applications and should support a broad range of reconfiguration operations. It should address relevant consistency issues that may arise during reconfiguration and minimize the impact of reconfiguration on application execution. The requirements imposed on the development process should be minimized.

The support for dynamic reconfiguration should be integrated in the middleware infrastructure, and it should be perceived by the users of the infrastructure as a service that provides reconfiguration transparency. The dynamic reconfiguration service should minimize the requirements on the application layer and maximize the transparency for the involved CORBA objects.

This thesis does not focus on the design activities that are undertaken to obtain a new version of a system or of its constituting parts. Neither does it intend to provide tools to support the description, testing and validation of reconfiguration. It rather aims at the activities to be undertaken in order to apply a set of well-defined reconfiguration operations to a running application.

1.3 Structure

The sequence of the chapters in this thesis reflects the order in which these issues have been dealt with throughout the research process. This thesis is structured as follows:

- Chapter 2 provides a brief introduction to the distributed-software model adopted in this thesis, and describes the particular middleware technology chosen for the architecture, design and implementation of our dynamic reconfiguration service.

- Chapter 3 is the result of a literature study in the area of dynamic reconfiguration of distributed systems, and introduces some important terminology, definitions and concepts used to discuss about dynamic reconfiguration. Furthermore, it presents some of the current approaches to dynamic reconfiguration and compares these approaches.

- Chapter 4 proposes a new dynamic reconfiguration approach that exploits particularities of object-middleware distributed systems. This approach has been used in the definition of our dynamic reconfiguration service.

- Chapter 5 describes the architecture of the proposed dynamic reconfiguration service, presenting both the application developer's view and the change designer's view.

- Chapter 6 describes the design and implementation of the dynamic reconfiguration service realized in a prototype. Performance measurements obtained with the prototype are also presented in the chapter.

- Chapter 7 illustrates the usage of the dynamic reconfiguration service through different validation scenarios.

- Finally, Chapter 8 presents important conclusions and future directions.

2 Object Middleware

This chapter introduces the object model adopted in this thesis, and presents a particular object-middleware architecture that uses this model. The dynamic reconfiguration approach proposed in this thesis assumes this model, and the dynamic reconfiguration service populates the described middleware architecture.

This chapter is further structured as follows: Section 2.1 introduces object-middleware and its relevance to supporting distributed applications, Section 2.2 presents the role of the Object Management Group (OMG) in object middleware, Section 2.3 describes OMG's object model and Section 2.4 describes OMG's Object Management Architecture (OMA) and the components that populate it. This chapter provides a brief introduction to OMG's object model and the OMA. For a more detailed description, refer to OMG's standards [16, 22].

2.1 The Role of 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, such as CORBA [16], Java RMI [29] and DCOM [12].

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

2.2 OMG's Specifications

The Object Management Group (OMG) was founded in 1989 with the objective of providing standards in distributed object computing. Currently, the OMG has more than 800 members, including system vendors, users and academic institutions.

The first key specifications produced by the OMG – the Object Management Architecture (OMA) [22] and its core, the Common Object Request Broker Architecture specification [16] – provide an architectural framework that accommodates a wide variety of distributed systems. These specifications supply a set of general-purpose abstractions and concrete services needed to realize practical solutions for the problems associated with distributed heterogeneous computing.

The OMA provides the conceptual infrastructure upon which supporting specifications are based. It uses an object model [16, 22] to define the entities that comprise a distributed system, how they interoperate, and how designers can specify their behavior. This object model is described in Section 2.3, using the terminology introduced in [28].

The OMA is populated by concrete elements which are supplied by a different number of vendors and comply with OMG's specifications. Some of these elements are described in Section 2.4.

2.3 The Object Model

According to the CORBA object model, distributed applications consist of a collection of objects, which are possibly geographically distributed, i.e., they may be deployed in different computer nodes.

From an abstract point of view, an object is an identifiable, encapsulated entity that provides services to other objects through its interface. The interface of an object shields the other objects from the internal characteristics of this object, like its internal data representation, algorithms or executable code. An object plays the role of a *client object* when it uses the service of another object, which is called a *target object*.

A client object can use the service of a target object by issuing a request on the interface of that object. An interface defines the set of operations of a target object that a client object is allowed to invoke by issuing a request. An interface definition provides the syntax for invoking operations on this interface, like the parameters required by the operations and their possible results and exceptions. Interfaces can be related by inheritance, in which case a derived interface is constructed by incremental modification of other interfaces.

An instance of interaction between objects consists of the sequence of activities starting when a client object issues a request for a target object and ending when a response to the request is delivered to the client object. Since the client and the target objects are possibly distributed, an object middleware infrastructure is used to locate the target object, forward information on the request to this target object and forward a response back to the client object. Furthermore, in order to interact with a target object, a client object is only allowed to refer to the interface of the target object. In this way the middleware infrastructure supports not only

distribution transparency, but it also enables the target object to be implemented in many alternative programming languages.

Figure 1 shows the interaction between a client and a target object, and the role of the middleware infrastructure in supporting this interaction.

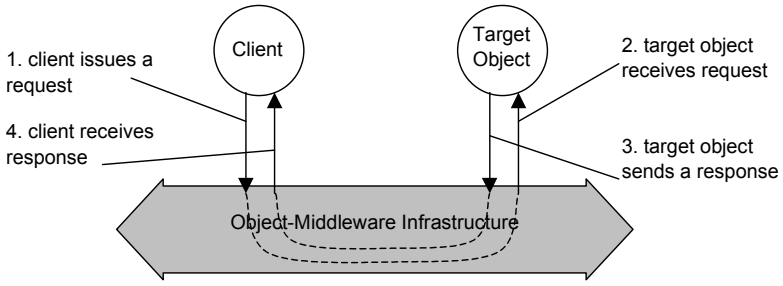


Figure 1 – Example of request

A target object is implemented by a programming language construct that carries out the computational activities necessary to process a request. In case a target object should be capable of handling multiple requests simultaneously, it should be implemented using a multi-threading execution model; otherwise a single-threaded model suffices.

A target object may issue requests on other objects in order to process a pending request. In this case, this object plays the role of a client in another instance of interaction. A *nested request* is a request issued by an object and that has to be processed in order to allow the processing of a pending request. An *invocation path* is the path traversed by a sequence of nested requests. A *re-entrant invocation* is a nested request issued on an object that issued a previous request of the same invocation path. Figure 2 illustrates nested and re-entrant invocations.

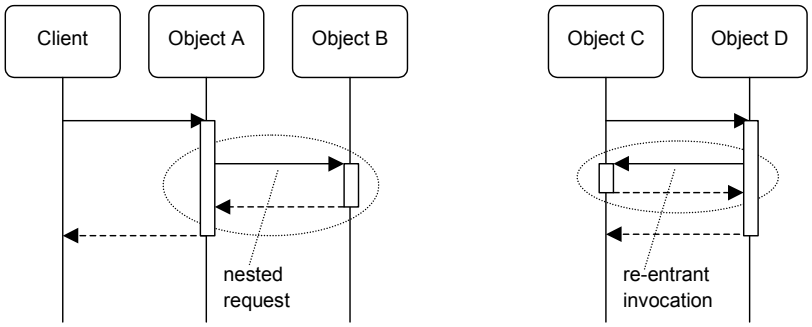


Figure 2 – Nested request and re-entrant invocations

In Chapter 4, we describe our dynamic reconfiguration approach, which assumes the object model presented in this section.

2.4 The Object Management Architecture

The OMA is composed by three interface categories and an Object Request Broker (ORB). These interface categories are:

- *Application Interfaces* – application-specific interfaces that fall outside of the OMG standardization;

- *Domain Interfaces* – domain-specific interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecom, Electronic Commerce, and Transportation;

- *Object Services* – domain-independent interfaces for general services which are widely available and are likely to be useful in any distributed object application. Examples of object services are the OMG Naming Service, the OMG Event Service, and the OMG Trading Service.

An overview of the architecture is depicted in Figure 3.

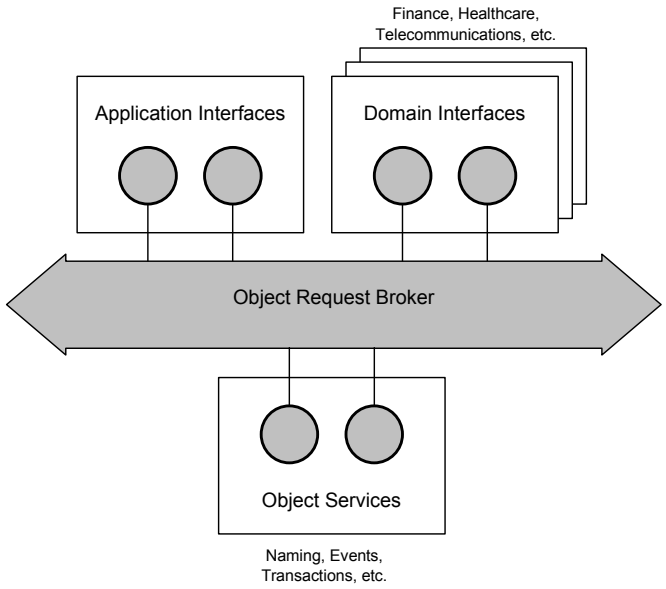


Figure 3 – Interface categories and the ORB

The Object Request Broker (ORB) provides the basic mechanism by which objects transparently make requests and receive responses. A client does not have to know the mechanisms used to communicate with or activate an object, how the object is implemented, nor where the object is located. The ORB thus forms the foundation for building applications constructed from distributed objects and for achieving interoperability between applications in both homogeneous and heterogeneous environments. The Common Object Request Broker Architecture (CORBA) [16] defines the programming interfaces to the ORB.

The ORB Core is the part of the ORB that provides the basic representation of objects and the communication of requests. It supports the minimum functionality to enable a client to invoke an operation on a target object. The ORB may provide additional features via ORB Services, which in some ORB implementations are layered as internal services over the core, or in other cases are incorporated directly into the kernel of the ORB implementation.

ORB Services differ from Object Services in that they run below the application and are invoked transparently and implicitly in the course of application-level interactions. However, many ORB Services include interfaces which correspond to conventional Object Services in that they are invoked explicitly by the application. Security is an example of service with both ORB Service functionality and Object Service interfaces, the ORB functionality being that associated with transparently authenticating messages and controlling access to

objects, while the necessary administration and management functions resemble conventional Object Services [16].

In Chapter 5, we introduce our dynamic reconfiguration service, which provides reconfiguration transparency for CORBA applications with ORB Service functionality and Object Service interfaces.

3 Dynamic Reconfiguration

This chapter introduces some important terminology, definitions and concepts used to discuss dynamic reconfiguration, and presents and compares some of the current approaches to dynamic reconfiguration. This chapter also presents a model for dynamic reconfiguration, and discusses the importance of consistency preservation. Furthermore, it considers the consequences of dynamic reconfiguration on application execution.

This chapter is further structured as follows: Section 3.1 presents the definition of dynamic reconfiguration and the dynamic reconfiguration model we adopted in this work, Section 3.2 discusses the activities performed before the application of reconfiguration, and Section 3.3 presents the main issues to be addressed by a reconfiguration approach during reconfiguration in a running system. Finally, Section 3.4 reviews and compares some of the current reconfiguration approaches.

3.1 Process overview

The purpose of dynamic reconfiguration is to make a system evolve incrementally from its current configuration to another configuration. A system configuration is defined as a structure of software entities. Dynamic reconfiguration should introduce as little impact as possible (ideally no impact at all) on the system execution.

Figure 4 depicts the dynamic reconfiguration model based on [9, 10], which has been adopted in this thesis.

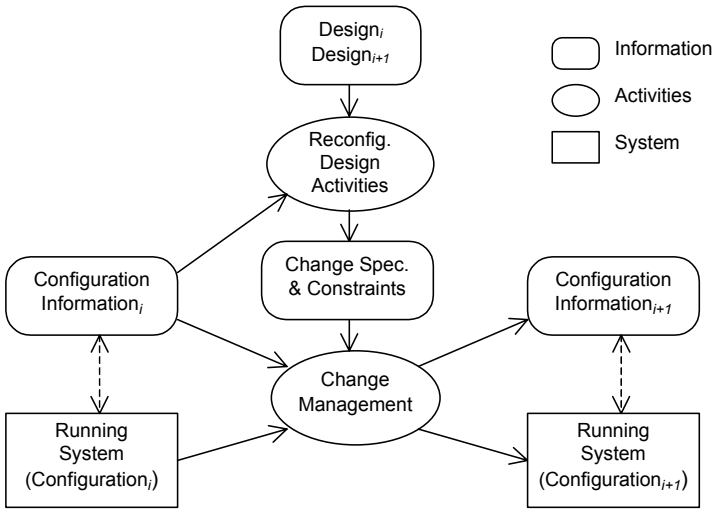


Figure 4 – Dynamic Reconfiguration model

In this model, *reconfiguration design activities* produce the specification of well-defined changes and constraints to be preserved during reconfiguration. Changes are specified in terms of *entities* and *operations* on these entities, and are applied under the control of *change management functionality*, making the system evolve from its *current configuration* to a *resulting configuration*. Change management functionality uses *configuration information*, which refers to the relationship between entities.

Change management functionality [9, 13, 15] controls the reconfiguration process of a distributed system. This functionality must guarantee that (i) specified changes are eventually applied to a system, (ii) a (useful) correct system is obtained, and, (iii) reconfiguration constraints are satisfied. *Reconfiguration constraints* are predicates on the reconfiguration process that restrict its execution, such as, e.g., "the reconfiguration process must be completed within 10s", or "entity *A* should be available during the whole reconfiguration process".

3.2 Reconfiguration design activities

Reconfiguration (or change) design activities are part of the design activities that are executed during the lifetime of a system. These activities succeed system deployment, in case of unforeseen changes, and precede the application of reconfiguration to a system. They are performed by *reconfiguration (or change) designers*.

Reconfiguration designers make use of the initial system configuration and the new configuration, identifying modifications introduced, to produce a well-defined set of changes to be applied to a system.

Changes are specified in terms of *entities* and *operations* on these entities. The definition of entity depends on the level of granularity of reconfiguration. Examples of entities include objects, groups of objects, components, group of components, sub-systems, bindings and groups of bindings. Examples of operations on entities are replacement, migration, creation, and removal.

The procedures for obtaining of a new system configuration are beyond the scope of this work. These procedures are performed subsequently to design activities

and may include transformations on the initial system, re-specification, re-design and re-implementation, re-validation, re-test of parts of the system, acquisition and integration of new system parts, etc.

3.3 Change Management

Operating systems, distributed object platforms and programming languages do have mechanisms to enable system evolution, by allowing modules to be located, loaded and executed during run-time. However, these mechanisms normally do not ensure consistency, correctness, or desired properties of run-time change. Therefore, the sole use of these mechanisms to perform reconfiguration is error-prone [15].

Performing reconfiguration on a running system is an intrusive process [13]. Reconfiguration may imply, for example, creation, removal, migration or replacement of reconfigurable entities, and interference with ongoing interactions between entities. Reconfiguration management must assure that system parts that interact with entities under reconfiguration do not fail because of reconfiguration.

Preservation of system consistency is a major reconfiguration requirement. A system can become useless in case the preservation of consistency is ignored. The system under reconfiguration 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 [13]. 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 resulting running system S_{i+1} is said to be a *correct incremental evolution* of a running system S_i, if S_{i+1} is correct, and if the behavior of the affected entities complies with the behavior expected by the unaffected system parts in case the reconfiguration had not taken place. Each aspect of the correctness notion is addressed in the remainder of this section.

3.3.1 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 might be put together.

Reconfiguration may affect the structural integrity of the whole system, so that corrective measures must be taken. For example, in the CORBA object model let us consider the replacement of one object by its new version. Clients of the object being replaced should be capable of invoking the operations of this object during reconfiguration and after reconfiguration has taken place. This implies that two conditions on the structural integrity of the system must hold: (i) the new version of the object must satisfy the interface definition of the original object, providing its service through the operations of this interface, and (ii) the clients should be able to access the service provided by the object through the interface, i.e., in CORBA terms the clients should obtain the object reference of the new object.

3.3.2 Mutually consistent states

Entities in a distributed system need to be in mutually consistent states if they are to interact successfully with each other. 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 [13]. *Interactions* are the only means by which entities can affect each other's state.

In order to provide an example, we can consider that object *A* invokes an operation on *B*. Objects *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 is 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 forever for a response.

Reconfiguration approaches normally provide mechanisms to transform systems with entities in mutually consistent states into resulting systems that maintain this mutual consistency. Suppose we have capabilities for *coping with the temporary interruption of interactions* for the purpose of reconfiguration and for continuing these interactions in the resulting system. In this case the application developer still has to develop means to restore the control state of the reconfigured entities, allowing the interrupted interactions to continue after reconfiguration. This control state would typically include the state of the invocation stack, program counter or thread context information. This information would be closely tied to specific characteristics of the implementation code, and it would be typically platform-dependent. The mapping of the control state from one implementation to the implementation of the new version would require deep knowledge of both implementations and would hardly be manageable by the change designer, preventing us from upgrading systems with arbitrary level of modification. Therefore most approaches to reconfiguration do not consider this alternative.

When considering reconfiguration, we introduce the term *affected entities* to denote those entities that are replaced, removed or migrated as a result of the reconfiguration process. 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 5 shows a classification of reconfiguration approaches according to their choices on the preservation of mutual consistency.

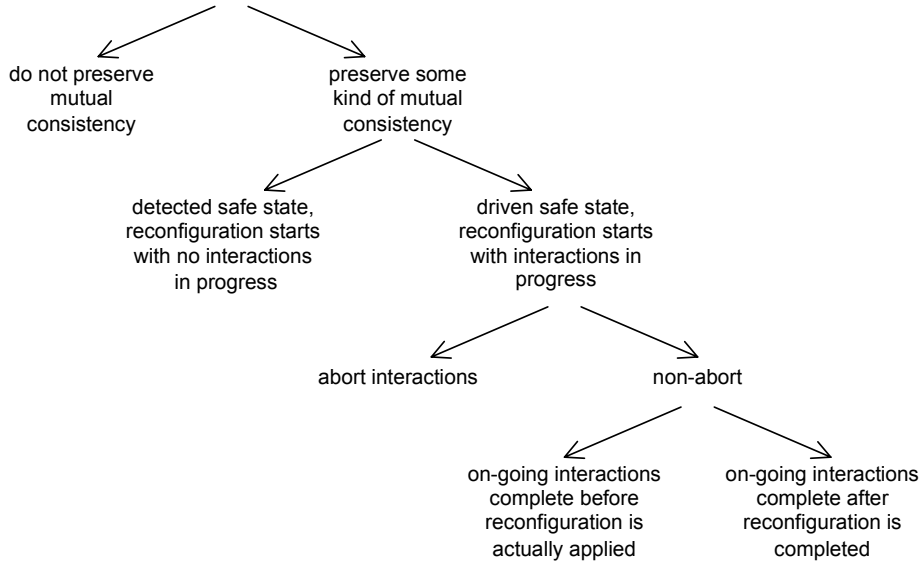


Figure 5 – Reconfiguration approaches and preservation of mutual consistency

In the classification on Figure 5, approaches that preserve some form of mutual consistency fall into two categories: the ones that reach the reconfiguration-safe state by observing the system execution, and the ones that reach the reconfiguration-safe state by driving the system to it. In the former case, the reachability of the safe state depends on the behavior of the application. For systems in which entities may interact continuously, there is no guarantee that reconfiguration will ever take place. If interactions are always in progress, reconfiguration is postponed indefinitely. In case the system is driven to a safe state, it is the role of the reconfiguration algorithm to guarantee the reachability of the safe state.

Existing approaches that work with a driven safe state fall into two major categories [13]: those in which during reconfiguration interactions are aborted and that rely on entities to recover from abortions, and those which avoid interactions to be aborted. Mechanisms based on interaction abortion (e.g., [4]) require the application developer to provide rollback mechanisms to recover from abortions without proceeding to errors. Therefore, the range of applications to which these mechanisms can be used is quite limited.

Mechanisms that do not abort interactions are designed to assure that interactions in progress are eventually completed, either before reconfiguration has started or after reconfiguration has finished.

This thesis proposes a mechanism that assures that on-going interactions complete before reconfiguration takes place, by driving the system to a reconfiguration-safe state. This mechanism is discussed in Chapter 4.

3.3.3 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 [13].

For example, let us 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 version. So, either (i) the set of all used identifiers is provided to the new version of the object, or (ii) the last used identifier is provided to the new version of the object. The latter alternative would require knowledge of the assignment mechanism used by the original version.

If dynamic reconfiguration is to be useful in a broad range of scenarios, it ought to provide mechanisms to allow the re-establishment of application invariants.

Most existing reconfiguration approaches rely on embedding the extra functionality for dealing with invalidated invariants into reconfigurable entities [13]. In this way, the responsibility to re-establish application invariants is solely delegated to application entities, which must determine what course of actions is needed to re-establish application invariants. For example, in Conic [10], application designers are required to supply modules with embedded routines (*initialization* and *finalization*) that are called whenever a reconfiguration operation is executed. The complexity of these routines depends largely on the nature of the application.

As pointed out in [13], this approach has serious drawbacks. Due to the generality of possible changes to a system, individual entities are rarely in a position to determine the course of actions to re-establish application invariants. This is especially true when, as is often the case, invariants are expressed over the combined state of a number of entities of the system.

Application entities developed to re-establish application invariants are likely to lose their potential generality, since they embed configuration specific concerns that prevent them from being used in other configurations. This is hardly acceptable in flexible architectures, as it conflicts with the tendency to develop systems based on the composition of general-purpose components.

Since embedding the necessary functionality to deal with invalidated invariants into application entities is undesirable, the support platform should provide mechanisms for change designers to specify how to re-establish application-state invariants.

[13] proposes a scheme whereby invalidated invariants can be identified and re-established by the change designer with little assistance from the application developer. This scheme consists of requiring reconfigurable entities to provide general-purpose state access-methods that can be invoked by a third party to query or adjust the state of entities. These are called *state-access* methods, and would be invoked by the change designer to query and alter a selected subset of an entity's internal state at runtime. The particular subset of the state that is exposed by these access-methods is decided upon by the application designer. In general, entities should provide "get" and "set" methods for state variables that control synchronization and computational behavior of the entity. One might argue that this scheme breaks encapsulation, once it allows external access to a component's internal state. Nevertheless, some form of introspection is necessary anyway for the manipulation of run-time aspects of an entity.

The nature of the safe state discussed in Section 3.3.2 should be such that in the safe state the invocation of state-access methods yields meaningful results. Thus,

a reconfigurable entity in a reconfiguration-safe state must have a consistent, self-contained and accessible state.

3.3.4 Impact on Execution

Reconfiguration is an intrusive process, since during reconfiguration, some system entities may become partially or totally unavailable, which can affect the performance of the system as a whole. Determining to what extent a system is affected during reconfiguration is relevant to assess the risks and costs in performing dynamic reconfiguration. If the system during reconfiguration fails to satisfy some non-functional requirements (e.g., hard response times), it may not be feasible to reconfigure during run-time. For instance, dynamic reconfiguration may be shown to be unacceptable due to safety reasons. This may be the case for process control, where a failure to perform a critical activity within time can put people's lives in danger.

Impact on system execution can be defined as the effect of reconfiguration on the provision of Quality of Service (QoS), i.e., on meeting a set of quality requirements on the system's behavior [8]. Therefore, the effects of reconfiguration should be placed in the broad context of QoS provision.

The quantification of the impact of reconfiguration on system execution is not trivial. Some reconfiguration approaches [9, 13] quantify the impact on system execution as proportional to the number of system entities affected by reconfiguration. These entities become idle or partially idle due to reconfiguration and would otherwise execute normally. In [3] a more fine grained quantification is proposed in which impact is said to be minimal if the reconfiguration affects the smallest possible set of execution threads in system objects. In [34], it is argued that more attention should be given to the period of time system entities are affected by reconfiguration.

Application characteristics are important when determining the impact of reconfiguration on execution. The reconfiguration algorithms chosen for an approach cannot be evaluated if we refrain from considering, for example, the level of coupling between system parts and the duration of the interactions between these parts.

In order to better understand the implications of application characteristics to system execution during reconfiguration, let us consider an application where interactions might take up to some hours to complete. Further, let us consider the replacement of an entity that has just initiated an interaction, using a reconfiguration approach based on a driven safe state. If we choose for an approach that allows on-going interactions to complete before reconfiguration, the reconfiguration will have a large impact on system execution, as it might take hours before the safe state is reached. During this time period, the affected system parts may not initiate new interactions, which might prevent the rest of the system from functioning. In contrast, if we choose an approach that aborts interactions, the reconfiguration time can be reduced drastically.

Ultimately, the maximum acceptable level of disturbance on the provision of QoS during reconfiguration is determined by the application.

3.4 Current Reconfiguration Approaches

This section describes some available approaches to dynamic reconfiguration reported in the literature, extending the survey presented in [13]. The selected approaches preserve mutual consistency without aborting interactions and strive to minimize impact on system's execution.

For each approach we present its overall considerations on reconfigurable distributed systems, as well as the way it structures reconfiguration functionality, the configuration information required to allow reconfiguration and its specific mechanisms to guarantee correctness on the resulting system. These mechanisms are compared throughout the section.

3.4.1 Kramer and Magee

The work of Kramer and Magee [9, 10] has been influential in the subsequent works on dynamic reconfiguration. The definition of change management and the reconfiguration model in Section 3.1 stem mostly from their work. Kramer and Magee promote a strict separation between the structural description of a system and the description of individual nodes. The first realization of their approach could be seen in the Conic environment [9], and led to the development of the approach called Configuration Programming and a configuration language named Darwin [11].

In the Configuration Programming approach, a system is seen as a directed graph whose nodes are the system nodes and whose arcs are connections between nodes. The model assumes at most one connection between any pair of nodes. Nodes can only affect each other states via transactions. A transaction is an instance of information exchange between two and only two nodes, initiated by one of the nodes, and consists of a sequence of one or more message exchanges between the two connected nodes. The model also assumes that transactions complete in bounded time and that the initiator of a transaction is aware of its completion. Figure 6 shows an example of a simple system, in which nodes A_1, A_2 and A_3 are able to initiate transactions on a node B.

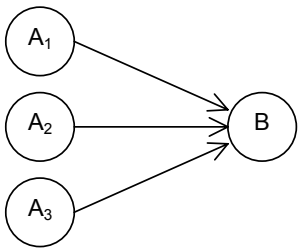


Figure 6 – A simple system

In this approach, a change is described in terms of modifications to the structure (configuration) of the application system. Changes take the form of *node creation* and *deletion*, and *connection establishment* and *removal*, and are applied by a Configuration Manager.

The set of management primitives for both specifying and modifying the structure of systems is:

- **create N:T [at L]** - Create node N of type T, optionally specify a physical location L;
- **remove N** - Remove node N;
- **link N1 to N2** - Create a connection from node N1 to node N2;
- **unlink N1 from N2** - Remove a connection between node N1 and node N2.

Reconfiguration-safe state

This approach has been the first to propose an avoidance-based mechanism to ensure that reconfigurations do not result in mutually inconsistent node states.

Kramer and Magee's approach uses the description of the changes and the current system configuration:

1. to identify the set of nodes whose activities must be restricted if reconfiguration is to proceed without leaving them in mutually inconsistent states, and;
2. to instruct these nodes to restrict their behavior by becoming *passive,* so that the safe state for reconfiguration *quiescence* is brought about over the affected nodes.

In this approach, node interactions are bounded transactions which are assumed to be the only means through which connected node can affect each other's states. Both parties involved in a transaction are informed of its completion. A transaction t is said to be *dependent* on the consequent transactions $t_1, t_2, \ldots t_n$ (written $t/t_1t_2..t_n$), if t can complete only after $t_1, t_2, \ldots t_n$ complete, and *independent* otherwise. This approach supports reconfiguration in systems with independent and dependent transactions.

Reachability of the safe state

The safe state for reconfiguration is reachable in finite time. This is discussed below for systems with only independent transactions and then generalized for systems with dependent transactions.

For systems with independent transactions a node is said to be in the *passive* state if it:

(a) continues to accept and service transactions, but

(b) does not initiate new transactions, and

(c) any transactions it has already initiated have completed.

A node reaches the passive state by refraining from starting new transactions and waiting for all the transactions it has started to terminate. A node is said to be passive if it is in a passive state. Passive nodes are not necessarily in a

reconfiguration safe state, since they continue to accept and service transactions. Therefore, the notion of quiescence is relevant. A node is said to be *quiescent* if:

(d) it is in the passive state, and

(e) it is not currently engaged in servicing any transactions (self initiated or otherwise), and

(f) no transactions have been or will be initiated by other nodes which require service from this node.

The passive state can be brought about by nodes unilaterally. The quiescent state, in contrast, can only be brought about by nodes in cooperation with other nodes in the system. A node *N* becomes quiescent if and only if all nodes in its passive set PS, denoted PS(*N*) are in the passive state. For independent systems, the membership of PS(*N*) is as follows:

(a) the node *N*, and

(b) all nodes that can directly initiate transactions on *N*, i.e., all nodes directly connected to *N*.

If all nodes in PS(*N*) are passive, *N* as well as all nodes that can initiate transactions on *N* are passive. Therefore, all transactions involving *N* are complete and new transactions will not be initiated, satisfying quiescence requirements a) and b). As the approach assumes transactions to complete in bounded time, it follows that quiescence is reachable within bounded time.

For systems with *dependent transactions* the situation is more complicated and the definition of passive and PS(*N*) need to be amended to allow for the initiation and service of consequent transactions. Consider the system depicted in Figure 7, consisting of three nodes *N*1, *N*2 and *N*3. Suppose that *N*3 is in the passive state and *N*1 has initiated transaction *a*. In this situation, *a* cannot complete because *b* depends on a consequent transaction *c* which *N*3 cannot initiate. This means that neither *N*1 nor *N*2 will be able to move into the passive state in bounded time if requested.

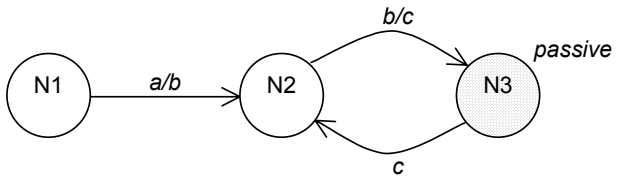


Figure 7 – A system with dependent transactions

To ensure the reachability of the passive state and consequently the reachability of the quiescent state, the requirements of the passive state have been modified as follows. For a dependent system a node is said to be in the passive state if it:

(a) continues to accept and service transactions and initiate consequent transactions, but

(b) does not initiate new (non-consequent) transactions, and

(c) any (non-consequent) transactions it has already initiated have terminated.

The set of passive nodes is extended to include all the nodes which are capable of initiating transactions indirectly on *N*. The *enlarged passive set* for a node *N* (EPS(*N*)) is defined as follows:

(a) all nodes in PS(*N*) are in EPS(*N*), and

(b) all nodes that can initiate dependent transactions that result in consequent transactions on *N* are in EPS(*N*).

This extension guarantees that node *N* reaches a quiescent state in finite time.

Reconfiguration rules

So far, we have discussed how nodes can reach quiescent states. Nevertheless, we have not discussed which set of nodes should be in the quiescent state for reconfiguration. In Kramer and Magee's approach, reconfiguration actions are *node deletion*, *node linking and unlinking*, and *node creation*. For each of these actions, reconfiguration rules in Table 1 apply:

Table 1 – Reconfiguration rules and justification

NODE REMOVAL	*Rule:* The node targeted for removal must be quiescent and isolated, where isolated means that no connections are directed to it from other nodes or from it to other nodes. *Justification:* An isolated node cannot affect the system and therefore can be independently removed.
NODE LINKING AND UNLINKING	*Rule:* The node N from which the connection is directed must be in the quiescent state. *Justification:* Quiescence of the initiator node ensures that its state is consistent and frozen with respect to that connection, and all transactions involving this node are complete.
NODE CREATION	*Rule:* The node should be quiescent. *Justification:* Trivially true, a newly created node is initially isolated and can neither respond to nor initiate transactions.

Using these rules it is possible to obtain the order in which nodes should be made passive, removed, created, connected and disconnected. Kramer and Magee also define an algorithm that allows multiple reconfigurations to be conducted simultaneously.

Some criticisms to Kramer and Magee's approach are:

- it places a heavy burden on the application programmer who must write all nodes of the system such that they respond correctly to the command to drive to a passive state [3, 13],

- even small reconfigurations involving a few nodes result in substantial disruptions to the system [13, 34], and

- the re-establishment of application invariants is done through routines embedded in nodes [13].

3.4.2 Moazami-Goudarzi

In [13], Moazami-Goudarzi proposes a framework that identifies the basic elements of a change management subsystem and establishes a separation between the responsibilities of the components that implement this subsystem. Figure 8 depicts this framework, which consists of a reconfiguration manager (RM), a reconfiguration database (CDB), the consistency manager (CM) and a number of runtime hooks in the application.

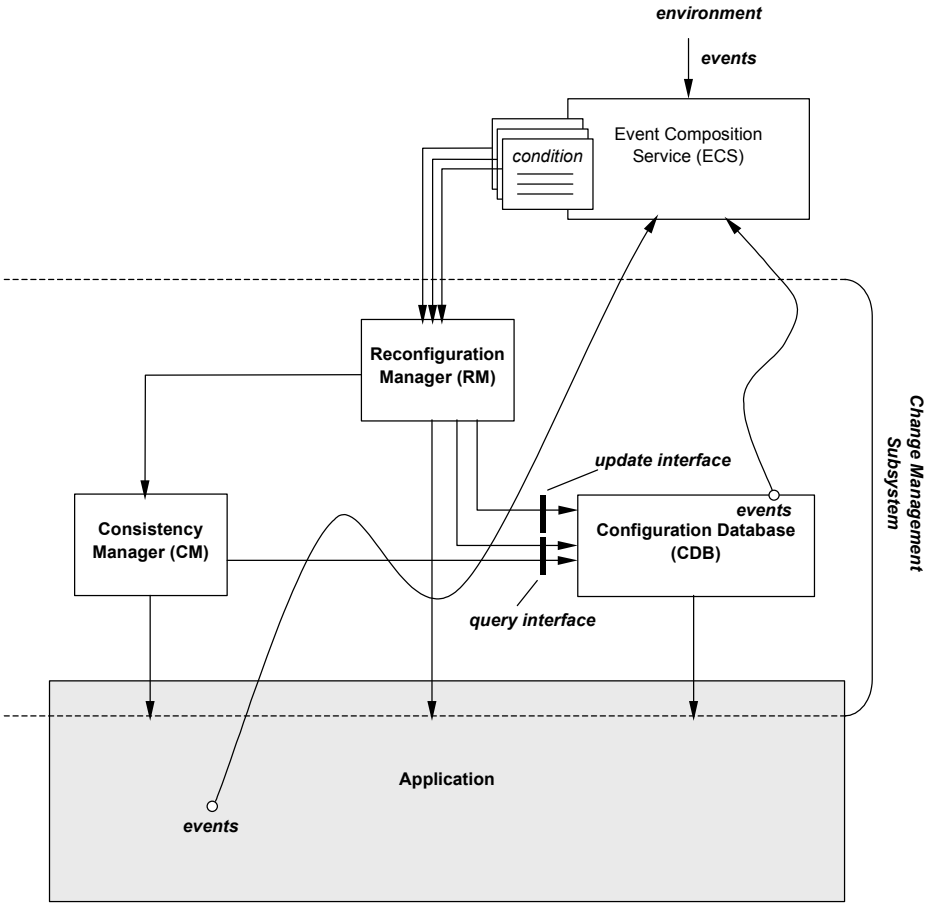


Figure 8 – Moazami-Goudarzi's Reconfiguration Framework [13]

The reconfiguration manager (RM) selects and executes reconfiguration scripts upon the arrival of triggering messages from an Event Composition Service (ECS). The RM coordinates the execution of the scripts with the consistency manager (CM) and configuration database (CDB), such that reconfiguration operations do not interfere with each other and leave components in mutually consistent states.

The consistency manager (CM) encapsulates the safety mechanism necessary to ensure that components are left in mutually consistent states after reconfiguration. Thus reconfigurations only proceed after the CM has been consulted and has signaled that they can proceed safely.

The configuration database (CDB) maintains and affects changes to the system configuration. It exports an interface that can be used to query and modify the system configuration. Interactions with the CDB are transaction-based and are performed through an internal concurrency control module that coordinates concurrent access to the system configuration.

The event composition service (ECS) evaluates the triggering conditions written by the change designers and generates messages that trigger the execution of the reconfiguration scripts. In this framework, reconfiguration scripts are written in a reconfiguration language.

The re-establishment of the application invariants is controlled from within the reconfiguration program, with the aid of specialized runtime hooks.

Preserving Consistency

[13] presents an alternative to Kramer and Magee's approach to reach a reconfiguration safe state. It assumes that components in the system do not interleave transactions, i.e., while a transaction is in progress, a component does not participate in any new one. In this way, it is possible to drive a component to a quiescent state by blocking its execution when no transactions are being serviced. As in Kramer and Magee's approach, for a component to block within finite time (therefore reaching quiescence), transactions are assumed to complete within finite time.

The basic algorithm is thus to request components in the quiescent set (called BSet, short for blocking set [13]) to block their execution. Consider that a component *Q* is to be driven to the quiescent state. Since some of the nodes that depend on *Q* may also have to block, *Q* must temporarily unblock to service some requests. However, the mechanism must guarantee that at some point no more such requests arrive and *Q* remains blocked. Therefore, a blocked node should be selective when serving transactions. Such a component cannot process just any incoming transaction, since the transaction might come from a component that is not affected in any way by the reconfiguration and as such it might initiate a new transaction any time. Thus the blocked component would have to unblock unpredictably and the safe state needed for reconfiguration to begin would never be reached. At least the transactions initiated by other BSet members will have to be serviced in order for them to become blocked. However, not every request from a non-BSet member can be ignored, since this might indirectly prevent another component in the BSet from blocking.

The BSet grows dynamically with outgoing transactions. When a component gets a request from a BSet member, it becomes a member too, and only requests from BSet members are attended; all other are queued and serviced after reconfiguration. A distinction is made between members of the original BSet and members of the extended BSet. Members of the original BSet are affected directly by reconfiguration. Members of the extended BSet are those that have blocked in order to let the members of the original BSet get blocked. When all the members of the original BSet are blocked, the components in the extended BSet can be unblocked. The BSet thus first grows and then shrinks.

This alternative addresses some of the criticisms to Kramer and Magee's work mentioned before. Nevertheless, the class of distributed systems to which this alternative can be applied is much more limited than in the case of Kramer and Magee, since components in this approach cannot treat more than one transaction

simultaneously. In a CORBA-based system this would imply that reconfiguration of systems with re-entrance and multi-threading would not be supported.

3.4.3 Bidan et al.

In [3], the implementation of a reconfiguration service in CORBA is considered. A distributed system in this case consists of a number of objects that communicate over an ORB. The reconfigurable entity is a CORBA object and the configuration information consists of a directed graph of objects connected through links. Objects *A* and *B* are said to be linked if *A* can potentially initiate a CORBA invocation on a target object *B*. Links are therefore equivalent to connections in Kramer and Magee's approach.

This approach offers node consistency, i.e., it is primarily concerned with preserving mutual consistent states, refraining from addressing application consistency. More specifically, they provide RPC-integrity, which is defined as "all RPCs initiated will be completed before the changes are effected." By providing node consistency they avoid addressing application state invariants and state transfer issues.

The reconfiguration service is designed for CORBA applications with multi-threaded objects, following the thread-per-request execution model, and extends the LifeCycle facilities [18] to support dynamic reconfiguration of a CORBA application. It provides the primitives *create* and *remove* to respectively create and remove objects, and 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.

Reconfigurable objects should implement functionality to passivate a link, i.e., to block the thread that may use the specific link.

Preserving Mutual Consistent States

In [3], the algorithm to guarantee mutual consistent states works at a finer granularity level than the approaches previously presented. This approach considers the passivation of links instead of quiescence, passivation or blocking of components. The advantage of this approach is that multi-threaded components can continue functioning partially, since only threads that may use the passivated links are required to block. Nevertheless, this implies additional burden to the application developer, which must provide functionality to restrict individual threads that use a specified link.

Unlike the approaches in [9, 34], this algorithm is not suitable for a system with re-entrant transactions. Since Bidan et al.'s work has focused on CORBA-based distributed systems, this means that the reconfiguration of systems with re-entrant invocations is not supported. Another major limitation of this approach is that it does not support multiple simultaneous object replacements.

3.4.4 Wermelinger

Wermelinger's approach [34] considers link passivation, as in [3]. Nevertheless, more fine-grained information on the reconfigurable entities is used in this approach than in [3].

A refined model for a distributed system is introduced, with the notion of *port dependencies.* A system is defined as a set of connected nodes, where a connection is given by an initiator port and a recipient port. Every node has a node interface, where port dependencies are specified. A port dependency is defined by a recipient port and an initiator port. Port *I* is said to be dependent of port *R*, if upon reception of a transaction in *R*, a transaction is initiated in connections leaving from *I*. This makes it possible to relate transactions and derive *transaction dependencies*, such as in [10]. Figure 9 shows an example of a simple system using the refined model.

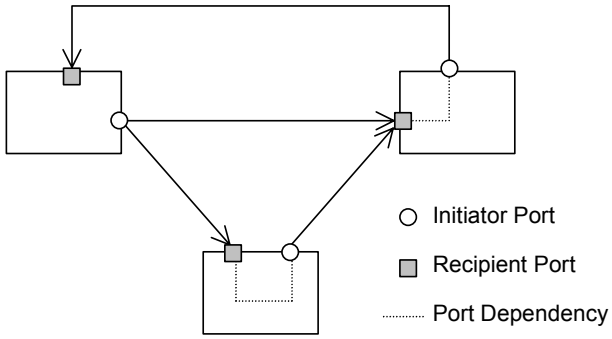


Figure 9 – A simple system with port dependencies

Wermelinger claims that the approach is suitable for multi-threaded and re-entrant nodes. The approach requires a component to be shipped with a description of a node's internal port dependencies, which are static and defined at design time. This sort of specification is typically not available, especially for off-the-shelf components.

Wermelinger's work is presented at a theoretical level, and so far it has not been implemented.

3.4.5 Observations

A common characteristic of the approaches we have studied is the definition of a reconfiguration-safe state. A system is driven into this safe state by algorithms that interfere with the execution of the system.

All the approaches studied use representations of the system, which are described in configuration languages [9, 34] or in configuration graphs [3, 13] to identify which activities of the system should be deferred in order to reach the safe state. The use of these representations may have implications on the scalability of the solutions.

The approaches studied do not assume the same computation model. For example, in the computation model assumed by Moazami-Goudarzi, an application entity cannot be involved in several interactions simultaneously, while in the computation model assumed by Kramer and Magee this restriction does not apply. The computation model assumed by an approach has direct implications on the definition of a safe state and the algorithms to reach this safe state.

4 A Dynamic Reconfiguration Approach

This chapter proposes an approach to dynamic reconfiguration that is tailored to object middleware. The approach assumes the object model presented in Chapter 2, and addresses each of the correctness aspects identified in Chapter 3.

This chapter is further structured as follows: Section 4.1 motivates the development of a new approach aimed at object-middleware-based applications, Section 4.2 states the requirements for such an approach, Section 4.3 presents the reconfiguration possibilities supported by our approach and Section 4.4 describes the mechanisms prescribed for change management. Finally, Section 4.5 discusses the limitations of the approach and Section 4.6 compares it to the approaches found in the literature.

4.1 Motivation

Most of the approaches found in the literature do not address object-middleware-based applications specifically. As a consequence, either they consider a computing model that is limited with respect to our object model, e.g. ruling out multi-threading or re-entrance, or they fail to address issues that are particularly relevant for object middleware systems, such as, e.g., interface evolution.

While some dynamic reconfiguration approaches that have not been originally developed for middleware platforms may be used in distributed object applications, only an object-middleware-based approach is able to profit from particular characteristics of object middleware.

Since object middleware facilitates transparencies for the application developer, it provides an opportunity for the provision of reconfiguration transparency. Application developers can profit from reconfiguration functionality with the benefits of a middleware-supported service, e.g., interoperability, application portability, language independence, and wide support, requiring minimal expertise in the field of dynamic reconfiguration.

4.2 Requirements

The following requirements have been considered in the conception of our approach:

- *Correct incremental evolution.* The approach must include mechanisms to obtain a correct incremental evolution of a system, as defined in Section 3.3. The integrity of the object model must be preserved under normal operation, i.e. when reconfiguration is not taking place, and during reconfiguration.

- *Impact on execution*. The approach should minimize impact on execution during reconfiguration, and it should account for little overhead during normal operation.

- *General applicability.* The approach should be applicable to a broad range of applications. It should be possible to reconfigure applications built from off-the-self components, applications with multi-threaded and single-threaded execution models, re-entrant objects, stateless objects and stateful objects.

- *Scalability*. The approach should account for scalability, particularly with respect to the number of clients. Distributed object systems often have a large and fast-changing number of clients.

- *Transparencies.* Reconfiguration should be fully transparent to clients, and as transparent as possible to the target object developer, requiring minimal expertise from application developers. In this way, application developers can focus on application specific functionality.

- *Language independence.* The approach must not require the use of specific programming languages for application development.

- *No additional formalisms.* The approach must not require the use of additional formalisms for application development.

4.3 Supported Reconfiguration

In our dynamic reconfiguration approach, entities subject to reconfiguration are called *reconfigurable objects*. A reconfigurable object is an object, as defined in the object model presented in Chapter 2, that can be manipulated through reconfiguration operations, namely *creation*, *replacement*, *migration* and *removal*.

4.3.1 Object Creation

Object creation allows an application to create an object at run-time. From the moment an object is created, a reference to its interface is used to communicate with it. Object creation is a trivial case from the perspective of change management, since applications are expected to cope with it.

Figure 10 depicts object creation from an abstract perspective.

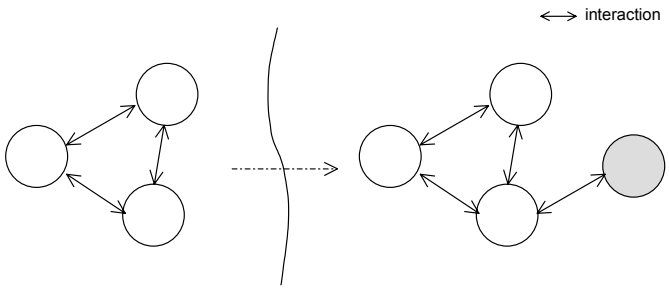


Figure 10 – Object creation

4.3.2 Object Replacement

Object replacement allows one version of an object to be replaced by another version, while preserving object identity. We use the term version of an object to denote a set of implementation constructs that realizes an object. The new version of an object may have functional properties that differ from the old version. For example, the new version may correct faults in the original version, or introduce improved algorithms that provide better and more precise results.

The new version may also have quality-of-service (QoS) properties that differ from the old version. For example, the new version may provide an implementation that performs better and uses less system resources. The change designer is responsible for assuring that the new version of an object satisfies both the functional and QoS requirements of the environment in which the object its inserted. In addition, the new version of an object may run in another execution environment supported by the object-middleware platform. For example, a version implemented in Java may be replaced by a version implemented in C++.

The aim of the reconfiguration approach is to provide replacement transparency, which masks the replacement of an object from objects that interact with it.

Object replacement requires special attention from the perspective of change management, since it threatens application consistency.

Figure 11 depicts object replacement from an abstract perspective. Object *A* is replaced, substituting its original version V_{A1} by a new version V_{A2}.

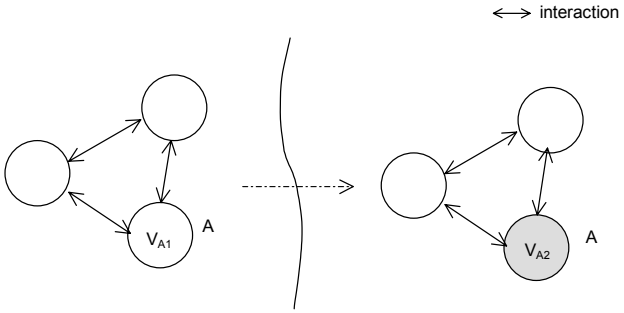


Figure 11 – Object Replacement

Replacement with Interface Changes

We define a version V_A' of an object *A conforming* with a version V_A if the interface of V_A' is identical to the interface of V_A or derived from it, and *non-conforming* otherwise.

Replacement of a current version by a non-conforming version is called non-conforming replacement. Our approach only supports non-conforming replacements in special cases that are explained later in Section 4.4.

4.3.3 Object Migration

Migration means that an object is moved from one location to another, which may also imply that the type of execution environment has to be changed. An object preserves its identity and state.

The aim of the reconfiguration approach is to provide migration transparency, which masks the migration of an object from objects that interact with it.

Object migration requires special attention from the perspective of change management, since it threatens application consistency.

Figure 12 depicts object migration from an abstract perspective. Object *A* migrates from its original location *X* to location *Y*.

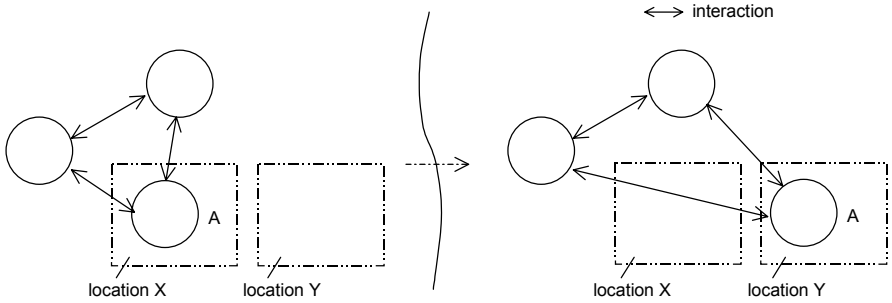


Figure 12 – Object Migration

4.3.4 Object Removal

Object removal allows an application to remove a reconfigurable object at run-time. From the moment an object is removed, the reference to its interface becomes invalid. Object removal is a trivial case with respect to reconfiguration, since applications are expected to cope with it.

Figure 13 depicts object removal from an abstract perspective.

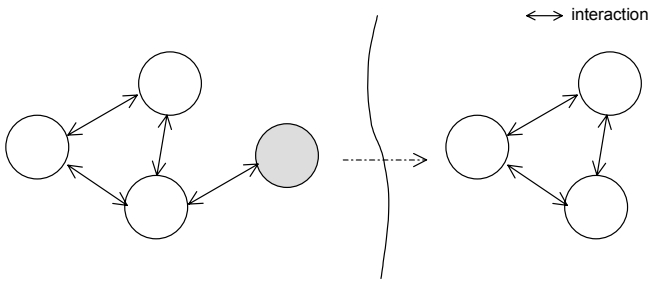


Figure 13 – Object Removal

4.3.5 Reconfiguration Steps

A system evolves incrementally from its current configuration to a resulting configuration in a *reconfiguration step*. A reconfiguration step is perceived as an atomic action from the perspective of the application.

A reconfiguration step consists of:

(i) the execution of a reconfiguration operation in an object, in which case it is called a *simple reconfiguration step;* or

(ii) the execution of reconfiguration operations in several distinct objects, in which case it is called a *composite reconfiguration step.*

Composite reconfiguration steps are often required for reconfiguration of sets of related objects. In a set of related objects, a change to one object *A* may require changes to other objects that depend on *A*'s behavior or other characteristics [21].

Figure 14 depicts a composite reconfiguration step from an abstract perspective, where object *D* is removed, objects *A* and *B* replaced, and object *E* is created.

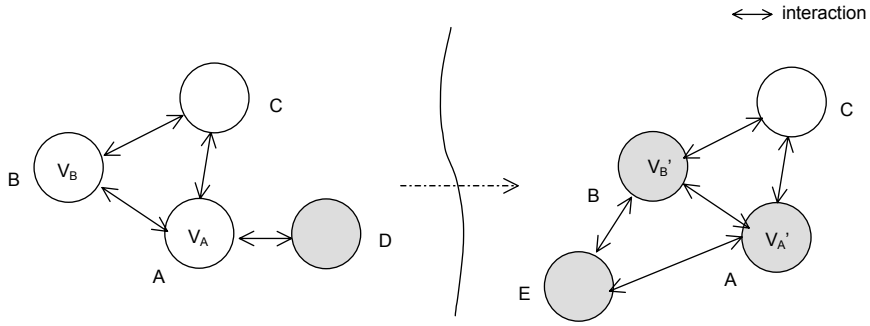


Figure 14 – Composite reconfiguration step

A particular case of composite reconfiguration step is the replacement of multiple objects. A common usage example would be to replace all objects of the same type with a new version.

Figure 15 depicts replacement of multiple objects from an abstract perspective, where, simultaneously, object *A* is replaced from version V_A to V_A', and object *B* is replaced from version V_B to V_B'.

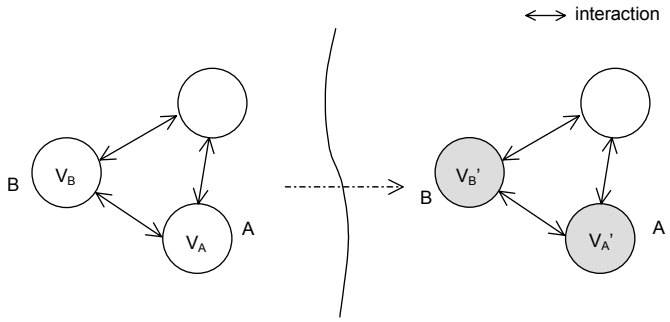


Figure 15 – Multiple Objects Replacement

4.4 Change Management

This section describes the change management mechanisms in our approach by addressing each of the correctness aspects identified in Section 3.3.

4.4.1 Structural integrity

In our 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 of an object from one host to another invalidates the IP address contained in the IIOP profile of an IOR (Interoperable Object Reference). 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 to find the objects with invalidated object references. This point of contact is called the location agent.

In the CORBA object model, interfaces satisfy the Liskov substitution principle [16]. 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, by inheritance.

It is possible to apply non-conforming replacements that promote arbitrary changes to the interface of a reconfigurable object if either one of the following conditions is satisfied:

(i) all clients of the reconfigurable object are also reconfigurable objects, or

(ii) the reconfiguration designer supplies a wrapper version of the object that is capable of translating requests to the new version.

Both cases require the use of a composite reconfiguration step. In the first case, the reconfigurable object and its clients are replaced. This case is depicted in Figure 16, with the non-conforming replacement of *A*. In the same reconfiguration step, the current version of *A* is replaced by the new version, and the current version of *B* is replaced by a version that is able to use *A*'s new interface.

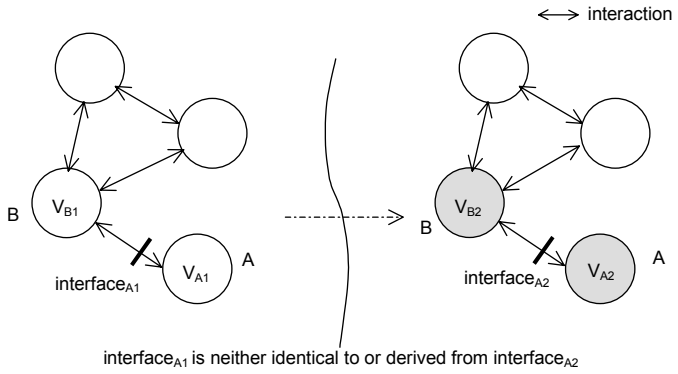


Figure 16 – Non-conforming Replacement (i)

In the second case, the new version is created and the reconfigurable object is replaced by the wrapper version. This case is depicted in Figure 17, with the non-conforming replacement of *A*. In the same reconfiguration step, the current version of *A* is replaced by the wrapper version, and the new version of *A* is created.

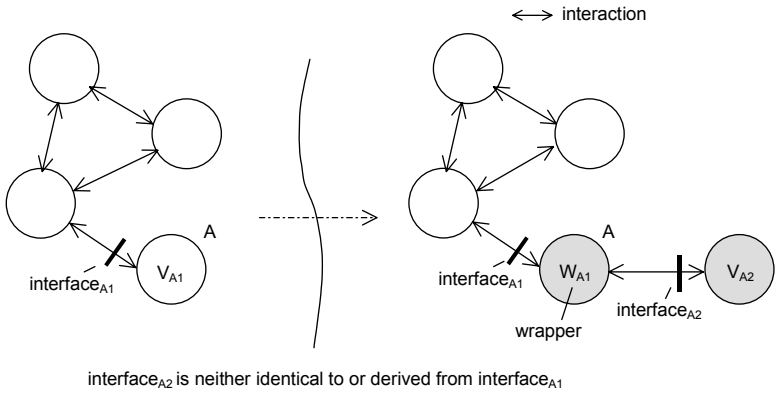


Figure 17 – Non-conforming Replacement (ii)

4.4.2 Mutually consistent states

We propose an algorithm 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 algorithm follows three stages:

1. Drive the system to the safe state by deferring interactions that would prevent the system from reaching the safe state;

2. Detect that the safe state has been reached; and

3. Apply reconfiguration;

We use the term *affected object* to denote an object that is replaced, migrated or removed as a consequence of 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 is processing requests or waiting for outgoing requests to be processed.

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 driving itself 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.

Reaching the safe state

We guarantee the reachability of the safe state by interfering with the activities of the system. All affected objects are requested to exhibit reactive behavior, and then pending invocations in the affected objects are allowed to complete.

In the case of a *simple reconfiguration step*, with the replacement or migration of a single *non re-entrant* 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 chance 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 case of replacement or migration of a single *re-entrant object*, we should not queue up re-entrant requests. A re-entrant request is not queued, since otherwise the affected object would have a pending outgoing request that would never complete. Consequently, the system would never reach the safe state. Figure 18 shows a re-entrant request that must be allowed to complete for reconfiguration to proceed.

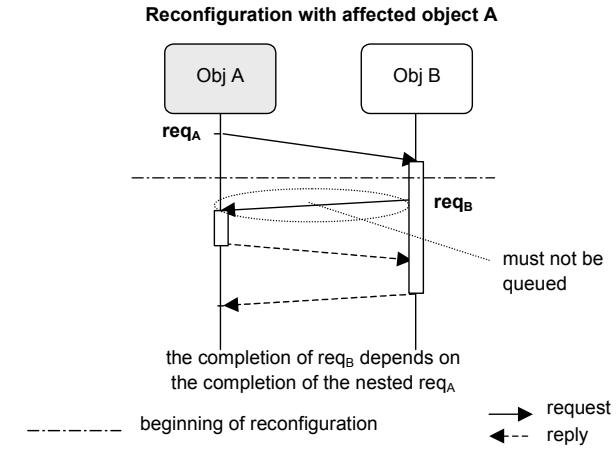


Figure 18 – Requests that must not be queued in simple reconfiguration step

In the case of *composite reconfiguration steps*, several affected objects have to be driven to the safe-state. In this case, we should neither queue up requests issued by an affected object nor the nested requests that are a consequence of requests issued by an affected object. If one of these requests would be queued, there would always be a pending outgoing request in the set of affected objects that

would never complete, and the system would never reach the safe state. Figure 19 shows requests that must be allowed to complete for reconfiguration to proceed.

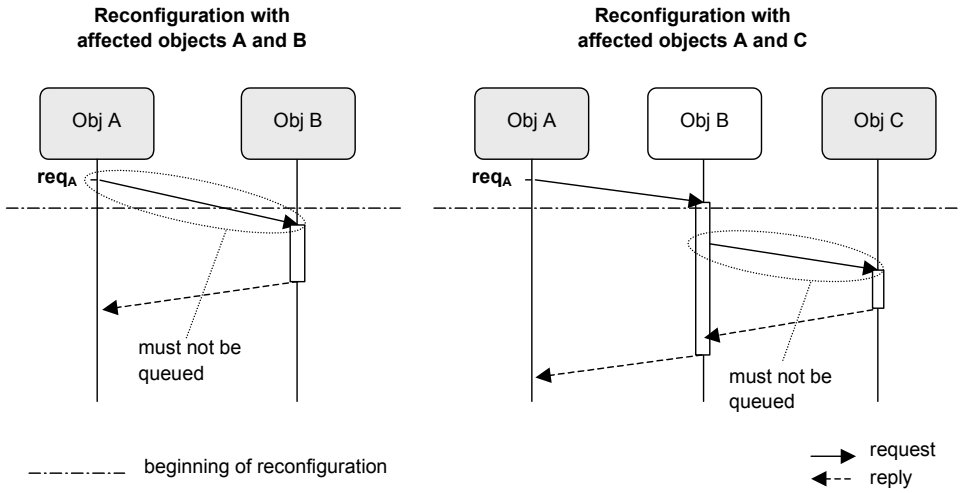


Figure 19 – Requests that must not be queued in composite reconfiguration steps

Therefore, in a system under reconfiguration, we can distinguish three sets of requests:

(i) *requests whose processing is necessary for the system to reach the reconfiguration-safe state* ('laissez-passer' set),

(ii) *requests whose processing could prevent the affected objects from reaching the reconfiguration-safe state* (blocking 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 be processed. This is done transparently for the application objects.

In order to identify requests that belong to the 'laissez-passer' set, we use the propagation of implicit parameters along invocation paths. For every reconfigurable object in an invocation path the middleware infrastructure adds the object's identification to the request as an implicit parameter.

Given a request and the set of affected objects, it is possible to determine if a 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, and should be allowed to complete.

Applying reconfiguration

When all affected objects are idle, reconfiguration can proceed. The affected objects' state can be inspected and used to derive the state of the objects being introduced. Once new objects, new and relocated versions of objects have been installed (which may be done before driving the system to the safe state), their state is properly modified. After their state is modified, they are allowed to exhibit active behavior. Queued requests and further new requests are redirected to the new or relocated version of an object after reconfiguration.

4.4.3 *Application-state invariants*

Each reconfigurable object must provide operations to access its state. These operations are used to inspect and modify a selected subset of the object's internal state. The application developer is responsible for deciding on the particular subset of the objects' state that is exposed by these state-access operations. In general, an object should provide operations to inspect and modify its control and data state. These operations are only invoked in the safe state. Since in the safe state an object is idle, the amount of control state to be externalized is minimized.

When the system to be reconfigured is in the safe state, the state of the affected objects can be accessed consistently through these state-access operations, and can be used as input to a state translation function supplied by the change designer. The state translation function determines the state of the new version of each affected object so as to guarantee that application invariants are not violated. Furthermore, the state translation function may have to adjust the state of an affected object so that its behavior is compatible with the behavior expected by its environment.

It is possible, however, that such state translation function does not exist for two given versions, preventing reconfiguration. This situation can be illustrated in a simple object replacement. It is possible that the new version of the object does not have a state that corresponds to the state of the original version. For example, let us consider an object that generates unique identifiers, with an initial version that generates identifiers counting up from *A* to *B*. The state variable of this implementation is the counter containing the last generated number *S*. Let us suppose that the new version also produces unique identifiers, but does it counting down from *B* to *A*. While the identifiers produced so far are known (all values smaller or equal to *S*), there is no value for the internal counter in the new version that can be derived so as to preserve the expected behavior of the generator. With any starting state, the new version would end up producing identifiers that have already been used in the previous version of the object, introducing inconsistency in the application.

4.4.4 *Impact on Execution*

While some reconfiguration approaches [9, 13] quantify the impact on execution as proportional to the number of affected reconfigurable entities, or proportional to the number of blocked execution threads in application objects [3], we intend to estimate the increase in response time experienced by clients of affected objects during reconfiguration.

Clients of an affected object may observe an increase in the response time of operations invoked in an affected object during reconfiguration. This increase only applies to 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 of the application. Its expected value is proportional to the average duration of interactions that involve the affected objects. Therefore, this increase is higher for systems with long-lived interactions. The increase is limited by the duration of the longest pending invocation in the set of affected objects at the moment the reconfiguration starts. For active objects, the amount of time taken for the object to exhibit reactive behavior should also be considered in the calculation of the upper bound of the increase in response time. We expect however this increase to be insignificant for most applications.

Considering the absolute increase in response time, the approach seems to be best suited for applications with short-lived interactions. Ultimately, the maximum acceptable increase in response time during reconfiguration is determined by the environment in which the affected object is inserted.

4.5 Limitations

Our approach ignores the preservation of architectural properties of an application, in contrast with the approaches presented in [9, 13, 15, 34]. We rather assume reconfiguration design activities to produce changes that have been validated *a priori,* and focus on the application of these changes. Reconfiguration design activities may be supported by tools that address the preservation of architectural properties.

Since we have opted for complete transparency for clients of reconfigurable objects, we can only support non-conforming replacements in restricted cases, as identified in Section 4.4.1. A less restrictive support to non-conforming replacements would require clients of reconfigurable objects to be developed with mechanisms to cope with arbitrary interface change.

Another limitation refers to the externalization of state. In the object model adopted, relationships between objects may be buried in the implementation of an object, in the form of object references. These object references cannot be easily manipulated by external entities. This forces the externalized state of an object to include all the object references that are still required for the object to continue operating and that otherwise would not be recovered from the system after reconfiguration. This problem could be solved in a component model (such as [24]) in which an external entity can reify connections between components at run-time and manipulate these connections.

Our approach does not exploit redundancy. Approaches that exploit redundancy promote object replacements by temporarily processing requests in both old and new versions of an object simultaneously. These approaches, such as the one adopted by [14], are often limited to object replacements where the new version of an object has the same externally visible functional properties when compared to the old version. Although it does not exploit redundancy, our approach can, in principle, be applied to redundant objects.

4.6 Comparison with Studied Approaches

This section presents distinctive features of our approach and compares it with the reconfiguration approaches studied.

4.6.1 Application-description Models

Our approach does not require the use of specific description formalisms for application development. Some of the dynamic reconfiguration approaches studied [10, 13, 15, 34] prescribe the use of architectural or configuration models to describe an application. These models are produced by the application designer during the development process, and are described in Architecture Description Languages (ADLs) or Configuration Languages (CLs).

These models are used by dynamic reconfiguration approaches to derive how to apply changes to a system under reconfiguration. For example, in Kramer and Magee's approach [10] an application is represented as a directed graph, whose nodes are system entities and whose arcs are connections between entities. An entity *A* is connected to an entity *B* if *A* can initiate a transaction with *B*. For an entity *Q* to be replaced, all the entities that are capable of initiating transactions directly or indirectly on *Q* should exhibit passive behavior, as well as *Q* itself. In this case, the configuration graph is used to identify which entities must exhibit passive behavior for the system to reach the reconfiguration-safe state. In Wermelinger's approach [34], application entities must be supplied with a description of internal port dependencies, which relate input ports and output ports.

A drawback of prescribing an ADL or CL for application design is that the conventional development process has to incorporate the production of a description of the application using the specific formalism or language. Our approach differs from these in the sense that it does not prescribe the use of an ADL or CL. The configuration information required to apply reconfiguration is obtained from the system at run-time. By doing this, we intend to separate the concerns of obtaining and maintaining configuration information for the reconfiguration design activities, and obtaining and maintaining configuration information for the application of reconfiguration.

Figure 20 shows a refinement of the model presented in Chapter 3, obtained by decomposing configuration information into configuration information obtained at design-time and obtained at run-time.

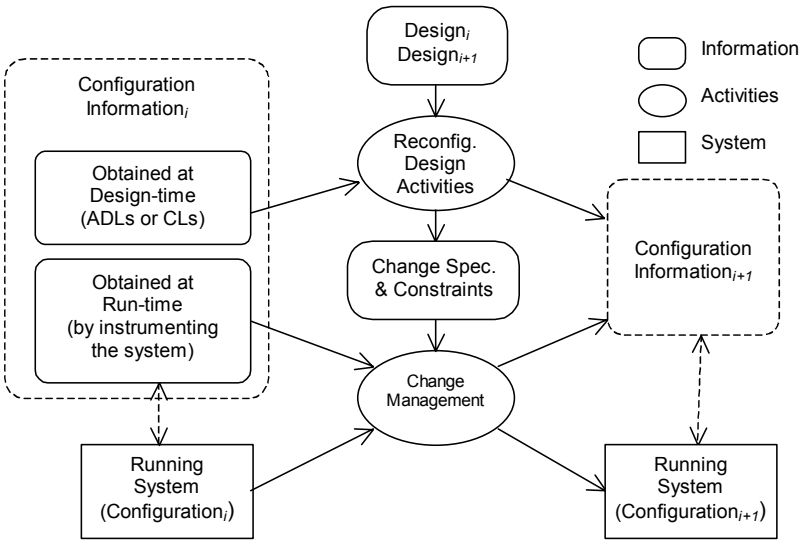


Figure 20 – Dynamic Reconfiguration Model with refined configuration information

Although we do not prescribe a specific ADL or CL, we acknowledge that ADLs can be useful to support software design activities. Moreover, ADLs can have an important role in reconfiguration design activities.

4.6.2 Reconfiguration Supported and Computation Model

Bidan et al. consider in [3] an approach to dynamic reconfiguration of CORBA-based applications. Similarly to 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.

From the approaches we have studied, this is the one that can be best compared to ours. Unlike our approach, Bidan et al.'s approach does not support composite reconfiguration steps. In sets of related objects, it is common that a change to one object may require changes to other objects that depend on the object's behavior or other characteristics [21]. Since only simple reconfiguration steps are allowed, the application of this approach is limited.

Furthermore, the approach does not support applications with re-entrant invocations. Therefore, an object that has initiated an invocation cannot play the role of server for some consequent invocation. The approach does not support re-establishment of application invariants and state translation either.

4.6.3 Impact on Execution

Our approach proposes a mutual consistency mechanism that only interferes with application activities that require interaction with affected objects during reconfiguration. This is not the case in most approaches we have studied [3, 9, 13, 34], which block all potential system activities that may prevent the system from reaching the safe state.

4.6.4 Transparencies

Our approach is completely transparent for the clients of reconfigurable objects, in contrast with [3, 10, 13, 34] where client applications have to provide support for reconfiguration.

Our approach facilitates the development of reconfigurable objects by incorporating change management functionality in the middleware infrastructure. Therefore, it requires minimal reconfiguration expertise from the object developer. The reconfigurable object developer is responsible to provide state-access operations and operations to drive an active object from an active state to a reactive state and back.

5 Dynamic Reconfiguration Service

This chapter describes the CORBA Dynamic Reconfiguration Service (DRS) we have developed. This chapter presents the DRS from the point of view of the users of the service. The DRS uses the mechanisms presented in Chapter 4, realizing the proposed approach in the CORBA platform, and supports both the application developer and the change designer.

This chapter is further structured as follows: Section 5.1 provides an overview of the architecture of the service, Section 5.2 describes the view of the change designer and Section 5.3 describes the view of the application developer. In this chapter, we use OMG Interface Description Language (IDL [16]) to specify the DRS interfaces.

5.1 Overview

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

The *Reconfiguration Manager* is the central component of the Dynamic Reconfiguration Service in that it interacts with all the other components of the service. It coordinates reconfiguration with Reconfiguration Agents and the Location Agent. The Reconfiguration Manager delegates object creation and removal to Reconfigurable Object Factories, it registers, re-registers and de-registers objects through interaction with the Location Agent and it co-ordinates the Reconfiguration Agents to drive the system to a reconfiguration-safe state.

A *Reconfiguration Agent* is created for each ORB instance that mediates requests for reconfigurable objects. Typically there will be an ORB instance per capsule, and a capsule will be a Java Virtual Machine or process. A Reconfiguration Agent is responsible for restricting the behavior of an affected object during reconfiguration through filtering of requests.

The *Location Agent* provides a registry for the location of reconfigurable objects. It produces location-independent object references, and is capable of translating 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 [16].

A *Reconfigurable Object* is the unit of reconfiguration. It provides state-access operations and is able to exhibit reactive behavior upon demand.

A *Reconfigurable-Object Factory* implements the Factory design pattern, creating and removing versions of 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 and the Java class loader.

Figure 21 shows an overview of the architecture.

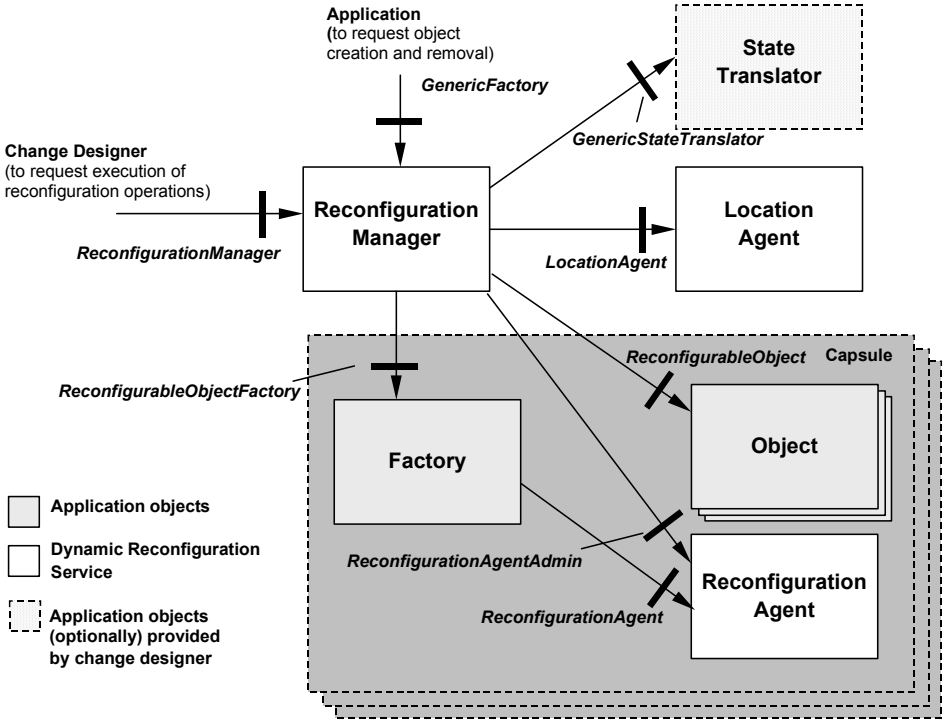


Figure 21 – Architectural Overview

The service concerns both the change designer and the application developer. The *change designer* can access the service of the Dynamic Reconfiguration Service to request the execution of reconfiguration steps. The *application developer* supplies application-specific Reconfigurable-Object Factories and Reconfigurable Objects that comply with the interfaces defined by the service.

5.2 Change Designer's View

The *change designer* interacts with the Dynamic Reconfiguration Service through the **ReconfigurationManager** interface. The **ReconfigurationManager** interface provides operations for creating and removing objects, managing factories and specifying reconfiguration steps.

From a client's point of view, there is no special mechanism for creating or destroying an object. Objects are created and destroyed as an outcome of issuing requests [16]. The same applies for reconfiguration, i.e., there is no special mechanism for requesting execution of reconfiguration steps.

5.2.1 Creation and Removal

Operations for object creation and removal are inherited from the **GenericFactory** interface defined in the Fault Tolerant CORBA specification [19].

The **create_object()** operation allows the application to request the creation of an object by specifying the identifier of the object's type and the criteria to be used in the creation.

```
interface GenericFactory {

   typedef any FactoryCreationId;

   Object create_object(
      in TypeId type_id,
      in Criteria the_criteria,
      out FactoryCreationId factory_creation_id
   )
   raises (
      NoFactory,
      ObjectNotCreated,
      InvalidCriteria,
      InvalidProperty,
      CannotMeetCriteria
   );

   void delete_object(in FactoryCreationId factory_creation_id)
   raises (ObjectNotFound);

};
```

The type identifier is the same identifier used in the interface repository to denote the most derived type of an interface. The type identifier is used in conjunction with the criteria to determine the local factory that creates the application object.

```
typedef CORBA::RepositoryId TypeId;
```

The criteria parameter allows application to define initialization parameters, and restrictions on how to create the object. Examples of criteria are initialization values, the required version of an object and the preferred location of an object. The criteria parameter is defined as a sequence of properties. A property is a name-value pair.

```
typedef CosNaming::Name Name;

typedef any Value;

struct Property {
      Name nam;
      Value val;
};

typedef sequence<Property> Properties;
typedef Properties Criteria;
```

The following names are reserved for criteria: **drs.VersionId**, **drs.Location**, **drs.IOR**, **drs.Id** and **drs.ApplicationObjectCreation**. The names **drs.VersionId** and **drs.Location** are used respectively to specify the required version of an object (**float**), and the preferred location of an object (**string** with the hostname or IP address). The names **drs.IOR**, **drs.Id** and **drs.ApplicationObjectCreation** are reserved for communication of the DRS with the application's local factories. All other criteria are implementation-specific and are interpreted only by the factory.

The **factory_creation_id** parameter allows the entity that invoked the factory and the factory itself to identify the object for subsequent manipulation. The **factory_creation_id** is an **Any** value that contains a **ReconfigurableObjectId**. This **ReconfigurableObjectId** is used to denote a reconfigurable object.

The object reference returned by the **create_object()** operation is a reference to the reconfigurable object, and is valid during the complete reconfigurable object lifetime (this reference continues to be valid after subsequent replacements and migrations).

5.2.2 Factory Management

The Reconfiguration Manager inherits the **FactoryManager** interface. This interface provides operations for adding and removing local application factories that are to be used by the Reconfiguration Manager.

The **add_factory()** operation registers a local application factory with the Reconfiguration Manager, and associates this factory to the types it can create. The information about the factory supplied to the Reconfiguration Manager consists of a reference to the factory, the location of the factory and the default criteria for object creation. The criteria include the version identifier of the objects created by the factory.

```
typedef Name Location;

struct FactoryInfo {
      GenericFactory factory_;
      Location the_location;
      Criteria the_criteria;
};

interface FactoryManager
{
      typedef any FactoryId;

      FactoryId add_factory(in FactoryInfo factory_info, in TypeIds type_ids);

      void remove_factory(in FactoryId factory_id)
      raises (FactoryNotFound);

      FactoryInfo get_factory_info(in FactoryId factory_id, out TypeIds type_ids)
      raises (FactoryNotFound);

};
```

The **get_factory_info()** operation provides information on a registered factory and the **remove_factory()** operation de-registers a factory.

5.2.3 *Reconfiguration Steps*

A system evolves incrementally from its current configuration to a resulting configuration in a *reconfiguration step*, which is perceived as an atomic action from the perspective of the application.

A reconfiguration step is modeled by a **ReconfigurationStep** object, which can be created through the **create_reconfiguratio_step()** operation of the **ReconfigurationManager** interface.

```
interface ReconfigurationManager : GenericFactory, FactoryManager
{
    ReconfigurationStep create_reconfiguration_step()
    raises (UnderReconfiguration);
};
```

The **UnderReconfiguration** exception is raised if another reconfiguration step is being defined. An implementation of the DRS may allow multiple reconfiguration steps to be executed in parallel, in which case this exception may not be raised.

Composing a reconfiguration step

The **ReconfigurationStep** interface provides means to compose a reconfiguration step from reconfiguration operations, namely, object creation, object replacement, object migration, and object removal. The change designer composes a reconfiguration step and commits it, i.e., requests its execution.

The operations for object creation and removal have the same syntax as defined in the **GenericFactory** interface. These operations differ from the ones defined in the **GenericFactory** interface in that these operations are executed when the reconfiguration step is committed. The object reference returned by **create_object()** should only be used after the reconfiguration step has been executed.

Object replacement can be done both on individual basis, i.e., by specifying **factory_creation_id**s, or on a type basis, i.e., by specifying the type of the objects. While replacement on individual basis provide a fine-grained control over the version of each object in the system, its use should be avoided when all objects of a type can be replaced simultaneously. Reconfiguration on a type basis simplifies version management, by preventing objects of the same type from having different versions.

The operation **replace_object()** requires the user to specify the object being replaced and the criteria to be used in the creation of the new version of the object. The criteria are used to identify a factory previously registered with the Reconfiguration Manager.

```
void replace_object(
    in FactoryCreationId factory_creation_id,
    in Criteria the_criteria
)
raises (
    ObjectNotFound,
    NoFactory,
    InvalidCriteria,
    InvalidProperty,
    CannotMeetCriteria
);
```

The operation **replace_type()** requires the user to specify the type being replaced, the new type and the criteria to be used in the creation of the new version of objects. The new type must be identical or derived from the original type. Requests for the creation of objects of a type being replaced are deferred until the end of reconfiguration. After reconfiguration, the identifier of the original type can still be used when requesting object creation, so that type replacements with sub-typing can be transparent for the client application. Nevertheless, the new derived type is used for the actual creation. **replace_type()** returns the list of objects replaced.

```
FactoryCreationIds replace_type(
      in TypeId current_type_id,
      in TypeId new_type_id,
      in Criteria the_criteria
)
raises (
      NoFactory,
      InvalidCriteria,
      InvalidProperty,
      CannotMeetCriteria
);
```

A type can be removed using the **remove_type()** operation.

```
void remove_type(
      in TypeId type_id
);
```

Object migration can be done both on individual basis, i.e., by specifying **factory_creation_id**s, or on a type-location basis, i.e., by specifying the type of the objects to be migrated and their current location. Migration on individual basis provides a fine-grained control over the location of each object in the system. The local factory that is used to create the relocated version of an object is determined by the criteria.

```
void migrate_object(
      in FactoryCreationId factory_creation_id,
      in Criteria the_criteria
)
raises (
      ObjectNotFound,
      NoFactory,
      InvalidCriteria,
      InvalidProperty,
      CannotMeetCriteria
);

FactoryCreationIds migrate_objects(
      in TypeId type_id,
      in Location origin,
      in Criteria the_criteria
)
raises (
      NoFactory,
      InvalidCriteria,
      InvalidProperty,
      CannotMeetCriteria
);
```

The default criteria for the creation of a type can be set by invoking **set_default_criteira()**. It influences the behavior of object creations after reconfiguration, e.g., by specifying the default location of new objects of the type.

```
void set_default_criteria(in TypeId type_id, in Criteria the_criteria);
```

The optional state translator for the reconfiguration step can be provided by invoking **set_state_translator()**.

```
void set_state_translator(
      in GenericStateTranslator translator
);
```

Requesting the execution of a reconfiguration step

A reconfiguration step can be executed in blocking mode, in which case the operation returns when the reconfiguration is complete (by invoking **commit()**), or in non-blocking mode (by invoking **deferred_commit()**), in which case the operation returns immediately. In the non-blocking mode, **is_completed()** should be invoked to determine whether the reconfiguration step has already been executed. **commit()** and **is_completed()** may raise a **ReconfigurationException** in case of errors during reconfiguration. The non-blocking mode is particularly useful for self-replacement, i.e., when the object that initiates the replacement is expected to be replaced. In this case, the blocking mode would lead to deadlock, since the object being replaced would have a pending request (**commit()**) and would never reach the idle state.

```
exception ReconfigurationException {};

void commit()
raises (ReconfigurationException);

void deferred_commit();

boolean is_completed()
raises (ReconfigurationException);

void dispose()
raises ( UnderReconfiguration );
```

5.2.4 State Translation

In the replacement operations, the change designer can optionally specify a state translator. The state translator is used by the DRS when the system has reached the safe state. In the safe state, all the states of the affected objects are consistent and stable. These states are used as input to the state translator, which translates them to the state of the objects being introduced to replace the affected objects.

A state translator is supplied by the change designer. A state translator implements the **GenericStateTranslator** interface. This interface defines the structure **Instance**, which comprises the type identifier, the reconfigurable object identifier, the state of a reconfigurable object instance and the reconfiguration operation being applied to it. The operation **types_supported()** returns the types supported by the state translator. In the absence of a supplied state mapping for a particular type, the identity function is used, i.e., the state is not modified. The operation **translate()** translates the states of a set of instances into derived states.

```
interface GenericStateTranslator {

   enum ReconfigurationOperationType {
         CREATION, REPLACEMENT, MIGRATION, DELETION
   };
```

```
    struct Instance
    {
       // and instance of an object is composed of its type,  its identity and its state
       TypeId type_id;                    // the type
       ReconfigurableObjectId id;     // the identity
       State the_state;                   // and the state
       ReconfigurationOperationType op_type;
    };
    typedef sequence<Instance> Instances;

    TypeIds types_supported();

    void translate(in Instances original, out Instances derived);

  };
```

The state of an object may include object references that are narrowed by a state translator. If a reference to be narrowed points to an object being replaced as part of a **replace_type()** with sub-typing, an unchecked narrow must be performed. Unchecked narrows have been incorporated in the CORBA standards with the introduction of CORBA Messaging [16]. For ORB implementations that do not support unchecked narrows, object references should be externalized as **CORBA::Object**s. These references should only be narrowed when first used by the new version of an object.

5.3 Application Developer's View

The *application developer* is expected to supply application-specific Reconfigurable-Object Factories and Reconfigurable Objects that co-operate with the Dynamic Reconfiguration Service.

5.3.1 Reconfigurable Objects

Reconfigurable Objects must implement the **ReconfigurableObject** interface, providing the state-access operations **get_state()** and **set_state()**, which are identical to the state-access operations in the Fault Tolerant CORBA specification. The state is encoded as a sequence of octets. The encoding of the state may be application-specific. Nevertheless, the application developer is strongly recommended to specify the state as a structure in IDL in order to guarantee interoperability and allow re-use of available CORBA functionality to encode data structures as sequences of octets (Common Data Representation [16]).

```
typedef sequence<octet> State;

interface ReconfigurableObject
{
   State get_state() raises(NoStateAvailable);

   void set_state(in State s) raises(InvalidState);
};
```

Active reconfigurable objects must also implement the **ActiveObject** interface, providing **passivate()** and **activate()** operations in addition to state-access operations. An active object may initiate non-nested requests, i.e., requests that are not causally related to incoming requests. An active object should react to the **passivate()** operation by refraining from initiating non-nested requests, i.e., by

exhibiting reactive behavior. The **activate()** operation is the inverse of **passivate()**, i.e., it informs an object that it is allowed to exhibit active behavior.

```
interface ActiveObject
{
   void passivate();

   void activate();
};
```

DRS Context Information

The DRS maintains some context information for the thread in which a request is being processed. This context information, called the DRS context, contains the invocation path of the request being treated and the identifier of the object treating the request. The DRS context is used in order to determine the invocation path that is sent implicitly with a request. Figure 22 shows the DRS context of a thread treating a request req_1. The DRS context contains the invocation path of req_1 ($\{O_1, \ldots O_N\}$) and the identifier of the object O_M treating req_1. The request req_2, which is a nested request of req_1, contains the invocation path of req_1 appended with the identifier O_M ($\{O_1, \ldots O_N, O_M\}$).

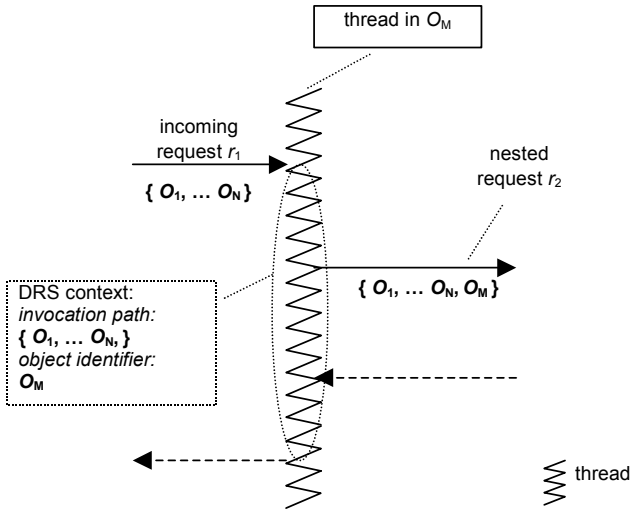


Figure 22 – The DRS context and the propagation of the invocation path

The DRS context is accessible through the **ReconfigurationCurrent** local object. An instance of the **ReconfigurationCurrent** object can be obtained by invoking **ORB::resolve_initial_references("ReconfigurationCurrent")**.

An active reconfigurable object must register each thread that issues non-nested requests with the DRS. This is done by using the operation **register_thread()** of the **ReconfigurationCurrent** object. The parameters for **register_thread()** are the identifier of the object adapter in which the object is located, and the object identifier used by this object adapter. For the Portable Object Adapter (POA) [16] these parameters are obtained through **POA::id** and **POA::reference_to_id()** respectively.

```
module ReconfigurationService {

   interface Current : CORBA::Current
   {
      void register_thread(in octets adapter_id, in octets object_id);
      …
```

When an incoming request is treated in only one thread, as depicted in Figure 22, the propagation of the invocation path is completely transparent for the reconfigurable object developer. In less conventional threading strategies, however, more support from the reconfigurable object developer is required, as explained in the remainder of this section.

In case the completion of an incoming request served in a thread T_1 depends on the completion of a nested request issued in another thread T_2, the DRS context information in thread T_1 must be transferred to thread T_2. This situation is depicted in Figure 23, where T_1 blocks waiting for nested request req_2 in T_2 to be processed.

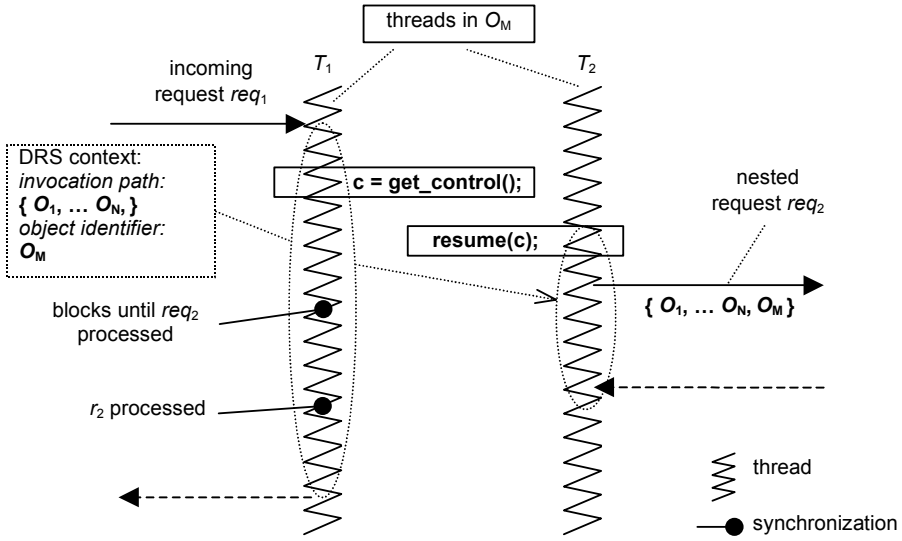


Figure 23 – Transferring DRS context information between threads

In this case, the **get_control()** operation of the **ReconfigurationCurrent** must be invoked in thread T_1 to obtain the **Control** structure that must be passed to the **resume()** operation of the **ReconfigurationCurrent** in thread T_2. The names **get_control()**, **resume()** and **Control** are adopted in order to resemble the Indirect Context Management with Explicit Propagation in the CORBA Transaction Service [23].

```
module ReconfigurationService {

   interface Current : CORBA::Current
   {
      void register_thread(in octets adapter_id, in octets object_id);

      struct CurrentSlotInfo
      {
        ReconfigurableObjectId id;
        ReconfigurableObjectIds invocation_path;
      };

      typedef CurrentSlotInfo Control;

      Control get_control();

      void resume(in Control which);
   };
};
```

The invocation path must also be propagated through non-reconfigurable objects that are in the invocation path between reconfigurable objects. If these non-reconfigurable objects are implemented with the unconventional threading strategies identified previously in this section, the object developer is responsible for transferring DRS context information between threads in the same way as required for reconfigurable objects with unconventional threading strategies.

5.3.2 Reconfigurable-Object Factories

Reconfigurable-Object Factories implement the **ReconfigurableObjectFactory** interface, which inherits the **GenericFactory** interface. These factories must provide **create_object()**, **delete_object()** and **get_reconfiguration_agent()** operations. The **get_reconfiguration_agent()** operation returns the **ReconfigurationAgent** associated to a given reconfigurable object.

```
interface ReconfigurableObjectFactory : GenericFactory
{
    ReconfigurationAgent get_reconfiguration_agent(in ReconfigurableObjectId id);
};
```

A Reconfigurable-Object Factory creates and deletes instances of objects on behalf of the Reconfiguration Manager, and registers and de-registers these instances with the Reconfiguration Agent.

Figure 24 depicts the participation of an object factory in the creation, replacement and migration of an object.

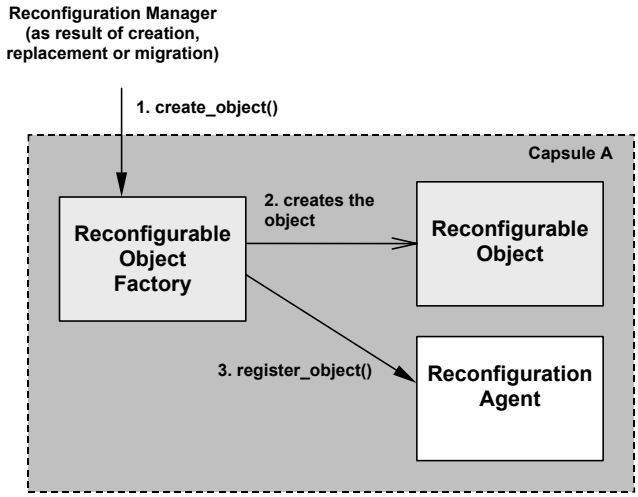


Figure 24 – Participation of Factory in Creation, Replacement and Migration

The **create_object()** operation is invoked by the Reconfiguration Manager (1) to create an instance of an object. **create_object()** may be invoked in the course of object creation, replacement or migration.

In the case of replacement or migration, the Reconfiguration Manager delegates the creation, by repeating the parameters supplied by the user and adding extra properties (name-value pairs) in the criteria parameter. These properties are the location-independent object reference to be used by the instance of the object (a property named **IOR**) and its reconfigurable object identifier (a property named **Id**). This allows the object to maintain its identity across subsequent reconfigurations, and publish the location-independent object reference as its object reference.

A reconfigurable object may retrieve its location-independent object reference and its reconfigurable object identifier from the Reconfiguration Agent, by invoking the **get_reference()** and **get_reconfigurable_object_id()** operations. A reference to the Reconfiguration Agent can be obtained by invoking **ORB::resolve_initial_references("ReconfigurationAgent")**. If the reconfigurable object invokes POA methods to retrieve its object reference, the POA supplies the conventional location-dependent object reference.

In case of an actual reconfigurable object creation, the Reconfiguration Manager includes in the criteria the **IOR** and the **Id** properties, and an extra **ApplicationObjectCreation** property. This allows the factory to distinguish, if necessary, between an actual object creation and a creation that results from replacement.

create_object() creates the instance of the object (2), registers it with the Reconfiguration Agent (3) and returns the location-dependent object reference to the Reconfiguration Manager.

```
interface ReconfigurationAgent
{
      typedef sequence<octet> octets;

      void register_object(
            in ReconfigurableObjectId id,
            in Object rec_obj_reference,
            in octets adapter_id, in octets object_id
      );

      void deregister_object(
            in ReconfigurableObjectId id
      );

      Object get_reference(
            in octets adapter_id, in octets object_id
      );

     ReconfigurableObjectId get_reconfigurable_object_id(
            in octets adapter_id, in octets object_id
      );

      boolean is_affected(
            in ReconfigurableObjectId id
      );
};
```

register_object() receives as parameters the identifier of the reconfigurable object and the location-independent object reference as sent by the reconfiguration manager, the identifier of the object adapter in which the object is located, and the object identifier used by this object adapter.

The **delete_object()** operation is invoked by the Reconfiguration Manager to delete an instance of an object. The execution of **delete_object()** may be invoked in the course of reconfigurable object removal, replacement or migration. If a factory finds it necessary to distinguish between object removal on the one hand, and replacement and migration on the other hand, it may invoke the operation **is_affected()** of the **ReconfigurationAgent** with the identifier of the object as a parameter. **is_affected()** returns **true** if the object is currently being replaced or migrated. The use of **is_affected()** allows us to maintain the syntax for **delete_object()** as defined in the Fault Tolerant CORBA specification.

Figure 25 depicts the participation of an object factory in the removal, replacement and migration of an object.

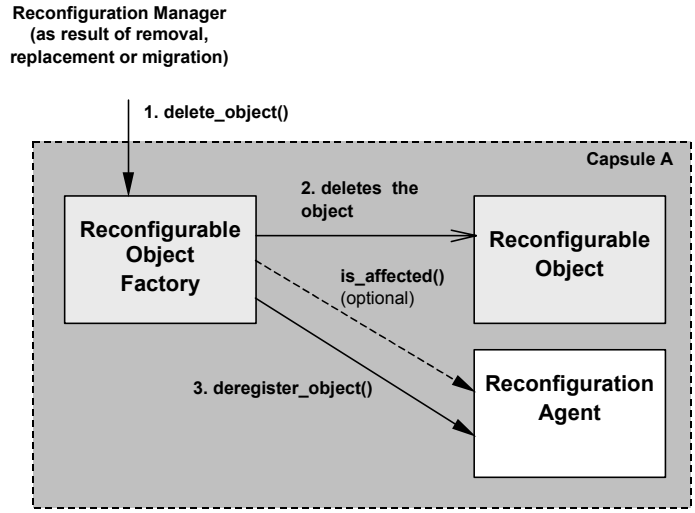


Figure 25 – Participation of Factory in Removal, Replacement and Migration

5.3.3 Clients View

The Dynamic Reconfiguration Service is transparent for client applications, which manipulate object references and issue requests to reconfigurable objects in the ways prescribed in the CORBA object model. During reconfiguration, requests may be queued by the ORB and re-directed to the target object, transparently for the client application.

One may think 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 target object processes the requests. This can be seen in example (1) of Figure 26. In addition, the order in which replies reach a client does not imply the order in which the server processed the requests. This can be seen in example (2) of Figure 26.

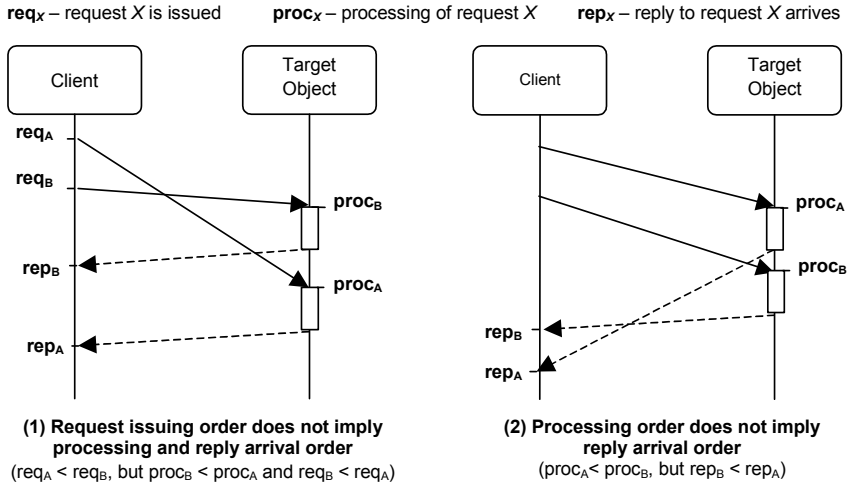


Figure 26 – CORBA ordering guarantees

Nevertheless, CORBA does guarantee that (i) the issuing of a request is eventually followed by the processing of the request, and that (ii) the processing of a request is eventually followed by the arrival of the reply at the client-side. A client can assume that requests are processed sequentially if it issues a request after the arrival of the reply of a previous request. Our DRS does not jeopardize these guarantees.

6 Design and Implementation

This chapter describes the design and implementation of the Dynamic Reconfiguration Service in CORBA. It presents the design and implementation choices made and discusses alternative implementations.

This chapter is further structured as follows: Section 6.1 discussed the implementation of the Location Agent to obtain location-independent object references, Section 6.2 discusses the implementation of selective queuing, and Section 6.3 describes how the DRS co-ordinates its components to perform reconfiguration operations, providing extra details on the implementation of each component. Finally, Section 6.5 presents an evaluation of the prototype implemented.

6.1 Location-independent Object References

In order to keep an object reference valid after reconfiguration, we make use of a Location Agent [6]. In case a request on a modified object reference is performed, an exception at the client ORB makes it contact the Location Agent, which uses of forward mechanism to inform a client ORB of the new location of the target object.

An alternative solution would be to preserve forwarding proxies in the location of the old targets. These forwarding proxies would forward requests to the new location (or version) of an object. The problem with this solution is that the forwarding chain grows each time an object is replaced or migrated. Therefore, when compared to the Location Agent solution, this approach introduces more overhead, is harder to manage and uses more system resources than necessary. Furthermore, locations where forwarding proxies exist cannot be actually taken off-line, e.g., in the case of a hardware upgrade.

Another solution requires the client-side ORB to be notified of the reconfiguration, substituting the current object reference with the new modified object reference. This solution can hardly be considered appropriate, since the communication overhead increases with the number of clients, which is often high. Furthermore, it would be necessary to keep trace of all clients of a

reconfigurable object. Keeping track of all clients would prevent the scalable implementation of the reconfiguration service. Object reference distribution in CORBA is not controlled by the ORB, since object references can be exchanged between objects by many different means.

6.1.1 Location Agent implementation

The Location Agent must fabricate object references that point to itself instead of pointing to the actual location of object. These object references are called location-independent object references. The Location Agent does this by creating an object reference that contains the location's agent address and the reconfigurable object identifier as the object-key.

When a request is issued by a client for the first time, the Location Agent is invoked. The Location Agent is implemented with a servant locator, which keeps a registry mapping a reconfigurable object identifier to the conventional location-dependent object reference. The servant location throws a **LocationForward** exception with the current location-dependent object reference that points to the current version of the object. This exception reaches the client ORB as a **LOCATION_FORWARD** GIOP (General Inter-ORB protocol [16]) message. As prescribed in the rules of GIOP, the client ORB reissues the request with the new object reference, until an error occurs when using this reference. Figure 27 depicts the basic functioning of the mechanism in the establishment of the binding.

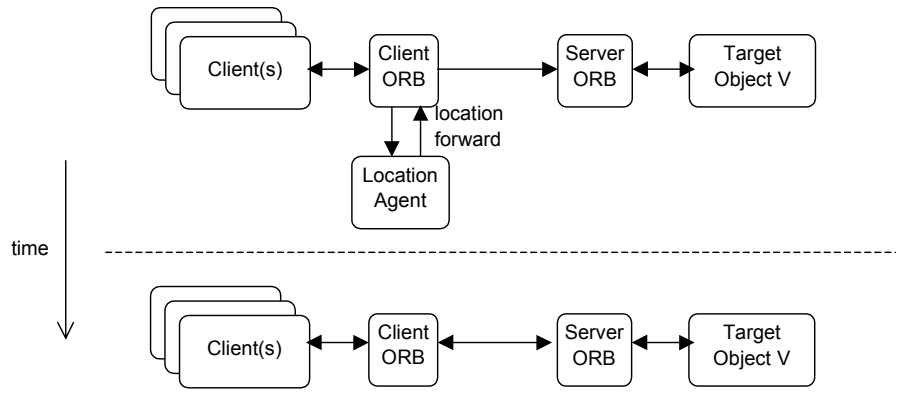


Figure 27 – Transparent binding establishment

When reconfiguration occurs, the reference being used by a client ORB is no longer valid. At this moment, GIOP mandates that the client ORB switches back to the original object reference, which in this case is the location-independent reference. The re-establishment of the binding follows the same procedure as in the first establishment, transparently for the client application. Figure 28 depicts the basic functioning of the mechanism in the re-establishment of a binding broken by reconfiguration.

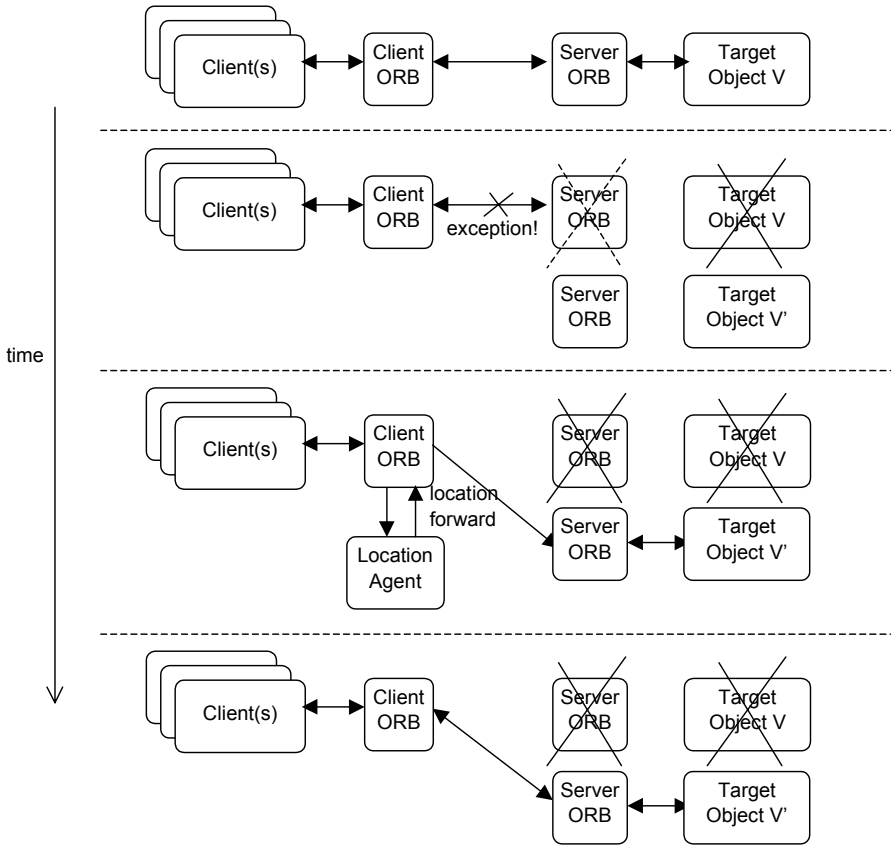


Figure 28 – Transparent binding re-establishment

This mechanism is fully transparent to the client application and the overhead for this solution is limited to the first invocation of a client on the reconfigured target object. The forward mechanism is already implemented in implementation repositories, although the interface between the implementation repository and the server ORB has not been standardized.

In our implementation, the Location Agent implements the **LocationAgentAdmin** interface, which allows the Reconfiguration Manager to register an object while retrieving its location-independent object reference (**register_object()**), get the location-independent object reference to an object (**get_reference()**), get the location-dependent object reference to an object (**get_target_object()**) and remove the current reconfigurable object identifier and location-dependent object reference association (**deregister_object()**).

```
module LocationAgent
{
    interface LocationAgentAdmin
    {
        Object register_object (
            in Object target,
            in ReconfigurationService::ReconfigurableObjectId id
        );

        Object get_reference (
            in ReconfigurationService::ReconfigurableObjectId id
        );

        Object get_target_object (
            in ReconfigurationService::ReconfigurableObjectId id
        );
```

```
            void deregister_object (
                in ReconfigurationService::ReconfigurableObjectId id
            );
        };
    };
```

6.2 Selective Request Queuing

In order to bring the system to the reconfiguration-safe state we have to implement a selective queuing of requests. Requests that do not belong to the 'laissez-passer' set should be queued transparently for clients and target objects. We identify two objects that realize selective queuing: a *selector* and a *queue*. These objects can be built in different ways: they can become internal ORB objects, CORBA objects or even request bridges.

6.2.1 Selector and Queue Objects

For each request directed to an affected object, the selector determines if the request belongs to the 'laissez-passer' set. If it does, the request is forwarded to the target object as in normal operation. Otherwise, the request is sent to the queue. Figure 29 shows this scheme, abstracting from the location of the objects in the middleware platform and the way they are built.

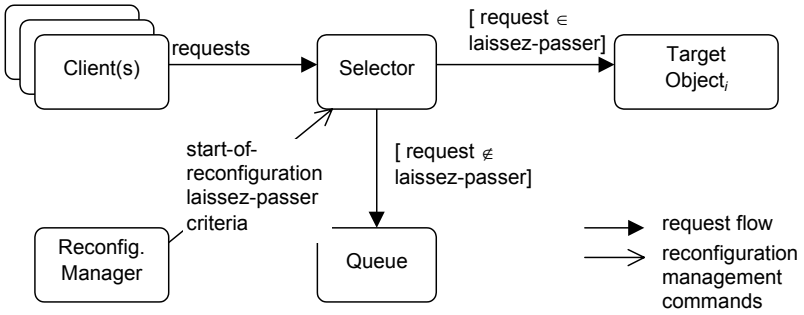


Figure 29 – Selector and Queue functions, reaching the safe state

The queue is responsible for storing requests until reconfiguration is complete. Stored requests are redirected to the new version of the target object after the reconfiguration, as depicted in Figure 30.

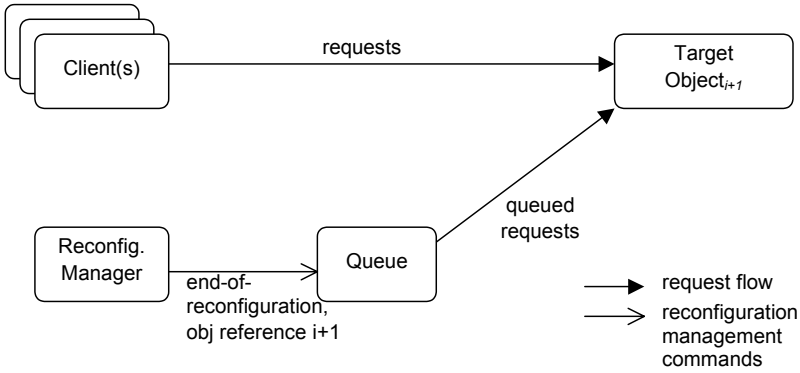


Figure 30 – Redirecting requests to new version of the implementation after reconfiguration

Initially we investigated the implementation of our approach as a service exclusively on top of the ORB, in which the selector and the queue would be combined to form a single CORBA object. Since the ORB threading strategy determines how threads are allocated to requests, and each request being handled by the combined selector/queue would block its thread, this solution is too dependent of the ORB threading strategy to work. Typically there would be as many threads as requests that have to be handled by the selector/queue object, such that this solution is not scalable. Therefore we ignore this alternative.

Another alternative is the embedding of the selector and the queue in the middleware infrastructure. In this alternative, a distributed application that uses the dynamic reconfiguration service consists of Reconfigurable Objects, Reconfigurable-Objet Factories and clients, which interact with a Reconfiguration Manager over a DRS-enabled ORB, as depicted in Figure 31. The DRS-enabled ORB realizes the selective queuing mechanism (introduced in Chapter 4) transparently for the application layer.

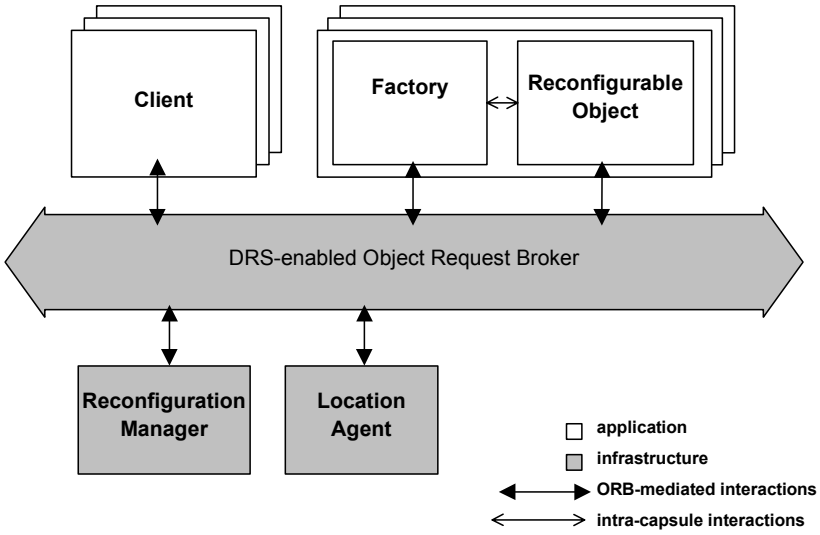


Figure 31 – Overview of Application with Reconfigurable Objects

6.2.2 Allocation of Selector and Queue Objects

Implementation alternatives can be generated by considering the allocation of the selector and the queue to different parts of the middleware infrastructure, namely the client-side ORB (client ORB) and the server-side ORB (server ORB). The benefits and drawbacks of each alternative are the following:

- *Pure client-side solution*. Selector and queue are implemented as extensions of the client ORB. Requests are selected and blocked at the client side, imposing no overhead to the communication infrastructure. Nevertheless, there is a serious scalability problem since all potential clients of an affected object must be known a priori, and all these clients must be notified of the set of affected objects. This drawback applies to all solutions that place the selector in the client ORB. Moreover, this solution complicates management, since the client ORB extensions have to be deployed in every potential client of the reconfigurable objects;

- *Pure server-side solution*. Selector and queue are implemented as extensions of the server ORB. This solution offers better scalability than the pure client-

side solution, as clients do not need to be known a priori and do not need to be informed of the reconfiguration. Since the client ORB does not have to be extended, management and deployment can be simplified;

- *Hybrid solution*. The selector and the queue are implemented as extensions of the server ORB and the client ORB, respectively. Clients that attempt to issue a request to an affected object are informed to block the request and re-issue it when they get a notification that the reconfiguration has been completed. In effect, the queue becomes distributed among all clients that attempt to issue a request to an affected object during reconfiguration. This solution requires more communication overhead than in the case of the pure server-side solution.

6.2.3 ORB Instrumentation

The solutions discussed above imply that the ORB has to be extended somehow, i.e., the ORB has to be instrumented. This can be realized by either making proprietary modifications to the ORB code or by using some kind of request reflection [33], in which we can operate on the reified request object. Request reflection can be implemented using interceptors (or filters), which are supported by most commercial ORBs. Interceptors make it possible to add new functionality to an ORB without altering or accessing the source code of the ORB. Interceptors offer standard support for extending ORB functionality. OMG is currently finalizing the CORBA Portable Interceptors standard [20], which offers somewhat limited but portable instrumentation capabilities for ORB implementations. The pure client-side solution can be directly implemented using portable interceptors in the client ORB. The implementation of the pure server-side solution with portable interceptors has the same kind of problems with respect to threading as the implementation as a service on top of the ORB discussed above. The hybrid solution can be directly implemented using portable interceptors in the client ORB and in the server ORB. In section 6.2.4, we present our implementation of the hybrid solution.

The selector can use the service context of a request to determine a request belongs to the ‘laissez-passer’ set. Service contexts allow implicit arguments to be passed in a method invocation. When a reconfigurable object issues a request, it adds its identifier to the service context. During the first stage of the reconfiguration process, when a request arrives at the selector the request’s service context is inspected. If the identifier of an affected object is included in the service context, the request belongs to the ‘laissez-passer’ set and should not be queued.

6.2.4 Solution based on Portable Interceptors

The implementation considered in this section is based on the use of portable interceptors [20] to extend the functionality of the ORB. Portable interceptors allow the extension of the ORB through a limited request reflection mechanism in an ORB-independent manner. Interceptors allow a service to reify requests at specific interception points.

Figure 32 depicts an overview of the implementation with a brief description of the actions undertaken at the client- and server-side request interception points.

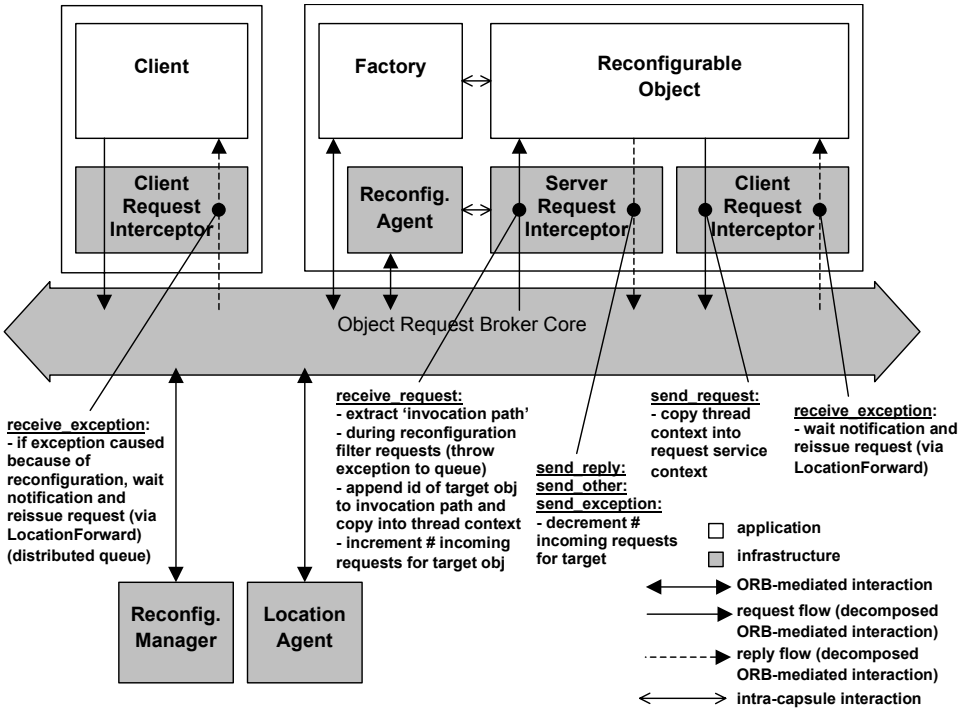


Figure 32 – Elements of the implementation and request reification points

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.

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 request interceptors, which block the thread of execution and reissues the blocked requests later by raising a **LocationForward** exception.

We consider two different policies for determining the moment in which a request is reissued: the *wait-and-retry* policy and the *wait-for-notification* policy. The policy is determined by the Reconfiguration Manager prior to reconfiguration. In the *wait-and-retry* policy, the Reconfiguration Manager sends a time interval to the Reconfiguration Agents of the affected objects. This time interval is passed in the reply service context of the exception that is sent to clients during reconfiguration. The client-side request interceptor waits for the time interval specified and reissues the request. If the reconfiguration is not over yet, the server-side request interceptors will raise the exception again, and the client-side request interceptor will block for the time internal again. In the *wait-for-notification* policy, the Reconfiguration Manager creates a **ReconfigurationManagerCallback** object before the start of a reconfiguration. The Reconfiguration Manager sends

this reference to the Reconfiguration Agents of the affected objects. This reference to the callback object is passed in the reply service context of the exception that is sent to clients during reconfiguration. The client-side request interceptor invokes the **block_until_ready()** method of the **ReconfigurationManagerCallback** object, which blocks until the end of the reconfiguration. The client application is not at any moment aware of the reconfiguration, potentially observing an increase in the response time of invocations that are queued waiting for reconfiguration.

One may think that the use of the event service (or the notification service) would be appropriate for the communication between Reconfiguration Manager and blocked clients. In this case, the Reconfiguration Manager would send an event with the semantics "the reconfiguration is over" to all blocked clients. In Figure 33, we show an execution in which the use of the event service would prevent the client-side ORB from unblocking. In this execution, the client's **pull()** request reaches the event channel after the event that indicates the end of reconfiguration. According to the specification of the event service, the event is not delivered, and the client is left waiting for a response to **pull()** indefinitely.

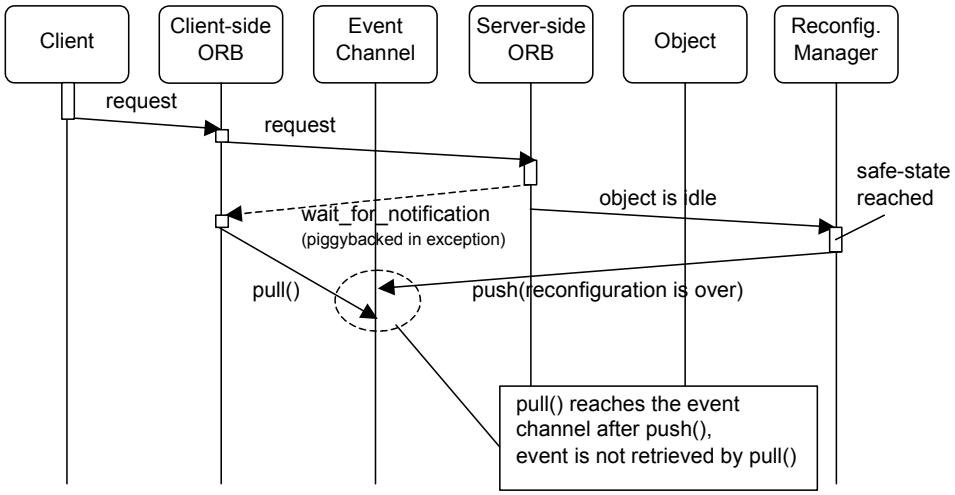


Figure 33 – Problem using the event service for notifying clients

6.3 Performing Reconfiguration Steps

The DRS components, namely the Reconfiguration Manager, the Location Agent and the Reconfiguration Agents, cooperate to perform a reconfiguration step. In the sequel we detail the activities executed to perform simple and composite reconfiguration steps.

6.3.1 Object Creation

Figure 34 shows the creation of an object. 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). After that, 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).

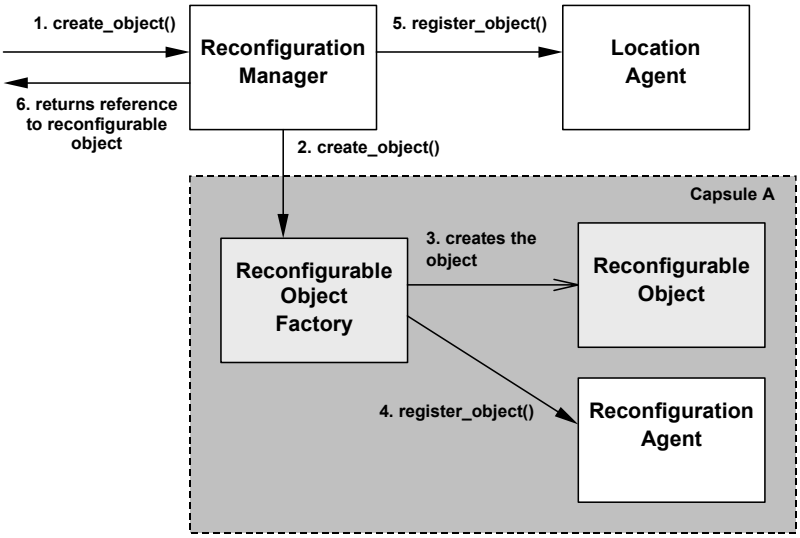


Figure 34 – Object Creation

No special support is required from the ORB for performing this step.

6.3.2 Object Replacement

Figure 35 shows the replacement of an active object. Initially, the Reconfiguration Manager delegates the creation of the new version of the object to a local Reconfigurable Object Factory (2). After that, 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 object is allowed to exhibit active behavior (10), the new location of the object is registered with the Location Agent (11), and the local factory is requested to remove the previous version of the object (12). In Figure 35 we do not show state translation that may take place.

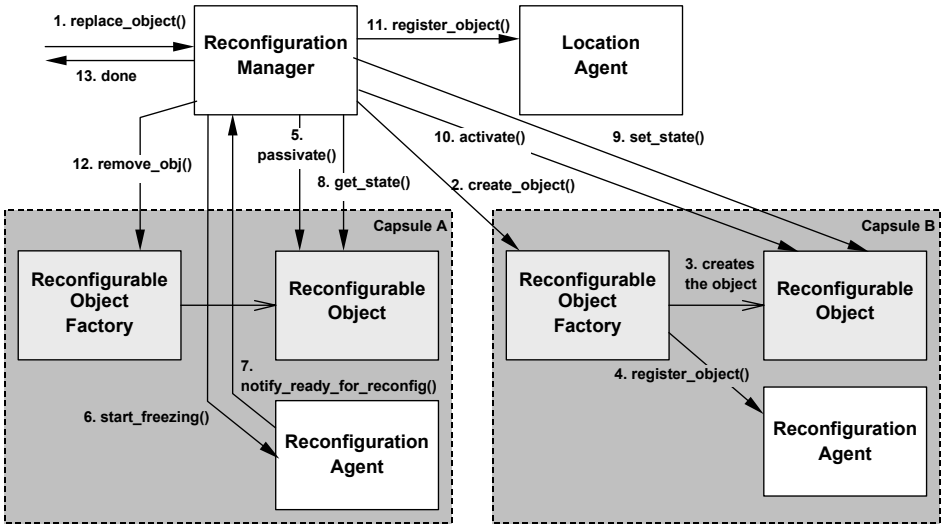


Figure 35 – Object Replacement

6.3.3 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.

6.3.4 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.

6.3.5 Composite Reconfiguration Steps

The procedures for the execution of simple reconfiguration steps described in sections 6.3.1 to 6.3.4 are special cases of the procedure to execute a composite reconfiguration step. This procedure consists of the following activities:

1. for each object creation, migration and replacement, the Reconfiguration Manager invokes the **create_object()** operation of the appropriate local object factory (determined by type and criteria);

2. the Reconfiguration Manager invokes the **passivate()** operation of all active objects in the affected set;

3. the Reconfiguration Manager invokes the **start_freezing()** operation of all active objects in the affected set. The parameters of **start_freezing()** include the set of affected objects and the information for the queuing policy adopted;

4. all the affected objects eventually reach the idle state and the Reconfiguration Agents invoke the **notify_ready_for_reconfig()** operation on the **ReconfigurationManagerCallback** object. The safe-state is reached;

5. the Reconfiguration Manager reads the states of the affected objects, translates them with the **translate()** operation of the state translator, when this is supplied, and sets the states of the new or relocated versions;

6. the Reconfiguration Manager (re-) registers the location-dependent object reference with the location agent;

7. the Reconfiguration Manager invokes the **activate()** operation of the new or relocated versions;

8. the client-side ORBs are notified that the reconfiguration is over, in case reissuing policy is *wait-for-notification*; and

9. for each object removal, migration and replacement, the Reconfiguration Manager invokes the **delete_object()** operation of the local object factory that holds the version to be discarded.

6.4 Portability and Interoperability Considerations

In order to guarantee the portability of applications built using the DRS, an implementation of the DRS must provide the interfaces presented in Chapter 5. These are, namely: **ReconfigurationManager**, **GenericFactory**, **FactoryManager**, **ReconfigurableObjectFactory**, **ReconfigurableObject**, **ActiveObject**, **ReconfigurationAgent** and **ReconfigurationCurrent**. The implementation of the DRS should not use proprietary ORB interfaces.

The interfaces **ReconfigurationAgentAdmin**, **LocationAgent**, and **ReconfigurationManagerCallback** are internal interfaces. These interfaces are used for the interaction between parts of the DRS and are specific to an implementation of the DRS. In order to guarantee the interoperability between the Reconfiguration Manager, Reconfiguration Agents, DRS-ready client ORBs and DRS-ready server ORBs from different vendors, these internal interfaces should be standardized, as well as the exceptions used to indicate reconfiguration, and the identifier and structure of the DRS service context.

Figure 36 shows the interfaces that should be implemented to guarantee the portability of applications built using the DRS, and the internal interfaces.

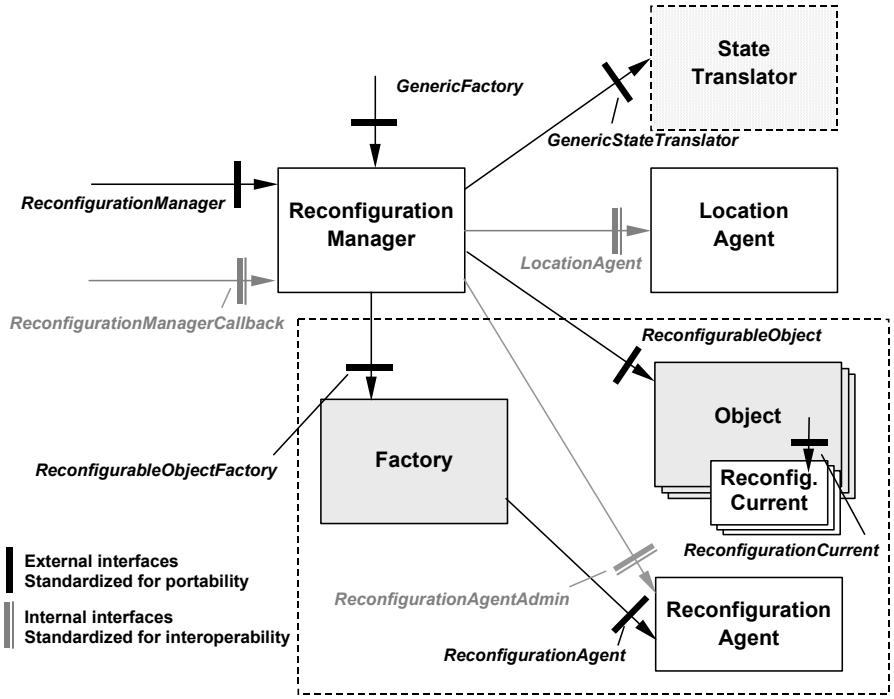


Figure 36 –Interfaces that should be standardized for portability and interoperability

6.5 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 [26].

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 this section we

present the results obtained from these tests. This section gives an estimation of the overhead introduced by the Dynamic Reconfiguration Service during normal operation and an estimation of the impact of reconfiguration on execution.

6.5.1 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 approximate R by considering that it consists 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 $\Delta_{application} \cong 0$, 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 2 (values are the average of 4 x 10^4 invocations):

Table 2 – 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%

Figure 37 shows these results in a graphical representation:

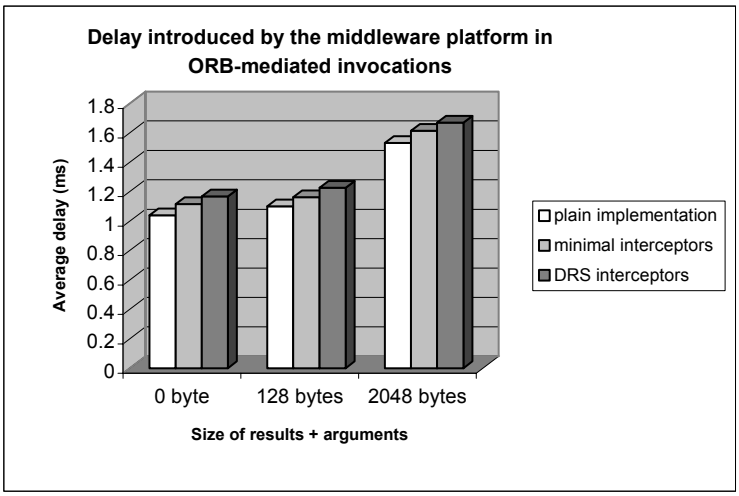


Figure 37 – Normal and increased response times

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 expect that the overhead of the reconfiguration service will be acceptable.

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.

6.5.2 Impact on execution during reconfiguration

In Section 4.4.4 we have stated that the increase in response time is upper-bounded by the duration of the longest pending invocation in the set of affected objects at the moment the reconfiguration starts. Nevertheless, we have neglected the overhead introduced by the reconfiguration service for coordinating the reconfiguration.

The delay introduced by the reconfiguration service forms 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. However, these values can be reduced once we optimize some code segments, something that we have not done yet.

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. Since queued requests are all delivered to a new object right after reconfiguration, this object may get overloaded and the performance of the overall system can be affected.

7 Usage Examples

This chapter discusses the use of the dynamic reconfiguration service with two different application scenarios. The objectives of this chapter are twofold: to validate the use of the dynamic reconfiguration service in different application scenarios, and to provide extended examples to reconfigurable object developers, complementing the description of the interfaces of the service in Chapter 5.

This chapter is further structured as follows: Section 7.1 presents the reconfiguration of a banking application and Section 7.2 presents a load-balancing manager based on object migration.

7.1 Reconfiguration of a Banking Application

The objective of this example is to illustrate the usage of the DRS to perform a sequence of reconfiguration steps to a simple banking application. The reconfiguration steps considered are object creation, multiple object replacements with change in internal implementation (both without state conversion and with state conversion), replacements with interface sub-typing, and a composite reconfiguration step, which consists of multiple replacements and creation.

7.1.1 Initial Configuration

Initially, the banking application consists of **BankAccount** reconfigurable objects and a **ReportGenerator** reconfigurable object.

The bank account provides methods to consult the current balance (**getBalance()**), withdraw (**withdraw()**), deposit (**deposit()**), get the history of balances (**getHistory()**), get the maximum overdrawn limit allowed for the account (**getMaxOverdrawnAllowed()**).

The factory for a bank account implements the **BankAccountFactory** interface, which is derived of the **ReconfigurableObjectFactory** interface.

```
module Banking
{
    interface BankAccount : ReconfigurationService::ReconfigurableObject
    {
        float getBalance();
        void withdraw(in float amount);
        void deposit(in float amount);

        typedef sequence<float> floatSeq;
        floatSeq getHistory();
        float getMaxOverdrawAllowed();
    };
    interface BankAccountFactory
            : ReconfigurationService::ReconfigurableObjectFactory
    {
    };
};
```

The bank account is a stateful object. The state has been defined as a structure in IDL.

```
module Banking
{
    typedef sequence<float> floatSeq;
    struct BankAccountState_initial
    {
        float balance;
        floatSeq history;
    };
};
```

The report generator implements the **ReportGenerator** interface, which provides an operation to generate a report for a given account. The report generator is stateless.

```
module Banking
{
    interface ReportGenerator : ReconfigurationService::ReconfigurableObject
    {
        string report(in BankAccount account);
    };
    interface ReportGeneratorFactory
        : ReconfigurationService::ReconfigurableObjectFactory
    {
    };
};
```

Figure 38 shows the initial configuration of the banking application.

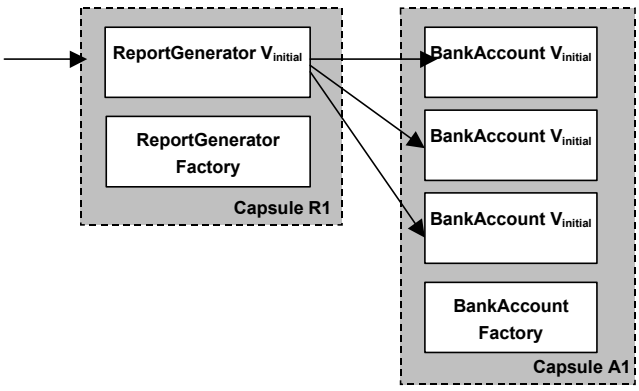


Figure 38 – Initial Configuration of the Banking Application

The objects have been implemented in Java. We show the implementation of the operations which are relevant to the application developer. These are **BankAccount**'s **get_state()** and **set_state()** and the **BankAccountFactory**'s **create_object()** and **delete_object()**.

The **get_state()** operation returns the state of the object encoded as a sequence of octets. In our example, we re-use CORBA's CDR operations to encode the state in an interoperable way. Initially, a **CodecFactory** is obtained. After that, The **CodecFactory** is used in order to create a **Codec** that implements the desired CDR encoding. The state is obtained from the application, in this case local Java calls to **this.getBalance()** and **this.getHistory()**, and it is placed into the structure **BankAccountState_initial** that defines the state in IDL. This structure is placed into a CORBA **Any**, and the operation **encode()** of the **Codec** is invoked in order to encode this **Any**. The implementation of **get_state()** is shown below. Exception handling is omitted in this section.

```
public byte[] get_state()
 {
    org.omg.IOP.CodecFactory codec_factory =
       org.omg.IOP.CodecFactoryHelper.narrow(
         orb.resolve_initial_references("CodecFactory"));

    org.omg.IOP.Encoding cdr_encoding =
       new org.omg.IOP.Encoding(
          org.omg.IOP.ENCODING_CDR_ENCAPS.value, (byte)1, (byte)0);

    org.omg.IOP.Codec codec = codec_factory.create_codec(cdr_encoding);

    org.omg.CORBA.Any any = this._orb().create_any();

    BankAccountState_initial state = new BankAccountState_initial(
              this.getBalance(), this.getHistory());
    BankAccountState_initialHelper.insert(any, state);

    byte result[] = codec.encode(any);
    return result;
}
```

IOP codec is obtained

Any is created with the state of the object

Any is encoded with the codec

The implementation of the **set_state()** operation is symmetric to the implementation of **get_state()**. Initially, a **CodecFactory** is obtained. After that, the **CodecFactory** is used in order to create a **Codec** that implements the desired CDR encoding. The encoded state is decoded with the operation **decode()** of the **Codec**, which returns an **Any**. The structure **BankAccountState_initial** is extracted from this **Any**, and the state can be restored to the application. The implementation of **set_state()** is shown below.

```
public void set_state(byte s[])
{
    ...

    org.omg.CORBA.Any any = codec.decode(s);

    BankAccountState_initial state = BankAccountState_initialHelper.extract(any);
```

IOP codec is obtained (see get_state()

Any is decoded and **BankAccountState_impl** extract from it

```
        this.balance = state.balance;

        this.history.clear();
        for(int i = 0; i < state.history.length; i++)
            this.history.add(new Float(state.history[i]));
    }
```

state is restored to the object

The application developer must also supply an implementation of the local factory. The **create_object()** operation of the local factory creates objects on behalf of the Reconfiguration Manager. Initially, the parameters for creation are extracted from the criteria. This includes the location-independent object reference, the reconfigurable-object identifier and the **ApplicationObjectCreation** property in case of an application object creation. After that, the servant is created, the CORBA object is activated. The object is registered with the Reconfiguration Agent (a reference to the Reconfiguration Agent has been obtained previously by invoking **ORB::resolve_initial_references("ReconfigurationAgent")**), and the factory creation identifier is returned. The implementation of **create_object()** is shown below.

```
public org.omg.CORBA.Object
   create_object(String type_id,
                 Property[] the_criteria,
                 org.omg.CORBA.AnyHolder factory_creation_id)
      throws NoFactory,
             ObjectNotCreated,
             InvalidCriteria,
             InvalidProperty,
             CannotMeetCriteria
{
    // search for "Id" in the properties
    long id = extract_id_criteria ( the_criteria );

    // search for "IOR" in the properties
    org.omg.CORBA.Object obj_being_created = extract_ior_criteria ( the_criteria );

    // search for "ApplicationObjectCreation" in the properties
    Boolean first_version = extract_app_obj_creation_criteria ( the_criteria );

    org.omg.CORBA.Object result = null;

    result = (new BankAccount_initial_impl(orb))._this_object(orb);

    byte object_id[], adapter_id[];

    object_id = _poa().reference_to_id(result);
    adapter_id = _poa().id;

    rec_agent.register_object(id, obj_being_created, adapter_id, object_id);

    registry.put(id, object_id, servant );

    factory_creation_id.value = _orb().create_any();
    factory_creation_id.value.insert_longlong(id);

    return result;
}
```

the parameters for creation are extracted from the criteria

the servant is created, and CORBA object is activated

the object is registered with the Reconfiguration Agent

factory-specific registration

factory creation id is returned (for later **delete_object()**)

The **delete_object()** operation of local factory deletes objects on behalf of the Reconfiguration Manager. The object is de-registered with the Reconfiguration Agent and the CORBA object is de-activated.

```
public void delete_object(org.omg.CORBA.Any factory_creation_id)
    throws ObjectNotFound
{
    long id = factory_creation_id.extract_longlong();
    rec_agent.deregister_object(id);
    this._poa().deactivate_object(registry.getObjectId(id));
}
```

the object is deregistered

the object is deactivated

7.1.2 Multiple Replacements

The initial implementation is defective in that the history grows indefinitely, using up more resources that necessary. Therefore we substitute it by a version which limits the number of balances stored. Since the **set_state()** operation of the new version truncates the number of balances stored, no state translation is necessary. Our first reconfiguration step consists of the replacement of all objects of type **BankAccount** from this version $V_{initial}$ by this new version V_{new}, as depicted in Figure 39.

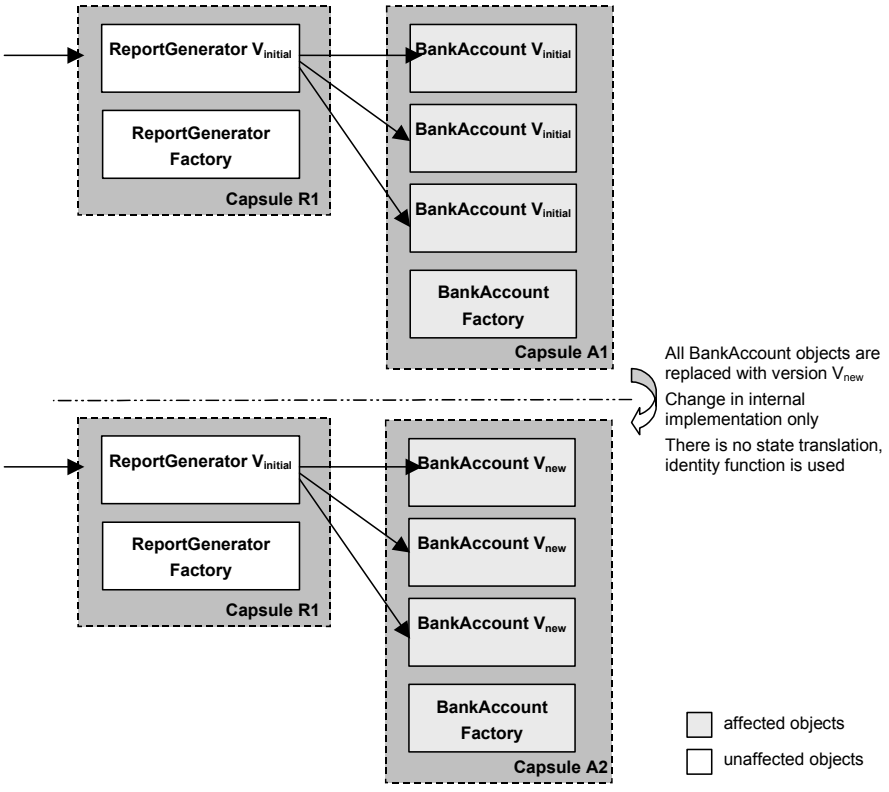


Figure 39 – Composite Reconfiguration Step: BankAccount objects are replaced

Before defining the reconfiguration step, the new factory is added with the Reconfiguration Manager. Initially, both the location (**drs.Location**) and version identifier (**drs.VersionId**) properties are set. The location refers to the location where the new factory creates objects and the version identifier refers to the version of the objects created. These properties form the criteria for object

creation using this factory. A reference to the factory is obtained and the factory is added by invoking **add_factory()** on the **FactoryManager** interface. The code to add the factory of objects of version V_{new} is given below.

```
NameComponent version_prop = new NameComponent[1];
version_prop[0] = new NameComponent("drs.VersionId","");

Any version_val = orb.create_any();
version_val.insert_float(1.1);
```

the version property of the criteria is set

```
NameComponent location_prop = new NameComponent[1];
location_prop[1] = new NameComponent("drs.Location","");

Any location_val = orb.create_any();
location_val.insert_string("");
```

the location property of the criteria is set

```
Property[] the_criteria = new Property[2];

the_criteria[0] = new Property(version_prop, version_val);
the_criteria[1] = new Property(location_prop, location_val);
```

the_criteria is created with both properties

```
ReconfigurableObjectFactory factory =
    ReconfigurableObjectFactoryHelper.narrow(
    orb.string_to_object(readReference("BankAccount_new_implFactory.ref")));
```

a reference to the factory is obtained

```
FactoryInfo finfo =
        new FactoryInfo(factory, location_prop, the_criteria);

String[] type_ids = new String[1];
type_ids[0] = BankAccountHelper.id();

rec_mgr.add_factory(finfo, type_ids);
```

parameters of **add_factory()** are prepared, factory is added

The reconfiguration step is created invoking the **create_reconfiguration()** operation of the Reconfiguration Manager. The identifier of the new version is placed in the criteria. **replace_type()** is invoked to request the replacement.

```
Property[] the_criteria = new Property[1];

NameComponent version_prop = new NameComponent[1];
version_prop[0] = new NameComponent("drs.VersionId","");

org.omg.CORBA.Any version_val = orb.create_any();
version_val.insert_float(1.1);

the_criteria[0] = new Property(version_prop, version_val);
```

the criteria are prepared, with the new version specified

```
ReconfigurationStep step = rec_mgr.create_reconfiguration_step();
step.replace_type(
      BankAccountHelper.id(), BankAccountHelper.id(), the_criteria);
step.commit();
```

a reconfiguration step is created and replacement requested

7.1.3 Multiple Replacements and Creation

In our second reconfiguration step, all **BankAccount** objects are replaced, and a **CreditCentral** object is created. The new version of **BankAccount** ($V_{factored}$) delegates requests to **getMaxOverdrawAllowed()** to the Credit Central. The Credit Central makes a re-entrant invocation to the **BankAccount** object to get the history of balances and calculates the overdrawn limit.

```
interface CreditCentral
{
    float getMaxOverdrawAllowed(in BankAccount account);
};
```

This reconfiguration step is depicted in Figure 40.

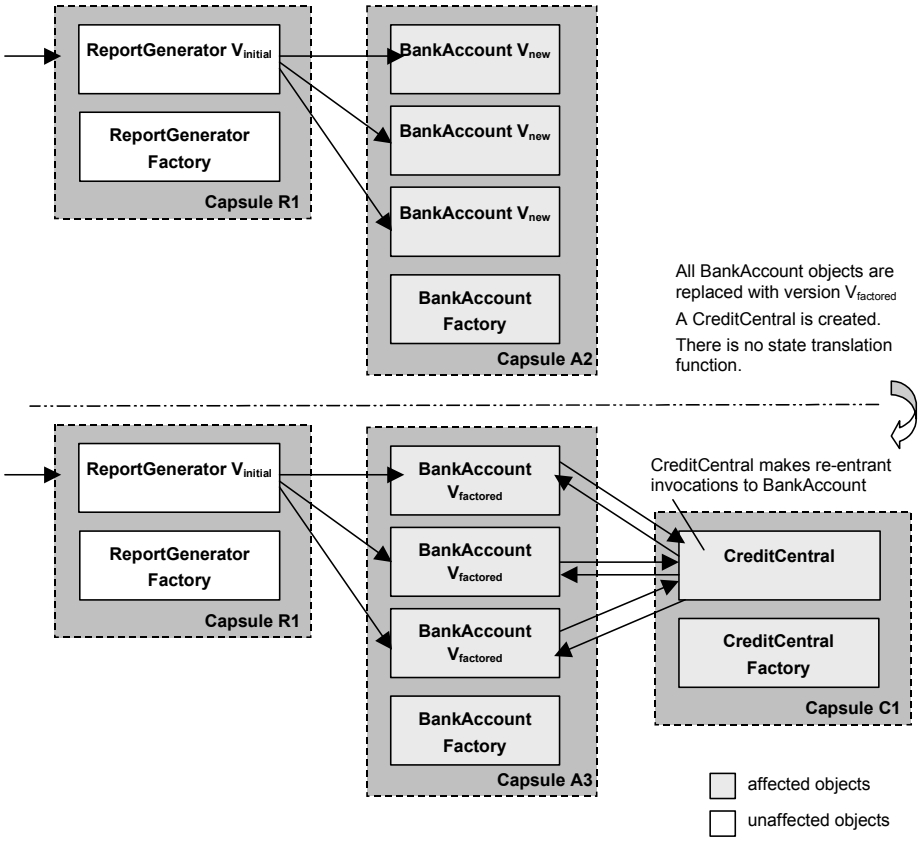
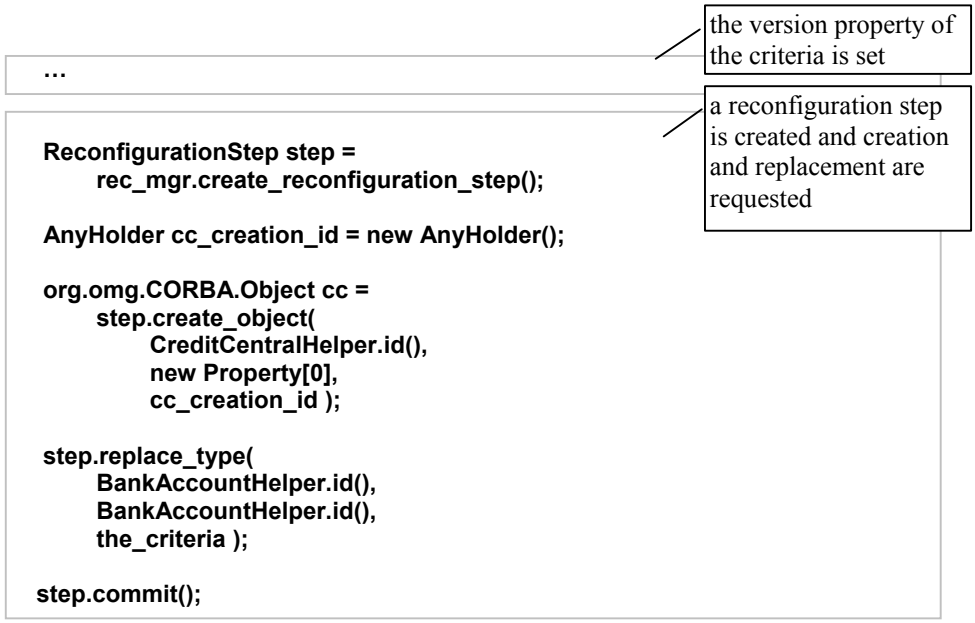


Figure 40 – Composite Reconfiguration Step: BankAccount objects are replaced and CreditCentral singleton created.

Before defining the reconfiguration step, the new factories are registered with the Reconfiguration Manager, as exemplified in Section 7.1.3.

The code that specifies this reconfiguration step is given below. The criteria for the replacement contain the identifier of the new version. The reconfiguration step is created and object creation and replacement are requested.




```
...

ReconfigurationStep step =
      rec_mgr.create_reconfiguration_step();

AnyHolder cc_creation_id = new AnyHolder();

org.omg.CORBA.Object cc =
      step.create_object(
            CreditCentralHelper.id(),
            new Property[0],
            cc_creation_id );

step.replace_type(
      BankAccountHelper.id(),
      BankAccountHelper.id(),
      the_criteria );

step.commit();
```

7.1.4 Multiple Replacements with Sub-Typing

In our third reconfiguration step, both the **ReportGenerator** and the **BankAccount** objects are replaced. The version $V_{factored}$ of **BankAccount** is replaced by V_{euro}. V_{euro} implements a sub-type of **BankAccount**, namely **EuroBankAccount**. V_{euro} stores the balance and history in Euros, while $V_{factored}$ stores the balance and history in Dutch Guilders. Therefore, state conversion is required. The new version of **ReportGenerator**, V_{euro} uses the derived interface to print reports in Euros by invoking the **getBalanceEuro()** operation. The interface of the Report Generator remains unchanged. The **report()** operation narrows the **BankAccount** parameter to from **BankAccount** to **EuroBankAccount**.

```
interface EuroBankAccount : BankAccount
{
      float getBalanceEuro();
};

interface EuroBankAccountFactory
      : ReconfigurationService::ReconfigurableObjectFactory
{
};
```

The reconfiguration step is depicted in Figure 41.

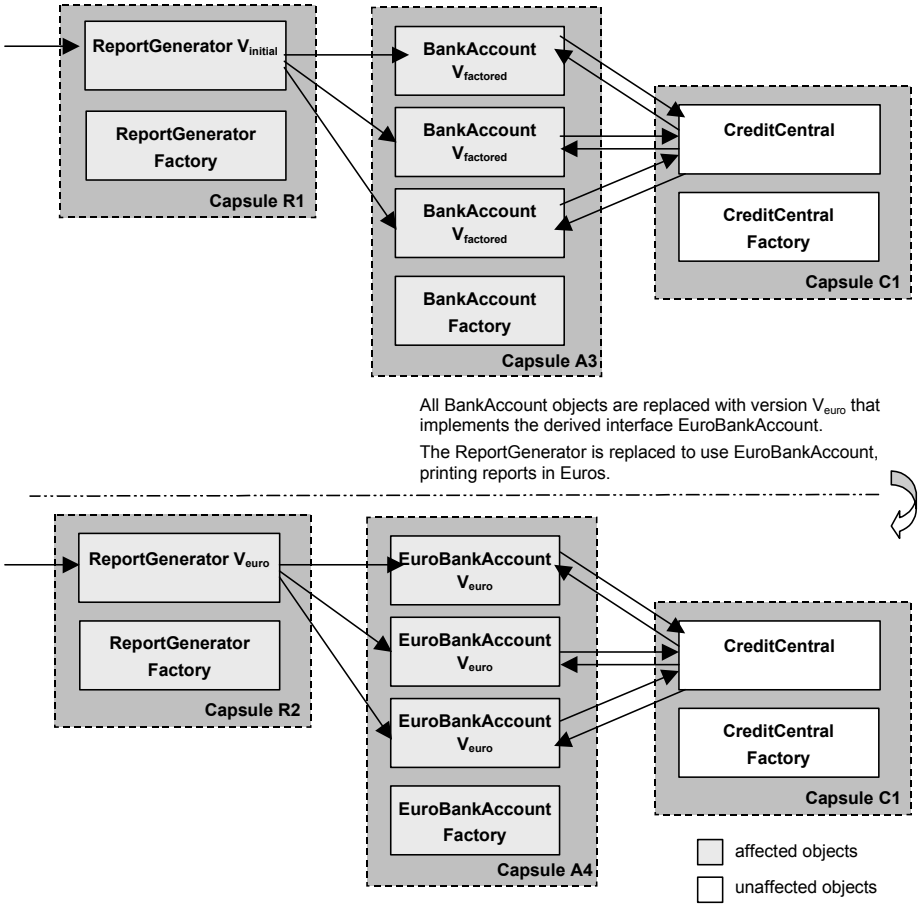


Figure 41 – Composite Reconfiguration Step with Replacements with Sub-Typing

Before defining the reconfiguration step, the new factories are registered with the Reconfiguration Manager, as exemplified in Section 7.1.3. The criteria for the replacements are defined as exemplified in Section 7.1.2.

The code that specifies this reconfiguration step is given below. A reference to the state translator is obtained. The state translator is used as a parameter for the operation **set_state_translator()**.

a reference to the state translator is obtained, and the state translator is associated to the step

```
GenericStateTranslator translator =
      GenericStateTranslatorHelper.narrow(
      orb.string_to_object(
      readReference("BankStateTranslator_factored_to_euro.ref")  ));

ReconfigurationStep step = rec_mgr.create_reconfiguration_step();

step.set_state_translator(translator);

step.replace_type(
      BankAccountHelper.id(),
      EuroBankAccountHelper.id(),
      the_criteria_acc );

step.replace_type(
      ReportGeneratorHelper.id(),
```

```
            ReportGeneratorHelper.id(),
            the_criteria_rg );

        step.commit();
```

The code for the state translator's **translate()** function is given below. The function decodes the state of an instance of **BankAccount**, converts both the current balance and the account history from Dutch Guilders to Euros and encodes the state appending it to the output array **derived**.

```
public void  translate(Instance[] original, InstanceSeqHolder derived)
{
    derived.value = new Instance[original.length];
    for (int i = 0; i < original.length; i++)       // for each instance
    {
     if (original[i].type_id.equals(BankAccountHelper.id()))
     {
        Any any = codec.decode(original[i].the_state);

        BankAccountState_factored original_state =
           BankAccountState_factoredHelper.extract(any);

        EuroBankAccountState derived_state =  new EuroBankAccountState();

        derived_state.balance = original_state.balance * NLG_TO_EUR;
        derived_state.history = new float[original_state.history.length];
        for (int j = 0; j < original_state.history.length; j++)
              derived_state.history[j] = original_state.history[j] * NLG_TO_EUR;

        Any any_derived_state = this._orb().create_any();
        EuroBankAccountStateHelper.insert(any_derived_state, derived_state);
        byte encoded_derived_state[] = codec.encode(any_derived_state);

        derived.value[i] = new Instance(
            original[i].type_id,  original[i].id,  encoded_derived_state);
     }
     else {
        derived.value[i] = original[i];     // use identity
      }
    }
}
```

decode the state

actual state translation

encode the derived state

place the derived state in the output variable

7.1.5 Conclusions

With this example, we have shown that composite reconfiguration steps can be supported by the DRS. We have shown (i) how simple reconfigurable objects and reconfigurable object factories can be implemented and (ii) how to request the execution of different composite reconfiguration steps. The example shows code fragments which can be re-used in other reconfiguration situations.

This example is a simplified version of an example that has been implemented and executed with the prototype of the DRS, so that it constitutes a simple proof-of-concept.

7.2 *Load-balancing Manager based on Migration*

This section describes a load-balancing manager that uses object migration to balance the load across a set of locations. We do not aim at proposing a load-balancing mechanism in this section, but only showing the use of the migration facilities of our DRS in a realistic application example.

Figure 42 shows an overview of the load-balancing architecture. This architecture consists of one central Load Manager and several Load Agents, one for each location available. The Load Manager regularly polls each registered Load Agent, which estimates the load of a location and returns this estimate to the Load Manager.

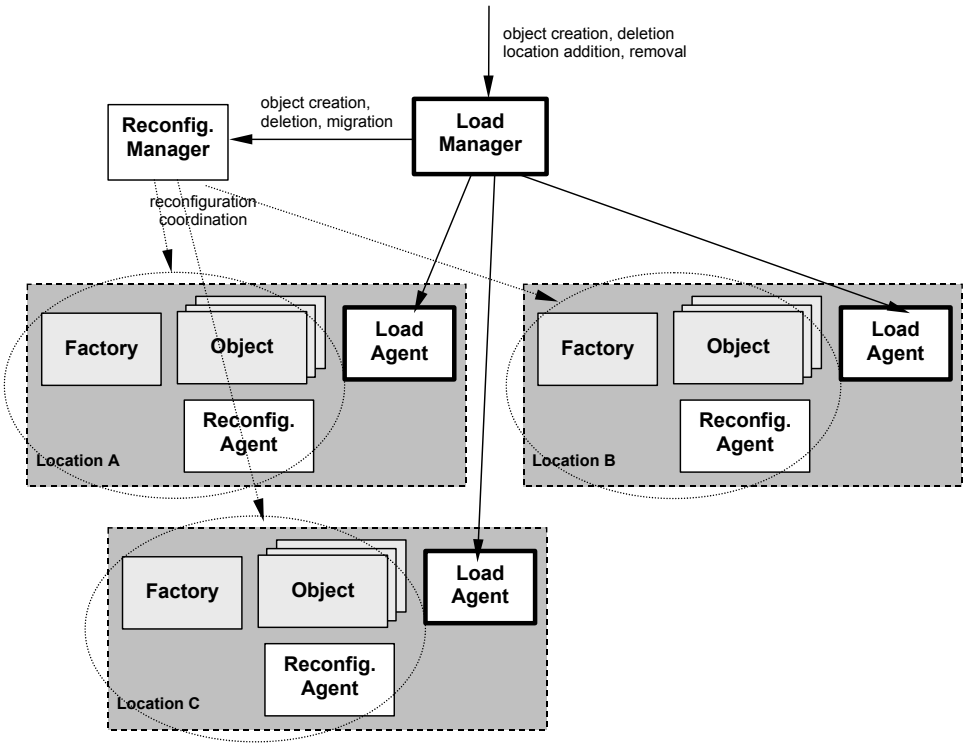


Figure 42 – Load-balancing manager, and load agents

The Load Manager implements the **LoadManager** interface. This interface inherits the **GenericFactory** interface and adds operations for location addition (**add_location()**) and removal (**remove_location()**).

Each location has an associated Load Agent. A Load Agent implements the **LoadAgent** interface, providing a **load()** operation, which returns the estimated load of the location as a **float**.

```
module LoadBalancing
{
 interface LoadAgent
 {
    float load();
 };
```

```
typedef ReconfigurationService::Location Location;

interface LoadManager : ReconfigurationService::GenericFactory
{
   void add_location(in Location locus, in LoadAgent agent);
   void remove_location(in Location locus);
};

};
```

7.2.1 Implementation of the Load Agent

We have implemented a Load Agent in Java that obtains a simple estimate of the load of the location where it runs. Our Load Agent has a load-measuring thread that constantly estimates the load of the location. This thread estimates the load by counting the number of iterations that can be made in a fixed measuring period of 2 seconds. The lower the number of iterations counted the higher the load. Therefore we take the negative of the number of iterations as our estimate of the load.

7.2.2 Implementation of the Load Manager

The Load Manager regularly polls the Load Agent of each Location registered to obtain the load of the location. If the difference between the maximum load and the minimum load is greater than a fixed threshold, a random object is migrated from the location with the maximum load to the location with the minimum load.

```
while (true)
    {
      // get an update of the load of the registered agents
      location_registry.update_view();

      Location loc_max_load = location_registry.location_max_load();
      Location loc_min_load = location_registry.location_min_load();

      if (  (loc_max_load.get_load() - loc_min_load.get_load())  > THRESHOLD )
      {
            org.omg.CORBA.Any creation_id = loc_max_load.random_object();

            if (creation_id!=null)
            {
             Property[] the_criteria = new Property[1];

             NameComponent location_prop = new NameComponent[1];
             location_prop[0] = new NameComponent("drs.Location","");

             Any destination = orb.create_any();
             destination.insert_string(loc_min_load.get_name());

             the_criteria[0] = new Property(location_prop, destination);

             try
             {
              ReconfigurationStep step = rec_mgr.create_reconfiguration_step();
              step.migrate_object(creation_id, the_criteria);
              step.commit();

              loc_max_load.remove_object(creation_id);
              loc_min_load.add_object(creation_id);
            }
            catch(Exception e)
```

allocation decision

new location is specified in the criteria

migration is requested

```
                {
                  System.out.println("Error migrating object.");
                }
              }
          }

          try {  this.sleep(PAUSE_DURATION);  } catch (InterruptedException e) { }
      }
```

7.2.3 Implementation of Example Application Objects

We have implemented a simple reconfigurable application object to populate the load-balancing example. **ExampleObject** has an operation **hello()** that returns a string and an operation **statistics()** that returns the number of times **hello()** has been invoked. The state of an **ExampleObject** is defined in IDL with the structure **ExampleObjectState**.

```
module Example
{
 interface ExampleObject : ReconfigurationService::ReconfigurableObject,
 {
    string hello();
    long statistics();
 };

 interface ExampleObjectFactory :
       ReconfigurationService::ReconfigurableObjectFactory
 {
 };

 struct ExampleObjectState
 {
  long statistics;
 };
};
```

The implementation of the **hello()** operation consists of a loop with a high number of iterations in order to simulate the application load.

7.2.4 Tests

We have started the application with one single location registered (Location 1). 12 objects were gradually created in Location 1, as well as one client for each object. With the addition of a second location, Objects 1 to 7 were migrated to this Location 2, while Objects 8 to 12 remained in Location 1. With the addition of Location 3, Objects 1, 8, 2, 3, 9, 4 were migrated from Locations 1 and 2 to Location 3. Figure 43 shows the evolution of the configuration.

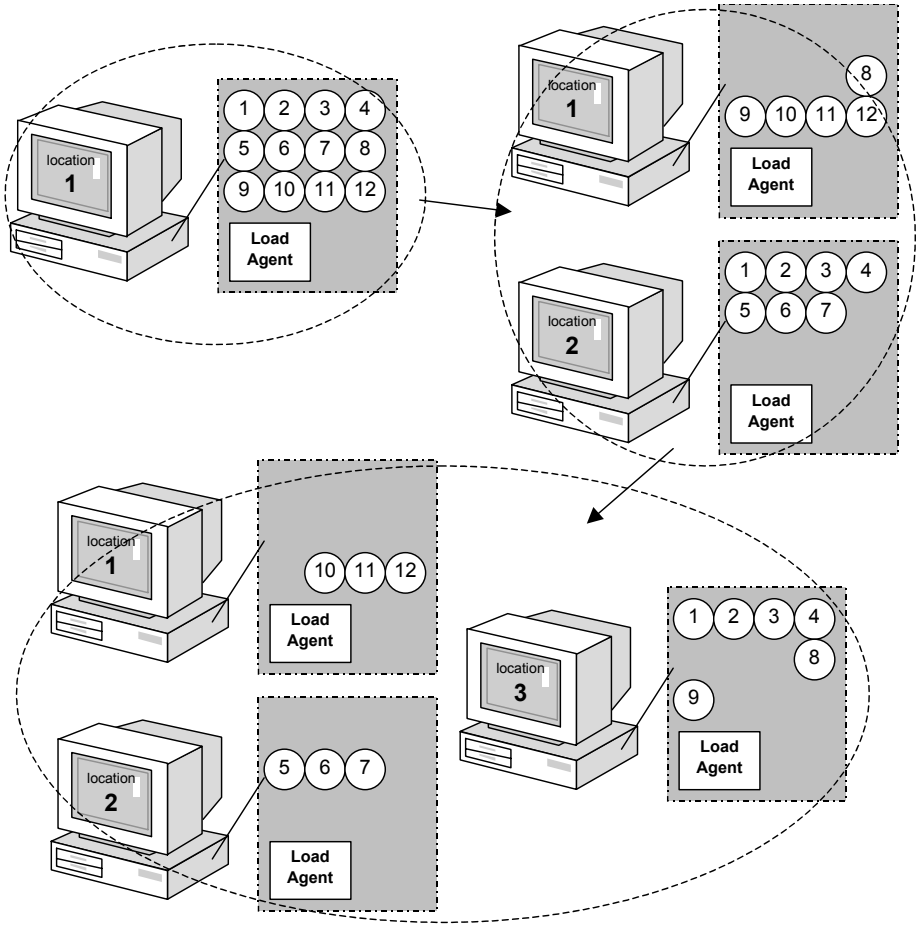


Figure 43 – Change of configuration with the addition of locations

The average response times perceived by clients for the **hello()** operation lie in the range [2.0; 2.5]s for the configuration with one location, in the range [1.0; 1.5]s for the configuration with two locations and in the range [0.5; 1.0]s for the configuration with three locations.

Figure 44 shows the response times perceived by the client of Object 1. Initially, there is an increase in response time that is a result of the creation of 11 other objects in the same location. With the addition of Location 2, Object 1 migrates to the new location (the addition of a location is represented as a vertical dotted line in Figure 44). The response times drop drastically, as Object 1 has all the resources available at Location 2. As Object 2 to 8 are also migrated to Location 2, the response times rise and stabilize, until the introduction of Location 3, which incurs new migrations. The increase in response times due to migration can be observed in the picture as small peeks during the migration process, immediately to the right of a vertical dotted line. Similar results have been obtained for Objects 2 to 8.

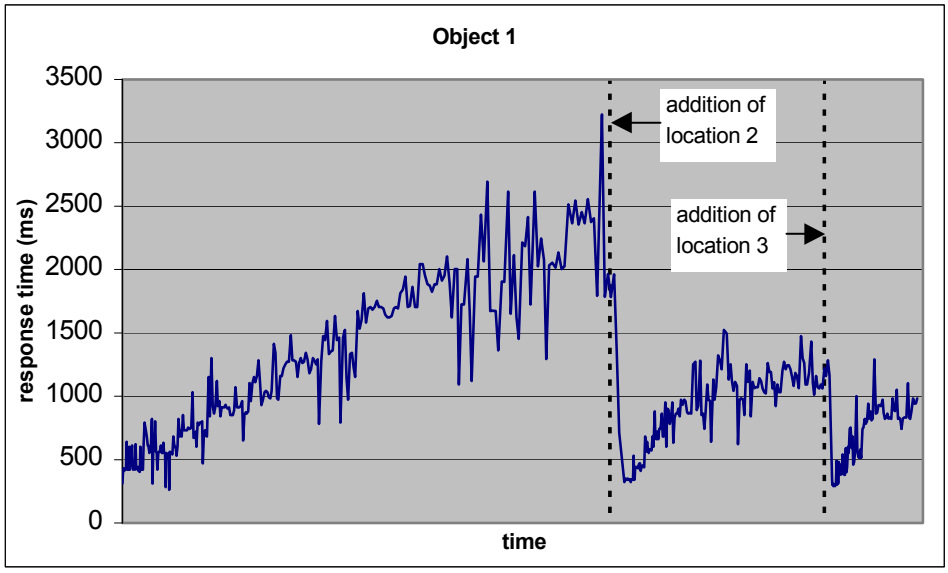


Figure 44 – Response times perceived by client of Object 1

The response times perceived by a client of Object 10 are depicted in Figure 45. Object 10 is not migrated during the execution. The decrease in response time is explained by the migration of other objects that leave Location 1.

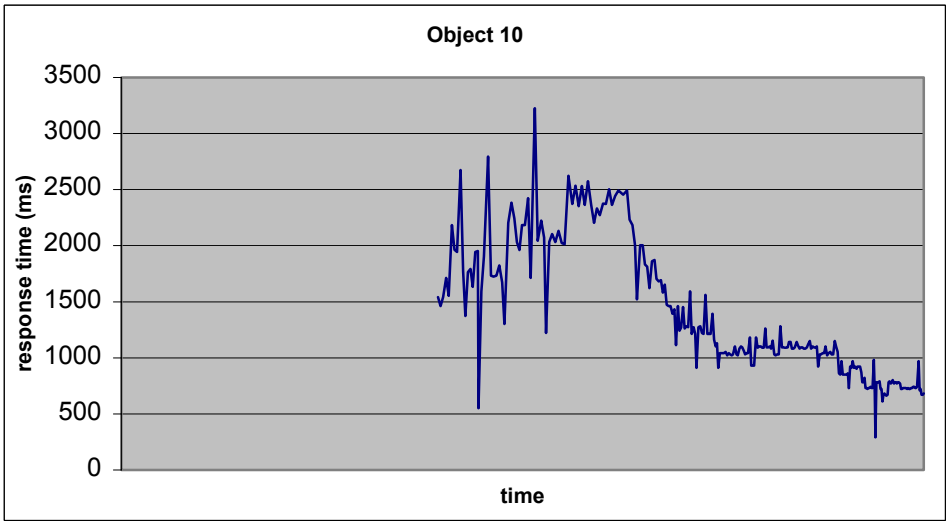


Figure 45 – Response times perceived by client of Object 10

7.2.5 Conclusions

In this example, we have shown the use of migration in a load-balancing application. In this application, a Load Manager uses the DRS to request the creation, removal and migration of objects.

Since the DRS provides support for migration, the Load Manager is responsible for determining which object should be migrated and to which location the object should be migrated. The Load Manager delegates the execution of the simple reconfiguration step with one object migration to the Reconfiguration Manager.

Objects are migrated from a highly loaded location to a less loaded location, in order to ultimately reduce the response times perceived by clients. In the test run shown in Section 7.2.4, we have seen that the increase in response times due to migration may be considered acceptable for the use of DRS as a migration mechanism for the load-balancing scenario presented. Nevertheless, we expect the increase in response time to be higher for applications in which objects are involved in longer interactions.

This example also constitutes a proof-of-concept for yet another application of the DRS.

8 Conclusions

This chapter presents the main contributions of this thesis, draws some relevant conclusions of our work and identifies areas where further investigation is necessary.

This chapter is further structured as follows: Section 8.1 presents the main contributions of this thesis, Section 8.2 presents general conclusions, and Section 8.3 identifies some further work.

8.1 Main Contributions

The main contributions of our work can be summarized as follows:

- we have proposed a new approach to dynamic reconfiguration of distributed applications built on top of object middleware;
- we have designed a dynamic reconfiguration service for CORBA that supports both the application developer and the change designer. This service can be implemented by using standardized mechanisms;
- we have implemented the dynamic reconfiguration service in a prototype and obtained some performance measurements; and
- we have illustrated the use of the dynamic reconfiguration service in different application scenarios, including an example banking application and a load-balancing application.

Our dynamic reconfiguration approach:

- support object creation and removal;
- supports replacement, in which a new version of an object may have functional and quality-of-service (QoS) properties that differ from the old version. This new version may run in another execution-environment type supported by the middleware platform;

- supports migration;
- supports composite reconfiguration steps, in which several objects are reconfigured in an atomic action from the perspective of the application;
- prescribes mechanisms to obtain a correct incremental evolution of a system, preserving the object model under normal operation and during reconfiguration;
- is applicable to a broad range of applications, including applications built from off-the-self components, multi-threaded applications, re-entrant objects, and stateful objects;
- minimizes impact on execution during reconfiguration and accounts for little overhead during normal operation;
- scales with respect to the number of clients;
- provides full reconfiguration transparency to client developers, and requires minimal reconfiguration expertise from the reconfigurable object developer;
- does not require the use of a specific programming language for application development;
- does not require the use of additional formalisms for application development.

The main limitation of the approach is that it ignores the preservation of architectural properties of an application, assuming reconfiguration design activities to produce changes that have been validated *a priori*.

We have submitted the approach presented in this thesis in Lucent Technologies' response [32] to the Request For Information (RFI) on Online Upgrades issued by the OMG in September 2000 [17], hoping that this approach becomes incorporated in a forthcoming CORBA standard.

8.2 General Conclusions

Most of the approaches we have investigated attempt to be general to distributed systems and hence do not exploit the particular characteristics of object-middleware. Middleware platforms are designed to provide several transparencies for the application designer, facilitating distributed application development. Embedding reconfiguration functionality in a middleware platform is a promising way to leverage this functionality with maximum transparency. In this thesis we have proposed an approach that can be realized with minimum additional burden on the development of the reconfigurable objects and that is fully transparent to the developer of client objects.

The proposed approach can be used in systems with a large and changing number of objects. We have proposed to use request reflection to instrument the middleware platform and 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 request reflection, we are able to freeze system interactions on demand. In this way, our approach only interferes directly

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.

In the approach, reconfiguration of objects that are involved in long-running interactions may implicate high increase in response time experienced by the clients of the affected objects. Ultimately, the maximum acceptable increase in response time during reconfiguration is determined by the environment in which the affected object is inserted.

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. The Dynamic Reconfiguration Service performs reconfiguration steps on behalf of the user. 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 expected to be acceptable for most application domains. We also presented preliminary measurements on the overhead introduced by the DRS during reconfiguration. The results of these measurements are satisfactory as well.

Adding reconfiguration transparency to a CORBA environment is not straightforward, especially if the objective is to preserve a sufficient level of compatibility with the CORBA standard. In our design, we account for extensions to be done through standardized mechanisms, such as portable interceptors. The benefits of using standardized extension mechanisms are that (i) the ORB specification does not have to be modified, and that (ii) implementations are portable and conform to the ORB specification.

The Dynamic Reconfiguration Service has been proven useful in different application scenarios, including an example banking application and a load-balancing application.

8.3 Future Work

The dynamic reconfiguration approach could be extended with the abortion of interactions that are possibly long running and do not affect the state of an object. This would lead to a hybrid abortion-avoidance and abortion approach that could decrease the impact on system execution, especially for systems with long-running interactions.

Our prototype of the dynamic reconfiguration service could be made available to a large number of developers and it could be applied in complex realistic applications. This would further validate the service and provide feedback that would lead to possible improvements.

We expect the approach to be applicable directly on a component-based middleware infrastructure (e.g. [24, 25]). The support to dynamic reconfiguration in this case may be located in the container of a reconfigurable component. A component could be declared to be reconfigurable in its deployment descriptor, thus providing a strict separation between application and reconfiguration concerns. With component-based middleware, it would be easier for the component developer to define the state access functions, because the relationships between components are not encapsulated in the implementation of a component, and these relationships can be reified and manipulated at run-time by a third-party, which, in our case, is the dynamic reconfiguration service. Since a

component is a deployment unit, it would also be possible to re-use or adapt the deployment facilities of a middleware infrastructure in order to include dynamic reconfiguration.

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, in *Proceedings of the19th Brazilian Symposium on Computer Networks (SBRC 2001)*, Santa Catarina, Brazil, May 2001.

[2] J. P. A. Almeida, M. Wegdam, M. van Sinderen, L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA, in *Proceedings of the 3rd International Symposium on Distributed Objects & Applications (DOA 2001)*, Rome, Italy, September 2001 (to appear).

[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/IEC 10746-1.

[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] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying Distributed Software Architectures, in *Proceedings of the 5th European Software Engineering Conference*, ESEC '95, Barcelona, 1995.

[12] Microsoft Corporation. *Distributed Component Object Model (DCOM)*. http://www.microsoft.com/com/tech/dcom.asp

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

[14] 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*, pp. 184-196, Hilton Head, SC, January 2000.

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

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

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

[18] Object Management Group. *Life Cycle Service*, formal/00-06-18, July 2000.

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

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

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

[22] Object Management Group. *Discussion of the Object Management Architecture*, formal/00-06-41, January 1997.

[23] Object Management Group. *Transaction Service Specification*, orbos/00-06-28, May 2000.

[24] Object Management Group. *Components FTF Edited Drafts of CORBA Core Chapters,* ptc/99-10-03, October 1999.

[25] Object Management Group. *CORBA 3.0 New Components Chapters,* ptc/99-10-04, October 1999.

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

[27] 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, pp. 95-111, Springer-Verlag, Berlin, 1999.

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

[29] Sun Microsystems, Inc. *Java Remote Method Invocation*. http://java.sun.com/products/jdk/rmi/

[30] SunSoft, Inc. and Hewlett-Packard Company. *Response to the Object Management Group Object Services Task Force Request for Information*, 1992/92-02-10. February 1992.

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

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

[33] M. Wegdam, D.-J. Plas, A. van Halteren, L. 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.

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

Appendix A IDL Interfaces

This appendix lists the OMG IDL interfaces that should be standardized for portability, as identified in Section 6.4.

A.1 The ReconfigurationService Module

Code in gray has been extracted from the Fault Tolerant CORBA specification.

```
// from FT CORBA
#include "OB/CosNaming.idl"          // 98-10-19.idl
#include "OB/ORB.idl"                // from 98-03-01.idl
// end of from FT

#ifndef DRS_IDL
#define DRS_IDL

module ReconfigurationService
{
   // from FT CORBA
   typedef CORBA::RepositoryId TypeId;

   typedef CosNaming::Name Name;

   typedef any Value;

   struct Property {
         Name nam;
         Value val;
   };

   typedef sequence<Property> Properties;

   typedef Name Location;

   typedef sequence<Location> Locations;

   typedef Properties Criteria;

   interface GenericFactory; // forward reference

   struct FactoryInfo {
         GenericFactory factory_;
         Location the_location;
         Criteria the_criteria;
   };

   exception ObjectNotCreated {};
   exception ObjectNotFound {};
   exception InvalidProperty {
         Name nam;
         Value val;
   };
   exception NoFactory {
         Location the_location;
         TypeId type_id;
   };
   exception InvalidCriteria {
```

```
        Criteria invalid_criteria;
};
exception CannotMeetCriteria {
        Criteria unmet_criteria;
};

interface GenericFactory {

        typedef any FactoryCreationId;

        Object create_object(
                in TypeId type_id,
                in Criteria the_criteria,
                out FactoryCreationId factory_creation_id
        )
        raises (
                NoFactory,
                ObjectNotCreated,
                InvalidCriteria,
                InvalidProperty,
                CannotMeetCriteria
        );

        void delete_object(
                in FactoryCreationId factory_creation_id
        )
        raises (
                ObjectNotFound
        );

};
// end from FT

typedef sequence<TypeId> TypeIds;

exception FactoryNotFound {};

interface FactoryManager
{
   typedef any FactoryId;

   FactoryId add_factory(
         in FactoryInfo factory_info,
         in TypeIds type_ids
    );

   void remove_factory(
         in FactoryId factory_id
   )
   raises (FactoryNotFound);

   FactoryInfo get_factory_info(
        in FactoryId factory_id,
        out TypeIds type_ids
   )
   raises (FactoryNotFound);
};


typedef any FactoryCreationId;
typedef sequence<FactoryCreationId> FactoryCreationIds;

exception UnderReconfiguration {};

exception ReconfigurationException {
};
```

```
interface ReconfigurationStep; // forward reference
interface GenericStateTranslator; // forward reference


interface ReconfigurationManager : GenericFactory, FactoryManager
{
      ReconfigurationStep create_reconfiguration_step()
      raises (UnderReconfiguration);
};

interface ReconfigurationStep
{

      Object create_object(
            in TypeId type_id,
            in Criteria the_criteria,
            out FactoryCreationId factory_creation_id
      )
      raises (
            NoFactory,
            ObjectNotCreated,
            InvalidCriteria,
            InvalidProperty,
            CannotMeetCriteria
      );

      void delete_object(in FactoryCreationId factory_creation_id)
      raises (ObjectNotFound);


      void replace_object(
            in FactoryCreationId factory_creation_id,
            in Criteria the_criteria
      )
      raises (
            ObjectNotFound,
            NoFactory,
            InvalidCriteria,
            InvalidProperty,
            CannotMeetCriteria
      );

      void migrate_object(
            in FactoryCreationId factory_creation_id,
            in Criteria the_criteria
      )
      raises (
            ObjectNotFound,
            NoFactory,
            InvalidCriteria,
            InvalidProperty,
            CannotMeetCriteria
      );

      FactoryCreationIds replace_type(
            in TypeId current_type_id,
            in TypeId new_type_id,
            in Criteria the_criteria
      )
      raises (
            NoFactory,
            InvalidCriteria,
            InvalidProperty,
            CannotMeetCriteria
      );

      FactoryCreationIds migrate_objects(
```

```
            in TypeId type_id,
            in Location origin,
            in Criteria the_criteria
        )
        raises (
            NoFactory,
            InvalidCriteria,
            InvalidProperty,
            CannotMeetCriteria
        );



        void set_default_criteria(in TypeId type_id, in Criteria the_criteria);

        void remove_type(
            in TypeId type_id
        );

        void set_state_translator(
            in GenericStateTranslator translator
        );

        void commit()
        raises (ReconfigurationException);

        void deferred_commit();

        boolean is_completed()
        raises (ReconfigurationException);

        void dispose()
        raises ( UnderReconfiguration );
};


typedef sequence<octet> State;

typedef long long ReconfigurableObjectId;

typedef sequence<ReconfigurableObjectId> ReconfigurableObjectIds;

interface ReconfigurationManagerAdmin; // forward reference

interface ReconfigurableObject
{
  State get_state();

  void set_state(
        in State s
   );
};

interface ActiveObject
{
  void passivate();
  void activate();
};

interface ReconfigurationAgent; // forward reference

interface ReconfigurableObjectFactory : GenericFactory
{
   ReconfigurationAgent get_reconfiguration_agent(
     in ReconfigurableObjectId id
```

```
  );
};

interface GenericStateTranslator
{
   enum ReconfigurationOperationType {
        CREATION, REPLACEMENT, MIGRATION, DELETION };

   struct Instance
   {
    TypeId type_id;
    ReconfigurableObjectId id;
    State the_state;
    ReconfigurationOperationType op_type;
   };

   typedef sequence<Instance> InstanceSeq;

   typedef sequence <TypeId> TypeIdSeq;

   TypeIdSeq types_supported();

   void translate(in InstanceSeq original, out InstanceSeq derived);
 };


 interface Current : CORBA::Current
 {
    typedef sequence<octet> octets;
    void register_thread(in octets adapter_id, in octets object_id);

    struct CurrentSlotInfo
    {
      ReconfigurableObjectId id;
      ReconfigurableObjectIds invocation_path;
    };

    typedef CurrentSlotInfo Control;

    Control get_control();

    void resume(in Control which);
 };


 interface ReconfigurationAgent
 {
        typedef sequence<octet> octets;

        void register_object(
              in ReconfigurableObjectId id,
              in Object rec_obj_reference,
              in octets adapter_id, in octets object_id
        );

        void deregister_object(
              in ReconfigurableObjectId id
        );

        Object get_reference(
              in octets adapter_id, in octets object_id
        );

       ReconfigurableObjectId get_reconfigurable_object_id(
              in octets adapter_id, in octets object_id
        );
```

```
            boolean is_affected(
                in ReconfigurableObjectId id
            );
        };

};

#endif
```