Adaptable Fault Tolerance Requirements on Component Models

Phuong-Quynh Duong[†], Elizabeth Pérez Cortés[‡*] , Christine Collet[†]

[†]LSR/IMAG Laboratory
BP 72, 38402 St Martin d'Hères, FRANCE
{*Phuong-Quynh.Duong, Christine.Collet*}*@imag.fr*

[‡]Depto. de Ingeniería Eléctrica UAMI
Av. San Rafael Atlixco 186 Col. Vicentina 09340, México D.F
pece@xanum.uam.mx

***Abstract*— Our work aims to provide adaptable fault tolerance for component-based systems through frameworks. In order to implement this approach, the underlying component model must fulfill some requirements. This paper focuses on the discussion of those requirements and the way current component models cope with. To motivate the discussion, the paper also describes briefly the approach and the preliminary results of our work.**

I. Introduction

Nowadays, applications are more and more developed under the approach of separation of concerns. This approach tries to formally separate the application specific algorithms from technical aspects[1]. Thus, it helps application developers to easily manage the non applicative tasks. Many efforts on isolation of technical aspects (e.g. concurrency control, persistency, load balancing etc.) from applicative concerns have been done so far.

Also, frequently the proposed solutions allow some degree of adaptability to provide a best suited functionality with respect to the application requirements. Some of them propose adaptability in a static way, that is technical aspects are chosen and integrated into the application before the execution. This approach is called **customization**. In other cases, resulting applications are able to evolve during execution if the requirements or the available resources in the system change. In this case we have dynamic adaptability which is called **adaptivity**.

The increasing need of highly fault tolerant computer systems make us to orientate our efforts to this aspect. Our work aims to define a framework handling both static and dynamic adaptability of the fault tolerance concern for component-based systems. For the moment we allow the customization. We have defined a framework whose instances can ensure a quality of service "on demand" to component-based systems using well proven fault tolerance techniques. The term **framework** refers to a set of interfaces and the description of the interactions among these interfaces that can be partly implemented. The customization is reached by allowing the application developer to specify, in a non ambiguous way, the required fault tolerance level. Once this is done, a specific framework is instantiated and integrated to the application code. The next step of this work is to chose a specific component model to implement our proposal. In this paper we focus on the requirements the chosen model has to fulfill and how current component models cope with. This work is a single research thread of the NODS[2] (*Networked Open Database Services*) project [**?**].

The rest of this paper is organized as follows. In Section **??**, the preliminary results of our work are presented. Section **??** discusses the requirements on component models so that our proposal can be applied and examines current component models with regard to our requirements. Finally, Section **??** describes our ongoing and future work.

II. Adaptable fault tolerance: current status

As mentioned in Section **??**, we defined fault tolerance levels [**?**] that enable the instantiation of the *customization fault tolerance framework*. In our definition, a fault tolerance level specifies the couples (*type of fault, fault tolerance form*).

We also defined the functional architecture of the customization fault tolerance framework. Figure **??** shows two main kinds of architectures we identified. Figure **??** copes with levels that do not require replication and Figure **??** handles levels that do require replication.

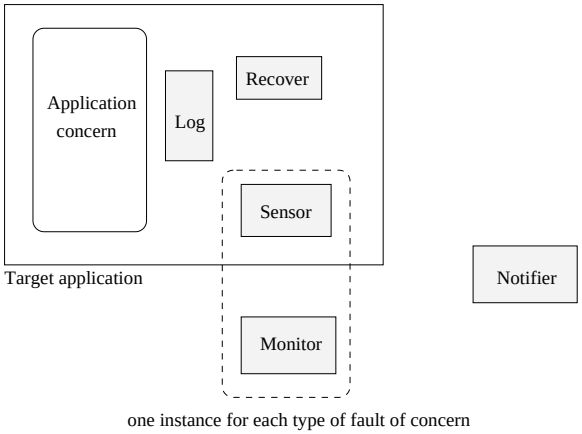
In the sequel, we describe briefly the elements of those architectures:

- **Sensor**: one instance of this element is added to the target application for each type of fault. It collects/generates (depending on type of fault) diagnosis information and sends this to **Monitor**.
- **Monitor**: One instance of this element has to be available for each type of fault. It receives diagnosis information from **Sensor**, analyzes it, detects the presence of faults, and signals the presence of faults to **Notifier**.

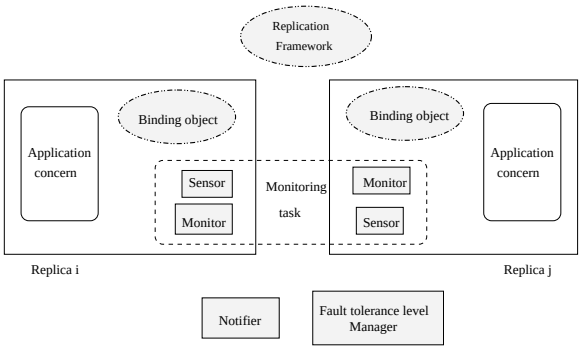
*Supported by the CONACyT 34230A project.

[1] *concern* and *aspect* are interchangeable in this context.

[2] http://www-lsr.imag.fr/storm.html



(a) Levels without use of replication



(b) Levels requiring replication

Fig. 1. Functional architecture of fault tolerance framework

- **Notifier**: receives subscriptions to the presence of a particular fault, receives the signalization of the presence of faults, discards multiple signalization of the same (instance of) fault, and notifies the subscribers the presence of fault that they have subscribed to.
- **Log**: The goal of **Log** is to maintain useful information so that the recovery can take place later. Depending on the required level, information kept by **Log** is determined by the target application. It may be snapshots of application state, performed operations, exchanged messages, etc. **Log** element is responsible for writing and retrieving information. It is not aware of the meaning of information it keeps.
- **Recover**: receives also the notification of the fault presence and starts up recovery process by using redundant element, i.e. **Log**, that has been installed in the target application.
- **Replication Framework** & **Binding objects**: **Replication Framework** provides support for replicas life cycle management, e.g. their creation and deletion, and for inter-replicas synchronization protocols [**?**]. One instance of the **Replication Framework** is created for each replicated entity. Indeed, the instantiation of **Replication Framework** is directed by a so-called replication policy. A **replication policy** is defined by four parameters: the replication time (when to create or to delete a replica inside the system), the replication degree (how many replicas have been or may be created?), the replicas placement (where to place a replica among a set of distributed nodes?), and the coherency model (what is the required coherency model among replicas?). Each instance of **Replication Framework** takes a replication policy as input parameter and implements it. In order to accomplish its goals, **Replication Framework** "installs" a **Binding object** at each replica. A detailed description of **Replication Framework** is out of the scope of our work but can be found in [**?**][**?**].
- **Fault tolerance level manager** (**FTL Manager**): keeps track of available resources, i.e. the number of available sites, the number of operational sites, and instructs the replication policy to **Replication Framework**.

In the sequel, the term *fault tolerance element* refers to any element in our framework.

III. Requirements for component model

From our point of view, a component model encompasses the following elements: the definition of component, the structure of a component (i.e. how a component is developed), and the application of component concepts in software development. In our context, we will discuss only the first two elements and in particular the second one.

There are several similar but not identical definitions of components. Two examples are:

" A component is a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger systems." [**?**]

and

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [**?**]

Although these definitions differ in detail, they both assert that a component is an independent software package that provides functionalities via well-defined interfaces [**?**]. Thus, components are runtime structures, i.e. they manifest during system execution.

In component context, each application is seen as a component. A **component** is formed out of two parts: a *controller* (or *container*[3]) and a *content*. The **content** depicts business rules or application specific logic.

The **controller** embodies behavior associated with a particular component. Each controller can be seen as the implementation of a particular semantic of its content. In particular, a component controller can:

- Intercept incoming and outgoing messages targeting or originating from the content it controls,
- Manage the life cycle of the component,
- Superpose a control behavior to the behavior of its content.

As mentioned in Section **??**, several elements of our framework will be integrated to the application component depending on the required fault tolerance level. Thus, the integration consists of adding appropriate fault tolerance elements to the target

[3]In the sequel, two terms are used interchangeably.

application or the target component. Those elements are essentially included in the container. Thus, the **first requirement** is the ability to increment the capabilities of the container.

In current standard component models like CCM (*CORBA Component Model*) or EJB (*Enterprise Java Bean*), the container is added automatically to the target application thanks to the so-called deployment descriptor. The capabilities of the container are determined via some predefined couples (parameter, value). Thus, new couples (parameter, value) for an existing capability or new capabilities of the container can be provided only by reimplementing the containers generator, e.g. EJB server in EJB model.

The fault tolerance can be seen as an incremental capability of the container. Thus, our fault tolerance elements can be instantiated by the containers generator. In order to implement our proposal, we need to re-implement the containers generator of the chosen component model. This approach is rather costly and less flexible.

So, the approach of extending containers in order to enable the fault tolerance capability of the target application is more appropriate to meet the requirement previously cited. In such a case, the underlying component model must allow to customize containers not only by specifying some predefined arguments but also by extending containers.

Added fault tolerance elements are implemented with some assumptions that other parts of the target application have to fulfill. More precisely, in some cases, **Sensor** elements have to be aware of the incoming and outgoing messages targeting or originating from the resulting component. However, **Sensor** elements are not responsible for intercepting these messages. Thus, for the **second requirement**, the target application or/and the component model has to provide some mechanisms allowing **Sensor** or other elements inside the resulting application (in a more general case) to access to the incoming/outgoing messages. We can say that this requirement is taken into account in most current component models.

For the adaptive fault tolerance, it is feasible only if the underlying component model allows the dynamical change of the component architecture. The dynamical change can be understood as the capabilities of stopping/removing/adding/restarting one or several parts of a component. Therefore, the **third requirement** states that the underlying model has to provide an explicit and causally connected representation of the application architecture. It must also provide operations to manipulate these representations.

Most current component models provide introspection mechanisms to enable the transparency between the container and its content. However, introspection mechanisms do not provide the capability of introspecting the architecture of the content.

In fact, the three above requirements are needed to provide adaptivity for any technical aspect. Fault tolerance is an example of such aspects.

In our proposal, we do not consider the "blinding logging"[4]. Instead, we adopt the following strategy: **Log** element provides common logging operations like writing information on a stable storage, reading information; but it is not aware of the meaning of these information. Figure **??** gives an example.

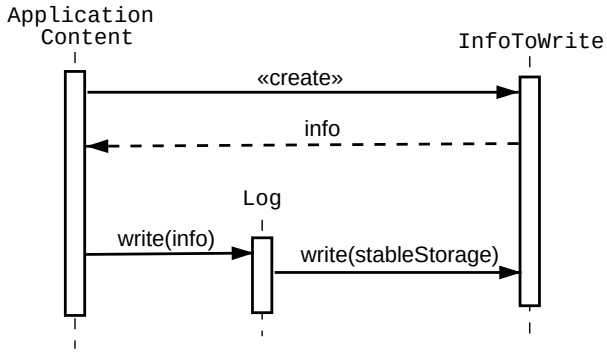


Fig. 2. Interactions while writing log

Whenever the target application wants to log the information about its state or/and its operations, it creates an instance of **InfoToWrite**. And then, the target application asks **Log** element to write this **InfoToWrite** instance on a stable storage. The stable storage is managed by **Log** element but the marshaling of data in the **InfoToWrite** instance is carried out by **InfoToWrite** instance itself. Note that **InfoToWrite** is implemented by the target application. Therefore, beside the three identified requirements, we add this **forth requirement**. It states that the added fault tolerance elements have to be aware of some knowledge related to the application logic. The existence of **InfoToWrite** in Figure **??** is an example of such knowledge.

IV. Ongoing and future work

As argued earlier, standard component models are not appropriate to achieve the adaptation proposed by our work. The requirements discussed in this paper are being investigated by the implementation of our proposal with Fractal component model [**?**]. We have chosen Fractal model as the underlying component model for our work because according to its specification, Fractal may help us to make our framework operational.

We took a component-based persistence service as the target application and specified a fault tolerance level. The implementation aims to evaluate the complexity[5] of the approach of extending containers in fault tolerance context. It also helps us to determine the componentization of our framework.

As mentioned, we have mostly worked on the customization aspect of the adaptable fault tolerance. We are planning to take the same component-based persistence service, specify another fault tolerance level and examine the complexity of changing the resulting application architecture from one level to another. This is going to be the first step towards the adaptivity.

References

[1] Szyperski C. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[2] Christine Collet. The NODS project: Networked Open Database Services. In *Proceedings of Symposium on Objects and Databases (ECOOP)*, LNCS1944, pages 153–169, June 2000.

[4] Blinding logging refers to mechanisms that log application information without any knowledge of the application logic.

[5] The complexity refers to the facility to implement a target application according to our requirements and to integrate our elements to obtain the resulting application.

[3] P.T. Cox and B. Song. A Formal Model for Component-Based Software. In *2001 IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering*, pages 304–311, Stresa, Italy, September 2001.

[4] D'Souza D.F and Wills A.C. *Objects, Components, and Frameworks with UML - the Catalysis approach*. Addison-Wesley, 1999.

[5] S. Drapeau, C. Roncancio, and P. Déchamboux. RS2.7: an Adaptable Replication Framework. In *18èmes Journes Bases de Données Avancées (BDA'02)*, October 2002.

[6] S. Drapeau, C. Roncancio, and P.Déchamboux. Overview of an Adaptable Replication Framework. In *Posters of the International Symposium on Distributed Objects and Applications (DOA'02)*. Extended abstract published as a technical report of the University of California at Irvine, October 2002.

[7] Phuong-Quynh Duong, Elizabeth Pérez Cortés, and Christine Collet. La tolérance aux fautes adaptable pour les systèmes à composants : application à un gestionnaire de données. In *18èmes Journes Bases de Données Avancées (BDA'02)*, October 2002.

[8] Technical Component Model Working Group. Fractal component model. http://www.objectweb.org/fractal/index.html.