

# Dynamic Reconfiguration for Middleware-Based Applications<sup>1</sup>

Maarten Wegdam<sup>1,2</sup>, João Paulo A. Almeida<sup>1</sup>,  
Marten J. van Sinderen<sup>1</sup>, Lambert J.M. Nieuwenhuis<sup>1</sup>

wegdam@lucent.com, almeida@cs.utwente.nl,  
sinderen@ctit.utwente.nl, l.j.m.nieuwenhuis@cs.utwente.nl

- 1) University of Twente, Department of Computer Science, Centre for Telematics and Information Technology,  
P.O. Box 217, 7500 AE Enschede, The Netherlands, phone: +31 53 4898041, fax: +31 53 4894524.
- 2) Lucent Technologies, Bell Labs Advanced Technologies EMEA, Bell Labs Twente, Capitool 5, 7521 PL,  
Enschede, The Netherlands, phone: +31 35 6875720, fax: +31 35 6875777.

**Abstract** – Distributed systems with high availability requirements have to allow reconfiguration of the system without being taken off-line. Examples of reconfigurations are the replacement of a component with a newer version, or the migration of a component to another node. A key issue for reconfiguration is maintaining the correctness of the system, which can be very complex due to the number of components, unclear relations between components, heterogeneity in operating systems and programming languages, and physical distribution of components. In this paper, we describe a new approach for dynamic reconfiguration of middleware-based applications that is more transparent for the application developer than existing approaches. We compare our approach with other approaches, and describe a prototype that implements our approach for CORBA-based applications.

**Index Terms** – Dynamic reconfiguration, distributed systems, middleware, CORBA

---

<sup>1</sup> Submitted to IEEE Transactions on Parallel and Distributed Systems, Special Issue on Middleware (Fall 2003).

# 1 Introduction

The dependency on distributed 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 disruptions in the service provided by these systems [14].

The aim of *dynamic reconfiguration* [4, 11, 12, 18, 22] is to allow a system to evolve incrementally from a configuration to another one at run-time. Dynamic reconfiguration exploits parallelism to improve a system's overall availability. While certain activities of a system are affected during reconfiguration, other activities are left unaffected. Developing systems that can be dynamically reconfigured is a complex task, since a developer must ensure that dynamic reconfiguration results in a correct and useful system.

Several approaches to dynamic reconfiguration found in literature [11, 12, 14, 18, 22] do not address component-middleware-based applications specifically. As a consequence, either they assume a computing model that is limited compared to the component model assumed by component middleware, e.g., ruling out multi-threading or re-entrance in system entities, or they fail to address issues that are particularly relevant for component-middleware-based systems, such as, e.g., interface evolution. Contrary to these approaches, we focus on the reconfiguration of component-middleware-based applications, such as, e.g., applications built on top of CORBA [6], Sun's Enterprise JavaBeans [7] and Microsoft's DCOM/COM+ [10]. The notion of component adopted in this paper is fairly unrestrictive in that components may be multi-threaded, active (as opposed to reactive), and may have re-entrant operations.

This paper shows that embedding reconfiguration functionality in a middleware platform is a promising way to provide this functionality with maximum transparency

to the application developer. Application developers can profit from reconfiguration functionality while requiring minimal expertise in the field of dynamic reconfiguration.

The remainder of this paper is structured as follows: Section 2 gives an overview of dynamic reconfiguration; Section 3 identifies the requirements for a middleware-based approach to dynamic reconfiguration; Section 4 describes our approach; Section 5 describes the Dynamic Reconfiguration Service for CORBA and a prototype that realizes our approach; Section 6 discusses and compares related work; and, finally, Section 7 contains our conclusions.

## **2 Overview of Dynamic Reconfiguration**

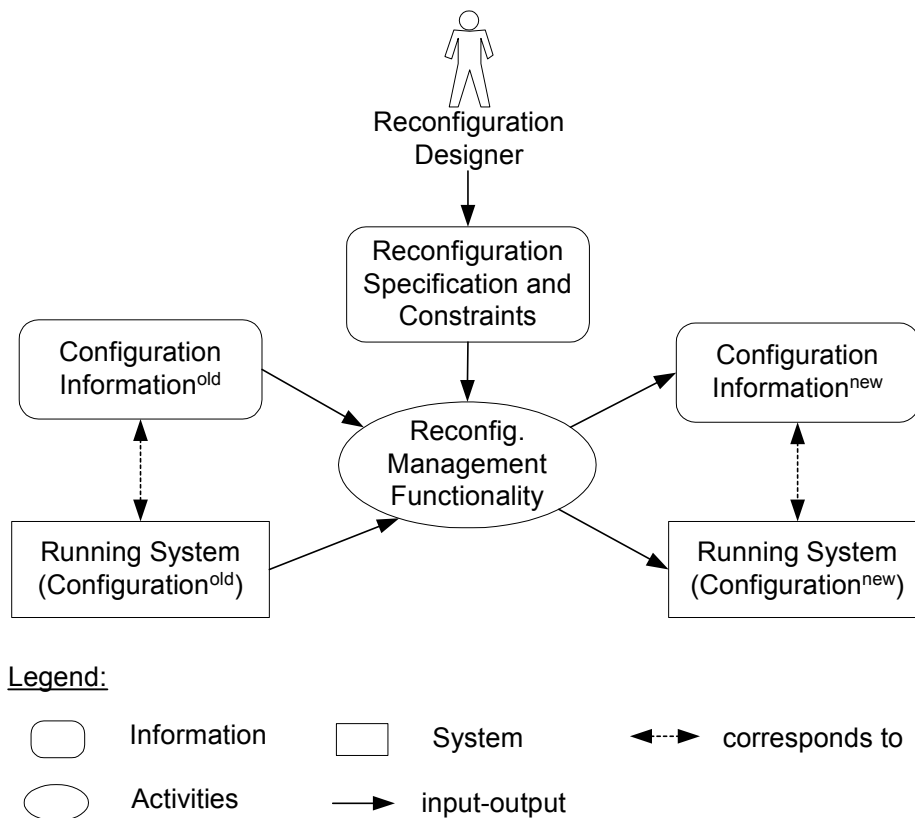
In this section we present an overview of dynamic reconfiguration by presenting the model of dynamic reconfiguration we adopt and by discussing the notions of correctness and impact on execution during reconfiguration.

### ***2.1 Model of Dynamic Reconfiguration***

The purpose of dynamic reconfiguration is to make a system evolve incrementally from its current configuration to another configuration at run-time. A *system configuration* is defined as a set of software entities, and how they are related to each other. The definition of entity depends on the level of granularity of reconfiguration. Examples of entities include objects, groups of objects, components, groups of components, sub-systems, modules, bindings and groups of bindings. Reconfiguration is specified in terms of entities and operations on these entities. Typical operations on entities are replacement, migration, creation and removal.

## Model

Figure 1 depicts the dynamic reconfiguration model based on [11, 12], which we adopt in this paper.



**Figure 1 Model of Dynamic Reconfiguration**

In this model, the *reconfiguration designer* is responsible for producing the specification of well-defined reconfigurations and constraints that have to be preserved during reconfiguration. *Reconfiguration constraints* are predicates on the reconfiguration process that restrict its execution, e.g., “the reconfiguration process must be completed within 10s”, or “entity *A* should be available during the whole reconfiguration process”.

*Reconfiguration management functionality* [11, 14] controls the reconfiguration process of a distributed system. It has as input the system in its current configuration and its configuration information, and the reconfiguration specification and

reconfiguration constraints. *Configuration information* refers to the relationship between entities. Reconfiguration management functionality makes the system evolve from its old configuration to a new configuration, and produces updated configuration information.

The reconfiguration management 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.

## 2.2 Correctness

Operating systems, middleware platforms and programming languages have mechanisms that facilitate system evolution, by allowing modules to be located, loaded and executed during run-time. However, these mechanisms normally do not ensure correctness, or observe reconfiguration constraints. Therefore, the sole use of these mechanisms to perform reconfiguration is error-prone [18].

Performing reconfiguration on a running system is an intrusive process that may interfere with ongoing interactions between entities. Reconfiguration management functionality must ensure that the system is left in a correct state after reconfiguration. A system is said to *be in a correct state* if these three aspects of correctness hold [14]:

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 reconfigured running system  $S_{i+1}$  is said to be a *correct incremental evolution* of a running system  $S_i$ , if  $S_{i+1}$  is in correct state, and if the behavior of the affected entities complies with the behavior expected by the unaffected system parts in case reconfiguration had not taken place. The term *affected entities* denotes the set of

entities on which some reconfiguration operation is executed. Each aspect of the correctness notion is addressed in the remainder of this section.

### **Structural Integrity**

Structural integrity requirements constrain the structure of a system, i.e., they constrain how entities are related [14].

Reconfiguration may affect the structural integrity of a system, and corrective measures must be taken to prevent this. For example, us consider the replacement of one component by a new version of the component in a component-middleware-based system. Clients of the component being replaced should be capable of invoking the operations of this component 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 component must satisfy the interface definitions of the original component, and (ii) the clients should have a valid reference to the new version of the component.

### **Mutually Consistent States**

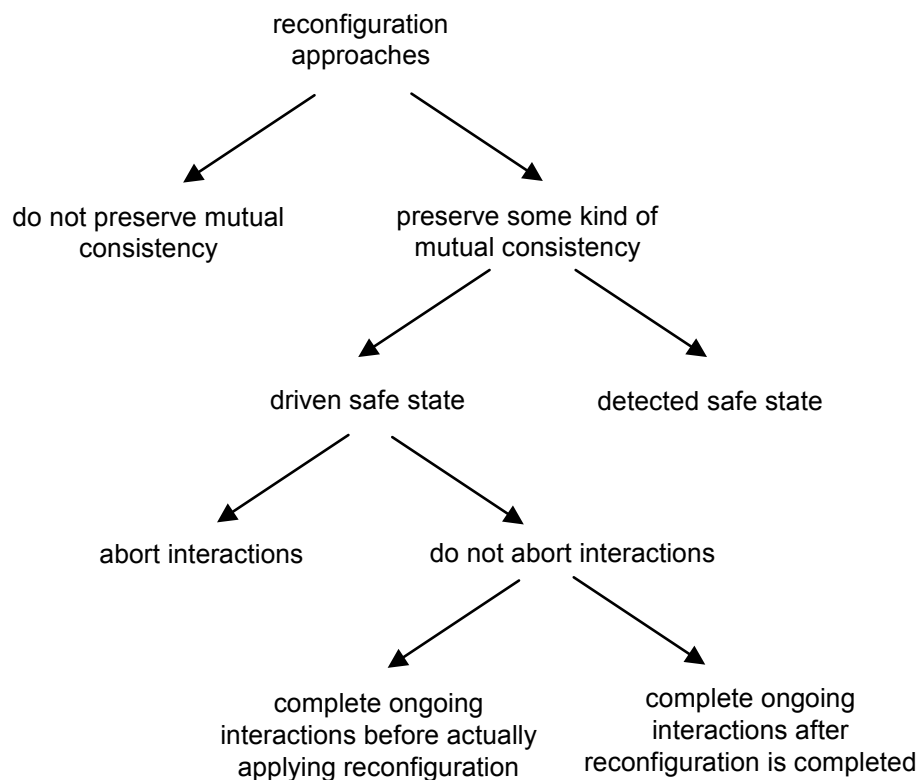
Entities in a distributed system need to be in mutually consistent states if they are to interact successfully with each other. Entities are said to be in mutually consistent states, if each interaction between them, on completion, results in a transition between well-defined and consistent states for the parts involved [14]. We define interactions to be the only means by which entities can affect each other's state.

For example, in a system with two components, component *A* invokes an operation on *B*. Components *A* and *B* are said to be in mutually consistent states if and only 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.

Reconfiguration approaches provide mechanisms to transform systems with entities in mutually consistent states into resulting systems that maintain this mutual consistency. This is done by defining a *reconfiguration-safe state* (or shortly safe state) in which reconfiguration can be applied while maintaining mutual consistency.

Figure 2 shows a classification of reconfiguration approaches according to their choices regarding the preservation of mutual consistency.



**Figure 2 Classification of reconfiguration approaches**

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 [14]: 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., [5]) 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.

In case of an approach in which ongoing interactions are interrupted and completed when reconfiguration has finished, the application developer has to implement functionality to restore the control state of the reconfigured entities, allowing the interrupted interactions to continue after reconfiguration. This control state typically includes the state of the invocation stack, program counter or thread context information. This information is closely tied to specific characteristics of the implementation code, and it is typically language- and operating system-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 reconfiguration designer. Therefore most approaches to reconfiguration do not consider this alternative. An exception is [8].

In this paper, we propose a mechanism that drives the system to a safe state without aborting interactions and that allows ongoing interactions to complete before reconfiguration is applied. This mechanism is discussed in Section 4.

### **Application 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 [14].

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 generated by the original object.

### **2.3 Impact on Execution**

Dynamic reconfiguration should introduce as little impact as possible (ideally no impact at all) on the system execution. However, during reconfiguration some system entities may temporarily become partially or totally unavailable, which can affect the performance of the system as a whole. The quantification of the impact of reconfiguration on system execution is not trivial. Some reconfiguration approaches [11, 14] 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 [4] 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 entities. In [22], it is argued that more attention should be given to the period of time during which system entities are affected by reconfiguration.

In this paper, we focus on the temporary degradation in performance of the affected entities during reconfiguration, and therefore quantify the impact on execution as the increase in response time from the perspective of the unaffected system entities.

### 3 Requirements

In the conception of our approach, we have identified the following requirements:

1. *Transparency* – our approach should hide as much as possible the complexity associated with dynamic reconfiguration from the application developer, thereby reducing the required expertise for developing reconfigurable applications and reducing the amount of effort spent on dynamic reconfiguration functionality. There should be no need for manually provided configuration information on the system. Reconfiguration support should be completely transparent to developers of clients of reconfigurable entities.
2. *Heterogeneity* – our approach should not depend on, e.g., specific implementation languages or operating systems. It should be possible to replace an entity by a new version implemented in another implementation language and with another operating system than the original version.
3. *General suitability* – our approach should minimize restrictions on applications. In particular, it should be suitable for systems with multi-threaded, stateful, re-entrant and active components.

4. *Composite reconfiguration steps* – our approach should allow reconfigurations that involve multiple entities.
5. *Correctness* – our approach should provide facilities to allow a correct and useful system to be obtained by reconfiguration.
6. *Impact on execution* – our approach should minimize the disruption to the system execution during reconfiguration.
7. *Scalability* – our approach should be suitable for large-scale systems.
8. *Common middleware* – our approach should be realizable with common middleware technologies. For example, in the case of CORBA, no changes should be required to the CORBA specification, or to CORBA implementations.

## 4 Approach

In this section, our approach to dynamic reconfiguration is detailed. We present the supported reconfiguration operations and explain how these operations can be combined in single and composite reconfiguration steps. In the sequence, we explain how the approach addresses each correctness requirement, and we conclude by defining upper bounds for the impact on execution during reconfiguration.

### 4.1 Supported Reconfiguration

Entities subject to reconfiguration are called *reconfigurable components*. A reconfigurable component is an application component that can be manipulated through reconfiguration operations, namely *creation*, *replacement*, *migration* and *removal*.

Component *creation* allows run-time creation of an application component. Component *removal* is the converse of component creation. These operations do not

offer challenges from the perspective of reconfiguration management, since applications are expected to cope with them.

Component *replacement* allows one version of a component to be replaced by another version, while preserving component identity. We use the term version of a component to denote a set of implementation constructs that realizes the component. The new version of a component may have functional and Quality-of-Service (QoS) properties that differ from the old version. For example, the new version may correct faults in the original version, or implement additional functionality. The reconfiguration designer is responsible for assuring that the new version of a component satisfies both the functional and QoS requirements of the environment in which the component is inserted. In addition, in our definition of replacement, the new version of component may run in another location or in another type of execution environment supported by the component-middleware platform, e.g., a different programming language and/or operating system.

The aim of the reconfiguration approach is to provide replacement transparency, which masks the replacement of a component from components that interact with it.

Replacement requires special attention from the perspective of reconfiguration management, since replacement threatens application consistency.

Component *migration* means that a component is moved from its current location to a new location. Because in our approach replacement can involve changing the location, we consider migration a special type of replacement in which the version of the component does not change.

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:

1. the execution of a reconfiguration operation on a single component, in which case it is called a *simple reconfiguration step*; or
2. the execution of reconfiguration operations on several distinct components, in which case it is called a *composite reconfiguration step*.

Composite reconfiguration steps are often required for reconfiguration of sets of related components. In a set of related components, a change to one component  $A$  may require changes to other components that depend on  $A$ 's behavior or other characteristics. A particular case of a composite reconfiguration step is the replacement of multiple components. A common usage example would be to replace all components of the same type with a new version.

Since the replacement is the most complex operation in our dynamic reconfiguration approach, we will describe our approach by explaining how replacements are realized.

## ***4.2 Structural Integrity***

The main issues that have to be dealt with for a middleware-based system with respect to preserving structural integrity are referential integrity and interface compatibility.

*Referential integrity* becomes an issue whenever a component reference changes. A component reference is defined as a value that denotes a particular component, and is used by the middleware infrastructure to identify and locate the component.

References acquired by clients prior to reconfiguration may be invalidated due to reconfiguration. If a reference points to a component that no longer exists, the established logical binding between a client and a target component is broken. In order to re-establish the binding after reconfiguration, we provide a logically central

point of contact for clients to find the component with invalidated component references.

To preserve *referential integrity*, a new version of a component must satisfy the original interfaces. In component-middleware technologies, interfaces satisfy the Liskov substitution principle [13]:

*“If for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$ , then  $S$  is a subtype of  $T$ .”*

This means that to satisfy the original interfaces, the new interfaces have to implement the original interfaces, or implement interfaces that are subtypes of the original interfaces. We refer to this as a *conformant* replacement. A non-conformant replacement is possible by introducing a wrapper component that is conformant, or by replacing all the clients of the replaced component as part of the same (composite) reconfiguration step.

### ***4.3 Mutually Consistent States***

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 postponing interactions that would prevent the system from reaching the safe state;
2. Detect that the safe state has been reached; and
3. Apply reconfiguration;

In this approach, the system is said to be in the reconfiguration-safe state when each affected component: (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 components is serving requests or waiting for outgoing requests to complete.

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

An active component 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. These capabilities should be provided by the component developer. Once the set of affected system components is defined, all active components 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 components 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 component.

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 components.

In the case of a simple reconfiguration step entailing the replacement of a single non re-entrant component, all requests issued to this component are queued by the middleware platform before they reach the component. In this way, new requests are prevented from being served before the reconfiguration, and the component gets the chance to finish handling ongoing requests. In case the component is active, it is requested to exhibit reactive behavior. 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 composite reconfiguration steps entailing the replacement of a single re-entrant component, or multiple (possibly re-entrant) components simultaneously, selective queuing of requests directed to affected components is necessary. Requests issued by an affected component are part of the ‘laissez-passer’ set, since these requests have to be executed for the safe state to be reached. This implies that requests with an invocation path that contains at least one affected component also have to be included in the ‘laissez-passer’ set. In particular, re-entrant requests initiated by affected components are also included in the ‘laissez-passer’ set. All components that could otherwise issue new ‘laissez-passer’ requests are set to exhibit reactive behavior, so that no new ‘laissez-passer’ requests are generated. At some point, the existing requests are treated, all affected components 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 component in an invocation path adds its own identification to the request as an implicit parameter. Given a request and the set of affected components, 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 components has been included in the request's implicit parameters, the request belongs to the 'laissez-passer' set.

We assume that the completion of a request does not depend on other non-nested requests. Dependency on non-nested requests is an uncommon construction, which can be avoided by an application developer.

### **Applying Reconfiguration**

When all affected components are idle, the reconfiguration process can proceed. The affected components' state can be inspected and used to derive the state of the components being introduced. The reconfiguration designer may provide functions for state translation. Once new components or new versions of components have been installed, their state is properly modified. Queued requests and further new requests are redirected to the new version of a component.

A reconfiguration step can be aborted and rolled-back in case failures are detected during reconfiguration. Optionally, reconfiguration may be aborted and rolled-back in case the duration of the reconfiguration step exceeds some configured timeout.

### ***4.4 Application-state Invariants***

We adopt a scheme proposed in [14], in which invalidated invariants can be identified and re-established by the reconfiguration designer with little assistance from the component developer. This scheme consists of requiring components to provide general-purpose state access operations that can be invoked by a third party to query or adjust the state of these components.

In this scheme, the component developer decides on the relevant application state to be exposed by these access operations. In addition, the reconfiguration designer may provide state translation functionality.

One might argue that this scheme breaks encapsulation, since it allows external access to a component's internal state. Nevertheless, this form of introspection is unavoidable for dynamic reconfiguration.

#### ***4.5 Impact on Execution***

We quantify the impact on a system's execution in terms of the increase in response time experienced by clients of affected components during reconfiguration.

In our approach, invocations initiated by the clients of an affected component after the beginning of the reconfiguration and before the end of the reconfiguration are queued. This causes an increase in response time that is dependent on the application. Since we wait for ongoing interactions involving the affected components to finish, the expected increase in response time is proportional to the expected duration of those interactions. Therefore, this increase is higher for applications with long-lived interactions. For reactive components, the upper bound for this increase is the duration of the longest pending invocation in the set of affected components at the moment reconfiguration is initiated. For active components, the amount of time taken for the component to exhibit a reactive behavior has to be added to this upper bound.

### **5 Proof-of-Concept**

We have designed and prototyped a Dynamic Reconfiguration Service (DRS) that serves as a proof-of-concept for our approach. This service is based on CORBA 2.x [6], and reconfigurable components are realized as CORBA objects. The DRS has been used to implement object migration in a load distribution service for CORBA [21].

## 5.1 High Level Design

The DRS consists of a Reconfiguration Manager, a Location Agent and one or more Reconfiguration Agents. Figure 3 depicts its high level design.

The *Reconfiguration Manager* is the central component of the DRS 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 Object Request Broker (ORB) instance that mediates invocations for reconfigurable objects. Typically, there is one ORB instance per 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.

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, or at request of the application. Factories shield the DRS from the specific support to deployment offered by different languages, operating systems or virtual

machines, such as, e.g., Windows Dynamic Linking Libraries and the Java class loader.

The *State Translator* implements a state translation function, if needed for a reconfiguration. This state translation is application dependent.

As Figure 3 also shows, the service concerns both the reconfiguration designer and the object developer. The reconfiguration designer accesses the service of the DRS to request the execution of reconfiguration steps, and, if necessary, to supply the State Translator. The object developer has to supply the application-specific Reconfigurable-Object Factories and Reconfigurable Objects that comply with the interfaces defined by the service. The Reconfiguration Manager, Location Agent and the Reconfiguration Agents are supplied by the developer of the DRS.

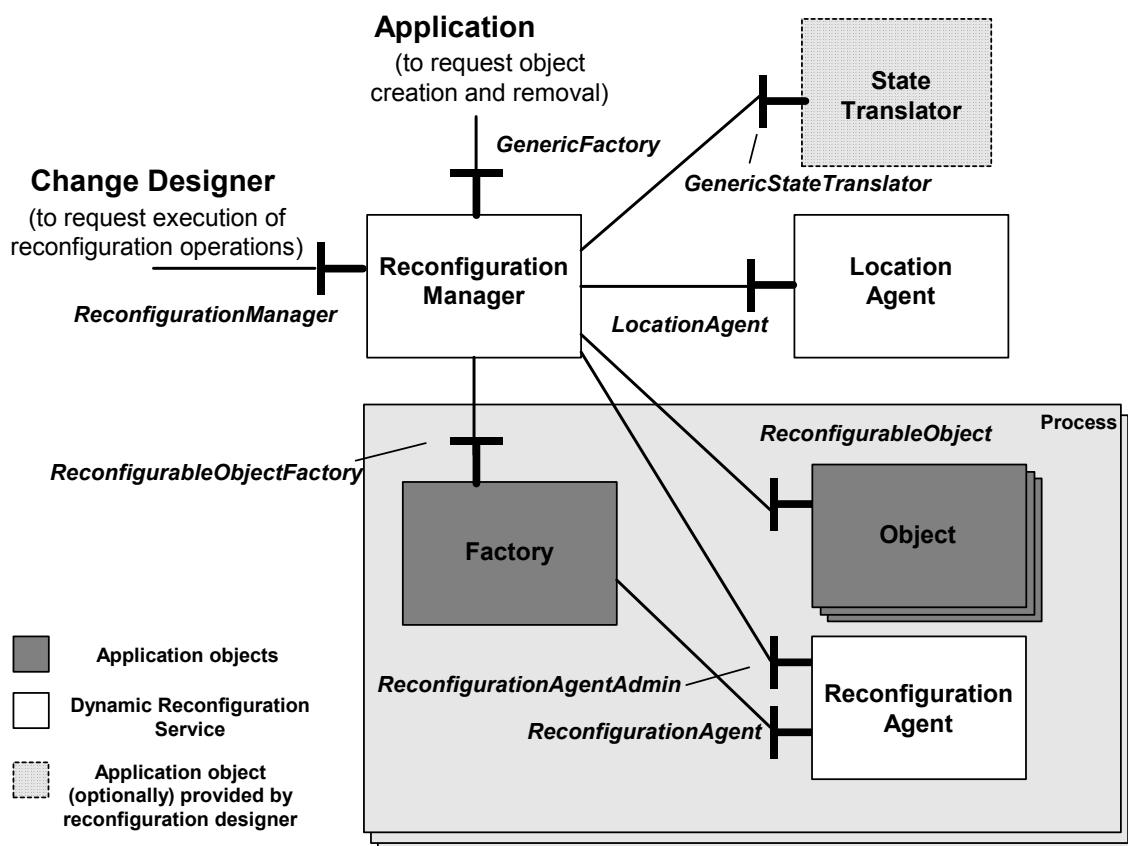


Figure 3 High level design of the Dynamic Reconfiguration Service

## **Performing a Reconfiguration Step**

In this section, we describe how reconfiguration steps are applied in our design. We first explain a simple reconfiguration step with a single object replacement, and then generalize the explanation to a composite reconfiguration step with multiple object replacements. All other cases are left as an exercise to the reader.

In a reconfigurable object replacement, the Reconfiguration Manager delegates the creation of the new version of the object to a local Reconfigurable Object Factory. In the sequence, the Reconfiguration Manager notifies the affected reconfigurable object to exhibit reactive behavior and its Reconfiguration Agent to queue incoming requests that are not part of the ‘laissez-passer’ set. When the object is ready for reconfiguration, the Reconfiguration Agent notifies the Reconfiguration Manager. The state transfer and optional state translation are conducted, the new location of the object is registered with the Location Agent, and the previous version of the object removed through interaction with its factory. The queued requests are redirected to the new version of the object.

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

In the following two paragraphs we present solutions to two relevant technical issues addressed in the design of the DRS: the queuing and redirection of requests. The detailed design can be found in [2] and [3].

## **Location Independent Object Reference**

To maintain structural integrity, clients have to be redirected to the new object, which typically has a different object reference. To solve this we introduce a location independent object reference that remains valid even if the object changes location.

We can do this transparently for the client side application developer by exploiting the request forwarding mechanism that is part of the CORBA specification [6]. The Location Agent directs client to the actual location of the object using this request forwarding mechanism. If this location is changed, the client ORB will go back to the Location Agent and will then be re-directed to the updated location. The overhead for implementing this solution is limited to the first invocation of a client on the reconfigured target object.

## **Selective Queuing**

In order to bring the system to the reconfiguration-safe state, we implement selective queuing of requests. Requests that do not belong to the ‘laissez-passer’ set should be queued, and after reconfiguration should be redirected to the new version of the object. We implement selective queuing and redirection in the middleware, thereby making it transparent for client- and reconfigurable object developers.

The selection is done at the server-side ORB, based on the invocation path contained in implicit parameters. For requests to be queued, a message is sent to the client-side ORB, which queues the request. The client-side ORB reissues the request when notified by the Reconfiguration Manager or after some configurable time-out.

We use CORBA Portable Interceptors [6] on both client and server-sides to add the invocation path as an implicit parameter to every request, and to implement selective queuing of requests. Portable Interceptors are a standardized mechanism to add

reflective capabilities to CORBA [20]. By using Portable Interceptors in our implementation, no modifications to application and ORB code are required.

## ***5.2 Overhead during Normal Operation***

Because of the functionality of the DRS implemented in client and server-side interceptors, the DRS causes some extra overhead for every invocation during normal operation. In our test scenarios (described in more details in [3]), this caused a fixed increase of approximately 0.13 ms in invocation response time.

To establish an upper bound on the relative increase in response time, we have measured the increase in response time for the scenario with the lowest response time. In this scenario we have a network with low latency, server service time approximating zero and an operation without any parameters. The average response time observed without the DRS was 1.04ms. With the DRS, this response time increased to 1.17ms, constituting a relative increase of 12.4% in response time.

These measurements however are valid for the implementation environment adopted (namely, the ORBacus ORB implementation and the Java implementation language). Considerations for other particular implementation environments should be made.

## **6 Related Work**

In this section we discuss related work, and compare it to our approach. We only consider approaches that address correctness requirements and that drive the system to a reconfiguration-safe state without abortion interaction.

## **Kramer and Magee**

The early work of Kramer and Magee [11, 12] has influenced the subsequent works of many others on dynamic reconfiguration. Our definition of reconfiguration management and the reconfiguration model from Section 2 is based on their work.

This approach requires the developer to provide configuration information in the form of directed graphs that represent the entities that are present in the system, the transactions between these entities, and possible dependencies between these transactions. Apart from the fact that this does not fit the computational model underlying component-middleware-based systems, providing this configuration information violates our transparency requirements because it requires time and expertise from the developer. This approach also violates our scalability requirement because for large-scale systems it will be undoable to provide and maintain this information for all entities and transactions.

Since all entities capable of initiating a transaction directly or indirectly with an affected entity have to be passive to reach the safe state, even small reconfigurations involving a few entities can result in a substantial impact on execution. The transparency requirement is also violated in this approach because the application programmer must implement all active entities of the system with capabilities to become passive. This contrary to our approach, in which this is only required for active reconfigurable components.

## **Goudarzi**

Goudarzi's approach [14] assumes that components in the system do not interleave interactions, i.e., a component participates in one interaction at a time. Therefore, it is possible to drive a component to a passive state by blocking its execution when no interactions are in progress, and only temporarily unblocking its execution in case of

interactions that originate from an affected component. The main benefit of this approach is that component developers do not have to implement functionality to drive components to the passive state. Nevertheless, this approach violates our general suitability requirement because the class of distributed systems that can be reconfigured is limited to components that participate in at most one interaction at a time. For middleware-based systems this would mean that multi-threading and re-entrance cannot be supported.

### **Bidan et al.**

In Bidan et. al.'s approach [4], the implementation of a reconfiguration service in CORBA is considered. As is the case for our prototype, a reconfigurable entity is a CORBA object. In this approach, the reconfiguration service 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*. All client applications and target objects must implement a 'passivate' operation to block the initiation of requests on a specific outgoing link. The algorithm guarantees the reachability of a safe state by sending 'passivate' messages to all the clients of an object and then to the object itself.

Unlike our approach, Bidan et. al.'s approach is not suitable for a system with re-entrant invocations, violating the general suitability requirement. Furthermore, the approach does not support composite reconfiguration steps.

Compared to our approach, the functionality for dynamic reconfiguration is more at the application layer, requiring not only server objects but also client applications to incorporate support for reconfiguration, for example to re-establish connections to maintain structural integrity. This violates the transparency requirement.

## **Tewksbury et al.**

Tewksbury et al. [19] propose an approach that extends a Fault Tolerant CORBA [17] implementation named Eternal [15] with dynamic reconfiguration capabilities. This approach exploits the replication functionality provided by Eternal. The basic idea behind this approach is to replace old versions of the objects by an intermediate version that implements both the behavior of the old version, and of the new version of the object. The intermediate version of the object is mostly generated based on the code for the old and the code for the new object. After all old objects have been replaced by intermediate objects, and all intermediate objects are in a safe state in which they do not process any invocations, all intermediate objects simultaneously switch to the new behavior using a special method that is implemented by the intermediate objects. The intermediate version can then be replaced by the actual new versions of the objects. Eternal is based on operating system interceptors, which are ORB independent, but cannot intercept invocations between co-located objects [20].

The requirements for strong replica consistency presented in [15] imply that an object can only service one invocation at a time, thus limiting scalability and violating our general suitability requirement. Similarly to Goudarzi et. al., re-entrant invocations are also ruled out.

Because the intermediate object mixes the old and new implementation code, access to the source code is required, and the old and new version of the code have to be in the same (version of) programming language. This violates the heterogeneity requirement.

## **Online Upgrades Draft Specification**

The OMG is in the process of defining an Online Upgrades specification [16]. The draft specification is limited to the upgrade of a single object. The application

developer has to implement similar state access functionality as in our approach. In addition, the application developer has to determine when the component is in a safe state, violating our transparency requirement.

## **7 Conclusions**

Dynamic reconfiguration functionality increases the availability of a distributed system. We propose a new approach for dynamic reconfiguration of middleware-based systems that fulfils our requirements with respect to transparency, heterogeneity, general suitability, support for composite reconfiguration steps, correctness, minimal impact on execution, scalability and common middleware. Our approach does not only make it possible to dynamically reconfigure middleware-based systems, but also uses the middleware to achieve this in a transparent manner by embedding the functionality of dynamic reconfiguration in the middleware.

We exploit the middleware's reflective capabilities to discover at run-time the configuration information required by reconfiguration management, to preserve structural integrity by transparently updating component references and to ensure a mutually consistent state by driving the application to a reconfiguration-safe state. Because of our usage of middleware, we do not have to rely on manually provided configuration information, which significantly increases transparency and scalability. The main burden our approach puts on an application developer is that he has to implement state access methods.

Our approach could be extended with the possibility to abort invocations that are long running and do not affect the state of a component. This would lead to a hybrid abortion-avoidance and abortion approach that could further decrease the impact on execution.

## Acknowledgements

This research was done as part of the Amidst project (<http://amidst.ctit.utwente.nl>).

We especially want to acknowledge Luís Ferreira Pires for his contribution, and thank Jeroen Schot and Bastien Peelen for their useful comments.

## 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”, *Proc. 19th Brazilian Symposium on Computer Networks (SBRC 2001)*, Santa Catarina, Brazil, May 2001. Also appeared as CTIT Technical Report TR-CTIT-01-06 in February 2001.
- [2] J. P. A. Almeida, “Dynamic Reconfiguration of Object-Middleware-based Distributed Systems”, *M.Sc. thesis*, University of Twente, June 2001.
- [3] J. P. A. Almeida, M. Wegdam, M. van Sinderen, L. Nieuwenhuis, “Transparent Dynamic Reconfiguration for CORBA”, *Proc. 3rd International Symposium on Distributed Objects & Applications (DOA 2001)*, Rome, Italy, September 2001.
- [4] C. Bidan, V. Issarny, T. Saridakis, A. Zarras, “A dynamic reconfiguration service for CORBA”, *Proc. IEEE International Conference on Configurable Distributed Systems*, May 1998.
- [5] T. Bloom, M. Day, “Reconfiguration and module replacement in Argus: Theory and Practice”, *IEE Software Engineering Journal*, 8(2), March 1993.
- [6] Object Management Group, “The Common Object Request Broker: Architecture and specification”, version 2.4.1, formal/00-11-07, November 2000.

- [7] SUN Microsystems, “Enterprise JavaBeans Specification”, Version 2.0, 22 August 2001.
- [8] C. Hofmeister, “Dynamic Reconfiguration of Distributed Applications”, Ph.D. thesis, 1993, University of Maryland, USA.
- [9] ITU-T / ISO, “Open Distributed Processing - Reference Model - Part 3: Architecture”, ITU-T X.903 | ISO/IEC 10746-3, November 1995.
- [10] M. Kirtland, “Object-Oriented Software Development Made Simple with COM+ Runtime Services”, *Microsoft System J.*, Vol. 11, p. 49, Nov. 1997.
- [11] J. Kramer, J. Magee, “Dynamic configuration for distributed systems”, *IEEE Transactions on Software Engineering*, 11(4), pp. 424-436, April 1985.
- [12] J. Kramer, J. Magee, “The evolving philosophers’ problem: dynamic change management”, *IEEE Transactions on Software Engineering*, 16(11), pp. 1293-1306, November 1990.
- [13] Barbara Liskov, “Data Abstraction and Hierarchy”, *ACM SIGPLAN Notices*, 23(5), pp. 17-34, May 1988.
- [14] K. Moazami-Goudarzi, “Consistency preserving dynamic reconfiguration of distributed systems”, Ph.D. thesis, Imperial College, London, March 1999.
- [15] P. Narasimhan, Transparent Fault Tolerance for CORBA, Ph.D. thesis, University of California, Santa Barbara, Dec. 1999.
- [16] OMG, “Online Upgrades Draft Adopted Specification”, ptc/02-07-01, 8 July 2002.
- [17] OMG, ‘Fault Tolerant CORBA Specification”, version 1.0, ptc/2000-04-04, April 2000.

- [18] P. Oreizy, N. Medvidovic, R. Taylor, “Architecture-based runtime software evolution”, *Proc. of the International Conference on Software Engineering*, April 1998.
- [19] L. A. Tewksbury, L. E. Moser, P. M. Melliar-Smith, “Coordinating the Simultaneous Upgrade of Multiple CORBA Objects”, *Proc. 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, Rome, Italy, September, 2001.
- [20] Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, Bart Nieuwenhuis, “Using message reflection in a management architecture for CORBA”, *Proc. of the 11th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2000)*, Springer-Verlag LNCS Vol. 1960, pp. 230-242, Dec. 2000, Austin, Texas, USA.
- [21] Maarten Wegdam, Lambert J.M. Nieuwenhuis, Marten J. van Sinderen, “Load Distribution for Component-Middleware-Based Distributed Systems”, submitted to *Bell Labs Technical J.*, Wiley Periodicals Inc., 8(3), 2003.
- [22] M. A. Wermelinger, “Specification of software architecture reconfiguration”, Ph.D. thesis, Universidade Nova de Lisboa, Portugal, Sept. 1999.