An approach to dynamic reconfiguration of distributed systems based on object-middleware

João Paulo A. Almeida[1,2], Maarten Wegdam[1,2], Luís Ferreira Pires[1], Marten van Sinderen[1]
almeida@cs.utwente.nl, *wegdam@lucent.com*, *pires@cs.utwente.nl, sinderen@cs.utwente.nl*

[1]Centre for Telematics and Information Technology, University of Twente

PO Box 217, 7500 AE, Enschede, The Netherlands

[2]Lucent Technologies, Bell Labs Twente

Capitool 5, 7521 PL, Enschede, The Netherlands

Abstract

There is an increasing demand for long running and highly available systems. This holds particularly for distributed systems based on object-middleware, which are becoming increasingly popular nowadays. Dynamic reconfiguration consists of modifying the configuration of a system during runtime, contributing to the availability of the system. This paper introduces a novel approach to dynamic reconfiguration of distributed systems based on object-middleware. The paper also discusses some issues related to the implementation of our approach and proposes a design for a Dynamic Reconfiguration Service using CORBA standard mechanisms.

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

Resumo

Existe atualmente uma demanda crescente de sistemas de longa execução e ampla disponibilidade. Isso se aplica particularmente a sistemas distribuídos baseados em middleware de objetos, que estão ficando cada vez mais populares. Reconfiguração dinâmica consiste na modificação da configuração de um sistema durante a sua execução, contribuindo assim para aumentar a disponibilidade do sistema. Esse artigo introduz uma nova forma de se realizar reconfiguração dinâmica de sistemas distribuídos baseados em middleware de objetos. Esse artigo discute alguns tópicos relacionados a implementação das nossas idéias e propõe um projeto para um Serviço de Reconfiguração Dinâmica baseado em mecanismos do padrão CORBA.

Palavras-chaves: reconfiguração dinâmica, sistemas distribuídos, middleware, CORBA, upgrade on-line

1 Introduction

Distributed computing systems have been in widespread use for several years in commercial, industrial and research environments. Such systems are currently deployed in mission-critical and long-running applications, such as, for example, telecommunication switches and e-commerce solutions. The reliance on software 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. They 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 [10].

The aim of *dynamic reconfiguration* [8, 9, 7, 1, 4, 11, 2, 10, 14, 18] is to allow a system to evolve at run-time [8], 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. Changes can be classified with relation to the moment they are envisioned as *programmed* and *evolutionary changes* [10]. Programmed changes are foreseen and anticipated by the system designer, while evolutionary changes are unanticipated and become necessary over the execution lifetime of an application. Changes are applied to a system through reconfiguration. The possible changes applied to a system depend on the granularity of the reconfigurable entities and the operations that can manipulate such entities in the affected part of a system. Entities can be objects, groups of objects, components, group of components, sub-systems, bindings and groups of bindings. Operations on entities can be replacement, migration, addition, and removal.

New generation distributed applications often consist of co-operating objects, and make use of object-middleware technology, such as CORBA [3], Java RMI and DCOM. Object-middleware facilitates the development of distributed applications by providing distribution transparencies to the application designers. Object-middleware offers a widely accepted approach for the provision of flexible computing environments. As such, there are many systems that would profit from dynamic reconfiguration facilities for object-middleware, such as, e.g., critical and/or long-running systems. The development of such systems would be facilitated through the inclusion of (transparent) reconfiguration support in the middleware platform.

This paper is further structured as follows. Section 2 describes briefly the CORBA object model; Section 3 presents terminology, definitions and concepts used in our discussion of dynamic reconfiguration; Section 4 presents requirements for a dynamic reconfiguration service for object-middleware; Section 5 describes our dynamic reconfiguration approach; Section 6 discusses some implementation issues concerning the implementation of our reconfiguration support using CORBA standard mechanisms; Section 7 compares our approach to related work found in the literature. Finally, Section 8 presents conclusions and future work.

2 CORBA object model

This section briefly presents the CORBA object model [3], by using the terminology introduced in [16]. 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. The middleware platform (CORBA in our case) offers distribution transparency to the application objects and supports their interaction.

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 we call 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.

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, for the case of at-most-once semantics (request-response operations). In this paper we ignore the simpler case of best-effort semantics (oneway operations). Since the client and the target objects are possibly distributed, an Object Request Broker (ORB) 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 ORB 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 ORB in supporting this interaction.

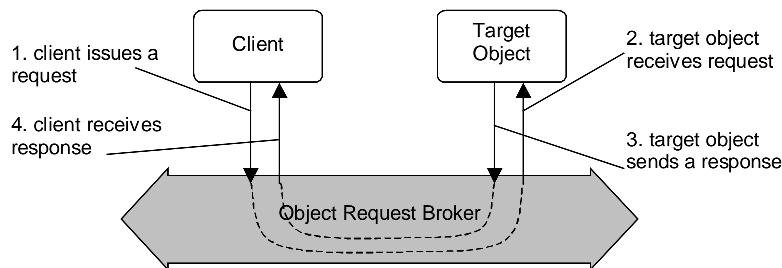

Figure 1 – Example of CORBA request

A *servant* is a programming language construct that implements a target object. A servant carries out the computational activities that are necessary to process a request. In case a servant should be capable of handling multiple requests simultaneously, the servant 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 in order to process 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.

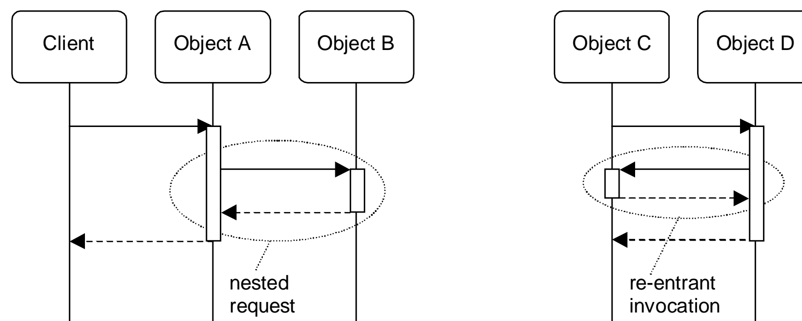

Figure 2 – Nested and re-entrant invocations

3 Dynamic reconfiguration

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

3.1 Process overview

Figure 3 depicts the dynamic reconfiguration model based on [8, 9], which has been adopted in this paper.

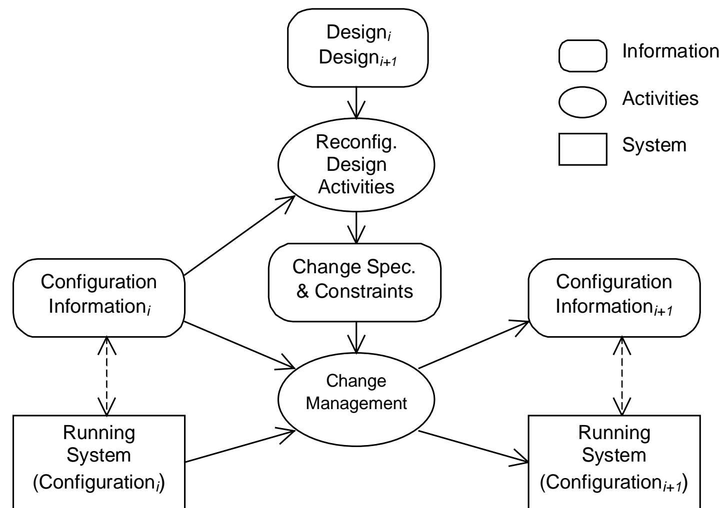

Figure 3 – Dynamic Reconfiguration model

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 [11, 10, 8] 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 last less that 10s”, “entity A should be available during the whole process”.

Performing reconfiguration on a running system is an intrusive process [10]. Reconfiguration may imply, for example, addition, 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 therefore 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 [10]. 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 sequel.

3.2 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 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 [10]. *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 has to be aborted, A is informed and synchronization is maintained; or (ii) B receives the request, finishes processing it and sends the response, and then is replaced by B'; or, (iii) B is replaced by B', and B' has to honor the invocation, by processing the request and sending a response to A. In case none of these alternatives occur, A might be kept waiting for a response forever.

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 4 shows a classification of reconfiguration approaches according to their choices on the preservation of mutual consistency.

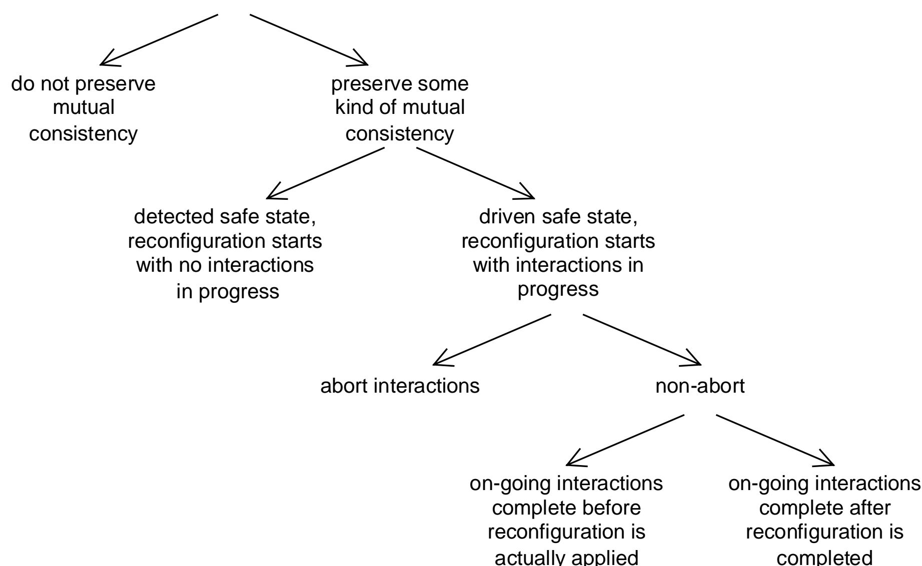

Figure 4 – Reconfiguration approaches and preservation of mutual consistency

In this classification, 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 [10]: 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., [1]) 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.

We have studied mechanisms that do not abort interactions. These mechanisms are designed to assure that interactions in progress are eventually completed, either before reconfiguration has started or after reconfiguration has finished.

In this paper, we propose 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 section 5.

3.4 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 [10].

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 object.

4 Requirements

The following requirements have been considered in the design of the *Dynamic Reconfiguration Service*:

- *Correct incremental evolution.* The service should provide facilities to allow a reconfiguration designer to obtain a correct incremental evolution of a system, as defined in Section 3;
- *General applicability.* The architecture should be applicable to a broad range of applications. It should be possible to reconfigure applications built from components-off-the-self, applications with multi-threaded and single-threaded execution models, re-entrant objects, stateless objects, stateful objects, etc.;
- *Scalability*. The dynamic reconfiguration service should be scalable, particularly with respect to the number of clients;
- *Impact on execution*. The architecture should minimize impact on execution during reconfiguration. *Impact on system execution* is 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 [20]. The design of the service should also account for little overhead during normal operation, i.e., when configuration is not taking place;
- *Responsibilities and transparencies.* Responsibilities should be assigned to the dynamic reconfiguration service, facilitating both the development of reusable reconfigurable entities and the use of the reconfiguration service. Ideally, the middleware platform should be extended to provide reconfiguration transparency. Reconfiguration should be fully transparent to clients. This would require less expertise from application developers and would allow them to focus on application specific functionality;
- *Configuration information.* The access to configuration information and system state information should be done in a principled way, as opposed to ad hoc solutions. The service should not require the application developer to provide extensive descriptions of

the system and its structure. The service requires instrumentation on the middleware platform and application in order to gather the information necessary for reconfiguration. The service should be able to hide instrumentation details and reconfiguration-specific concerns from change and application designers.

The *Dynamic Reconfiguration Service* should allow objects to be manipulated through reconfiguration operations. Reconfiguration operations are *replacement*, *migration*, *addition* and *removal*. Replacement allows an object to be replaced by another object. The new object may have both functional and quality-of-service (QoS) properties that differ from the old entity. Furthermore, the new implementation of the object may run in another execution engine supported by the object-middleware platform. For example, an object implemented in Java may be replaced by an object implemented in C++.

The service should be able to apply reconfiguration operations to several system entities atomically from the perspective of the unaffected system parts. Through the application of reconfiguration operations to several objects atomically, addition and removal may be used in conjunction with replacement, to provide general replacements, such as the atomic replacement of a group of objects.

5 Proposed approach

This section describes our proposed approach to the reconfiguration of systems based on object-middleware by addressing each of the aspects of correctness identified in Section 3.

5.1 Structural integrity

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

Referential integrity becomes an issue whenever an object reference changes. An object reference is defined as a value that denotes a particular object, and is used by the middleware infrastructure to locate the object. Object references acquired by clients prior to reconfiguration may be invalidated due to reconfiguration. For example, in CORBA platforms, migration invalidates the IP address and port number contained in the IIOP profile of an IOR. If a reference points to an object that no longer exists, the established logical binding between a client and a target object is broken. In order to re-establish the binding after reconfiguration, we provide a logically central point of contact for clients to find the objects with invalidated object references. Section 6.1 presents a CORBA-based solution to this problem that is transparent to the clients of the reconfigured objects.

In the CORBA object model, interfaces satisfy the Liskov substitution principle [3]. This means that if interface B is derived from interface A, then references to an object that supports interface A can be used to denote an object that supports interface B. To avoid that object replacements violate the object model, a new object must satisfy the old interface. This can be done either by implementing the old interface or by implementing an interface derived from it, e.g., by inheritance. If all clients of a reconfigurable object are also reconfigurable objects, it is possible to promote arbitrary changes to the interface by upgrading both clients and target objects atomically.

5.2 Mutual consistency

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

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

Affected objects are the objects that are replaced, removed or migrated as a result of the reconfiguration process.

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

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

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

Reaching the safe state

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

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

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

In the more complex cases of reconfiguring *multiple (re-entrant) objects simultaneously,* selective queuing of requests directed to affected objects is necessary. Requests issued by an affected object should get 'laissez-passer' status, since its requests have to be executed for the safe state to be reached. This implies that requests in invocation paths that contain at least one

affected object should also be included in the ‘laissez-passer’ set. In particular, re-entrant requests initiated by affected objects are also included in the ‘laissez-passer’ set. All objects that could otherwise issue new ‘laissez-passer’ requests are set to exhibit reactive behaviour, so that no new ‘laissez-passer’ requests are generated. At some point, the existing requests are treated, all affected objects are idle, and the system reaches the safe state.

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

Applying reconfiguration

When all affected objects are idle, the reconfiguration process can proceed. The affected objects’ state can be inspected and used to derive the state of the objects being introduced. Once new objects or new versions of objects have been installed, their state is properly modified. Queued requests and further new requests are redirected to the new version of an object.

5.3 Application invariants

In [10] a scheme is proposed in which invalidated invariants can be identified and re-established by the change designer with little assistance from the application developer. This scheme consists of requiring objects to provide general-purpose state access-methods that can be invoked by a third party to query or adjust the state of objects. These methods would be invoked by the change designer to inspect and modify a selected subset of an object’s internal state at runtime. In this scheme, the application designer decides on the particular subset of the objects’ state that is exposed by these access methods. In general, objects should provide methods to inspect and modify state variables that control synchronization and computational behavior of the object.

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

6 CORBA implementation issues

This section discusses the implementation issues related to our approach. In particular we discuss the functionality that has to be placed on the client-side of an ORB (client ORB) and on the server-side of an ORB (server ORB) in order to implement our approach.

6.1 Referential integrity

In order to keep an object reference valid after reconfiguration, we can 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 standard LOCATION_FORWARD mechanism to inform a client ORB of the new location of the target object. This mechanism is fully transparent to the client application and the overhead for implementing 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 with the server ORB has not been standardized.

6.2 Selective 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.

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 5 shows this scheme, abstracting from the location of the objects in the middleware platform and the way they are built.

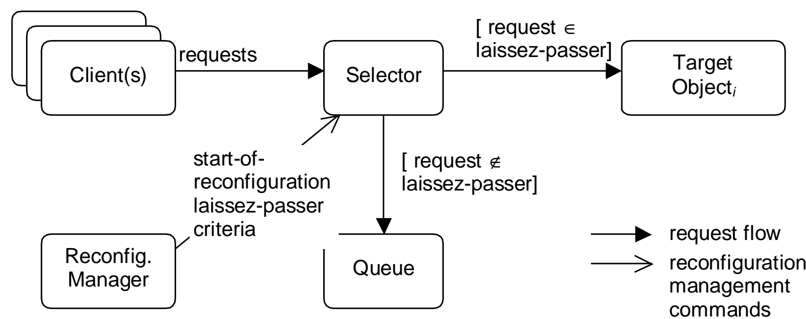

Figure 5 – Selector and Queue functions, reaching the safe state

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 6.

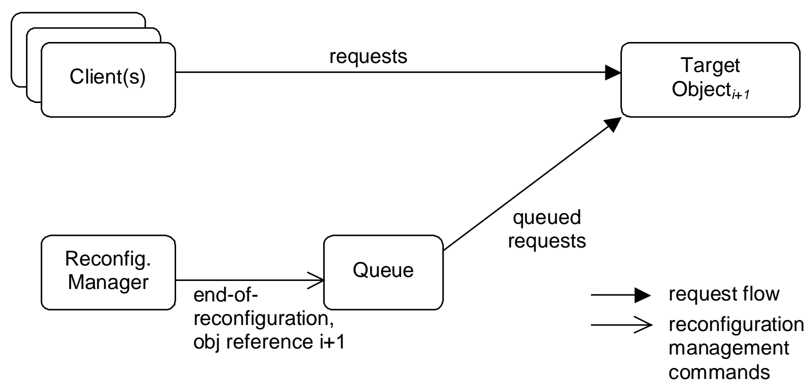

Figure 6 – Redirecting requests to new version of the implementation after reconfiguration

Initially we investigated the implementation of our approach as a service 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 in this paper we ignore this alternative.

Other alternatives can be generated by considering the allocation of the selector and the queue to different parts of the middleware platform. 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.

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 [19], in which we can operate on the request. 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 [12], 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. Implementing the hybrid solution using portable interceptors is quite straightforward.

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 identity 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 identity of an affected object is included in the service context, the request belongs to the 'laissez-passer' set and should not be queued.

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

the requests. In addition, the order in which replies reach the client does not imply the order in which the server processed the requests.

6.3 Object creation

Since reconfigurable objects need to have some common characteristics, we intend to create them by using factory objects in a similar way as defined in the Fault Tolerant CORBA specification [5]. Another benefit of using factories for creating objects is that a factory can shield the application and reconfiguration designer from the specific support to object deployment offered by different languages, operating systems or virtual machines, such as e.g., DLLs, the Java class loader and interpreted languages. This would make it possible, for example, to replace an object implemented in C++ by another object implemented in Java. The direct use of this specific support would reduce the generality of the middleware reconfiguration facilities.

7 Related work

This section presents and compares some developments on dynamic reconfiguration found in the literature. These developments are also evaluated and related to our approach.

7.1 Other approaches to dynamic reconfiguration

Kramer and Magee's approach [8, 9] has been influential in the subsequent works on dynamic reconfiguration. The definition of change management and the reconfiguration model from section 2 stems mostly from their work. In this approach, a system is seen as a directed graph whose nodes are the entities and whose arcs are connections between entities. The model assumes at most one connection between any pair of entities. Entities 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. 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. The approach works for systems with independent and dependent transactions. For an entity Q to be replaced, all the entities that are capable of initiating transactions *directly* or *indirectly* on Q should become passive, as well as Q itself. When these entities have reached the passive state, all transactions involving Q are complete and no new transactions will be initiated. At this point Q has reached the quiescent (reconfiguration safe) state.

Using this approach, even small reconfigurations involving a few entities result in substantial impact on the system's execution [10, 18]. This approach also places a heavy burden on the application programmer who must implement all entities of the system with capabilities to act correctly to a passivate command that sets an entity to a passive state [10, 2]. In a passive state an entity shows reactive behavior.

Moazami-Goudarzi's approach [10] assumes that components in the system do not interleave transactions, i.e., a component participates in one transaction at a time. Therefore, it is possible to drive a component to a passive state by blocking its execution when no transactions are being serviced. The main benefit of this approach is that components do not have to implement a command to drive them to the passive state. Nevertheless, the class of distributed systems to which this alternative can be applied is much more limited, since

components in this approach cannot treat more than one transaction simultaneously. In a CORBA-based system, this implies that re-entrance and multi-threading would be forbidden.

In *Bidan et. al.'s approach* [2], the implementation of a reconfiguration service in CORBA is considered. The reconfigurable entity is a CORBA object and the configuration information maintained consists of 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.* Links are therefore equivalent to connections in Kramer and Magee's approach. In this work, the algorithm to guarantee consistency works at a relatively fine granularity level, since it considers passivation of *links* instead of quiescence, and passivation or blocking of *objects*. Unlike the approach of Kramer and his colleagues, this algorithm is not suitable for a system with nested and re-entrant invocations.

In this work, functionality for dynamic reconfiguration was not fully embedded in the middleware platform, requiring client applications and server objects to incorporate support for reconfiguration. All clients and servers must be multi-threaded, and all of them have to run a reconfiguration thread that implements the command to make a link passive.

Wermelinger's approach [18] considers link passivation, as in [2]. Nevertheless, more fine-grained information on the components is used than in [2]. A system is defined as a set of connected nodes, where a connection is given by an initiator port and a recipient port. For each node, 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*.

This approach requires a component to be shipped with a description of the component's internal port dependencies. However, 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.

7.2 Evaluation

A common characteristic of the approaches investigated is the use of configuration information, usually made concrete in configuration graphs, as input to algorithms for mutual consistency preservation. In these algorithms, configuration information is used to freeze the affected entities and the entities that can interact with them directly or indirectly, in order to drive the system to the reconfiguration-safe state. The maintenance of such kind of configuration information by a reconfiguration infrastructure (in a middleware platform) has serious drawbacks. Firstly, it might be too costly or even impossible to obtain the required configuration information. Secondly, relationships between entities typically have a temporary and transient nature. This makes it even more expensive to keep the configuration information up-to-date and accessible, especially in a centralized fashion. Configuration information is stored implicitly and can be spread over the middleware infrastructure and in the implementation of entities (e.g., CORBA Interoperable Object References).

Keeping track of all relationships between entities would possibly prevent the scalable implementation of a reconfiguration service, if fast changing configuration information from the whole system would have to be centralized, and kept consistent with the system itself. The number of clients in middleware-based systems is potentially high and may change quite quickly. For example, a system may have a large number of clients that have short sessions

with the target objects. This would require frequent updates in the configuration information repository.

In our approach, we consider requests individually, being able to decide whether a request belongs to the blocking set or the ‘laissez-passer’ set when the request is issued. In contrast, the approaches mentioned above freeze entities that could *potentially* initiate requests that would prevent the system from reaching the safe state. In this way, they may interfere with more system activities than necessary, e.g., by blocking entities that may indirectly influence the affected entities (as in [9]) or by blocking threads that might potentially start invocations on the affected entities (as in [2]), instead of blocking threads that actually invoke operations on the affected entities. In some sense these approaches are more conservative than our approach.

8 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. 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 reconfiguration algorithm only interferes with interactions that are required to be blocked for the reconfiguration, imposing minimal impact on the execution of the system.

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, avoiding modifications to the ORB specification. We have also submitted the approach presented in this paper in Lucent Technologies' response [17] to the Request For Information (RFI) on Online Upgrades issued by the OMG in September 2000 [13], hoping that this approach becomes incorporated in a forthcoming CORBA standard. The approach presented in this paper is being implemented in a prototype and the execution overhead imposed by the proposed mechanisms will be evaluated using realistic example applications.

References

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

[2] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] ITU-T X.901 | ISO/IEC 10746-1. *Open Distributed Processing Reference Model. Part 1 - Overview.*