Electronic Notes in Theoretical Computer Science 65 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 9 pages

Component-Based Applications: A Dynamic Reconfiguration Approach with Fault Tolerance Support

Thais Vasconcelos Batista [1,2], Milano Gadelha Carvalho [3]

Informatics Department
Federal University of Rio Grande do Norte
Natal-RN, Brazil

Abstract

This paper presents a mechanism for dynamic reconfiguration of component-based applications and its fault tolerance strategy. The mechanism, named *generic connector*, allows composing a component-based application as a set of services with no previous knowledge about the specific components that will provide some services. The components will be selected at runtime. The objective of this work is to offer a mechanism that tries to satisfy every invocation under its responsibility and makes failures as transparent as possible. It is an important tool to compose applications through the reuse of existing components because it frees the programmer from the task of searching components in order to use them and, also, from solving some types of failures.

1 Introduction

Development of distributed applications using a component-based approach has gained importance in software engineering as a promising manner for the reduction of costs and time of software development. Component-based development focuses on composing applications by assembling prefabricated piece of software named components [15].

In this context, there is a challenge for mechanisms that support the development of distributed applications, including fault-tolerance behavior [8] in order to produce well-functioning applications. Fault tolerance is the ability of a system to behave in a well-defined manner once fault occurs [6]. While

[1] Research for this paper was done with valuable support from CNPq (Brazilian Council for Development of Science and Technology) under process 68.0103/01-5
[2] Email: thais@ufrnet.br
[3] Email: milano@lcc.ufrn.br

component-based development is widely accepted as an approach for the composition of distributed applications, support for fault-tolerance is still found lacking.

Fault Tolerance is especially important when dynamic reconfiguration is a main requirement of an application. In general, applications that typically need dynamic reconfiguration support are mission-critical applications such as flight control systems, financial and health applications. This class of applications does not permit the occurrence of faults, once availability is a key issue.

In this paper, we present a mechanism used in application composition development that offers support for dynamic selection of components and includes fault-tolerance treatment, avoiding that faults occur when the mechanism is controlling the application. This mechanism, named *generic connector* [2], does the dynamic selection of components using as a search parameter the signature of a method that the component must provide. The generic connector makes it possible to configure an application as a set of services with no previous knowledge about the specific components to execute the services. The components will be found by the generic connector at runtime. Besides, the generic connector does the invocation of the method upon the selected component and returns the results. The objective of the fault tolerance support is to guarantee that an invocation done through the generic connector is a fault-free invocation. The novelty of this paper is to discuss a fault-tolerance strategy to the generic connector once the original proposal does not cope with this issue.

This mechanism is an important tool to compose applications through the reuse of existing components because it frees the programmer from the task of searching components in order to use it and, also, from solving some types of failures. In a distributed environment the overload of searching components spread in various places may discourage the programmer to reuse components. With the generic connector the programmer merely uses services of unknown objects in the same way as he/she uses services of known objects.

This paper is organized as follows. Section 2 presents the generic connector mechanism. Section 3 describes the fault tolerance support provided by the generic connector. Finally, section 4 presents the final remarks.

2 The Generic Connector

The generic connector is one of the tools of LuaSpace [1], an environment for the development of component based applications that combines the CORBA platform with the interpreted and procedural scripting language Lua [7] used to glue components. The application is written in Lua and can be composed by components implemented in any language that has a binding to CORBA. Another tool of LuaSpace is LuaOrb [5], a binding between Lua and CORBA, based on CORBA's Dynamic Invocation Interface (DII) that provides dynamic

```
p = generic_createproxy()
p:play(‘‘musicfile.mid’’)
```

Fig. 1. Configuration program example with the generic connector

access to CORBA components as any other Lua object.

The generic connector is a Lua object created by `generic_createproxy` function that returns a proxy that is not bound to a specific component but refers to the generic connector itself. To invoke a service of an unknown object, an "orphan" service is invoked in the same way as a service of a known component is invoked, but with the proxy that represents the generic connector in the place of the name of a component, suggesting that the service will be provided by the generic connector.

To illustrate the mechanism we present a simple example. Figure 1 shows a piece of a configuration program that invokes `play` method through the generic connector, i.e., without specifying the component that will provide the service. In the execution of this program, the generic connector will find a component whose interface describes a `play` method with *file* as input parameter, such as the interface *Player* shown in Figure 2.

```
interface Player {
   void play(in file);}
```

Fig. 2. Player Interface

When a method is called upon the proxy of the generic connector, the Lua interpreter intercepts the invocation and implicitly invokes the generic connector implementation that is illustrated in Figure 3. The generic connector invokes a search function to look for some component that offers the service specified in the call. The search can proceed upon the standard repository (Naming or Trading services) or upon the `configuration table`, a table managed by the generic connector that caches the identification of the components selected to execute services called by an application via the generic connector. The generic connector can be instructed by the application programmer to always look for services in the repository, gaining in reconfigurability, or to first check its cached data in the configuration table, gaining in efficiency. The default behavior is first to do the search upon the configuration table and the search process proceeds upon the repository only if no component is found in the configuration table to provide the service. In order to change this default behavior, the programmer can set to true the boolean value of the `userRequest` variable and the search process will be done upon the standard repository.

After selecting the component that provides the service, the generic connector mounts the solicitation, a sequence of commands written in Lua to create a proxy for the selected component and to activate the service. The next step

to be performed by the generic connector is to register the method with its provider in the configuration table. Finally, the generic connector does the invocation and, after execution, returns the result sent by the method invoked.

This mechanism allows configuring an application as a set of services with no previous knowledge about the specific components that implements the service. Those components, if available, would be inserted into the application during its execution. At runtime, the generic connector searches for components that can provide the service stated in the application configuration program, activates the service and returns the result to the client. This mechanism introduces a great flexibility in application modeling once the developer can abstract away about specific components. It also addresses dynamic reconfiguration because different invocations of the same service may result in the selection of different components.

3 Fault Tolerance in the Generic Connector

When a method invocation is done using the generic connector proxy, the generic connector search function uses the signature of the invoked method to do a syntactic matching between this signature and the method signatures of the components registered in the repository. This issue avoids syntactic error associated to mismatch types. However, the generic connector original proposal does not detect and correct runtime errors associated with the execution of a service invoked by the generic connector. In this way, the application may exhibit unexpected behavior due to failures associated to the components selected by the generic connector. A proper behavior for a dynamic reconfiguration mechanism that provides support for application development is to avoid to propagate failures to the programmer of the application. Despite the failures, the application should be able to continue operating.

Two different types of failures can occur when a method of a component is invoked by the generic connector:

- `The component or the method is not available anymore`. In general this situation occurs when the invoked method or the component is not running anymore. Since the generic connector looks for a component in the repository or in the configuration table, this situation occurs due to the following problems:
  - The search is done upon the repository:
    * the component stopped running and it did not delete its entry in the repository;
    * the component interface has changed and it did not register the new interface in the repository. In this case, the repository has an old piece of information.
  - The search is done upon the configuration table and the information about the component selected is old

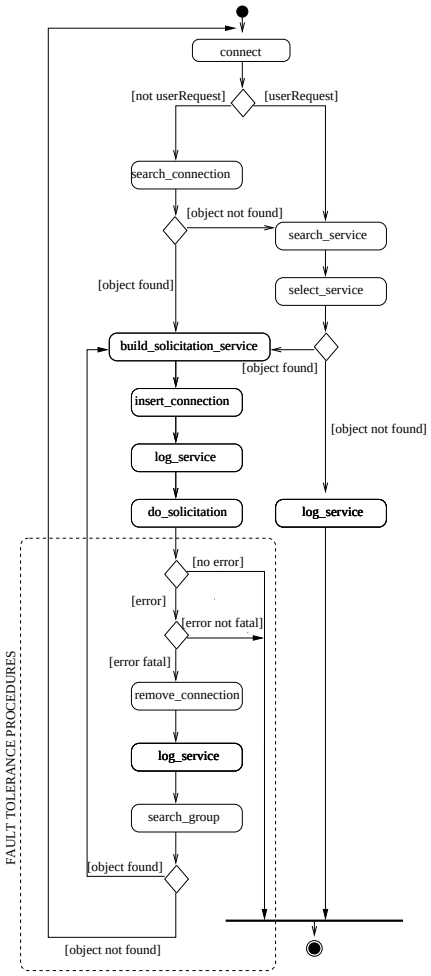

Fig. 3. Structure of the generic connector with fault tolerance

- `The method cannot be executed`. This failure occurs due to problems regarding method execution, such as a malformed parameter passed in the method invocation. In this case, both component and method invoked are ready to execute and, during the execution an internal problem not associated with the component or method happened . For instance, in the example of Figure 1, an implementation of *Player* interface (shown in Figure 2) will open a file before playing it. The configuration program passes the file name as a parameter of this method. If the file name is wrong, an error is returned (corresponding file does not exist). We classify this kind of error as a *non fatal error* because the method can be executed if the programmer repairs the file name.

In this context, we propose a fault tolerant mechanism that, as usual in fault-tolerance works [6], applies the idea of dividing fault-tolerance actions into detection and correction phases. The general idea of the mechanism is to intercept the runtime error returned after the invocation made by the generic connector and to search for another component that provides the same

service. This process must be repeated until a component that can execute the service is found or until there are no more components that offer the service. This procedure takes place transparently to the programmer and to the configuration program. The objective is to mask fault of components and to, automatically, substitute them.

Middleware platforms, such as CORBA, provide a component repository that probably contain different components offering a same service. As the generic connector works upon a CORBA platform, the redundancy of service providers in CORBAs repository can be explored by the fault tolerance mechanism.

Figure 3 illustrates the UML [4] activity diagram of the generic connector including fault tolerance support. The dotted line area highlights the fault tolerance procedures. Below, we explain them.

In the *detection procedure* that begins after the `do_solicitation` qfunction activates the service, the generic connector remains waiting messages sent by the running component or by the underlying execution platform (CORBA and LuaOrb). When it receives a message, it verifies if the message is an error or a service execution result. When an error is captured, the next step is to classify it as a fatal or a non fatal error. A fatal error is an error impossible to recover with respect to the specific component invoked, such as *component or service not available anymore*. A non fatal error means that the component and the method are running and can execute the service but there is an internal error, for instance, an error in the parameter passing that causes the failure of the invocation. This kind of error is reported to the programmer that can, probably, remove it. In the previous example, if the programmer types a wrong letter in the file name, this is a not fatal error and it can be easily corrected by the programmer.

In order to support fault tolerance, the generic connector implementation was modified to include a new table, named *component group table*. This table caches the list of identifications of the components that provide the method invoked upon the generic connector. In the first search done upon the standard repository, the `search_service` function will return a list of components whose interface has the method invoked upon the generic connector and records this list in the component group table. This facility avoids a new search in the repository in case of failures of components or methods. In the original implementation this table does not exist. It is included in order to cope with fault tolerance. Without this table, in case of failure, the search procedure will query the standard repository and the recovery will be a delayed task. The component group table follows the idea of grouping objects with any kind of redundancy useful for fault tolerance support [9].

Only fatal errors are handled by the *correction procedure*. After detecting a fatal error, the *correction procedure* invokes the `remove_connection` function to remove the register corresponding to the fault component from the configuration table and from the component group table. In this way, the *cor-*

rection procedure updates both tables. Then, the *correction procedure* invokes `log_service` function that logs all information about the invocation done upon the generic connector, for instance, the `status` parameter informs if the corresponding execution has completed or failed. Next, the `search_group` function is used to search, in the component group table, for another component that offers the method whose invocation failed. If there is no cache in the *component group table* regarding such method, the correction procedure forwards the invocation to the generic connector beginning procedure. This procedure will search, in the repository, for another component that also offers the method invoked upon the generic connector. It is not necessary to select a component that adheres to the same high-level interface as the faulty one. It is only necessary to select a component that provides the method invoked upon the generic connector, regardless of the other methods that the component provides.

The generic connector does not guarantee that there is an available component that offers the method invoked upon the generic connector but, if there is at least one recorded in the repository, it will be selected.

After selecting the component (from the component group table or from the repository), the generic connector does the invocation. In this way, the generic connector dynamically substitutes faulty components and so it is a dynamic reconfiguration mechanism.

With this behavior, the efforts of handling failures are shifted to the generic connector implementation, releasing the programmer of doing this task.

4 Final Remarks

This paper presented the generic connector and its fault tolerance strategy. The generic connector is a mechanism that dynamically selects components to provide services required by an application. The objective of this work is to offer a mechanism for component-based applications development that tries to satisfy every invocation under its responsibility and makes failures as transparent as possible. The fault tolerance strategy includes detection and correction phases. In order to cope with fault tolerance, the original proposal of the generic connector was extended: new functions are inserted and a new table was created to record a list of different components that can satisfy an invocation. This information is used by the fault tolerance procedures.

The purpose of inserting fault tolerance capability to this mechanism is to shift the burden of searching and substituting faulty components from the programmer to the mechanism.

The level of fault tolerance that the generic connector can provide depends on the availability of components that offer the method invoked upon the generic connector. If there is a set of components available to satisfy the method invocation, the level of fault tolerance support can be high. Otherwise, in the extreme case of not existing any component to provide the service, the

programmer will be informed and the fault tolerance strategy will not be successful.

Support for searching distributed objects is so recognized as a fundamental service for component-based programming that CORBA platform offers Naming and Trading services [10] for this proposal. However, the generic connector functionality goes beyond that provided by these services that only supply repositories with some well-defined ways to query about component information stored there. The generic connector gives the programmer a uniform way to use services from previously selected components or from unknown components. Besides, it also activates the required service from the selected component and returns the results. In addition, no fault tolerance support is provided by Naming and Trading services. There is another CORBA service [11] to cope with fault tolerance issues. However, there is no available implementation of this service, probably, because it depends on other CORBA services, such as the CORBA notification service and the CORBA object group service that are not easily found in the current CORBA implementations.

Although the generic connector has been implemented using Lua and a specific binding to CORBA (LuaOrb), the same mechanism can be implemented using other programming languages and other binding to CORBA. It is necessary that the language and the binding to CORBA provide dynamic features that allow configuring applications without relying on previous declaration of components that will only be found at runtime.

This mechanism can represent an important role in reusing existent components because it provides a way for the programmer to use a component without being involved in searching it. The search process can be a laborious task mainly in a distributed environment and its complexity can discourage the programmer to reuse components. With the generic connector, the search is done in an automatic way and fault tolerance is also supported. These facilities release the programmer from all low-level technicalities associated with searching components and with solving some types of failures, such as "component unavailable".

An extension of the generic connector is under development and consists in allowing the programmer to associate some properties to the method invoked. The search process will look for a component that provides the method and also satisfies the properties. It can then produce a semantically more adequate result with respect to the original search process that only computes a syntactic match between a method signature and the services registered in the repository.

References

[1] Batista, T. and N. Rodriguez, Dynamic Reconfiguration of Component-based Applications, *5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000)*, IEEE Computer Society, (Limerick,

Ireland, 2000) 32–39.

[2] Batista, T., C. Chavez and N. Rodriguez, Dynamic Reconfiguration through a Generic Connector, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, CSREA Press, (Las Vegas, USA, 2000) 1127–1132.

[3] Bernstein, P., *Middleware*, Communications of the ACM, 39:**2**, (1996).

[4] Booch, G., J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide" Addison-Wesley, 1999.

[5] Cerqueira, R., C. Cassino and R. Ierusalimschy, Dynamic Component Gluing Across Different Componentware Systems, *International Symposium on Distributed Objects and Applications (DOA'99)*, IEEE Press, OMG, (Edinburgh, Scotland, September, 1999) 362–371.

[6] Gartner, F., *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments*, ACM Computing Surveys, 31:**1**, March,(1999),1–26.

[7] Ierusalimschy, R., L. H. Figueiredo and W. Celes, *Lua - an extensible extension language*, Software: Practice and Experience, 26:**6**, (1996), 635–652.

[8] Jalote, P., "Fault Tolerance in Distributed System," Prentice-Hall, 1994.

[9] Maffeis, S., Adding Group Communication and Fault-Tolerance to CORBA, *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, USENIX, (Monterey, California, June, 1995).

[10] Object Management Group (OMG), "CORBA services: Common Object Services Specification – Revised Joint Submission", OMG Document Formal,orbos/99-08-01, 1998.

[11] Object Management Group (OMG), "Fault Tolerant CORBA", OMG Document Formal,orbos/99-12-19, 1999.

[12] Orfali, R., D. Harkey and J. Edwards, "The Essential Distributed Objects Survival Guide," John Wiley & Sons., 1996.

[13] Siegel, J., *OMG Overview: CORBA and the OMA in Enterprise Computing*, Communications of the ACM, 41:**10**, October, (1998), 37–43.

[14] Stikeleather, J., *Why Distributed Computing is inevitable*, Object Magazine, March,(1996),35–39.

[15] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 1998.