Abstract

This paper presents the NODS fault tolerance framework. It encompasses the framework itself and a definition of fault tolerance level. Depending on the required fault tolerance level, different elements of the framework will be integrated to the target application. Two main kinds of architectures are identified and described. In the resulting architectures, fault tolerance of the framework itself is also provided.

Keywords: Fault Tolerance, Fault Tolerance Level, Customization, Functional Architecture
Functional Architecture of the NODS Fault Tolerance Framework

Phuong-Quynh Duong\textsuperscript{1}, Elizabeth Pérez Cortés\textsuperscript{2\*}, Christine Collet\textsuperscript{1}

\textsuperscript{1}LSR/IMAG Laboratory
BP 72, 38402 St Martin d’Hères, FRANCE
\{Phuong-Quynh.Duong, Christine.Collet\}@imag.fr

\textsuperscript{2}Depto. de Ingeniería Eléctrica UAMI
Av. San Rafael Atlixco 186 Col. Vicentina 09340, México D.F
pece@xanum.uam.mx

Abstract

This paper presents the \textit{NODS fault tolerance framework}. It encompasses the framework itself and a definition of fault tolerance level. Depending on the required fault tolerance level, different elements of the framework will be integrated to the target application. Two main kinds of architectures are identified and described. In the resulting architectures, fault tolerance of the framework itself is also provided.

\textbf{Keywords:} Fault Tolerance, Fault Tolerance Level, Customization, Functional Architecture

1 Introduction

The increasing use of computers in almost every aspect of modern life has led to the need of highly fault tolerant computer systems. The more they are used, the more potential negative impact they have if they fail. Since the middle of the last century, a lot of work has been done to endow computer systems of a higher fault tolerance level [1].

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\textsuperscript{1}. Thus, it helps application developers to easily manage the non applicative tasks. Also, frequently the proposed solutions allow some degree of customization to provide a best suited functionality with respect to the application requirements. Many efforts on isolation of technical aspects (e.g. concurrency control, persistency, etc.) from applicative concerns have been done so far. The goal of our work is to provide a framework that handles fault tolerance concern. The term \textbf{framework} refers to a set of interfaces and the description of the interactions among these interfaces that can be partly implemented. Indeed, existing fault tolerance frameworks focus only on some particular predefined fault tolerance requirements. Our work aims to propose a more generic, flexible and customizable framework.

Traditionally, a fault tolerant system is built by including redundant elements that in a normal case are superfluous but that are necessary in the case of faults. Then, at the runtime, system is monitored to detect the presence of a fault and in such a case, recovery is done by

\textsuperscript{1}Supported by the CONACyT 34230A project.
\textsuperscript{1} \textit{Concern} and \textit{aspect} are interchangeable in this context.
using redundant elements previously included. The redundant elements are chosen according to considered faults and desired behavior. In our work, these elements make up the framework handling fault tolerance concern. 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.

Our work is a single research thread of the NODS\textsuperscript{2} (Networked Open Database Services) project \cite{2}. The paper presents our proposal on functional architecture of the NODS fault tolerance framework. It mainly identifies and analyzes different instances of the framework, the elements composing these instances, the functionalities of these elements as well as their interactions. It also discusses the fault tolerance for the framework itself. The way the functionalities are mapped to interfaces is out of the scope of this paper. We assume that the target applications do not have any capability of fault tolerance with respect to operational faults\textsuperscript{3}.

The rest of this paper is organized as follows. In Section 2, the definition of fault tolerance level is presented. Section 3 describes the functional architecture of the framework. Related work is discussed in Section 4. Finally, Section 5 gives some concluding remarks and our ongoing work.

2 Fault tolerance expression

According to the consequences of a fault that an user\textsuperscript{4} can perceive, it is possible to define four different forms of fault tolerance as follows:

- \textit{Masking}: The properties of a system are preserved in the presence of faults,

- \textit{Unmasking}: Some properties of the system are violated when the fault occurs, but when the fault stop occurring, the system resumes and its properties hold again,

- \textit{Signaling}: Some properties of the system are violated when the fault occurs and the system is capable of recognizing this violation,

- \textit{Nothing}: Nothing is done in the presence of faults.

In the first two cases, we are talking about real fault tolerance capability. In the third case, the system does not do anything about the fault but raise a signal. And in the last case, no fault tolerance capability is present. For the sake of uniformity, all these forms will be called fault tolerance forms.

Each fault tolerance form matches a level of consequence of fault users perceive. Each form has also impacts on the cost of implementing mechanisms that ensures it. So, it is possible to establish a \textit{stronger relation} among the four above forms. \textit{Masking} is stronger than \textit{unmasking}, which, in turn, is stronger than \textit{signaling}. \textit{Nothing} is the weakest form of fault tolerance. The stronger the fault tolerance form is, the more expensive it is to be built a system that ensures it, and the less bad consequences of fault users perceive.

Considering hierarchical nature of fault classes \cite{7}, we define the \textit{parent relation} between two fault classes as follows:

\textbf{Definition 1} For two fault classes $F_i$ and $F_j$, $F_i$ is the parent of $F_j$ iff the set $F_i$ is a superset of the set $F_j$. $F_i \supseteq F_j$ for short.

In our work, we consider a subset of the classes of faults defined in \cite{7}. More precisely, we consider the following classes of operational faults: crash, omission fault, late timing fault, value fault and Byzantine fault. The parent relation for these classes of faults is shown in Figure 1.

\footnote{http://www-lsri.imag.fr/storm.html}

\footnote{Operational faults are ones which appear during the system life and are caused due to physical reason\cite{7}.

\footnote{Users may be another system or a human being.}
Considering the four different forms of fault tolerance previously defined and the parent relation among fault classes, we define the level of fault tolerance $FT$ for a system $S$ as follows:

**Definition 2** $FT(S) = \{< f, l > | f$ is a fault class and $l \in \{nothing, signaling, unmasking, masking\} \}$

where $l_j$ is stronger than or equal to $l_i$ if $f_i \sqsubseteq f_j$.

The constraint $l_j$ is stronger than or equal to $l_i$ if $f_i \sqsubseteq f_j$ is explained as follows. All faults belonging to $f_i$ and $f_j$ are perceived at most with consequences associated to $l_i$ and $l_j$ respectively. Since all faults belonging to $f_j$ belong also to $f_i$, all faults belonging to $f_j$ are perceived at most with consequences associated to $l_i$. So, the consequences associated to $l_j$ must be less bad than or equal to the consequences associated to $l_i$. In other words, $l_j$ is stronger than or equal to $l_i$. To illustrate better this constraint, Figure 2 shows a valid fault tolerance level.

$$FT(S) = \{< Byzantine, nothing >, < Value, nothing >, < Late timing, unmasking >, < Omission, masking >, < Crash, masking >\}$$

Figure 2: Example of fault tolerance level

Since late timing fault class is the parent of omission fault class and late timing faults need an unmasking form, omission faults will not have a fault tolerance form weaker than unmasking form. Thus, omission faults may have an unmasking or a masking form. In the example, we choose masking form that leads to the masking form for crashes.

In the sequel, the term level stands for fault tolerance level.

### 3 NODS fault tolerance framework architecture

The fault tolerance level definition given in Section 2 allows application developers to specify the application requirements on fault tolerance. Such a level gives information enabling the instantiation of our NODS fault tolerance framework. In fact, the constraint on valid forms for fault classes in definition 2 limits the number of valid levels over classes in Figure 1 to a finite set. This allows us to establish an exhaustive mapping from a required level to redundant, monitoring, diagnosis and recovery elements to be integrated. During the execution, redundant elements collect useful information that will be used by recovery elements in the case of a fault. Diagnosis elements collect information useful for detecting the presence of a particular fault. Monitoring elements analyze collected information and determine if a fault has probably occurred.
Indeed, these elements make up an instance of the NODS fault tolerance framework for the required level. The nothing form of fault tolerance does not require any element among four kinds of elements described above. The signaling form requires only monitoring and diagnosis elements. The unmasking and masking forms require all four kinds of elements: redundant, monitoring, diagnosis and recovering. Depending on the kind of redundant and recovering elements, the unmasking or masking form will be ensured.

In the sequel, we will investigate the functional architecture of our framework in two main cases. The first case discussed in Section 3.1 copes with fault tolerance levels which do not require replication. The second case dealing with fault tolerance levels that require replication is presented in Section 3.2. Section 3.3 analyzes the fault tolerance capability of the elements in our framework. And the final architectures are presented in Section 3.4.

3.1 Architecture for levels without replication

All the levels discussed in this section consider three types of faults: crash, omission fault and late timing fault. Thus, when we mention a type of fault, we mean one of them.

Figure 3(a) shows the integrated architecture for fault tolerance levels which require only signaling form. For such levels, monitoring and diagnosis elements are enough. The elements of the NODS fault tolerance framework in this case are Sensor, Monitor and Notifier. Their functional description is as follows:

- **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 diagnosis information 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.

- **Notifier**: A Notifier element 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.
The interaction between an instance of Sensor and an instance of corresponding Monitor can be performed in pull or push mode.

When at least one type of fault need to be in an unmasking form, Log and Recover elements are added to the integrated architecture as shown in Figure 3(b).

- **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, or exchanged messages. Log element is responsible for writing and retrieving information. It is not aware of the meaning of information it keeps.

- **Recover**: A Recover element 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.

Note that the couple (Sensor, Monitor) in Figure 3(b) is instantiated for each fault class with signaling or unmasking form in the required level. When an unmasking form has been required, the subscribers (of Notifier) to the presence of a particular type of fault are target applications.

Figure 4 depicts the most important interaction among added fault tolerance elements that have been discussed.

![Figure 4: An interaction among elements in framework without use of replication](image)

### 3.2 Architecture for levels with replication

As in Section 3.1, we have to distinguish two sub-cases using replication. The first sub-case copes with levels which do not require masking form for value and Byzantine faults, and the second deals with which require masking form for value and Byzantine faults.

**Without masking form for value and Byzantine faults** Most of the levels requiring replication can be ensured with the architecture shown in Figure 5. The exception are those including masking form for value and Byzantine faults. Note that this architecture handles three types of faults: crashes, omission faults and late timing faults. Thus, as in Section 3.1, when we mention the type of faults, we mean one of them. In this case included elements are:

- **Sensor and Monitor**: The tasks for Sensor and Monitor are the same as in the first case with only a slight difference: Sensor may send diagnosis information to several Monitors in place of one as in the first case. As a result, Monitor may receive diagnosis information from several Sensors, and at least one instance of Monitor has to be available for each type of fault.
• **Notifier:** This element has the same tasks described in Section 3.1.

• **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 [3]. 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 [3][4]. Thus, although we show Binding object in Figure 5, we will consider Replication Framework and Binding objects as a whole element.

• **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.

The interaction among the described elements is illustrated in Figure 6.

Replication Framework is a subscriber to the presence of all the types of faults in the required level. Upon the reception of a fault notification, Replication Framework removes the faulty replica or does the state synchronization.

Note that in order to ensure some required level, our framework has to delegate some tasks to the Replication Framework. This clear separation of tasks between fault tolerance and replication makes our framework design more modular.

**With masking form for value and Byzantine faults**  
Masking form is the only valid form for value and Byzantine faults. This form can be handled by Replication Framework using its
Consensus Manager: In this case, no additional treatment is needed on behalf of the NODS fault tolerance framework with regard to the previous sub-case. Thus, the differences between this sub-case and the previous one are the number of replicas to be created and the specific instance of the Replication Framework.

3.3 Fault tolerance of elements in the NODS fault tolerance framework

The added elements are divided in two categories: replicas of application components and fault tolerance elements (FT elements). The FT elements are Notifier, Monitor and FTL Manager. In our architecture, Monitor for the levels considered in Section 3.2 and Sensor are strongly coupled to the target application. Their fault tolerance is considered in the replicas fault tolerance management.

In order to provide fault tolerance for replicas, FTL Manager has to ensure the existence of the required number of replica instances. Upon the reception of a crash notification, it finds an available site and asks Replication Framework to install a new replica there. So, FTL Manager is a subscriber to crash notifications.

Since FT elements are critical, masking form of fault tolerance for them is necessary. Thus, masking all classes of faults, that is the strongest level, is the first level we want to provide for these elements. As this level is rather expensive and FT elements that are under the control of our framework will suffer most likely crashes, masking crashes is so an alternative level for FT element(s). Therefore, we propose two levels of fault tolerance for the FT element(s):

- masking crashes, i.e. masking form for crashes and nothing form for other classes of faults, and
- masking all classes of faults.

If the masking crashes level is chosen, FT element is replicated by using an instance of Replication Framework. To distinguish different instances of Replication Framework, we call this instance FT Replication Framework. The principle of masking such an element is the following: An instance of Crash Sensor and an instance of Crash Monitor are added to each replica of FT element. The presence of a replica crash is notified to FT Replication Framework so that the latter can manage the replica list.

If the masking all classes of faults level is chosen, each FT element is replicated by using an instance of the Replication Framework with a Consensus Manager inside as in case of masking Byzantine faults. We call this instance FT Replication Framework too. Besides, each replica of FT element will have a couple of (Sensor, Monitor) for each fault class of the following classes:
crash, omission and late timing fault. The presence of a fault in a replica is notified to FT Replication Framework so that the latter can manage the replica list.

In both above levels, FT Replication Framework signals the need of installing new replicas of FT elements to Notifier when only one replica is operational. An operator may be a subscriber to such an event.

3.4 Final framework architectures

This section gives an integrated view on framework architectures whose different aspects have been discussed in 3.1, 3.2, and 3.3. Therefore, Figure 7 shows the final architecture for all levels that do not require replication. Figure 8 shows the final architecture for all levels that do require replication.

(a) Masking crashes for added elements           (b) Masking all types of faults for added elements

Figure 7: Final architecture without use of Replication to ensure the required fault tolerance level

4 Related work

Many efforts have been done on isolating fault tolerance activities from the application specific computation into a framework. Depending on application requirements, an instance of such a framework is instantiated and integrated to the application.

AQUA [10] and Chameleon [8] propose complete framework like ours. By complete framework we mean a framework that copes with failure detection, recovery and replication aspects of fault tolerance. AQUA is specific for CORBA applications while while Chameeleon is mobile agent oriented. However, they handle less faults and less forms of fault tolerance than our proposal. AQUA ensures masking form for value faults and crashes like the case discussed in Section 3.2. Chameeleon proposes some mechanism to translate an user requirement to fault tolerance execution strategy that is very similar to our idea of translating a fault tolerance level to redundant, recovery, monitoring and diagnosis elements. However, Chameeleon does not propose a precise way to express user requirement like our work, and thus, the translation is not addressed in their proposal.

8
Figure 8: Final architecture with use of Replication to ensure the required fault tolerance level

Fault Tolerant CORBA specification [9] guides interactions among elements needed to ensure fault tolerance for CORBA applications. This guideline is somehow more detailed than interactions presented in Figures 4 and 6. The interactions are described in terms of interfaces. Our final work must reach this goal.

FRIENDS [5] proposes only mechanisms of integrating fault tolerance into target applications with a minimum impact on application codes. It does not cover fault detection aspect nor adaptability regarding user requirements. It achieves the goal by using reflection. Fault tolerance mechanisms are implemented in meta-object level. Thus, the approach taken by FRIENDS may be a hint for implementing our framework.

5 Conclusion

In this paper, we have presented the NODS fault tolerance framework. We have briefly presented our definition of fault tolerance level that allows to instantiate the framework correctly according to the target application requirement. The paper focuses on the functional architecture of the framework. We have identified and analyzed different instances of the framework, the elements composing these instances, the functionalities of these elements as well as their interactions. We have also discussed the fault tolerance capability of the framework itself.

Our work aims to provide a more complete solution to the problem of providing customized fault tolerance than existing works. This solution ranges from a precise definition of fault tolerance requirement, i.e. fault tolerance level in our term, to a framework describing interactions among elements needed for ensuring fault tolerance. The fault tolerance level definition allows an application developer to specify the application requirements on fault tolerance. Based on information in the required level, we are able to identify the instance of the framework to be used so that this level is guaranteed. The proposed framework functional architectures are complete enough to fill all the levels that can be defined by application developers.

From our point of view, the framework will work efficiently in component context, in particular gray component context. Gray components do not hide all information of their internal structure, enabling efficient interaction with our framework, especially with activities of Log and Recover elements. The implementation of our framework in a concrete component model [12] is
being investigated.

Several design patterns have been proposed to describe the interactions between different participants implementing a fault tolerance aspect like failure detection [6], recovery [11]. These patterns may give hints for the implementation of our FT elements. Thus, modeling the interactions among elements of the NODS fault tolerance framework in terms of design patterns is also one of our ongoing threads of work.

References


