Architecting Resilient Computing Systems: a Component-Based Approach

Miruna Stoicescu

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse

Discipline ou spécialité :

Systèmes Informatiques

Présentée et soutenue par :

Miruna STOICESCU

Le 9 décembre 2013

Titre :

Architecting Resilient Computing Systems: a Component-Based Approach
Conception et implémentation de systèmes résilients par une approche à composants

---

JURY

Rapporteurs :	Alexander Romanovsky	Université de Newcastle
	Lionel Seinturier	Université de Lille
Examinateurs :	Charles Consel	Université de Bordeaux
	Animesh Pathak	INRIA Paris-Rocquencourt
	Alain Rossignol	ASTRIUM SAS Satellites
Directeurs de Thèse :	Jean-Charles Fabre	INP de Toulouse
	Matthieu Roy	LAAS-CNRS

École doctorale :

EDSYS

Unité de recherche :

LAAS-CNRS

Directeurs de Thèse :

Jean-Charles Fabre, Matthieu Roy

To my parents and my friends,
who believe more in me than I believe in myself.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AC	**A**utonomic **C**omputing
ACID	**A**tomicity, **C**onsistency, **I**solation, **D**urability
ADL	**A**rchitecture **D**escription **L**anguage
AFT	**A**daptive **F**ault **T**olerance
AOP	**A**spect-**O**riented **P**rogramming
API	**A**pplication **P**rograming **I**nterface
ATaG	**A**bstract **Ta**sk **G**raph
CBSE	**C**omponent-**B**ased **S**oftware **E**ngineering
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
DSL	**D**omain **S**pecific **L**anguage
FT	**F**ault **T**olerance
FTM	**F**ault **T**olerance **M**echanism
LAN	**L**ocal **A**rea **N**etwork
LFR	**L**eader-**F**ollower **R**eplication
NVP	**N**-**V**ersion **P**rogramming
PBR	**P**rimary-**B**ackup **R**eplication
RB	**R**ecovery **B**locks
SCA	**S**ervice **C**omponent **A**rchitecture
SCC	**S**ense-**C**ompute-**C**ontrol
SCDL	**S**ervice **C**omponent **D**efinition **L**anguage
SOA	**S**ervice-**O**riented **A**rchitecture
TR	**T**ime **R**edundancy
UML	**U**nified **M**odeling **L**anguage
WAN	**W**ide **A**rea **N**etwork
WSN	**W**ireless **S**ensor **N**etwork
XML	e**X**tensible **M**arkup **L**anguage

Introduction

> "There is truth, my boy. But the doctrine you desire, absolute, perfect dogma that alone provides wisdom, does not exist. Nor should you long for a perfect doctrine, my friend. Rather, you should long for the perfection of yourself. The deity is within you, not in ideas and books. Truth is lived, not taught."
>
> — Hermann Hesse, *The Glass Bead Game*

Evolution in response to change during service life is a key challenge for a plethora of systems. Changes originate from the environment (ranging from interactions with the user to physical perturbations due to natural phenomena) or from inside the system itself. Internal changes may occur in different layers of the computing system, ranging from the hardware support and the network infrastructure to the software.

Whatever the origin, the system either directly copes with change or it does not. In the first case, it may either continue to deliver the original service in its entirety or a subset of functionalities (called a degraded mode of operation), according to the change impact. This case corresponds to a foreseeable change for which provisions were made by the system designers and/or developers (in the form of a predefined/preprogrammed reaction). However, one cannot predict everything and, even if it were possible, developing a system equipped for the worst case scenario on all imaginable dimensions is often cost-prohibitive. In the second case, when there is no preprogrammed reaction, the system must be modified to tackle the unpredicted change. There are two possibilities: either the system cannot be modified in a satisfactory manner and it is unusable in the current situation or it can be adapted to accommodate changes. As a result, the capacity to easily evolve in order to successfully accommodate change is a requirement of utmost importance. A disciplined design and a flexible architecture are well-established solutions to this problem, although often easier said than done, especially with the increasing complexity of systems.

The situation is more challenging in the case of dependable systems, which have to continuously deliver trustworthy services. Dependable systems must evolve in order to accommodate change but this evolution must not impair dependability. Resilient computing, defined by the addition of this evolutionary dimension to dependability, represents the core of the present work. On the one hand, resilience encompasses several aspects, among which evolvability, i.e., the capacity of a system to evolve during its execution. On the other hand, dependability relies on fault-tolerant computing at runtime, which is enabled by the presence of fault tolerance

mechanisms attached to the system. As a result, enabling the evolvability (also called adaptivity in this work) of fault tolerance mechanisms appears to be a key challenge towards achieving resilient computing.

Fault-tolerant applications generally consist of two abstraction layers. The first one, usually represented at the base, is the functional layer and it implements the business logic. The second one contains the non-functional features in the form of fault tolerance mechanisms, in this case. Other examples of non-functional features include cryptography and QoS policies. This separation of concerns facilitates the distribution of roles and tasks in the system development process. On the one hand, the design and implementation of the functional layer are executed by teams with a well-defined hierarchy and according to a thoroughly documented system development lifecycle. On the other hand, the non-functional layer usually requires so-called expert knowledge and can be delegated to a third-party.

Evolvability has long been a prerogative of the functional layer. A solid body of knowledge exists in this field, spawning from encountered problems and best practices. Solutions come in the form of programming paradigms, architectural patterns, runtime support, methodologies for the software development process etc.. In comparison, little effort has been done for the adaptivity of fault tolerance mechanisms, especially for benefiting from the aforementioned solutions. Nevertheless, their adaptivity is crucial and sometimes proves to be more important than the adaptivity of the functional layer. It is the case for long-lived autonomous systems, among others. Satellites and deep-space probes belong to this class of systems. These applications are mission-critical, resource-constrained, mostly non-repairable and their service life consists in a series of operational phases characterized by varying requirements. A key aspect to take into account is the dynamics of the environment (e.g., physical interferences and perturbations) that often leads to new threats. Embedding exhaustive fault tolerance strategies is unfeasible due to resource limitations and to the practical impossibility of foreseeing all events occurring during the service life of the system (that can span over decades). Evolution is generally tackled through ad-hoc modifications. In this context, we propose a methodology and accompanying tools for the systematic adaptation of fault tolerance mechanisms in operation, from design to the actual fine-grained modifications of mechanisms at runtime.

Autonomic computing has gained tremendous attention since its introduction and many systems adhere to its principles, among which future NASA swarm-based spacecraft systems. In order for a system to be autonomic, it must be self-configuring, self-healing, self-optimizing and self-protecting. Self-healing is defined as the automatic discovery and correction of faults. This definition is extremely similar to that of fault tolerance. However, changes can also be perceived as faults because they represent events that deviate from the expected behavior. A fault tolerance mechanism is effective if the change/fault falls in its effectiveness domain. Otherwise, the fault tolerance mechanism must be adapted to accommodate the specific change, so the system must be resilient. Given that autonomic computing also targets systems with ever-increasing complexity operating in dynamic environments, it is our belief that there is a significant overlap between resilience and self-healing. In this context, the contributions presented in this thesis are part of the broader spectrum of autonomic computing.

In the present work, multiple facets of the adaptivity of fault tolerance mechanisms are discussed. First of all, in Chapter 1, we elaborate on the relations between adaptivity, depend-

ability and resilience. Then, we present several categories of adaptation and identify the most suitable one for our case, namely on-line agile fine-grained adaptation. The overall approach is discussed and related research efforts are presented.

In Chapter 2, we identify a change model and establish a frame of reference in which the dynamics faced by fault-tolerant systems can be illustrated. This frame of reference unifies the parameters that determine which fault tolerance mechanism to attach to an application, among several options. Next, a classification of fault tolerance mechanisms is presented and a subset of strategies is described. Several transition scenarios, based on the variation of the aforementioned parameters, are discussed.

Next, in Chapter 3, the selected mechanisms are thoroughly analyzed, in order to reveal their similarities and their specificities. The iterative analysis results in a generic execution scheme for the "design for adaptation" of fault tolerance mechanisms. More precisely, the execution of protocols is broken down in three steps corresponding to their variable features.

In Chapter 4, we map the above mentioned generic execution scheme on a reflective component-based middleware. The choice of this support is discussed. Next, we leverage the control features provided by the platform for executing on-line fine-grained transitions between fault tolerance mechanisms. Several transitions, stemming from the adaptation scenario presented in Chapter 2, are illustrated.

Chapter 5 is structured in two main parts. The first one focuses on the evaluation of the proposed approach in terms of transition execution time and agility. The second one focuses on the usability of the proposed fine-grained adaptive fault tolerance mechanisms. To this aim, we describe their integration, first, in an overall development process centered on the traceability of requirements and, second, in a toolkit for wireless sensor network-based applications.

Chapter 1

Context & Problem Statement

> "The only people who see the whole picture are the ones who step out of the frame."
>
> — Salman Rushdie, *The Ground Beneath Her Feet*

1.1 Introduction

In this chapter, we present the context of our research endeavor and define several key concepts from the field of dependable computing. The motivation of our research endeavor is presented through an illustrative example. We state the challenge of this thesis, namely, enabling the evolution of dependable systems in operation.

Next, we describe the overall approach for achieving this research goal. Last but not least, we present related work from the various research areas connected to the topic of this thesis and put our contributions into perspective.

1.2 From Dependability to Resilience

1.2.1 Resilience

Resilience (from the latin word resilio, to rebound, to spring back) is a concept appearing in many different fields, such as psychology, materials science, business, ecology and, last but not least, computing, which is our area of interest. These fields share a similar understanding of this notion based on two key elements: disturbance and the ability to recover from it. Disturbances take different forms, according to the field under consideration: stress, a physical shock/impact, a natural disaster etc. Recovery from disturbance can be immediate (e.g., an object recovering after being struck) or span over a longer time interval (e.g., a business recovering from a terrorist attack, an ecosystem recovering from human activities such as deforestation or overfishing).

In the field of dependable computing systems, resilience was defined in [Laprie, 08] as "the persistence of dependability when facing changes" or, when incorporating the definition of dependability [Avižienis *et al.*, 04], "the persistence of service delivery that can justifiably be trusted, when facing changes". As a result, in order for a system to be resilient, first, it must be dependable and, second, it must conserve this attribute despite changes. In the remainder of this section, we explain dependability-related concepts and discuss their relationship with resilience.

Changes are classified in [Laprie, 08] according to three dimensions: nature, prospect, and timing (see Figure 1.1). It is worth noting that, in the context of dependable systems, changes can concern or lead to modifications in the threats the system is facing. With respect to the prospect criterion, it is obvious that unforeseen changes (e.g., an explosion in the number of client requests) are more difficult to tackle than foreseen (e.g., software versioning) and foreseeable ones (e.g., the advent of new hardware). This change classification, extracted from [Laprie, 08], provides a higher-level perspective than the one we propose in the next chapter, especially regarding the nature criterion. However, it provides a good view of the different aspects of change that must be taken into account and illustrates key concepts recurrent in our work such as prospect and timing that we further detail.

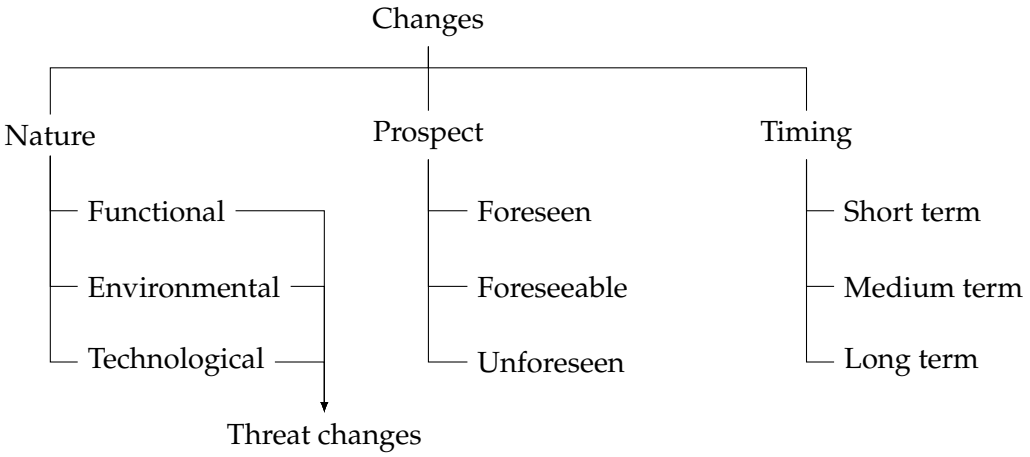


Figure 1.1: Change Classification

1.2.2 Dependability

As previously mentioned, a resilient system must be dependable in the first place. Dependability is a well-documented concept, older than resilient computing, and with a thoroughly defined taxonomy. A cornerstone of any *-critical system (*=safety, mission, business), dependability was defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as the "trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers". Dependability is defined by its attributes, the threats to it and the means by which to achieve it [Avižienis *et al.*, 04]. The attributes it encompasses are the following:

- availability, i.e., readiness for correct service;
- reliability, i.e., continuity of correct service;
- safety, i.e., absence of catastrophic consequences on the user(s) and the environment;
- integrity, i.e., absence of improper system alterations;
- maintainability, i.e., ability to undergo modifications and repairs.

The threats to dependability are failures, errors and faults. A *failure* is an event that occurs when the service delivered by the system deviates from the correct service (i.e., the one implementing the system function expected by the client). A service can fail either because it does not comply with the specification or because the specification does not accurately describe the system function. A service failure is a transition from correct service to incorrect service. Given that a service consists in a sequence of the system's external states (observed by the client), the occurrence of a failure means that at least one of the external states deviates from the correct service state. The deviation is called an *error*, which represents the part of the total state of the system that may lead to a failure, in case it reaches the interface of the system service. The determined or presumed cause of an error is called a *fault*.

As systems generally consist of a set of interconnected subsystems, they are most often subject to a recursive chain of threats as illustrated in Figure 1.2. A failure occurring at the service interface of a subsystem is perceived as a fault by the subsystem calling the service. This fault, when activated, produces an error, which in turn can lead to a failure at the service interface of the second subsystem. Dependability aims at breaking this causal chain and stopping a fault from propagating from one subsystem to another one and from producing a failure perceived by the user.

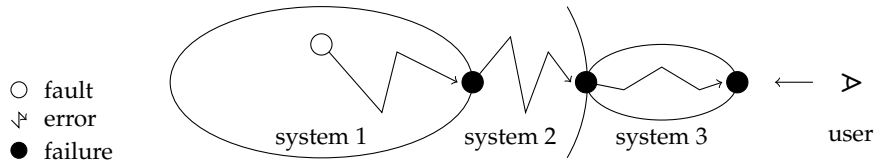


Figure 1.2: Recursive Chain of Dependability Threats

The means for achieving dependability are grouped in four categories [Avižienis *et al.*, 04]:

- fault prevention aims to prevent the occurrence or introduction of faults;
- fault tolerance aims to avoid service failure in the presence of faults;
- fault removal aims to reduce the number and severity of faults;
- fault forecasting aims to provide an estimation of the present number of faults, future incidence and possible consequences of faults.

1.2.3 Bridging the Gap

Resilience encompasses four major tracks of investigation[1]: evolvability, assessability, usability and diversity. An important feature within evolvability is adaptivity, which is the capacity to evolve during execution. These so-called technologies for resilience [Laprie, 08] are obviously related to the means for achieving dependability. Among these, fault tolerance is a non-functional activity accompanying the system throughout its operational life while fault prevention, removal and forecasting are generally performed off-line, before deployment. As adaptivity (and more generally, evolvability) deals with changes during operational life, it is only natural to address resilient computing by adding an evolutionary dimension to fault tolerance (and, more specifically, to fault tolerance mechanisms). Adaptive Fault Tolerance (AFT) was defined in [Kim and Lawrence, 90] as: *"an approach to meeting the dynamically and widely changing fault tolerance requirement by efficiently and adaptively utilizing a limited and dynamically changing amount of available redundant processing resources"*. Considered at the time a branch of fault tolerance still in its incipient form, adaptive fault tolerance has meanwhile gained interest and, in our view, represents a key aspect of resilient computing.

The contributions brought by the present work to the field of resilient computing consist in a methodology and a set of tools for enabling agile on-line fine-grained adaptation of fault tolerance mechanisms (FTMs). These attributes and their role are further explained. The evolution of FTMs may be required due to environmental changes (e.g., electromagnetic interferences on aircraft or spacecraft), application changes due to versioning (e.g., a deterministic version is updated to a new non-deterministic one), runtime support changes (e.g., loss of hardware devices) or fault type changes (e.g., transient hardware value faults to be tolerated due to hardware aging).

1.3 A Motivating Example

Satellites are autonomous objects providing complex functions with limited resources and energy. Many of the functions provided are critical, at least from an economic viewpoint. As described in [James *et al.*, 10], despite thorough verification processes, software faults may remain in operation and both internal and external faults (radiation effects, temperature, vibration or operator faults) may impair space missions. Fault-tolerant computing has been used since the 60s in spacecraft and is essential for systems such as deep-space autonomous probes to handle errors with limited ground connexion.

[1] Definitions and overview can be found in the ReSIST European Network of Excellence web pages: `http://www.resist-noe.org/`

A satellite may have an expected or contractual lifetime of up to several years (most often achieved or even largely exceeded in practice). An important characteristic with regards to its maintenance and evolution is that on-site reparation after launch is extremely difficult and costly, if problems occur. In an autonomous device, not all possible problems can be predicted (loss of resources, fault impact, hardware aging, etc.) and fault tolerance solutions cannot be anticipated for unknown problems. The evolution of the functional software, of the configuration and health state of resources and of the operational procedures is also something that must be considered during the lifetime of the satellite, that may change some assumptions for the selection of the fault tolerance strategy. In this context, when the need presents itself, the adaptation of fault tolerance mechanisms (and of the functional layer as well), is executed in an ad-hoc manner. Depending on the case, adaptation takes various forms, e.g., changes in parameter values, low-level updates of memory regions, i.e., software in-flight (re)programming, or even full replacements of the embedded procedures.

The approach we propose has many merits that can be illustrated with the above example. Defining and validating a priori all possible dependability mechanisms and their combinations according to failures and threats in operation is impossible (undecidable a priori, an NP-complete problem). Moreover, loading too many combinations in a spacecraft would be unacceptable due to memory space limitations. Our approach aims at designing FTMs as a Lego system, leveraging concepts and tools from the field of software engineering such as component-based architectures and runtime support. Applications/functions are initially equipped with some FTMs according to the initial non-functional requirements. The monitoring done within the satellite informs ground engineers about the current state of the satellite. The proposed approach enables engineers to design a fault tolerance solution off-line at any time during the satellite's operational life, to upload the needed Lego pieces to update the FTM, to install and attach a new FTM to the target function. In summary, according to monitoring done on-board (characteristics of the functions, available resources, priority among functions, etc.), we propose a differential update of dependability mechanisms, alleviating the burden of managing the whole embedded software as a single block with limited resources and connexion. Compared to ad-hoc modifications of FTMs, our approach enables their methodical adaptation. By leveraging existing software engineering tools, the proposed approach provides means to systematically develop the necessary bricks and to integrate them in the embedded software architecture.

The benefits of this work go far beyond long-lived space missions, as resilience is a challenge as well for small autonomous devices that need to survive without human intervention. This is true for autonomous smart submarine earthquake-detection probes or fire-detection probes in forests. This work is thus a contribution to self-healing techniques required for autonomic computing [Kephart and Chess, 03], targeting many autonomous non-repairable devices that will be part of our daily life in the near future.

1.4 Problem Statement

In general, dependable systems are complex systems having a long service life which often consists in a sequence of operational phases. Operational phases are established at design time.

For a rocket placing a satellite into orbit, these phases are: launch, flight according to flight plan, and release of the satellite. In highly dynamic environments, phases must be triggered by the system manager, who observes changes in the environment and in the system, during the system's service life, e.g., for a military aircraft, these phases can be flight, observation, attack, defense, retreat, with phases occurring more than once and not always in the same order.

Each phase has a set of functional and non-functional requirements, e.g., reliability, availability. In this work, we focus on fault tolerance requirements, particularly the type(s) of faults to tolerate. While application characteristics are expected to have a fixed value during an operational phase (i.e., the version of an application module will not be changed during its execution), resources are subject to change due to the dynamics of the environment.

1.4.1 From Fault Model to Change Model

In this context, the evolution of a dependable system is analyzed through the following aspects:

- the required dependability properties and, in particular, its fault tolerance capabilities (*FT*);
- application characteristics at a given point in time (*A*);
- available resources (*R*).

Dynamics (change) occurring along these dimensions may have an impact on dependability, more specifically, on the fault tolerance mechanism(s) attached to the application. This represents the change model we consider. The degree of impact ranges from a drop in the quality of the provided service to a total impairment of the mechanism. An FTM change is required when applications evolve (different characteristics *A*) or when the operational conditions change either because available resources *R* vary or because the fault tolerance requirements *FT* change (similar to the threat changes in Figure 1.1) during the service life of the system.

In this work, *FT* encompasses one or several FTMs that are obviously strongly related to the fault model and the required dependability properties (reliability, availability, integrity, confidentiality). The characteristics of the application *A* have an impact on the validity of the selected FTMs. To run the selected mechanisms, a set of resources *R* is required and should be available. The choice of an appropriate FTM for a given application is based on the values of the parameters *(FT,A,R)* and, at any point in the lifetime of the system, this triplet must be consistent with the current FTM in order to fulfill dependability properties. Any change(s) in the values of these parameters may invalidate the initial choice and require a new FTM. Transitions between FTMs are needed when inconsistency is found between an FTM and operational conditions.

1.4.2 Types of Adaptation

Providing fault-tolerant systems with flexibility and adaptivity [Gouda and Herman, 91] is a crucial aspect. Evolution has traditionally been tackled off-line, with systems being stopped, updated and restarted. However, this is not satisfactory for dependable systems whose service cannot be stopped for a long period of time, e.g., air traffic control systems, and which must

comply with stringent requirements in terms of availability, responsiveness etc. In this context, on-line adaptation of FTMs (i.e., while the system is in operation) appears to be the only acceptable solution for dependable systems to accommodate change.

Further on, the adaptation of FTMs at runtime can be performed in two substantially different manners, according to its degree of flexibility:

1. *preprogrammed adaptation*: several FTMs coexist from the beginning and according to changes in the environment, user policies etc., one FTM is selected over the others. Certainly, this provides an amount of flexibility to the fault-tolerant system compared to static fault tolerance but the degrees of freedom are limited to the initial set of FTMs which are hard-coded. Basically, this type of adaptive fault tolerance can be implemented as a switch statement where each case is represented by one of the considered FTMs. On-line adaptation consists in changing the current branch/case based on some triggers. Providing from the very beginning a set of FTMs rich enough to accommodate all changes demands a great amount of design effort and is often impossible.

2. *agile adaptation*: the application is deployed together with a minimal set of FTMs complying with the initial requirements. Some FTMs may be modified or added after the system is put in operation. During operational life, using the FTMs in real conditions may lead to slight upgrades, parameter tuning or composition of FTMs. According to such runtime observations, new FTMs can be designed off-line but integrated on-line with a limited effect on service availability. This provides much greater flexibility than preprogrammed adaptation. Agile adaptation of FTMs can be considered a type of "compositional adaptation" [McKinley *et al.*, 04] defined as exchanging "algorithmic or structural system components with others that improve a program's fit to its current environment. [...] an application can adopt new algorithms for addressing concerns that were unforeseen during development. This flexibility supports much more than simply tuning of program variables or strategy selection. It enables dynamic recomposition of the software during execution-for example, to switch program components in and out of a memory-limited device or to add new behavior to deployed systems". The means for putting agile adaptation into practice are further shown.

A second criterion for analyzing the adaptation of FTMs is the granularity of changes performed during transitions:

1. *coarse-grained adaptation*: FTMs are replaced in a monolithic manner, the initial one is removed and a new one is introduced. This can imply a lot of duplicated code, thus rendering the application difficult to maintain, as many FTMs share certain features. Furthermore, certain complex operations such as state transfer between the two FTMs are mandatory in the case of monolithic transitions.

2. *fine-grained adaptation*: FTMs are replaced in a differential manner. By analyzing the initial FTM and the final one, common and variable features become apparent. In order to minimize the impact on the architecture of the fault-tolerant application, adaptation should only affect the latter, i.e., a fine-grained modification should be performed. The identification of such variable features has additional benefits, such as reducing the amount of

time and effort spent on designing and developing new FTMs, as common parts can be reused. Besides, it can alleviate the task of transferring the state between the two FTMs.

1.4.3 The Scientific Challenge

Given the above considerations, the issue we focus on can be summarized as follows: how to perform *agile* adaptation of FTMs *at runtime* by modifying, in a *fine-grained* approach, only the specific points which are different between the considered FTMs? To tackle this challenge, we propose a novel solution for the *on-line* evolution of FTMs that minimizes the impact on system behavior and architecture at runtime. This solution leverages recent advances in the field of software engineering.

The benefits of agile adaptation are obvious, especially in the case of complex systems having a long lifespan in a dynamic environment, but also for resource-constrained systems which cannot afford to load a plethora of FTMs from which only one is used at a time. A fine-grained approach is beneficial both for agile and preprogrammed adaptation as it can substantially reduce the amount of time necessary for performing a transition between FTMs and building new ones from existing bricks. To the best of our knowledge, agile fine-grained adaptation of FTMs leveraging a standard component model has not yet been tackled. Based on related work we have examined, adaptation of FTMs has so far been performed in a coarse-grained preprogrammed manner, with some exceptions.

1.4.4 Two Ways of Stating the Problem

In software engineering and in other fields of science, there are two natural approaches for reasoning about a problem or for designing a system: top-down and bottom-up. The first one consists in breaking down the initial system into subsystems, which are then refined in greater detail, sometimes on several additional levels, until the elementary building blocks are reached. The second one consists in assembling systems for building larger systems. When used individually, each one of these techniques can present some drawbacks: in a top-down design, one might end up with a nonuniform design granularity for the different subsystems, refining too much certain parts and losing sight of the overall system objectives, while in the bottom-up approach, one might end up with a complex mixture of subsystems and links very far from the initial big picture. In practice, these two techniques are usually applied in conjunction in software development: the problem is broken down is subproblems (top-down) while code reuse and the introduction of Commercial-Off-The-Shelf (COTS) software are typical examples of bottom-up design.

The scientific challenge that we aim to solve can be stated from two different points of view, similar to the above-mentioned reasoning methods. In a top-down view, the evolution of dependable systems and the transition from dependability to resilience represents an issue which needs to be addressed. Fault tolerance is one of the facets of dependability particularly targeting change and dynamics during the operational life. As such, fault tolerance mechanisms must no longer be predefined statically at design time but in a flexible way enabling subsequent modification, composition and reuse, should the need present itself. This calls for special software development tools enabling separation of concerns, fine-grained de-

sign and architectural modifications at runtime. Such tools correspond ideally to reflective component-based middleware. In a bottom-up way of seeing things, recent advances in software engineering such as Component-Based Software Engineering (CBSE) [Szyperski, 02], Service Component Architecture (SCA) [Marino and Rowley, 09], and Aspect-Oriented Programming (AOP) [Kiczales *et al.*, 97] clearly open unprecedented perspectives and opportunities for research in other fields, among which dependable computing. Consequently, it is definitely worth trying to see what benefits these technological advances bring to dependable computing and what are their limits. The scientific challenge lies at the intersection of these two reasoning methods: while it is essential to keep in sight the big picture, to thoroughly understand what must be accomplished and to be able to break down the overall goal in smaller ones, it is also important to let oneself be guided by the features provided by the technological advances which can offer a fresh perspective over an older topic. To the best of our knowledge, targeting agile fine-grained adaptation of fault tolerance software at runtime using such approaches has not yet been investigated.

1.5 Overall Approach

Separation of concerns [Dijkstra, 82], is a generally accepted idea for implementing highly modularized software [Parnas, 72] and enabling the separate development of the functional behavior and of the cross-cutting concerns [McKinley *et al.*, 04] such as Quality of Service, security, dependability etc.. In the case of fault-tolerant applications, the obvious two concerns are the functional services and the non-functional ones, i.e., the fault tolerance mechanism(s).

Computational reflection [Maes, 87], consists in a program's capacity to reason about its structure (i.e., structural reflection) and its behavior (i.e., behavioral reflection) and, possibly, modify them. Reflection consists of two activities: introspection, enabling the application to observe itself, and intercession, enabling the application to modify itself.

Software architectures built on these principles consist of two abstraction levels where the base level provides the required functionalities (i.e., the business logic) and the top level/meta-level contains the non-functional mechanisms (e.g., FTM(s) [Fabre, 09]). As we target the adaptation of FTMs, we must manage the dynamics of the top level, which can have two causes:

- The application level remains unchanged but the FTM must be modified either because the fault model changes (e.g., due to physical perturbations) or because fluctuations in resource availability make it unsuitable or, at least, suboptimal from a performance viewpoint (i.e., changes in *FT* or *R*).

- Changes in the top level are indirectly triggered by modifications in the application level which make the fault tolerance mechanism unsuitable (i.e., changes in *A*). In this case both levels execute a transition to a new configuration.

The adaptation of FTMs can be viewed as a third concern, thus requiring a third abstraction level. This separation has the same benefits as the separation between the functional and the non-functional levels, ensuring flexibility, therefore, reusability and evolvability of the adaptation mechanisms [Redmond and Cahill, 02]. As previously mentioned,

agile adaptation of FTMs can be regarded as a particular form of compositional adaptation [McKinley *et al.*, 04]. Figure 1.3 shows the main technologies enabling compositional adaptation identified in [McKinley *et al.*, 04]. We share this view of necessary technologies and concepts, and the further enumerated steps of the overall approach emphasize it.

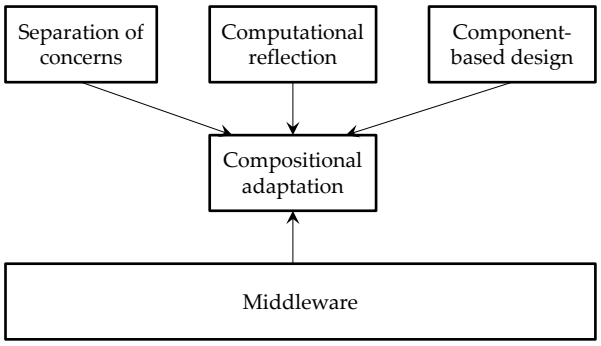


Figure 1.3: Main Technologies Supporting Compositional Adaptation ([McKinley *et al.*, 04])

In order to tackle this particular aspect of resilient computing, namely on-line fine-grained adaptation of FTMs, several milestones must be achieved. First of all, the previously mentioned change mode is detailed, in order to enumerate all the parameters responsible for the choice of a particular FTM over another. The values of these parameters indicate, at design time, which is the most appropriate FTM to attach to the application, if such an FTM exists. The variation of these parameters may invalidate the initial choice and trigger a transition towards a new FTM, determined by the new values. A classification of well-known FTMs is obtained based on this change model (and its accompanying frame of reference). Furthermore, it provides the logical base for a set of transition scenarios between FTMs.

Next, we thoroughly analyze a subset of FTMs from the classification in terms of structure and behavior. The purpose of this analysis is to identify common points and variable features that form a generic protocol execution scheme. Needless to say, on-line fine-grained transitions are difficult (impossible?) to perform without some previous preparation steps and without the proper tools. These preparation steps are what we call a "design for adaptation" approach, which means that the considered FTMs must be designed in such a way as to isolate common and variable features for subsequent reuse and modification. This approach has benefits not only for runtime adaptation but also for the development of static FTMs as it provides building blocks which can be reused and customized. The result of this step is a system of patterns for fault tolerance. The effectiveness of the approach is evaluated by measuring the impact of design refinements on the development time necessary for building new FTMs. This preparation step should not be regarded as another form of preprogrammed adaptation limited to this subset of FTMs. Based on the lessons learned from the "design for adaptation" step, particularly the generic protocol execution scheme, the transition scenario can be extended to encompass new variants of the FTMs from the illustrative subset, other FTMs from the classification or other non-functional mechanisms (e.g., authentication, encryption, logging etc.).

Once the variable features between FTMs are clearly identified, the next step is to leverage specific tools for runtime adaptation such as reflective middleware [Coulson, 01], i.e., a mid-

dleware providing introspection and intercession capabilities, and map the previous design patterns for fault tolerance on a component-based architecture. At this stage as well, design choices are extremely important as they determine the granularity of on-line modifications: the variable features must be mapped to individual component entities which can be identified and modified at runtime, otherwise the "design for adaptation" approach is fruitless. We implement our previously mentioned transition scenarios in order to prove the feasibility of our approach and to illustrate the dynamics brought by this state-of-the-art software engineering support to our adaptive FTMs. The benefits of our approach are discussed and further evaluated by measuring the impact of runtime differential modifications on the architecture of the initial FTM in terms of number of modified components and duration of reconfiguration (i.e., of executing the transition from the initial component-based architecture/configuration to the final one). The consistency of the adaptation process, a crucial aspect as on-line transitions must modify FTMs in a transactional manner, is also discussed.

Last but not least, we evaluate the usability of our adaptive fault tolerance middleware through two examples of integration. In the first one, we integrate adaptive fault tolerance in a design-driven development process for Sense-Compute-Control (SCC) applications (and underlying tool suite named DiaSuite). This example proves that our adaptive FTMs are general enough to be attached to external applications provided by different stakeholders. The design-driven methodology ensures a traceability of fault tolerance requirements from design to deployment and is meant to assist the developers of Sense-Compute-Control applications in the process of adding fault tolerance to their application by rendering it as transparent as possible. In the second example, the proposed adaptive fault tolerance mechanisms are integrated in Srijan, a toolkit for developing wireless sensor network-based applications using data-driven macroprogramming. The goal is, first, to develop a language for specifying fault tolerance requirements and describing the FTMs and, second, to integrate the FTMs and to customize them for the targeted platform and applications. It is worth emphasizing that DiaSuite and Srijan entered the integration process with no previous explicit notion of fault tolerance. At the end of this work, the platforms can either use the FTMs as stand-alone solutions or take advantage of their adaptive nature.

1.6 What This Thesis Is NOT About

Having stated the research goal and given an overview of how to reach it, we establish the perimeter of the present contributions. As such, the risk of creating false/unrealistic expectations is minimized.

- We do not build new software engineering tools, e.g., a new component-based middleware, a new DSL etc. One of the research goals is to examine the benefits and limitations of using already existing software engineering tools (built by specialists in a different field of expertise than ours) for tackling an issue in our field of research.

- We do not design new FTMs. There is a rich body of research on the topic dating back as early as the 60s [Avižienis, 67, Lyons and Vanderkulk, 62], and the intuitive principles lying at the foundations of fault tolerance, such as replication and design diversity can be

traced back long before the era of digital computers, e.g., in Dionysius Lardner's article on Babbage's calculating engine [Lardner, 34]. Furthermore, FTMs have long been used in industry. In this work, we revisit existing (static) FTMs under the angle of adaptation and evolvability.

- The decision for adaptation is not a part of the contributions presented in this thesis. We focus on what happens in the system once this decision is taken (by a system manager, be it human or not) and on how the system must be prepared for adaptation. The triggers of adaptation are discussed but we do not provide a decision manager, a monitoring system or a control-command system.

1.7 Related Work

The topic of this dissertation is at the intersection of several research areas:

1. Fault Tolerance (FT) and Fault Tolerance Mechanisms (FTMs)
2. Adaptive Fault Tolerance (AFT)
3. Design patterns for fault tolerance
4. Component-Based Software Engineering (CBSE)
5. Autonomic Computing (AC)
6. Reconfigurable (ubiquitous) systems

1.7.1 Fault Tolerance and Fault Tolerance Mechanisms

An extensive body of research exists on the topic of fault tolerance in dependable computing systems [Lee and Anderson, 90, Laprie *et al.*, 92, Knight, 12]. The set of fault tolerance mechanisms covered in our transition scenarios are thoroughly discussed in Chapter 2.

1.7.2 Adaptive Fault Tolerance

The field of adaptive fault tolerance has attracted the interest and efforts of researchers for some time now. Just as the principles lying behind fault tolerance are very intuitive (i.e., redundancy of data, hardware, design diversity), the shortcomings of applying static fault tolerance in any situation are also easy to perceive. Static fault tolerance demands complete understanding and control over the system, its mission and its environment throughout its entire service life, blocks the evolution of user requirements and requires an amount of resources which often proves to be prohibitive for real-life systems. Although the motivation is straightforward, the means to implement a solution are much less.

Certainly, there are a lot of projects in the literature focusing on adaptive fault tolerance. However, the meaning of adaptation of fault tolerance mechanisms is not uniform. Adaptation most often has a parametric form, e.g., changing the number of replicas inside one particular FTM or tuning the interval of time between two consecutive checkpoints in the case of passive

replication. In our work, adaptation implies executing a transition between one FTM which is no longer consistent with current operational conditions to another one, i.e., an evolution which can be represented as a trajectory in a frame of reference defined by certain parameters, as explained in Chapter 2.

The first attempt we have found to establish the concept of Adaptive Fault Tolerance (AFT) and the major underlying research issues is presented in [Kim and Lawrence, 90]. The definition of AFT, extracted from this work, has already been cited on page 8. It is extremely interesting to analyze this theoretical work dating back more than twenty years ago, in light of research advances and trends that have taken place meanwhile, e.g., the autonomic computing initiative, the introduction of Aspect-Oriented Programming (AOP), of standards such as Service Component Architecture (SCA), the birth and consolidation of Wireless Sensor Networks (WSNs) and ubiquitous computing etc., and to compare it with our own view and contributions (both theoretical and practical) to AFT, largely influenced by the aforementioned advances. AFT, which was back then “still at the concept-formulation stage" is viewed as “a far-fetched notion in comparison to the notion of fault tolerance that has been dealt with in the past three decades" (today, this means more than fifty years ago). “In a sense, the AFT can be viewed as a highly specialized branch of fault-tolerant computing since it is relevant only in large-scale complex distributed computing systems applications which involve widely varying dynamic changes in their operating conditions." As the authors do not give a quantitative definition of “large-scale complex distributed computing system", it is difficult to say whether this restriction on the applicability of AFT in modern systems is still valid. However, it is worth mentioning that, while at that time a complex distributed computing system generally consisted of a given number of computers interconnected through a Local Area Network (LAN) or a Wide Area Network (WAN), nowadays, systems are becoming increasingly heterogeneous, incorporating devices ranging from small sensors to smartphones and computers, e.g., intelligent building management systems. These modern systems might not necessarily be extremely complex in terms of application logic (e.g., start heating rooms when temperature goes beyond a given threshold and there is a person inside) but do require fault tolerance and taking into account evolution of user requirements and changes in resources, e.g., sensor hardware aging.

There are several common points between the claims of the authors of this paper and the ones we make in our work. They first notice that “the distinctive nature of AFT as compared to more conventional fault tolerance becomes more evident when the fault tolerance requirement or the resource availability changes in a noticeably discrete fashion, i.e., from one mode to another mode". In our change model, parameters also change in a discrete manner, i.e., from one fault model to another one, or from determinism to non-determinism. Further on, they state that a Fault Tolerance Management System (FTMS) that is able to enter a new mode of operating fault tolerance capabilities (an “adaptive multi-mode FTMS") is “bound to have a highly modularized structure and thus easier to implement reliably than monolithic static FTMSs which may mobilize all available fault tolerance mechanisms all the time". Although in our work we do not build a fault tolerance management system (i.e., we do not tackle adaptation decision-making), our fine-grained fault tolerance mechanisms are built on this core assumption that a highly modularized structure can evolve much more easily than a monolithic one. Regarding the origin of changes, the authors consider that changes can come from the ap-

plication environment (external) or from the internal fault pattern. They identify two types of changes: anticipated and unforeseen. They do not discuss adaptation to unforeseen changes, as the adaptation to anticipated changes is already challenging enough.

In [Armstrong, 94], the Adaptive Fault Tolerance Program is described, aiming to provide complex distributed military systems with greater degrees of survivability and graceful degradation than currently available. The definition of AFT [Kim and Lawrence, 90] also stems from this program. The need for an adaptive approach to fault tolerance in the context of Battle Management Command, Control, Communication, and Intelligence (BMC3I) systems is clearly stated as such systems must cope with dynamics on various dimensions. A taxonomy of faults is established and several already existing FTMs are classified according to this taxonomy. The concepts are demonstrated on a case study represented by the monitor function of the Strategic Defense System Command Center Element (SDS-CCE). The adaptation consists in replacing the Recovery Blocks mechanism (RB) [Randell, 75] attached to the monitor function with Distributed Recovery Blocks (DRB) [Kim and Welch, 89]. Later, this led to the definition of the Adaptable Distributed Recovery Block (ADRB) scheme [Kim *et al.*, 97]. RB is a strategy for tolerating software faults consisting in running on a single processor a series of "try-blocks" coded differently for the same function and submitting the result to an acceptance test whose result determines whether the result can be sent to the client or the subsequent block must be executed. This mechanism requires application state restoration (rollback) between two consecutive blocks. DRB is the distributed version of RB. Each block is executed on a separate processor and as such it facilitates forward recovery and is appropriate for real-time applications, unlike the sequential version. In the adaptation scenario, the transition between RB and DRB is triggered based on the evaluation of the time spent in backward error recovery (i.e., the rollback between two subsequent blocks in RB) as the error rate increases. As this time increases, deadlines are missed more frequently. This requires replacing RB with DRB. The implementation is not discussed in detail and the project appears to have stopped prematurely in an interim demonstration stage.

In the analysis of existing work on AFT, we have considered the following comparison criteria:

1. what is provided?
2. target applications/adaptation scenario
3. change model (i.e., a frame of reference for describing system evolution composed of parameters whose variation triggers the adaptation of FTMs)
4. preprogrammed vs. agile
5. granularity (monolithic vs. differential replacement of FTMs)

What Is Provided?

Regarding the first point, contributions in adaptive fault tolerance generally take the form of a middleware leveraging a distributed communication support [Bharath *et al.*, 01] such as Common Object Request Broker Architecture (CORBA). The architecture of the middleware usually

follows a pattern, featuring an Adaptive Fault Tolerance Manager in charge of receiving and analyzing input from the user, system probes for monitoring resources, various logs regarding resource usage, system failures and deciding when the current FTM is no longer appropriate, which is the most suitable strategy among the available ones and triggering the transition.

In [Shokri *et al.*, 97], an architectural framework of an Adaptive Fault Tolerance Management (AFTM) middleware is presented. The authors start by identifying target applications for AFT, namely mission-critical systems such as military planning and monitoring applications, which are inherently distributed, subject to change and have an operational life consisting of multiple phases. The change model they briefly mention is similar to ours, featuring the following sources of change: environment, application requirements and resources. However, the definition of these axes is vague. Their middleware provides fault tolerance mechanisms (called modes) dealing with hardware and software faults: sequential recovery blocks, distributed recovery blocks and exception handler. It is not clear how transitions are performed between these modes, whether on-line or off-line and neither the granularity of change nor its reactive/proactive triggers are invoked. The practical result of this work is a CORBA-based Adaptive Fault Tolerance Management middleware, developed on top of Solaris and leveraging the multi-threaded programming and real-time facilities provided by this operating system. It is interesting to note that the authors, like us, stress the importance of using state-of-the-art software engineering methods for tackling the challenges of building their middleware. At the time, this meant using object-oriented design and programming for separation of concerns (both between functional and non-functional aspects and, within fault tolerance software, between application-independent and application-specific features) and location-transparent object interactions through CORBA. Method call interception is performed by using the filter facility provided by Orbix, the CORBA-compliant Object Request Broker (ORB) used in this work. This filter mechanism, like aspect-oriented programming, enables the application developer to introduce additional code to be executed before and after the actual server program.

A framework for building adaptive fault tolerance in multi-agent systems is presented in [Marin *et al.*, 01]. Such distributed systems are essentially decentralized and subject to host or network failures, changing requirements and environment. A particularity of multi-agent systems is that the relative importance of the various entities involved can change during the computation. An example of application domain given by the authors is the field of crisis management systems in which software is developed for assisting different teams (e.g., paramedics, policeman, firemen) and coordinating them. The focus shifts from one activity to another, therefore the criticality of each activity fluctuates according to the different stages of the crisis. As it is cost-prohibitive to attach fault-tolerance protocols all the time to all entities, it is essential to be able to tailor the FTMs to the requirements of each task. The authors present a Java-based framework called DARX for designing reliable distributed applications. Each task can be replicated any number of times resulting in a replication group completely hidden to other tasks using either passive or active replication. The framework enables each agent to change at runtime its replication strategy and to tune the parameters of the current strategy (e.g., number of replicas, interval between checkpoints in the case of passive replication). Each agent has an associated replication manager which keeps track of all the replicas in the group and of the current strategy. The manager is also in charge of switching between replication strategies. The

adaptation of FTMs is not tackled dynamically, in the sense that both replication strategies seem to be hard-coded and available all the time (e.g., in a switch statement) and the transition between them consists in changing the case branch to be executed. Furthermore, the granularity of the adaptation is not taken into account.

Other works, such as [Goldberg *et al.*, 93, Hiltunen and Schlichting, 94, Hiltunen and Schlichting, 96], focus more on conceptual frameworks, algorithms, on defining AFT and providing examples of target applications. For example, in [Hiltunen and Schlichting, 96], a model for describing existing adaptation algorithms is presented. This model breaks the adaptation process in three steps: change detection, agreement and action. In [Chang *et al.*, 98], an approach for adapting to changes in the fault model is presented. More precisely, a program that assumes no failures (or only benign failures) is combined with a component that is responsible for detecting if failures occur and then switching to a given fault-tolerant algorithm. Provided that the detection and adaptation mechanisms are not too expensive, this approach results in a program with smaller fault-tolerance overhead and thus a better performance than a traditional fault-tolerant program. In [Gong and Goldberg, 94], the authors present, in the context of the conceptual framework described in [Goldberg *et al.*, 93], two algorithms for adapting between Primary-Backup Replication [Budhiraja *et al.*, 93] and the State-Machine approach (more generally known as Active Replication or Triple Modular Redundancy in the literature) [Wensley *et al.*, 78, Schneider, 90]. Their approach is modular, in the sense that any Primary-Backup or State-Machine protocol can be used among the existing ones. In order to execute the heavy State-Machine protocol only when the need presents itself, i.e., in the presence of arbitrary or Byzantine faults, their adaptive algorithm executes a default Primary-Backup algorithm and also attempts to detect the presence of non-manifest faults, i.e., faults such as crash and omission. For this, the authors modify the Primary-Backup protocol by letting the backup servers participate in the service as well. The backup servers receive not only checkpoints containing the state of the primary server after request computation (like in ordinary Primary-Backup Replication) but also the initial client request. They monitor the primary for any inconsistency by checking whether the primary computes requests in First In, First Out (FIFO) order and whether the response is correct. The authors make the assumption that servers have implemented a Byzantine agreement protocol for agreeing on the next client request to process. If the backup servers detect an inconsistency, they report the error to the client and start a Byzantine agreement protocol among themselves to recover from it. While this work explores the relationship between the two protocols and enables a transition based on a change of fault model, the approach is still preprogrammed, as the algorithm is hard-coded from the beginning.

In [Zheng and Lyu, 10] an adaptive fault tolerance algorithm and a middleware, both targeting reliable Web services, are presented. The authors propose a model of user requirements and Quality of Service (QoS) attributes and an algorithm for selecting the most appropriate strategy between sequential strategies (either Retry [Chan *et al.*, 07] or Recovery Blocks [Randell, 75]) and parallel ones, like N-Version Programming [Avižienis, 85] or Active Replication [Salatge and Fabre, 07]. The proposed QoS-aware middleware is inspired by the user-participation and user-collaboration concepts of Web 2.0. This user-collaborated middleware is used for obtaining and maintaining up-to-date information wrt user QoS requirements

and the actual QoS attributes of the available Web services, representing the necessary input for the algorithm which determines the most appropriate strategy. From our understanding, this is a preprogrammed adaptation as the set of strategies is predefined and cannot be enriched on-the-fly.

Target Applications & Change Model

Concerning the targeted applications and domains, examples come from: deep-space autonomous systems [James *et al.*, 10, Hecht *et al.*, 00], military/strategic defense systems [Armstrong, 94], crisis management systems [Marin *et al.*, 01], reliable SOA applications based on Web services [Zheng and Lyu, 10], etc. All these critical systems have in common the fact that they operate in highly dynamic environments and that they must adapt to changes during their lifetime.

Dynamics occurs along various dimensions, representing the considered change model: available resources, user preferences, QoS requirements, operational phases, external threats be they in the form of enemy attacks or cosmic radiation etc.

Preprogrammed vs. Agile Adaptation & Granularity of Modifications

With respect to the granularity of adaptations, as already discussed, transitions between FTMs can be executed either in a fine-grained or in a coarse-grained manner. Although not directly targeted at FTMs but at fault-tolerant protocols which are part of larger FTMs, e.g., group oriented multicast, [Hiltunen and Schlichting, 93] presents an interesting approach for achieving modularity. The idea is to implement the individual properties of a protocol as separate micro-protocols and then to combine the necessary micro-protocols in a composite protocol by using an event-driven software framework. Micro-protocols are structured as collections of event handlers, i.e., procedures which are called when a specific event occurs. For obtaining different types of reliable multicast, the authors propose the following micro-protocols: FIFO (message ordering) which can be reliable or unreliable, causality (reliable or unreliable transmission), total (message ordering) available in different versions/algorithms, reliability available in two versions (one based on negative acknowledgment and one based on positive acknowledgment), stability, membership, which is further broken down in micro-protocols dealing with process failures and joins, liveness and validity. The authors propose in [Bhatti *et al.*, 97] a similar approach for achieving fine-grained construction of communication services. In our work, more specifically in the "design for adaptation" step, we are guided by a similar idea: isolating the specific features of FTMs in components constituting a generic execution scheme. We go one step further, by leveraging this fine-grained design at runtime through differential adaptations.

In [Pareaud *et al.*, 08], the authors present an approach leveraging the reflective component-based middleware support provided by OpenCOM [Coulson *et al.*, 08] for enabling the fine-grained adaptation of FTMs. The fault tolerance software is decomposed in generic services, that are likely to remain unchanged during transitions, and specific inter-replica protocols (IRPs). A fine-grained componentization of the FTM is achieved based on the taxonomy of fault tolerance proposed in [Avižienis *et al.*, 04]. The transition between Leader-Follower Replication

(semi-active replication) and Primary-Backup Replication (passive replication) is presented as a case study in which only the component representing the inter-replica protocol must be changed. In our approach, the componentization of FTMs goes further, as the inter-replica protocol is also broken down in separate components based on a generic protocol execution scheme.

Another work leveraging component-based support, this time provided by CORBA Component Model (CCM) [ccm, 11], is presented in [Fraga *et al.*, 03]. The Adaptive Fault-Tolerance on the CORBA Component Model (AFT-CCM) enables the adaptation of the fault tolerance strategy at runtime transparently for the application layer. Based on fault occurrence rate and QoS requirements specified by the user at runtime, the AFT manager entity switches between a configuration with no replication, a configuration with passive replication (also known as Primary-Backup in the literature) [Speirs and Barrett, 89] and a configuration with semi-active replication (also known as Leader-Follower in the literature) [Barret *et al.*, 90]. The actual inter-replica protocol (corresponding either to passive replication or to semi-active replication) is implemented in one component, called the Replication coordinator. The granularity of modifications is coarser than in our approach as a transition implies replacing the whole Replication coordinator component. Furthermore, adaptation appears to be preprogrammed. An infrastructure tackling the same adaptation scenario, this time leveraging Fault-Tolerant CORBA (FT-CORBA) [ftc, 02] is presented in [Lung *et al.*, 06]. The authors propose a set of extensions to the standard (which provides static fault tolerance support) consisting in interfaces and service implementations enabling the change of the replication strategy at runtime. The set of FTMs among which transitions can be executed is preprogrammed (known at design time).

Chameleon [Kalbarczyk *et al.*, 99] is an adaptive software infrastructure enabling different degrees of fault tolerance requirements (and, consequently, different fault tolerance strategies) to be supported concurrently in a network environment. The system is based on the use of ARMORs (Adaptive, Reconfigurable, and Mobile Objects for Reliability) that control all operations. There are three types of ARMORs:

- *Managers* remotely install and uninstall ARMORs on other nodes, assign unique identifiers to them and keep an up-to-date list of them. There are three main types of managers:
  - the *Fault Tolerance Manager* interprets the user's fault tolerance requirements and chooses an appropriate fault tolerance strategy, initializes parameters characteristic to the strategy, chooses the hosts on which to deploy the strategy based on criteria such as load and failure;
  - the *Surrogate Manager* executes the application under the specified fault tolerance strategy and can install any other ARMOR necessary for executing the application under that strategy;
  - the *Backup Fault Tolerance Manager* prevents the Fault Tolerance Manager from becoming a single-point-of-failure by monitoring it, maintaining its own state consistent with it and replacing it in case of failure.
- *Common ARMORs* encapsulate specific techniques that are part of fault tolerance strategies, e.g., heartbeat, voter, checkpoint etc.

- *Daemons* reside on every node and perform accessory functions such as installing ARMORs locally, monitoring local ARMORs, acting as gateways between ARMOR and the Chameleon environment etc.

Fault tolerance mechanisms are broken down in common ARMORs, which shows concern for fine-grained design and for facilitating evolution and adaptation. Furthermore, the structure of ARMORs can be immediately translated in CBSE terms: they are composites containing basic building blocks which, in turn, may also be composites of smaller building blocks. Like in CBSE, to ensure the substitutability of building blocks, they must implement the same interface i.e., have the same inputs and outputs. Changing the structure of ARMORs at runtime is possible thanks to the CompositeController, a basic building block in charge of replacing the other building blocks, after ensuring that they are in a quiescent state, i.e., not currently processing a message. New ARMORs can be developed and integrated on-the-fly (i.e., adaptation can be performed in an agile manner) in systems allowing dynamic linking. Otherwise, all components which are likely to be necessary during the lifetime of the application must be included a priori. This work is similar to ours as fault tolerance strategies have a fine-grained design and adaptation can be performed both in a preprogrammed and in an agile manner. However, as this is a framework, much is left to the judgment of the developer. Furthermore, the considered fault tolerance mechanisms are not described methodologically, therefore their actual design is unclear. Another issue is the fact that Chameleon does not leverage an existing mature component-based support or a standard component model.

As already stated, the vast majority of works dealing with AFT which have come to our attention tackle adaptation in a preprogrammed manner, hard-coded at design time. Undoubtedly, the underlying middleware support (e.g., CORBA) plays a significant role in this: if the support does not provide "plug-and-play" capabilities for runtime reconfiguration, agile adaptation cannot be performed. In this work, we leverage a reflective component-based middleware which provides such facilities. In the case of CORBA-based approaches [Shokri *et al.*, 97, Narasimhan *et al.*, 05, Lung *et al.*, 06, Fraga *et al.*, 03], adaptation is done in a preprogrammed manner because the underlying support cannot accommodate unanticipated changes. However, by integrating the contribution presented in [Sadjadi and McKinley, 04], namely the Adaptive CORBA Template (ACT), adaptation could also be done in an agile manner in the aforementioned works. Based on generic interceptors, this framework template can be composed with existing adaptive CORBA frameworks to help tackle unanticipated changes.

1.7.3 Design Patterns for Fault Tolerance

Design patterns [Gamma *et al.*, 93] represent efficient and reusable solutions to recurring problems in different fields (e.g., architecture, mechanics, software engineering). In the field of object-oriented development process, design patterns are building bricks from which more complex designs are created. In addition to this, patterns do not exist in isolation but present interdependencies which become apparent when they are placed in pattern systems. A pattern system is defined as a "collection of patterns for software architecture, together with guidelines for their implementation, combination and practical use in software development" [Buschmann *et al.*, 96, Chapter 5: Pattern Systems]. In the dependability area, FTMs represent

well-established strategies providing solutions to specific problems described in terms of the (*FT,A,R*) parameters. These strategies are often strongly related, especially when they comply with the same FT requirements, e.g., duplex strategies for tolerating crash faults, voter-based strategies and acceptance-test-based strategies for tolerating software faults. Therefore, it is only natural to describe FTMs in the form of design patterns for fault tolerance and to place them in pattern systems which highlight their connections. We have identified several works on this topic.

In [Saridakis, 02], several well-known mechanisms for error detection (e.g., `Acknowledgment`, `I Am Alive`), error recovery (i.e., `Rollback` and `Roll Forward`) and error masking (e.g., `Active Replication`) are described in the form of design patterns. Detection, recovery and masking are three aspects of fault tolerance and the author uses this classification scheme to organize them in a pattern system. This classification can be further extended either with other existing mechanisms dealing with one of the three aspects or with new categories. In our pattern system, we go one step further by identifying the structural similarities inside the fault tolerance mechanisms through a generic execution scheme.

In [Daniels *et al.*, 97], the focus is on design patterns for software faults. The authors classify existing software fault tolerance techniques according to the adjudication method (i.e., voter, acceptance-test and hybrid) and extract a general pattern-like structure called the `Reliable Hybrid` pattern from which a great variety of FTMs can be instantiated, ranging from basic approaches (e.g., N-Version Programming, Recovery Blocks) to complex hybrid solutions (e.g., Consensus Recovery Block). This work is complementary to our pattern system which currently deals solely with hardware faults (i.e., crash and value faults). A direction for future work could be to compose this pattern with ours in order to address a broader spectrum of faults.

A more general pattern system is described in [Ferreira and Rubira, 98]. The core of this work is the `Reflective State Pattern`, which is a refinement of the `State` design pattern [Gamma *et al.*, 95, Chapter 5: Behavioral Patterns] based on the `Reflection` architectural pattern [Buschmann *et al.*, 96, Chapter 2: Architectural Patterns]. The `State` design pattern, labeled as a behavioral pattern, allows an object to change its behavior when its internal state changes. The `Reflection` architectural pattern, labeled as a pattern for “designing for change/evolution" in adaptable systems, provides a mechanism for changing the structure and behavior of software systems dynamically. This is achieved by splitting the application in a base-level providing the application logic and a meta-level, providing information about selected system properties. The resulting `Reflective State Pattern` addresses the difficulties of implementing the state machine for the `State` pattern by separating the control aspects from the functional aspects through the use of the `Reflection` pattern. By specializing this pattern for the fault tolerance field, the `Software Redundancy` pattern is obtained, a general pattern which defines a common structure that can be applied to three kinds of fault tolerance, namely hardware, software and environmental. In this pattern, the base-level classes represent the services and their replicates, while the meta-level classes implement the strategy specific to each FTM. To tolerate software faults, for instance, the base-contains a FTComponent providing the service and several RedundantComponents implementing different versions of the service (i.e., design diversity), while the meta-level implements the Voter for the N-Version

Programming technique or the Acceptance Test for the Recovery-Block technique. This is an abstract pattern that must be customized in order to implement environmental, software and hardware fault tolerance strategies, thus leading to a system of patterns. The pattern system is represented as a Pattern-Relationship Tree where patterns are connected by relationships such as refinement, variation or combination. Although this pattern system covers a broader spectrum of faults than ours, it is too coarse-grained for our purpose. In our work, we descend at a finer level of detail and break down the considered FTMs in order to identify structural similarities between them.

1.7.4 Component-Based Software Engineering

Component-based software engineering is a well-established branch of software engineering. The idea of software componentization enabling subsequent reuse can be traced back as early as 1968 in Douglais Mcllroy's talk "Mass Produced Software Components" [McIlroy, 68]. Since then, a plethora of component models, both commercial and academic, e.g., Enterprise JavaBeans (EJB) [ejb] developed by Sun Microsystems, Component Object Model (COM) [com] from Microsoft, CORBA Component Model (CCM) [ccm, 11, Wang *et al.*, 01], OpenCOM [Coulson *et al.*, 08], Fractal [Bruneton *et al.*, 06], have been introduced, using different technologies, targeting different purposes and based on different principles. Based on technologies such as object-oriented development, architecture description languages (ADL) etc., component-based development has not managed to provide a set of standard principles, unlike object-oriented programming [Crnković *et al.*, 11]. A commonly accepted definition of the notion of software component comes from [Szyperski, 02]: "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 party." However, alternative definitions exist emphasizing the importance of component models to which components must comply, as discussed in [Crnković *et al.*, 11]. We do not aim to provide a survey of existing component models as extensive work already exists on this topic stemming from the software engineering community [Szyperski, 02, Heineman and Councill, 01, Crnković *et al.*, 11, Feljan *et al.*, 09, Hošek *et al.*, 10]. The approach we propose abstracts away from the specificities of the various candidates to identify the minimal set of features that must be provided by a certain platform in order for it to be appropriate for our experimental work. We advocate the use of this kind of technologies for enabling agile fine-grained adaptation of fault tolerance mechanisms at runtime in order to make systems resilient. The specific technologies we used in this work are described in Chapter 4.

1.7.5 Autonomic Computing

One of the main challenges of modern systems, as stated by IBM in "The Vision of Autonomic Computing" [Kephart and Chess, 03], is the ever-increasing total cost of ownership (TCO) due to their continuous evolution. Given the growing complexity of both individual systems and dependencies between them, the solution, according to the authors, is autonomic computing, a term coined by IBM in 2001, i.e., computing systems which can manage themselves based on high-level policies provided by human administrators. The term *autonomic* originates from bi-

ology: the autonomic nervous system is a part of the nervous system which controls vital functions such as heart rate, body temperature, breath rate, digestive functions etc. By operating largely below the level of consciousness, it enables the latter to focus on more complex tasks. Similarly, an autonomic computing system should enable the human system manager to focus on high-level task by being self-configuring, self-healing, self-optimizing and self-protective.

Autonomic computing is also enticing for ubiquitous systems based on technologies such as Wireless Sensor Networks (WSNs). The goal of this "trend" is defined in [Sterritt, 05] as addressing "today's concerns of complexity and total cost of ownership while meeting tomorrow's needs for pervasive and ubiquitous computation and communication". One of our first areas of interest was autonomic computing due to its self-healing dimension, i.e., systems which are able to detect when their behavior deviates from the expected one and, if possible, take some repairing action. However, as also pointed out in [Sterritt, 05], many of the issues within autonomic computing have been at the core of other disciplines, e.g., fault tolerance, distributed systems, multi-agent systems etc. The novelty lies in the "holistic aim" of regrouping all relevant research efforts in a common framework. Focusing on the intersection between autonomic computing and fault tolerance, the same author, in [Sterritt and Bustard, 03], reaches the conclusion that dependability and fault tolerance are not only "specifically aligned to the self-healing facet" of autonomic computing but, on a closer view, *"all facets of autonomic computing are concerned with dependability"* (i.e., self-configuration, self-optimization and self-protection as well). As such, the approach presented in this thesis can be regarded as a contribution to autonomic computing.

1.7.6 Reconfigurable (Ubiquitous) Systems and Frameworks

A complex example of (re)configurable framework which also touches the areas of ubiquitous systems, AC and fault-tolerant computing is Gaia [Román *et al.*, 02], which builds on the core concept of *active spaces*. Heterogeneous devices, such as PCs, sensors, smart-phones, blend in the environment and interact with the user through a uniform interface. A method of adding an autonomic dimension to this framework is the use of a planner from the field of artificial intelligence, as presented in [Ranganathan and Campbell, 04], which has the role of creating a path between the current state and a goal state given by a user or by a developer. This planner can be considered analogous to our transition algorithm. In [Chetan *et al.*, 05], the authors add a fault tolerance dimension to Gaia by emphasizing the importance of dependability in pervasive systems, i.e., systems that interact closely with the users, possibly even in a healthcare context. They only endow Gaia with the property of tolerating fail-stop faults. Although this framework tries to cover issues from many areas, including dependable computing, the proposed solution appears to be closely linked to an underlying operating system called 2K, the case studies focus only on managing media presentations and the approach to fault tolerance is somewhat basic.

In the area of reconfigurable software architectures, Rainbow [Garlan and Schmerl, 02] builds on the use of architectural models for problem diagnosis and repair. The proposed framework includes a monitoring layer composed of two types of entities, namely probes that gather basic data from the running system and gauges [Garlan *et al.*, 01] that perform computations on the data in order to obtain measures of the system properties. An architecture manager is in charge of maintaining the architectural model at runtime and of verifying that

the constraints on the system elements are verified and to trigger reconfiguration in case they are not. The project is very complex as it includes a very expressive Architecture Description Language (ADL) called ACME [Garlan *et al.*, 00], a system in charge of verifying constraints, called ARMANI, a library of gauges and a language for describing adaptations, called Stitch [Cheng and Garlan, 12]. The idea of placing an ADL on top of a component-based middleware is the topic of [Batista *et al.*, 05], in which the authors describe their experience in associating an enriched version of ACME with the OpenCOM middleware [Coulson *et al.*, 08] for performing programmed and ad-hoc changes at runtime while maintaining certain constraints.

1.8 Summary

This chapter described the context of our research endeavor and the motivation of our work. Evolution is mandatory, especially in the case of long-lived systems. Dependable systems must be appropriately equipped to accommodate change. In this context, an approach is proposed for tackling a key aspect of resilient computing: on-line adaptation of FTMs in a fine-grained manner. Next, we discussed related work in different fields connected to our goal and emphasized our contributions compared to already existing approaches.

Chapter 2

Adaptation of Fault Tolerance Mechanisms (FTMs)

> "Imagination will often carry us to worlds that never were. But without it we go nowhere."
>
> — Carl Sagan, *Cosmos*

2.1 Introduction

In this chapter, a set of well-known fault tolerance mechanisms is presented, serving as a basis for our investigations on potential adaptations. Already well-documented in the past and used in industrial environments, these FTMs are represented in a frame of reference we have established in order to "visualize" and reason on evolution. This frame of reference is discussed and different representations of these FTMs are provided.

Several transition scenarios, stemming from the underlying characteristics of the considered FTM, are then presented. First, we provide a high-level view of possible transitions. Next, transitions are thoroughly analyzed and additional aspects such as adaptations triggers and stability of transitions are discussed.

2.2 Change Model and Associated Frame of Reference

As previously mentioned in Chapter 1, the choice of an appropriate FTM for a given application is based on the values of a set of parameters (*FT,A,R*). These parameters represent the dimensions along which dynamics may occur and invalidate the initial choice of FTM. At any point in time, (*FT,A,R*) must be consistent with the current FTM in order to guarantee dependability properties. Transitions between FTMs are needed when inconsistency is found between an FTM and operational conditions (i.e., (*FT,A,R*) not in the domain of the FTM). Updating an FTM is required when applications evolve (different characteristics *A*) or when the operational conditions change either because available resources *R* vary or when fault tolerance requirements *FT* change during the service life of the system.

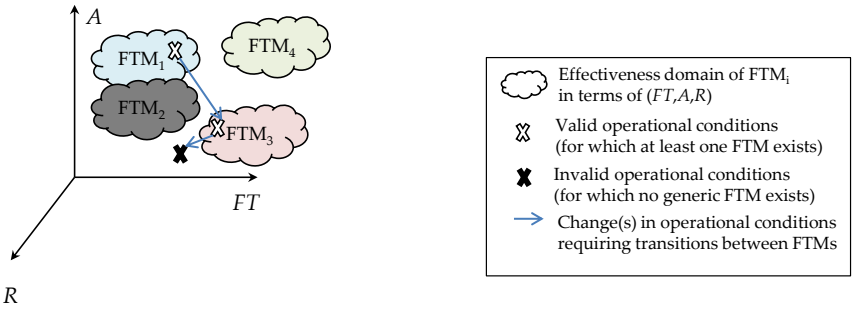


Figure 2.1: Change Model and Associated Frame of Reference

We proposed a 3-axis frame of reference [Stoicescu *et al.*, 11b], resulting from the change model, for depicting the evolution of a fault-tolerant application (see Figure 2.1). Each axis represents one of the multivariate variables (*FT,A,R*). In this space, each FTM occupies a certain place, that we call effectiveness domain, represented by a cloud in Figure 2.1, given by its parameter values. The initial values of (*FT,A,R*) (i.e., the operational conditions) place the application in the effectiveness domain of a certain FTM. When these values change, the application may either fall inside the effectiveness domain of another FTM (in which case, a transition between FTMs is required) or in an empty region, for which no generic solution is available. Evolution during the service life of the system can be visualized as a trajectory in the space defined by this frame of reference.

We make no pretense of this representation being complete and fully accurate. Firstly, the labels symbolize multivariate variables for which we do not have a total order relation. The only axis for which an order relation can be unequivocally established is *R*, e.g., FTM_1 requires more bandwidth than FTM_2 or 3 CPUs instead of 2. For the two others, an order relation could be based on the complexity of FTMs and the difficulty of developing them. In the case of fault models *FT*, a value fault is "harder" than a crash fault because a mechanism tolerating the former is structurally more complex, requiring an acceptance test/assertion/voter which are not present in a mechanism tolerating solely the latter and because, generally, an FTM tolerating the former also tolerates the latter. Further on, software faults are "harder" than hardware value faults because even if the corresponding FTMs have a similar external structure (i.e., N-Version Programming [Avižienis, 85] is similar to Active Replica-

tion [Lyons and Vanderkulk, 62, Chérèque *et al.*, 92], Recovery Blocks [Randell, 75] are similar to Time Redundancy, N-Self-Checking Programming [Laprie *et al.*, 90] is similar to Comparison&Double Duplex), they require design diversity, which is difficult to achieve. Secondly, the representation is inaccurate because the variables are discrete, not continuous. However, it is more difficult to visualize FTMs as sets of points in a three-dimensional grid. The role of this representation is to give a visual perspective of the dimensions of evolution and adaptation during the lifetime of a fault-tolerant application and of the aim of our research, namely providing the means for an application to remain fault-tolerant despite changes ergo to be resilient.

2.3 Classification of FTMs

The axes of our frame of reference represent multivariate variables, e.g., *A* (application characteristics) stands for determinism, statefulness of the application and state accessibility, which represent, in our approach, all the application-related parameters that have an impact on the choice of an appropriate FTM. By refining two of these axes, namely *FT* and *A*, a tree-graph classification of a set of well-established FTMs is achieved (see Figure 2.2) [Stoicescu *et al.*, 11a]. This representation can serve in the process of selection of an appropriate FTM when designing a fault-tolerant application (by browsing the tree in depth) and in deciding towards which FTM a transition must be executed, in case the operational conditions change (by browsing the tree from the leaf representing the current FTM towards the root).

The first criterion in this selection process is the fault models to tolerate. Our fault model classification is based on known types [Avižienis *et al.*, 04], i.e., software and hardware faults. Establishing the fault model leads to the selection of a subtree on which the second class of criteria is applied, namely application characteristics. In the design of fault-tolerant applications (based on separation of concerns), we assume that the functional layer cannot be modified for fault tolerance purposes. The application provides the required "hooks" enabling the FTM to reify service calls and, in some cases, to access service state. However, if the application presents some non-deterministic features, the FTM will not be able to intervene on them. This explains why in Figure 2.2, in case the application is stateful and non-deterministic but does not provide a method for managing state, no generic solution is provided. Non-determinism could be tackled on a case-by-case basis by interfering with the application to solve non-determinism points as done in [Barret *et al.*, 90] to deal with preemptive tasks in real-time systems. In our approach, we adopt a black-box model of the application to maintain a clear separation between the functional and the non-functional layer and the generality of the mechanisms.

For clarity, the tree-graph presents different levels of detail: duplex strategies are refined in more detail than the rest. For example, Active Replication only works in the case of deterministic applications. Also, the criterion related to resources is not shown. However, resources play an important part and represent the last step in the selection process. In case several solution exist, given the values of the parameters *FT* and *A*, the resource criterion can invalidate some of the solutions. For instance, in case the fault model is crash and the application is deterministic and provides state access, two potential FTMs are shown in Figure 2.2, namely Passive and Semi-Active Replication. A cost function can be associated to each solution, based on necessary resources (number of CPUs, memory, network bandwidth, energy consumption).

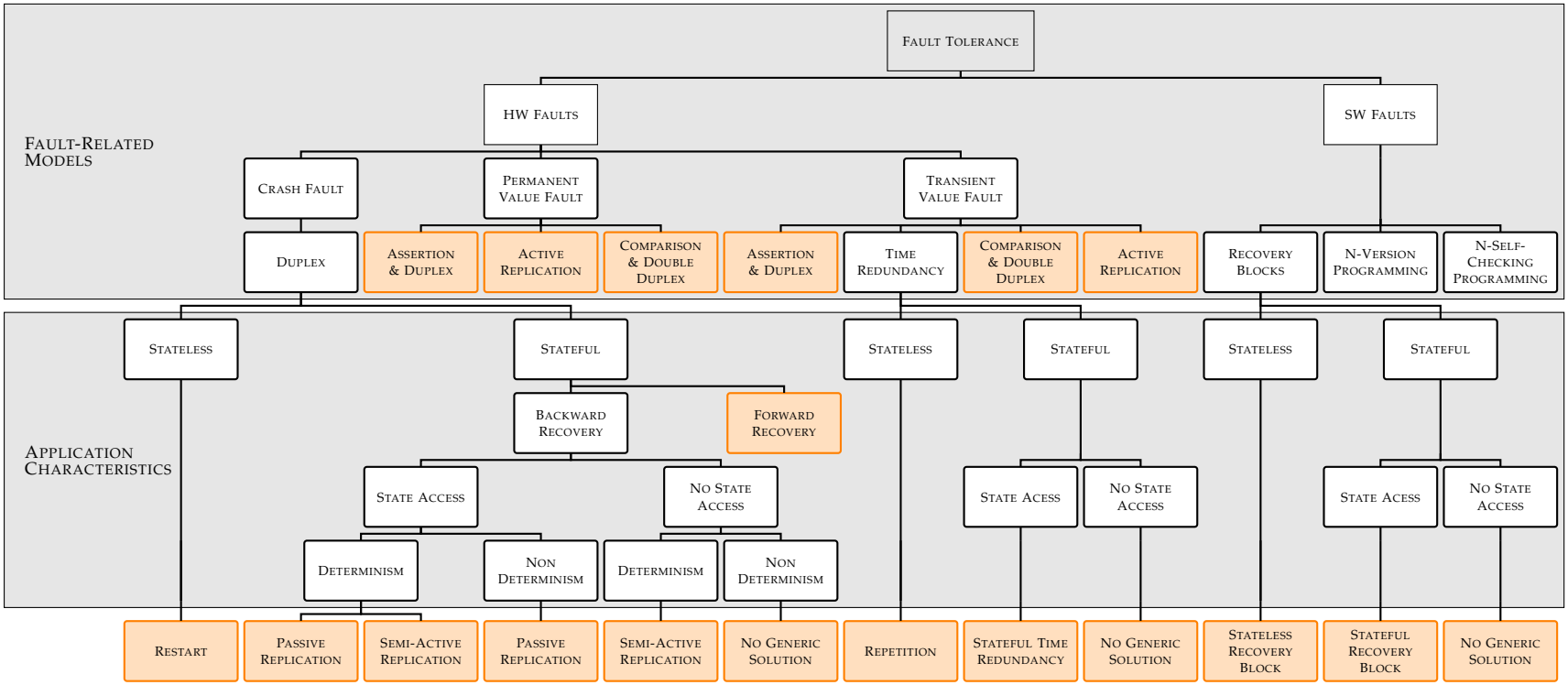


Figure 2.2: Classification of FTMs

2.4 Presentation of Considered FTMs

To illustrate our approach, we describe in detail a set of FTMs among the ones represented in Figure 2.2: the two variants of duplex tolerating crash faults and two mechanisms for tolerating hardware value faults, namely Time Redundancy and Assertion&Duplex. We discuss both the way these FTMs operate and their underlying characteristics in terms of the (*FT,A,R*) parameters. The discussion is limited to this subset of FTMs because they represent the practical basis for the rest of this work. The other FTMs contained in Figure 2.2 are well-established solutions thoroughly discussed in literature, e.g., N-Version Programming (NVP) is presented in [Avižienis, 85], N-Self-Checking Programming is presented in [Laprie *et al.*, 90] and Recovery Blocks are presented in [Randell, 75].

2.4.1 Tolerance to Crash Faults

Crash faults in client-server systems can be tolerated through duplex protocols by replicating the server on two or more hosts (master and slave(s)), assuming that a host running the server is fail-silent. A crash of the master is detected by a dedicated entity (e.g., heartbeat, watchdog) and triggers a recovery action through which the slave becomes master (or one of the slaves is elected master).

There are two main types of duplex protocols, represented in Figure 2.2: passive and semi-active (sometimes also called active). *Primary-Backup Replication (PBR)* [Speirs and Barrett, 89] is a passive strategy: only the master (primary) processes client requests and sends checkpoints containing its state to the slave(s) (backups). Checkpoints can be sent either for each request processed or at a certain interval or in a differential manner, depending on the application. *Leader-Follower Replication (LFR)* [Barret *et al.*, 90] is a semi-active strategy: all replicas process input requests but only the master (leader) replies to the client. Checkpoints are not necessary because each replica updates its own state.

2.4.2 Tolerance to Hardware Value Faults

Hardware value faults are tolerated using repetition of request processing or duplex processing with acceptance tests/assertions or parallel executions followed by voting, in the case of active replication [Chérèque *et al.*, 92] (also known as Triple Modular Redundancy [Lyons and Vanderkulk, 62]) or the state machine approach [Schneider, 90] in the literature. Several strategies exist, as shown in Figure 2.2 among which we analyze two. *Time Redundancy (TR)* tolerates transient value faults and requires only one host. A request is processed twice and results are compared. If results differ, it means a transient fault occurred, so the request is processed again and if two results out of three are identical, the reply is sent. *Assertion&Duplex (Assert&Duplex)* can tolerate both transient and permanent value faults. As the name indicates, it requires two separate hosts. A request is processed by both replicas. An assertion is applied to the result of the master: if the assertion succeeds, the master sends the reply to the client, otherwise, the assertion is applied to the result of the slave and, if it succeeds, this result is sent to the client.

2.4.3 Underlying (*FT,A,R*) Characteristics

The underlying characteristics of these FTMs, in terms of the parameters (*FT,A,R*) are shown in Table 2.1. We can see that although PBR and LFR tolerate the same fault model, their characteristics in terms of *A* and *R* are substantially different: LFR requires determinism of the application behavior (i.e., identical inputs must produce identical outputs because all replicas process input requests) but does not require state access for recovery. Conversely, PBR does not require determinism of the application behavior (the master propagates its state to the slaves) but imposes state capture and restoration for checkpointing that can be very complex actions. Depending on how checkpoints are built and verified, PBR can be more prone to fault propagation because of state transfer between replicas. In terms of resources, PBR requires more network bandwidth than LFR in normal operation (i.e., in the absence of faults) due to checkpoints while LFR requires more processing than PBR (and, consequently, more energy) with all replicas active, as indicated by ★ symbols in Table 2.1.

Characteristics \ FTM		PBR	LFR	TR	A&Duplex
Fault Model	Crash	✓	✓		✓
	Transient value			✓	✓
	Permanent value				✓
Application Characteristics	Deterministic	✓	✓	✓	✓
	Non-deterministic	✓			✓
	Requires state access	✓		✓	
Resources	Bandwidth	★★	★	n/a	★
	CPU	★	★★	★★	★★

Legend

PBR=Primary-Backup Replication
LFR=Leader-Follower Replication
TR=Time Redundancy
A&Duplex=Assertion&Duplex

Table 2.1: Underlying Characteristics of the Considered FTMs

TR requires state access because application state must be captured before the first request processing and restored between two consecutive executions. As it runs on only one host, it cannot tolerate crash and it has no bandwidth requirements. Assert&Duplex can tolerate both crash faults and value faults as two CPUs are used to run the replicas. Unlike TR, it does not require state access. Both TR and Assert&Duplex require more computation power (i.e., more energy) than PBR because of multiple request processing. However, with Assert&Duplex, the coverage of transient faults detection is lower than when using TR. This illustrates the kind of trade-offs that must be analyzed when choosing an FTM.

Here, we discussed in detail only the subset of FTMs that serves as a basis for our subsequent work. A less detailed view of the underlying characteristics of the entire set of FTMs contained in Figure 2.2 can be found in Table 3 in the Appendix.

2.5 Transitions Between FTMs

2.5.1 Possible Transitions

During the service life of the system, the values of the parameters enumerated in Table 2.1 can change. An application can become non-deterministic because a new version is installed. The fault model can become more complex, e.g., from crash-only it can become crash and value

fault due to hardware aging or physical perturbations. Available resources can also vary, e.g., bandwidth drops or constraints in energy consumption. Figure 2.3 shows a graph of possible transitions between the previously enumerated FTMs. The vertices represent the FTMs and the edges are labeled with the parameter (*FT,A,R*) whose variation triggers the transition. The PBR→LFR transition is triggered by a change in application requirements or in resources, while the PBR→Assert&Duplex transition is triggered by a change in the considered fault model. Transitions can occur in both directions, according to parameter variation. As we further show, there is a subtle difference between a transition and its reverse, in particular to prevent oscillations.

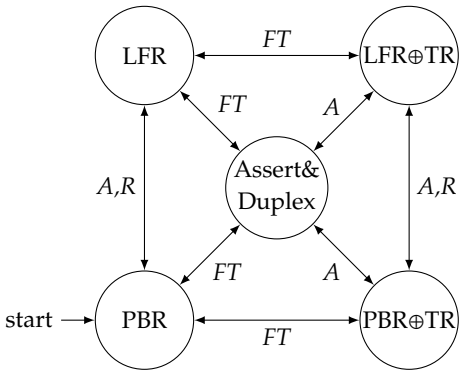


Figure 2.3: Possible Transitions Between FTMs

The fault model can change by becoming more or less complex. Its variation most often requires a *composition of FTMs*, e.g., if we start by tolerating only crash faults with PBR and we want to add tolerance to transient value faults, we will compose PBR with TR and obtain a composed FTM with the features specific to PBR and those specific to TR.

In this work, the ⊕ operator denotes composition of FTMs (e.g., PBR⊕TR in Figure 2.3). Generally, two FTMs are composed in order to tolerate both types of faults that they target individually (i.e., a complex fault model). Using our approach, an FTM can also be composed with another non-functional strategy, e.g., authentication, encryption, logging.

The state machine representation provided in Figure 2.3 is complementary to the tree-graph in Figure 2.2. While the tree can be used for identifying the most appropriate FTM to deploy in the beginning (i.e., which leaf to choose), the state machine illustrates, like the frame of reference associated to the change model in Figure 2.1, the dynamics along the three dimensions occurring during the lifetime of the system and towards which FTM to execute a transition.

2.5.2 Anticipation of Changes

As previously mentioned, dependable systems usually have a long service life consisting in a sequence of operational phases that can be known a priori, at design time, or even not at all and may require a change of FTM, if the values of the (*FT,A,R*) parameters vary. The point now is to distinguish if the situations triggering changes can be anticipated (i.e., if operational phases can be defined a priori or not) and if this has an impact on the dependability of the system.

Changes in Resources (*R*)

The amount of available resources may increase or decrease. When less resources such as network bandwidth, memory space, CPU time, are available, the system manager can search for a suitable FTM requiring less resources in normal operation (i.e., in the absence of faults). If the drop in available resources is too abrupt, the impact can be more acute than a simple undesirable overhead, going as far as impairing either the FTM or the functional layer. Indeed, the application and the FTM might find themselves in competition for resources such as computation power or energy. This is the case of satellite systems, for instance, in which the trade-off between dependability and energy consumption is extremely important, but also changes, according to the operational phase.

Changes in Application Characteristics (*A*)

When an application changes, i.e., a new version with different characteristics *A* (e.g., determinism, state accessibility) is available, the system manager can define a priori a suitable FTM for the new version, taking input from the application designer w.r.t. the new characteristics of the application. Thus, the on-line update encompasses changing the application version together with the FTM. This change can be done using our fine-grained approach to reduce impact on the system architecture and service disruption as much as possible. On-line fine-grained evolutions are of particular interest for dependable large-scale systems, e.g., air traffic control systems, because the overall system must deliver continuous service and cannot be stopped for updating a part of it.

Changes in the Fault Model (*FT*)

This is the most complex case. In the context of operational phases, one can understand that the fault model for a given phase (take-off, landing, cruise, ...) has been anticipated a priori. For critical phases, the fault model is usually stronger than for non-critical ones. For each phase, a given FTM is defined a priori and is instantiated using our differential approach when switching from one phase to the next.

For unanticipated changes, that require the intervention of the system manager, the situation is more problematic because unpredicted faults occurring at some point may not be tolerated by the FTM currently attached to the application. In other words, some operational faults are out of the scope of the fault model that was considered in order to define the FTM for the applications in the system. Unanticipated faults exiting the fault model of the current FTM have an impact on reliability, availability and safety. Improving at least availability by changing the FTM and extending the fault model seems to be a real advance. We can consider that a dependable system has several lines of defense, preventing a fault from propagating too far before being detected. In Chapter 1, the recursive chain of dependability threats was presented. Even if the fault has affected one subsystem that was not equipped at the time with the necessary FTM, applying the appropriate FTM at any point afterwards in the other subsystems can improve detection coverage and error latency.

For instance, let us consider that a duplex strategy was selected beforehand to tolerate crash faults. Transient hardware faults have been ignored because of their low probability of occur-

rence. With hardware aging, the probability of transient fault increases, leading to value faults affecting the output results.

Ideally, in this context, the evolution of the fault model in operation must be addressed in a *proactive* way that performs FTM updates in advance, either because the system is getting to a new operational phase or because of an early detection of fault model changes. If, for example, confidentiality and/or integrity attacks on the system are suddenly observed, cryptography mechanisms can be added or replication strategies can be set in case of denial-of-service attacks.

Based on the analysis of the three cases, we can state that application changes *A* and resource changes *R* can be tackled through *reactive* adaptation. While the former also require input from the system manager regarding new application characteristics (e.g., determinism and state accessibility), the latter can be dealt with by probes measuring the amount of resources and, based on thresholds, triggering transitions between FTMs. On the other hand, fault model changes *FT* require *proactive* adaptation and input from the system manager regarding evolution of the fault model. Proactive adaptation triggers are out of the scope of this work but can be based on error log analysis supplied by various lines of defense (nested assertions, defensive programming, safety bags...).

2.5.3 Detailed Analysis of Transition Scenarios

To illustrate all types of changes requiring transitions between FTMs (*FT,A,R*), we focus on the two duplex strategies (PBR and LFR) and on the mechanisms which tolerate value faults (TR and Assert&Duplex). We analyze in details the bidirectional transitions between PBR and LFR, between LFR and LFR⊕TR (i.e., composition of LFR with TR) and between LFR and Assert&Duplex. As previously shown in Table 2.1 and in Figure 2.3, the transition between PBR and LFR can be triggered either by changes in *A* (i.e., determinism and state accessibility) or in *R* (i.e., bandwidth and CPU), while the transition between LFR and LFR⊕TR and the one between LFR and Assert&Duplex are triggered by changes in *FT* (i.e., the fault model goes from crash to crash and value fault). Figure 2.4 shows a graph of transition scenarios between the FTMs in our subset. We call it a graph of scenarios because there are several events which lead to a transition between FTMs. For instance, we start by deploying PBR, which is appropriate for the system, given the initial values of (*FT,A,R*). When the bandwidth between replicas drops below a given threshold, LFR is chosen over PBR. Later, when a more critical operational phase starts, transient value faults must also be tolerated, so LFR⊕TR is chosen. For the sake of clarity, we present a simplified view of adaptation triggers. In general, they consist in more complex logical expressions.

Mandatory vs. Possible Transitions. As we can see in Figure 2.4, there are three types of transitions: mandatory transitions (continuous red lines), possible transitions (dashed green lines) and intra-FTM transitions (dotted black lines). There are parameters whose variation invalidates the initial FTM or affects its performance and therefore requires a transition towards an appropriate one. These are mandatory transitions. When starting from PBR, there are two such cases: bandwidth drop (which introduces undesired overheads) and state inac-

cessibility (which makes checkpoints impossible). On the other hand, there are parameters whose variation only makes possible the use of another FTM, without invalidating the initial strategy. These are possible transitions. When starting from PBR, there are two such cases: increase in available CPU (because LFR demands more processing than PBR) and application determinism (because both PBR and LFR work in this case). While mandatory transitions can be executed automatically, possible transitions are executed only if the system manager decides to. The intra-FTM transitions are only represented for the sake of completeness. If the system manager decides not to execute a possible transition towards a new FTM (e.g., to go from "PBR with non-determinism" to "LFR with state access" when the application becomes deterministic), the values of the parameters characterizing the current FTM will still change, i.e., an intra-FTM transition is executed (e.g., "PBR with non-determinism" goes to "PBR with determinism" when the application becomes deterministic).

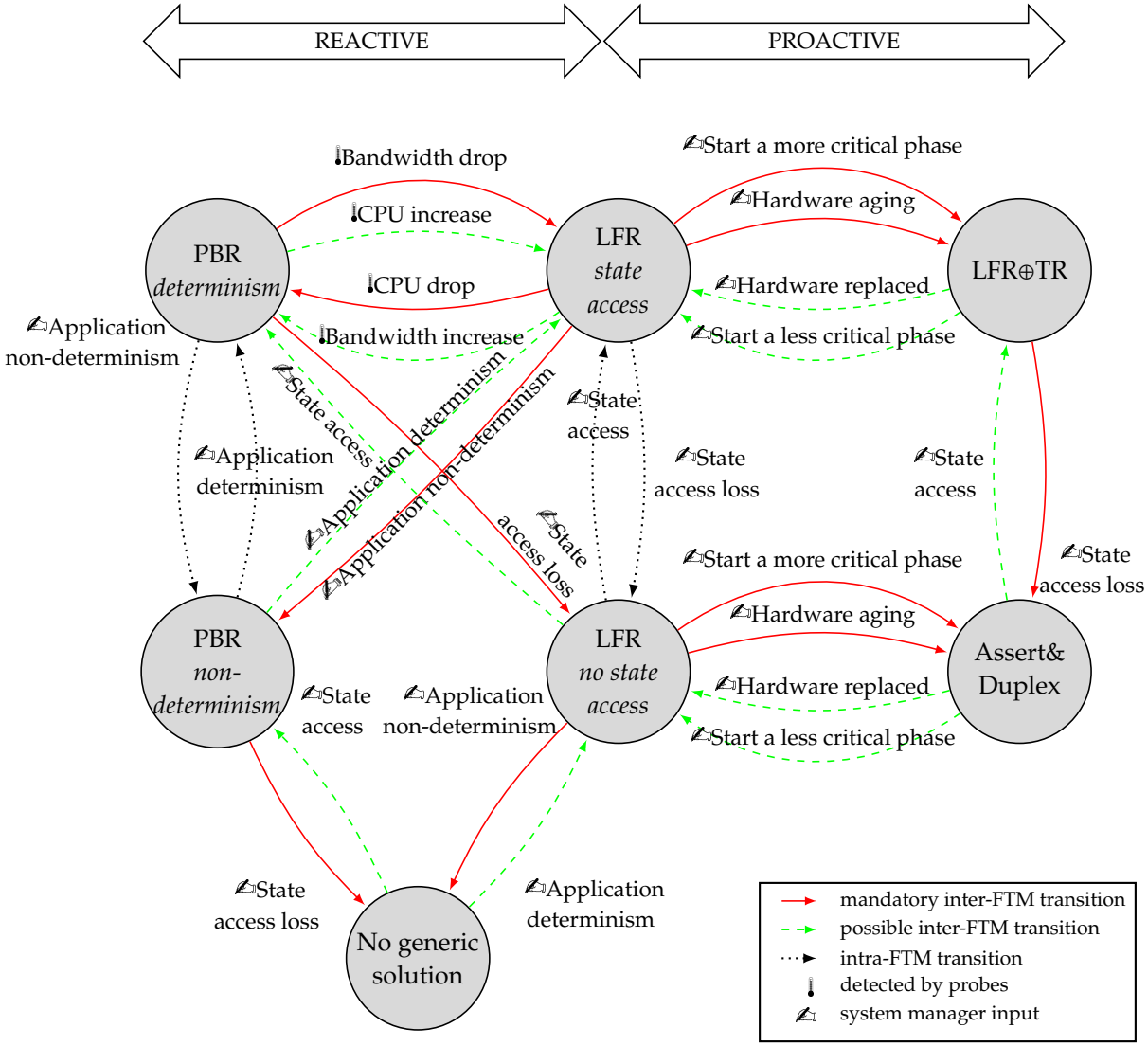


Figure 2.4: Transition Scenarios Between FTMs

Stability of Transitions. A problem which can be encountered in adaptive fault tolerance, perceived as a closed loop system, is the oscillation between FTMs: if a transition is triggered by the variation of a parameter that oscillates near the reconfiguration threshold, the system can become unstable and reconfigure itself too often, thus reducing its availability. By distinguishing between mandatory and possible transitions, this issue is solved: as Figure 2.4 shows, the reverse of a mandatory transition is always a possible one. The risk of oscillation is partially eliminated because once the system executes a mandatory transition due to the variation of a parameter, it will not be able to revert to the previous FTM, unless the system manager decides to. For instance, the system executes a mandatory transition PBR→LFR because the bandwidth drops below a given threshold. After a very short period of time, the bandwidth increases just above the threshold, which makes it possible for the system to revert to PBR. However, if the bandwidth drops again, the system would have to revert to LFR; should such variations occur too often, the system might find itself spending more time reconfiguring itself than in normal operation. This is countered by the system manager who decides not to execute the reverse transition from LFR→PBR based on bandwidth, considering other parameters, e.g., forecasted demands. This approach prevents instability in the case of independent parameter variation. Should the variation of one parameter impact another one (as is the case of bandwidth and CPU), additional measures must be taken for ensuring stability. For instance, an approach used in signal processing and electronics [Moghimi, 00] is to add an amount of hysteresis on triggers, to reduce instability and noise. The definition of threshold values on CPU and bandwidth is out of the scope of this work.

Probe-Triggered vs. Human-Triggered. Figure 2.4 shows another distinction between transitions: the variation of some parameters can be detected automatically by using probes ℹ, while others require input/observations from the application developer or from the system manager ✍. The former encompasses *R* variations, the latter concerns *A* and *FT* variations, as previously discussed.

Reactive vs. Proactive. Last but not least, the reactive/proactive nature of transitions is also visible in Figure 2.4. In Subsection 2.5.2, we discussed the issue of change anticipation. The transition between PBR and LFR is caused either by variations in *A* or in *R*. As such, it can be tackled through reactive adaptation. The transition between LFR and LFR⊕TR and the one between LFR and Assert&Duplex are caused by variations in *FT*, the fault model becoming more complex. Ideally, they should be tackled through proactive adaptation.

2.6 Summary

This chapter focused on the dynamics faced by fault-tolerant systems during their lifetime, which leads to transitions between fault tolerance mechanisms. As dynamics occurs along various axes, first, a frame of reference was established, for visualizing evolution. We defined a set of parameters whose values must be known in order to choose the most appropriate FTM and whose variation may invalidate the initial choice. Then, several well-known FTMs were described and represented, in the context of our frame of reference. Next, various transition scenarios, stemming from the underlying characteristics of the considered FTMs, were analyzed. Last but not least, other aspects connected to transitions, such as adaptation triggers and stability were discussed.

Chapter 3

Design for Adaptation of FTMs

> "We make our world significant by the courage of our questions and by the depth of our answers."
>
> — Carl Sagan, *Cosmos*

3.1 Introduction

Design for adaptation, also called design for change/evolution, is a fundamental issue when designing the architecture of a software system as changes "should not affect the core functionality or key design abstractions, otherwise the system will be hard to maintain and expensive to adapt to changing requirements" ([Buschmann *et al.*, 96, p. 169]). In general, software systems are likely to be subject to change both before deployment (i.e., during design and implementation), because of incomplete or misunderstood client requirements, and after deployment (i.e., during the operational life), due to versioning, user customization etc. This calls for flexible architectures designed with evolution in mind.

In our specific case, the aim is to design fault tolerance mechanisms in such a way as to facilitate their on-line evolution through a minimal set of structural and/or behavioral modifications. The design for adaptation of FTMs is an iterative process representing a preparation step for the actual transitions at runtime. Being performed off-line, this step is part of what we call "cold resilient computing", as opposed to "hot resilient computing", covered in the next chapter.

In this chapter, the process of designing a set of FTMs for facilitating subsequent evolution is presented. The process consists in several design loops and results in the identification of a generic execution scheme of fault tolerance protocols reproducible both in object-oriented (as presented in this chapter) and in component-based (as presented in the next chapter) design and a toolbox of reusable and customizable Fault Tolerance Design Patterns (FTDPs).

3.2 Requirements and Initial Design

The initial requirements set for building the toolbox of FTMs were to develop mechanisms targeting the crash fault model, which can be attached to query-response systems and which preserve the "only-once" semantics. To illustrate our "design for adaptation" approach, we use a simple replication protocol. Following the classification in Figure 2.2, our choice fell on the passive version of duplex protocols, i.e., PBR. Figure 3.1 presents the "big picture" of this FTM. There are three main entities/hosts, the client, the primary and the secondary (backup). The client sends requests to the primary through an unreliable connexion (1). After processing the request, the primary synchronizes its internal state with the secondary by sending a checkpoint (2) through the reliable inter-replica connexion (3). Next, the primary sends the reply to the client (4). In parallel, a pair of monitors/crash detectors periodically query the liveness of their local server (a) and of the partner (b). In case the primary server crashes (either the process or the entire host), the secondary takes over.

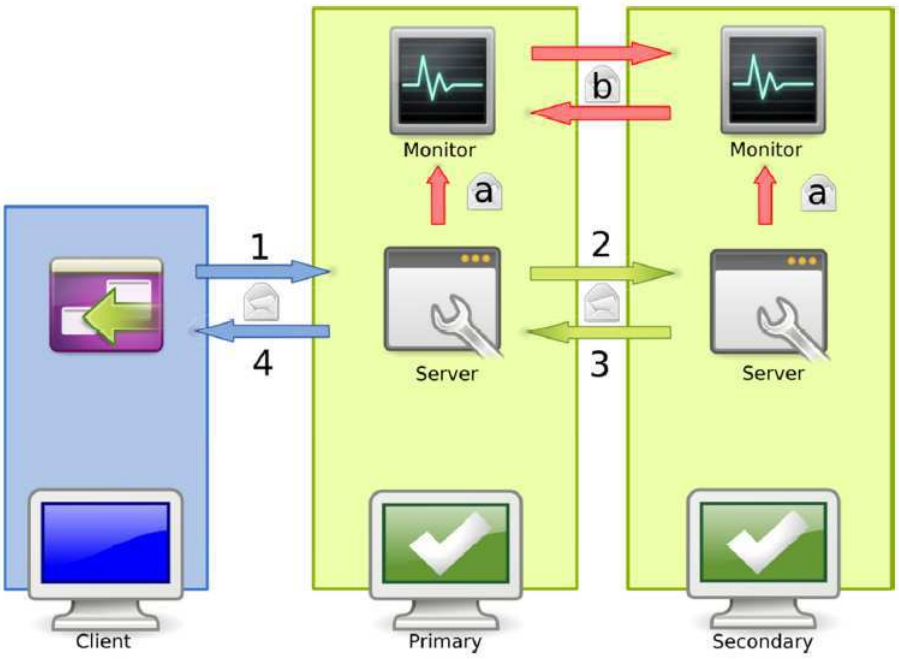


Figure 3.1: Overview of the Primary-Backup Replication

We first developed and validated a prototype both on a simple service (a basic calculator) and on a more complex use case, an application for controlling a Parrot AR.Drone. A full account of this work can be found in [André *et al.*, 11]. IBM Rational Software Architect v8.0 was used for UML design, code generation in C++ and synchronization between design and implementation. The class diagram in Figure 3.2 represents the design of the first prototype. The design consists of several packages: `server`, grouping the elements of the crash-tolerant server, `client`, `communication` and `monitor`. In the following, our focus is on the classes inside the `server` package constituting a crash-tolerant server implementing a set of services required by the client, which can be accessed remotely and which provide methods for state management. The actual inter-replica protocol is concentrated in the `DuplexProtocolPBR`

class encompassing general fault tolerance concerns, duplex concerns and specific PBR concerns. Having as a starting point this ad-hoc design, the aim of the subsequent design loops was to achieve a clean separation between all these concerns in order to easily develop both duplex variants and other FTMs without unnecessary code duplication.

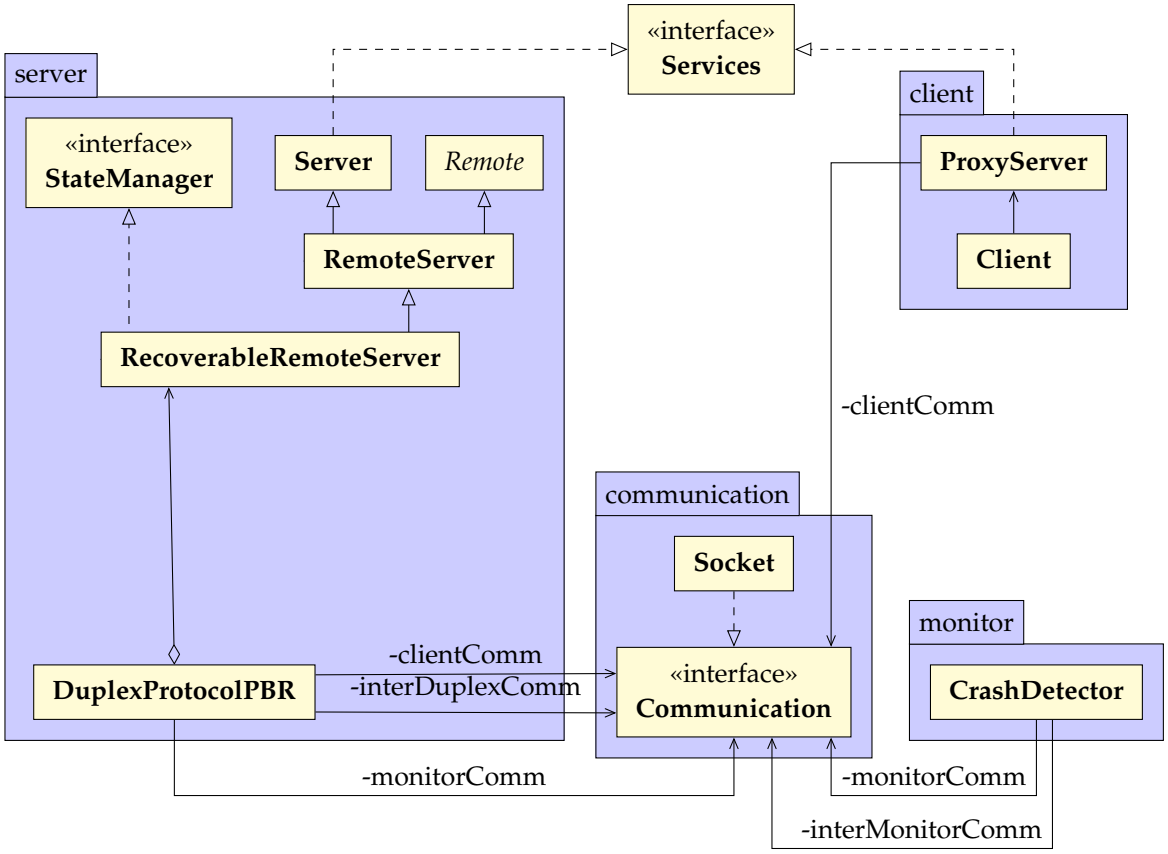


Figure 3.2: Initial Design of the Primary-Backup Replication

3.3 First Design Loop: Generic Protocol Execution Scheme

The analysis of the previously discussed set of FTMs led to the identification of a generic execution scheme. By breaking down an ordinary monolithic processing in separate execution steps (see Figure 3.3), we managed to capture their common parts and their variable features.

This scheme is inspired from Aspect-Oriented Programming [Kiczales *et al.*, 97]. Upon reception of a request from the client, a fault-tolerant server executes some actions *before* processing (e.g., filter inputs, synchronize with a replica). Then it *proceeds* with request processing. *After* processing, it executes some actions (e.g., filter results, apply assertions, synchronize with a replica) and finally it sends the reply to the client. We called this the *before-proceed-after* generic execution scheme of FTMs .

Table 3.1 shows the content of each execution step for the set of FTMs included in our transition scenario. This scheme can be directly applied to the other FTMs included in the classifi-

cation in Figure 2.2, e.g., in the case of N-Version Programming, *before* consists in multicasting the client request to all versions, *proceed* consists in processing the request (by all versions) and *after* consists in collecting the results from all the versions which reply before a certain timeout and voting on them. The steps of this generic execution scheme represent the variable features between FTMs.

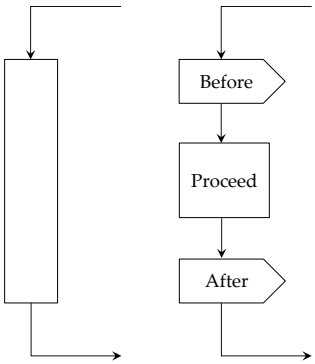


Figure 3.3: Monolithic Processing (left) vs. Generic Protocol Execution Scheme (right)

FTM	Before	Proceed	After
PBR (Primary)	Nothing	Compute	Checkpoint to Backup
PBR (Backup)	Nothing	Nothing	Process checkpoint
LFR (Leader)	Forward request	Compute	Notify Follower
LFR (Follower)	Receive request	Compute	Process notification
TR	Capture state	Compute	Restore state
A&Duplex	Nothing	Compute	Assert output

Table 3.1: Generic Execution Scheme of FTMs

It is worth noting that the behavior of each FTM can be described according to our generic execution scheme in a natural way. Furthermore, this scheme can be used to describe the protocol at different view levels and it can nest itself or be combined with another one following the same scheme.

When designing a duplex mechanism, this scheme can be translated in *sync_before-proceed-sync_after*, because an inter-replica synchronization takes place before request processing and another one after. This synchronization consists in sending/receiving client requests, computation results, checkpoints etc., depending on the protocol. Let us compare PBR and LFR in terms of inter-replica protocol: in the *sync_ before* step, in PBR, the master and the slave do nothing, while in LFR, the master forwards the request to the slave and the slave receives it; in the *proceed* step, in PBR, the master computes the request and the slave does nothing, while in LFR, both master and slave compute; in the *sync_ after* step, both in PBR and in LFR, the master sends some information (checkpoint and notification, respectively) to the slave which processes it.

This generic execution scheme enabled us to factorize in a class what is common to all

duplex protocols, `DuplexProtocol`, and then specialize, through inheritance, the concrete FTMs, `PBR` and `LFR` (see Figure 3.4). Other duplex variants can be added to the framework, either by inheriting from the abstract base class `DuplexProtocol` or from one of the concrete classes, `PBR` or `LFR`.

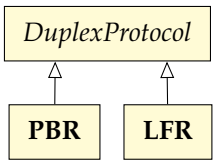


Figure 3.4: First Design Loop

Variable Features. The steps of our generic execution scheme represent the variable features between FTMs. Comparing, for instance, the execution scheme of PBR with that of LFR (see Table 3.1) gives us the intuition that by dividing the inter-replica protocol in isolated bricks/components which can be identified and modified on-line, we could execute a *differential* transition between PBR and LFR. This way, we replace only the components which contain the variable features between the two FTMs, without modifying the rest of the system (e.g., communication with the client, the actual processing to which *proceed* only forwards the requests). By identifying the variable features, we can easily develop FTM variants from existing ones (off-line) and, when using a component-based middleware support (as further shown in the next chapter), execute transitions between FTMs with minimal modifications (on-line).

3.4 Second Design Loop: Externalization of Duplex Concerns

Another separation can be done between what is common to all FTMs and what is specific to duplex ones. Communication with the client, preservation of “only-once" semantics (i.e., each request must be processed only once by the fault-tolerant server, therefore duplicate requests must be detected) and request forwarding to the concrete functional service in the *proceed* step are encapsulated in a class, `FaultToleranceProtocol` in Figure 3.5. This second factorization enabled us to introduce in our framework non-duplex protocols targeting other fault models than crash, more precisely value faults (transient and permanent, see Figure 2.2): `TimeRedundancy` and `Assertion`, which follow the same generic execution scheme, as already shown in Table 3.1.

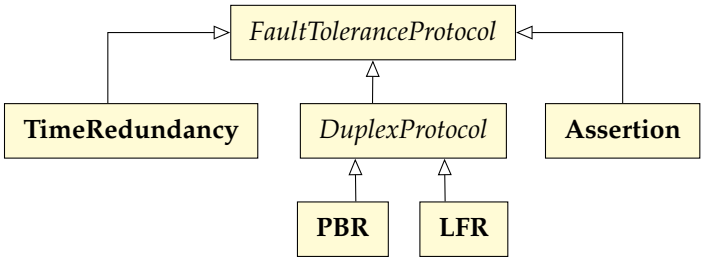


Figure 3.5: Second Design Loop

New protocols can be easily added to the framework either by extending the abstract base class `FaultToleranceProtocol` or one of the concrete classes. This is useful especially in the case of `Assertion`, which is usually application-tailored and can also depend on client requests.

3.4.1 Composing FTMs

As a direct consequence of the two design loops, the composition of FTMs is intuitive and almost immediate. By inheriting from a duplex protocol (tolerating crash faults) and from a value fault tolerance mechanism, we obtain four composed FTMs (Figure 3.6): `PBR_TR` and `LFR_TR`, corresponding to PBR⊕TR and LFR⊕TR respectively in Figure 2.3, and `PBR_A` and `LFR_A` which are two variants of Assert&Duplex. Figure 3.6 shows an excerpt of the final framework of FTDPs resulting from the two design loops, namely the `server` package. The client-side and the crash detection mechanism represented in Figure 3.2 were not modified during the design iterations.

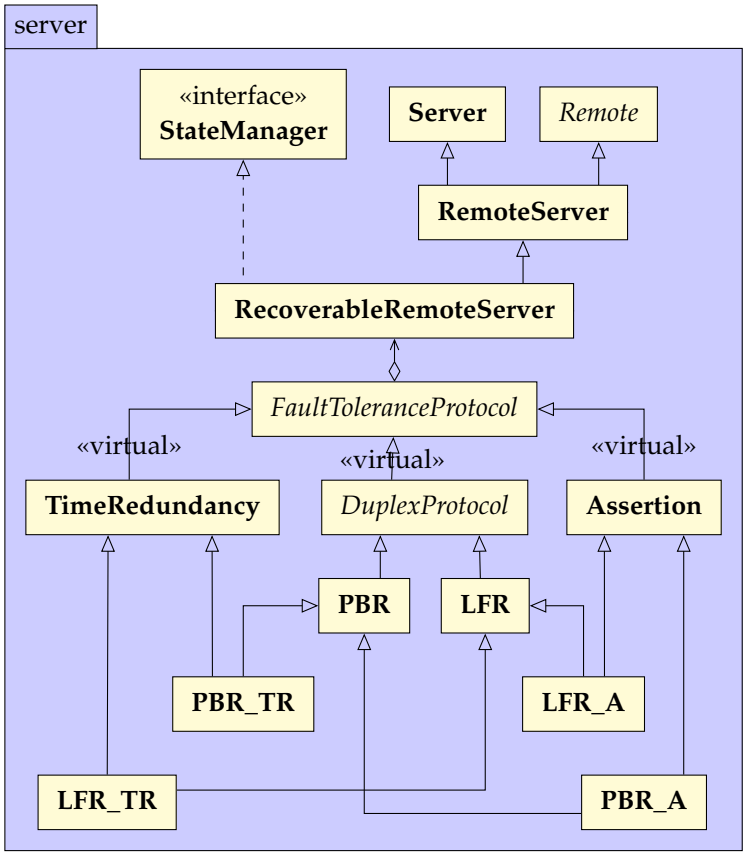


Figure 3.6: Excerpt from FT Design Patterns Toolbox (`server` package)

3.4.2 Fault Tolerance Design Patterns (FTDPs)

A definition of design patterns is provided in [Gamma *et al.*, 95]: "Design patterns capture solutions that have developed and evolved overtime. Hence they aren't the designs people tend

to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form." Similarly, our "design for adaptation of FTMs" approach consists of several design loops, each of which represents a refinement step towards achieving the optimal representation of the various concerns and protocols.

Several design patterns and systems of patterns for fault tolerance exist in the literature among which we have cited a few in the state of the art (see Subsection 1.7.3). Compared to them, the system of patterns we propose describes in greater detail the structural links between the considered fault tolerance mechanisms. Furthermore, to the best of our knowledge, this is the only system of patterns designed as a preparation step for subsequent on-line adaptation.

3.5 Validation & Evalution

All the above design versions have been implemented and validated in C++. The benefits of the "design for adaptation" approach lie in reducing the development time and effort necessary for implementing new FTMs and in increasing readability and separation of concerns/functionalities. It is worth noting that the implementation of composed mechanisms is almost immediate thanks to the two design loops. Their significance is nonetheless powerful: they extend the fault model initially considered (the crash fault model) to crash faults and value faults, making the application more robust to faults which were not taken into account in a first place.

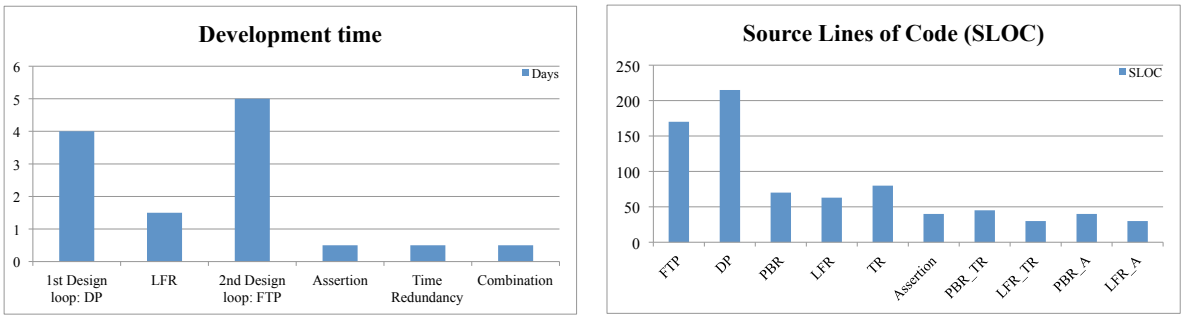


Figure 3.7: Design and Implementation: Development Time and Source Lines of Code

During the development of this FT toolbox, we observed that our design approach was very efficient, in terms of development time of new mechanisms and in terms of code reuse and lines of code to be produced (see Figure 3.7). Our statistics are based on the work done by a pair of junior developers, thoroughly described in [Gibert *et al.*, 12]. The results show that the development of a concrete protocol takes less than a quarter of the time spent on a design loop for adaptation. For instance, while the second design loop took five days, the development of Assertion and Time Redundancy each took half a day. The composition of FTMs, which is probably the most interesting result of this endeavor, only took half a day thanks to the two design loops. In terms of lines of code, as expected, the intensive use of inheritance results in small concrete classes/FTMs compared to the abstract base classes: the ratio is situated between one quarter and one third.

3.6 Summary

In this chapter, we presented the "design for adaptation" approach of fault tolerance mechanisms. The associated toolbox validates the approach, proving that careful design, evolution prediction, modularity and clean separation of concerns, in the large and in the small, enable us to reach a generic protocol execution scheme which is the cornerstone of our framework. The result is interesting in itself as this toolbox can be a starting point for developing an adaptive fault tolerance framework for a real-world application in which new FTMs can be easily developed and integrated off-line. Throughout this iterative process, we gained a thorough understanding of the considered mechanisms and, most importantly, of their variable features. In the context of our overall work, the "design for adaptation" approach is an essential step performed before the actual transitions between FTMs at runtime.

Chapter 4

Component-Based Architecture of FTMs for Adaptation

> "When thought becomes excessively painful, action is the finest remedy."
>
> — Salman Rushdie

4.1 Introduction

In this chapter, we describe in details the practical implementation of component-based adaptive FTMs for addressing resilient computing. This fundamental phase of our work proves its overall feasibility by bringing together the lessons learned from the "design for adaptation" phase presented in Chapter 3 and state-of-the-art software engineering tools.

To begin with, the standards and instruments we used are presented and our choice of tools is explained. Then, we show how these instruments are leveraged, first, for developing component-based stand-alone fault tolerance mechanisms and, second, for executing agile fine-grained on-line transitions between different mechanisms (i.e., while the overall system is in operation).

Next, we discuss a crucial aspect of the transition process, namely the consistency of adaptation, and present the steps we have taken to enforce the tolerance of the adaptation process to faults that might occur during transitions.

4.2 Standards, Tools and Runtime Support

Recent advances in the field of software engineering represent one of the catalysts of this research work. These advances take various forms, ranging from standards and specifications to complex middleware platforms providing a plethora of features. In this section, we present the novelties from software engineering that we have found to be particularly useful for achieving our aim, that of performing fine-grained on-line modifications of the fault tolerance software.

In our choice of tools, we were not primarily guided by the novelty of the proposed solutions because new does not necessarily mean better. We were mainly interested in finding a solution providing the features we had already identified as primordial for our purposes, summarized in what we call "a minimal API for on-line adaptation of fault tolerance mechanisms" that is further detailed. A second criterion was the presence of a strong support for these tools from their developers and the possibility to interact with them and give/receive feedback on our particular use of the chosen platforms.

4.2.1 The SCA Standard

The Service Component Architecture (SCA) [Chappell, 07, Marino and Rowley, 09] is a set of specifications for building and composing loosely-coupled, tailorable distributed applications based on the Service-Oriented Architecture (SOA) and Component-Based Software Engineering (CBSE) principles and encompassing a wide range of technologies. The specifications are hosted in the Open CSA section of the OASIS consortium which includes partners such as IBM, Oracle, Red Hat, SAP and TIBCO (`http://www.oasis-opencsa.org`).

While SOA [Margolis and Sharpe, 07] specifies a way of organizing software by exposing coarse-grained, loosely-coupled services which can be accessed remotely, it fails in specifying how these services should be implemented. SCA addresses this gap by specifying a language-independent programming model and promoting the use of components and architecture descriptors. SCA is not the only solution in the SOA world combining software components and services, another one being OSGi [OSG], that has received significant adoption. Although the two technologies do share a lot of similarities, what makes them essentially different is that OSGi is purely Java-based while SCA preaches programming language-agnosticism [Marino and Rowley, 09] for implementing components and defines a common assembly mechanism for combining components into applications.

The main idea behind SCA is that applications have a hierarchical structure based on *components* which can be included in *composite* components. Both components and composites can expose *properties*, i.e., configuration parameters represented as name-value pairs read at runtime. Components provide and require *services*. Required services are called *references* and they are connected to the actual service provider through *wires* (in case of local connections) or *bindings* (in case of remote connections). References and services owned by a component can be *promoted* at the level of the composite enclosing it. Composites can also act as components inside bigger composites (i.e., a component can either be a *primitive* component or a composite component). Figure 4.1 illustrates these notions. Components and composites are deployed and contained within a larger construct called *domain*.

By separating the interfaces (services and references) from the actual implementa-

tion [Szyperski, 02], this approach facilitates reuse, evolution and technology-agnosticism as components consume services provided by other components without being aware of how they are implemented and whether their implementation changes over time. The novel programming model of SCA enables developers to build distributed applications more easily. By abstracting away the intricacies of distributed computing and the specifics of underlying communication technologies, it allows them to focus on business logic and on structuring applications in well-cut reusable and customizable bricks.

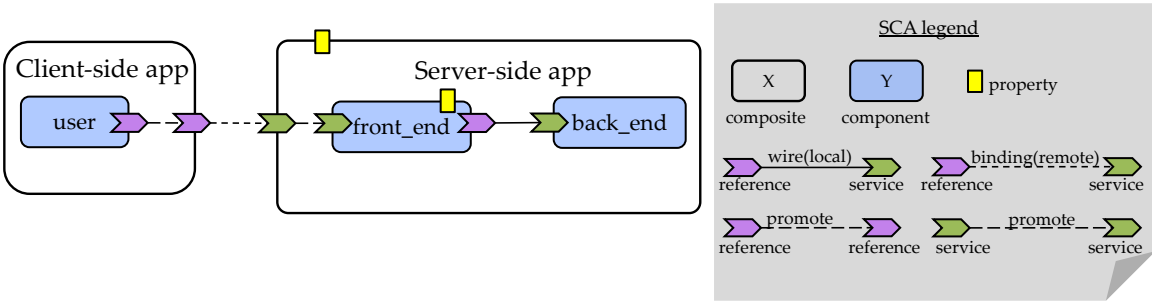


Figure 4.1: A Simple SCA-Based Application

To be more specific, the SCA standard is based on four sets of specifications [Chappell, 07]:

- *assembly language*: components are assembled in composites using an XML-based Architecture Description Language (ADL) called Service Component Definition Language (SCDL, commonly pronounced "skiddle"). This format makes use of the previously introduced concepts of service, reference, wire etc.;
- *component implementations*: these specifications define how services are implemented;
- *binding specifications* define how SCA services are accessed, both between two SCA-based applications and between an SCA-based application and any other service-oriented technology. Services and references are bound to a specific communication protocol (e.g., SOAP for Web Services, Java RMI etc.);
- *policy sets/intents*: non-functional features such as confidentiality and transactions must be provided by SCA platforms in order to separate application code from transport and communication protocol issues.

It is worth noting that all these specifications can be extended according to user needs and emerging technologies, e.g., one can specify how to implement components in a new (possibly proprietary) programming language or add policy sets for new non-functional requirements. All the previously introduced SCA notions are thoroughly explained in Section 4.3.

4.2.2 FraSCAti

As the SCA standard only provides a set of specifications, applications complying to them require an appropriate runtime support for deployment and execution. Several independent

SCA platforms exist, both commercial (e.g., IBM WEBSPHERE Application Server Feature Pack for SOA, Oracle Event-driven Architecture Suite) and open-source (e.g., Apache Tuscany, Fabric3, Service Conduit, and FRASCATI). Although all the above-mentioned commercial and open-source implementations have strong support from their communities and are continuously upgraded (which is to be expected in the case of commercial products), there are some differences between them. While the first three are better documented and more business-oriented (i.e., Apache Tuscany is thoroughly explained in [Laws *et al.*, 11], all code examples in [Marino and Rowley, 09] are based on Fabric3, and Service Conduit advertises itself as "the leading open source SCA implementation" on the official website), FRASCATI stems from an academic community and provides several key features for our work which are not present in the other platforms, neither in the open-source nor in the commercial ones. More precisely, these extra-features represent the previously mentioned "minimal API for on-line adaptation of fault tolerance mechanisms" which is further explained.

We developed our adaptive FTMs on FRASCATI (version 1.4 and 1.5) [Seinturier *et al.*, 11], an open-source platform providing runtime support for SCA-based applications and implemented according to SCA principles. The FRASCATI runtime provides support for SCA composite definitions following the SCA Assembly Model V1.0 specification, Java component implementation (SCA Java Component Implementation V1.0 and SCA Java Common Annotations and APIs V1.0), remote component bindings using Web Services (Soap or RESTful) and Java RMI protocols.

Unlike the other above-mentioned platforms, FRASCATI goes beyond the SCA specification as its creators have also identified several shortcomings of the original standard and addressed them as well. According to [Seinturier *et al.*, 11], a fundamental omission of the SCA standard is that it does not address runtime management of the application or of the underlying platform, as the assembly of components is only used to instantiate and initialize the application. Given that our adaptive fault tolerance mechanisms essentially depend on the runtime reconfiguration support provided by the underlying platform, a mere implementation of the SCA standard would be insufficient for our needs. FRASCATI addresses runtime reconfiguration of component-based architectures thanks to the integration of FRACTAL-based reflective computing capabilities [Bruneton *et al.*, 06]. FRACTAL is a programming language-independent component model designed for building highly configurable software systems. As a result of this integration, components inside SCA-based applications running on top of FRASCATI are also endowed with FRACTAL-specific *controllers* for life cycle management, for introspecting and reconfiguring the component-based architecture at runtime.

4.2.3 Runtime Reconfiguration Support

FRASCATI brings to SCA reflective and on-line reconfiguration capabilities. On-line architecture exploration and reconfiguration can be performed in several ways:

- through FRASCATI Explorer, which is a graphical management tool thanks to which users can interactively load and start composites, invoke methods, explore and reconfigure the component-based architecture of a running SCA application (see Figure 4.2);
- by using the FRACTAL control API in Java;

- by executing reconfiguration scripts written in FSCRIPT.

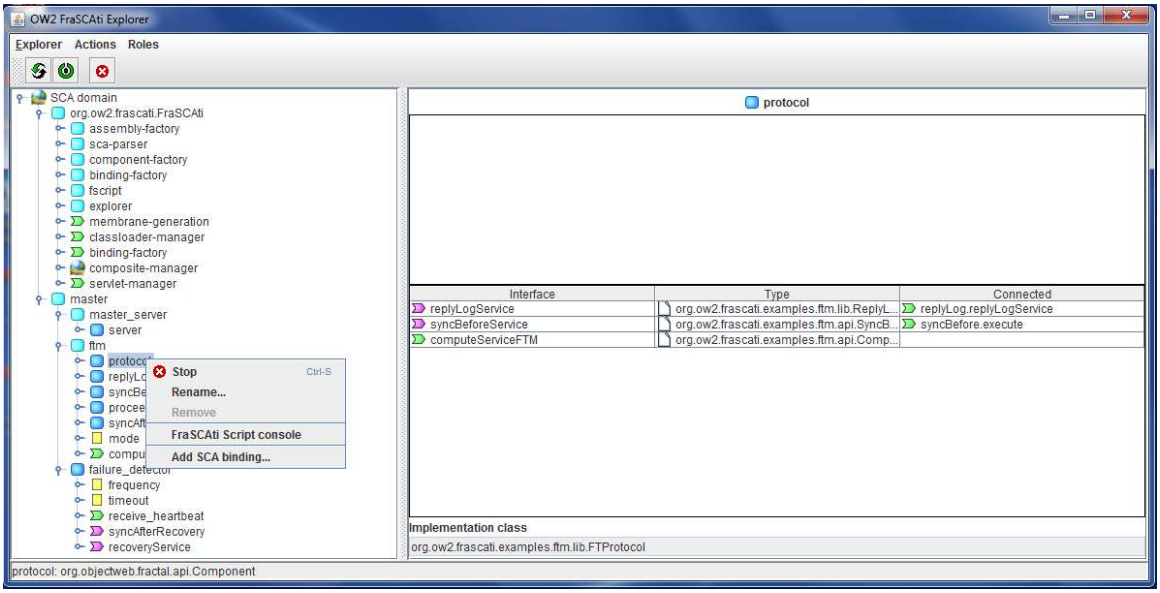


Figure 4.2: FRASCATI Explorer Showing Reconfiguration Features and the Architecture of the Platform and of the Running Application

For our purposes, the most convenient option is using FPATH and FSCRIPT [David *et al.*, 09] as they allow for a much more concise writing of reconfiguration actions than the basic FRACTAL API and also present some key additional properties. FPATH is a query language for introspecting and navigating inside FRACTAL-based architectures at runtime. Dedicated to modifying component-based applications, the FSCRIPT scripting language provides support for safe transactional changes [Léger *et al.*, 10], thus guaranteeing that a reconfiguration either takes a component-based architecture to a consistent state or leaves it unmodified in its initial state, should an exception be thrown during reconfiguration. This is an essential property for our transitions as the addition of dynamics to FTMs should not impair the reliability of the mechanisms. The consistency of the overall distributed reconfiguration process, which is further discussed, relies on the Atomicity, Consistency, Isolation, Durability (ACID) properties guaranteed by FSCRIPT.

4.2.4 Runtime Support Requirements for On-line Adaptation of FTMs

Before choosing the appropriate tools, we identified a set of features which must be provided by the underlying platform in order for it to become a candidate for our resilient computing framework. These features are all related to the degree of observability and control that the platform provides over the component-based architecture of applications at runtime:

- *control* over components' *life cycle* at runtime (add, remove, start, stop);
- *control* over interactions between components at runtime, for creating or removing bindings.

Furthermore, to ensure consistency before, during and after reconfiguration, several issues must be carefully considered:

- components must be stopped in a *quiescent* state, i.e. when all internal processing has finished;
- incoming requests on stopped components must be buffered, to ensure consistency of request processing.

This set of elements represents a minimal API and set of features which must be provided by a component-based middleware to make it a candidate for our practical experience. We chose FRASCATI (together with its FSCRIPT interpreter) because it provides these features, this being the first criterion for choosing a particular middleware support. FRASCATI also fills the second criterion, namely a strong support from its developers, as it is continuously upgraded with new features. Our direct interaction with its designers and developers, members of the ADAM research project-team from INRIA Lille, in the context of the "Software Engineering for Resilient Ubiquitous Systems" Collaborative Research Action (ARC SERUS), represented a significant advantage in the process of developing our adaptive FT middleware. It is worth noting that our approach is reproducible on any other runtime platform providing a minimal implementation of the adaptation API we have identified.

Unlike other SCA platforms, the FRASCATI middleware is itself developed according to SCA principles [Seinturier *et al.*, 11] and is a Software Product Line (SPL) enabling users to build configurable runtime platforms compliant with their requirements and their targeted systems (i.e., the memory footprint ranges from 256 KB for the minimal configuration to 25 MB for the full configuration). This way, we can choose among the modules of FRASCATI a minimal subset providing the required functionalities (e.g., FRASCATI provides support for several remote component binding technologies such as Java RMI, REST, Web Services etc. but as we only need one of them, we plug only that specific module in our version). The SCA-based components constituting FRASCATI itself are also endowed with reflective FRACTAL-based capabilities and as such can be explored and reconfigured at runtime. The component-based structure of the platform can be seen in Figure 4.2 which shows both our application (the `master` composite and everything underneath it) and the component-based middleware (the `org.ow2.frascati.FraSCAti` composite and its contents).

4.3 Component-Based Architecture of PBR for Adaptation

In the following, we describe in details a proof-of-concept consisting in the component-based architecture of a simple PBR strategy [Stoicescu *et al.*, 12a]. This proof-of-concept results from the implementation of our *before-proceed-after* generic execution scheme on FRASCATI according to the SCA specifications. The strategy is applied to client/server systems based on request-response interactions. The inter-replica communication channel is reliable while the client communication channel is supposed unreliable in this example. The client can issue a request up to three times, in case it does not receive a reply, and the FTM guarantees that requests are processed only once. For the sake of clarity, we consider in this work a single-threaded client/server application.

4.3.1 Separation of Concerns

Figure 4.3 shows an SCA diagram representing the component-based design of Primary-Backup Replication. We name the two replicas `master` and `slave` rather than primary and backup in order to have a uniform naming convention when passing to another duplex protocol, e.g. Leader-Follower Replication. Our architecture consists of three composite components and their interactions: the `client`, the `master` processing the requests and the `slave` processing the checkpoints sent by the master.

The design in Figure 4.3 emphasizes the separation of concerns between the functional layer of the application and the non-functional one. The former implements the functional service (`server` component), the aim being to trigger the execution of the FTM protocol. The application component is wrapped in an independent composite (`master_server` and `slave_server` composite, respectively) in order to maintain the clean separation of cross-cutting concerns should application complexity increase (e.g., the application could consist in a front end and a back end, like in Figure 4.1). The non-functional layer is further broken down into an inter-replica protocol (here, the FTM composite corresponding to PBR) and the crash detection mechanism, common to all duplex strategies, (`crash detector master` and `crash detector slave`, respectively) through which the replicas monitor each other's liveness and perform recovery actions in case a crash occurs.

4.3.2 SCA Entities

The component-based architecture of PBR in Figure 4.3 illustrates all SCA notions previously introduced:

- Components: bricks of business logic which can either be primitive components implemented in Java (e.g., `protocol`, `replyLog`) or in FSCRIPT (`recovery`) or composite components (e.g., `FTM` is a composite component inside the `master` composite);
- Composites: basic units of deployment, containing one or more components, e.g., `FTM`, `master`;
- Services: components implement interfaces representing the services they provide, e.g., `server` implements `ComputeService` (a Java interface) and provides a service named `compute` in our example;
- References: relations such as composition, association, aggregation are transformed in reference-service connections, e.g., `protocol` has an attribute of type `ReplyLogService`, an interface implemented by `replyLog`;
- Properties: configurable attributes of components or composites, e.g., `crash detector slave` has a property specifying the *timeout* for the watchdog mechanism, which can be configured according to network latency and a property specifying the *frequency* at which it checks the master's liveness, which can also be configured according to network resources;

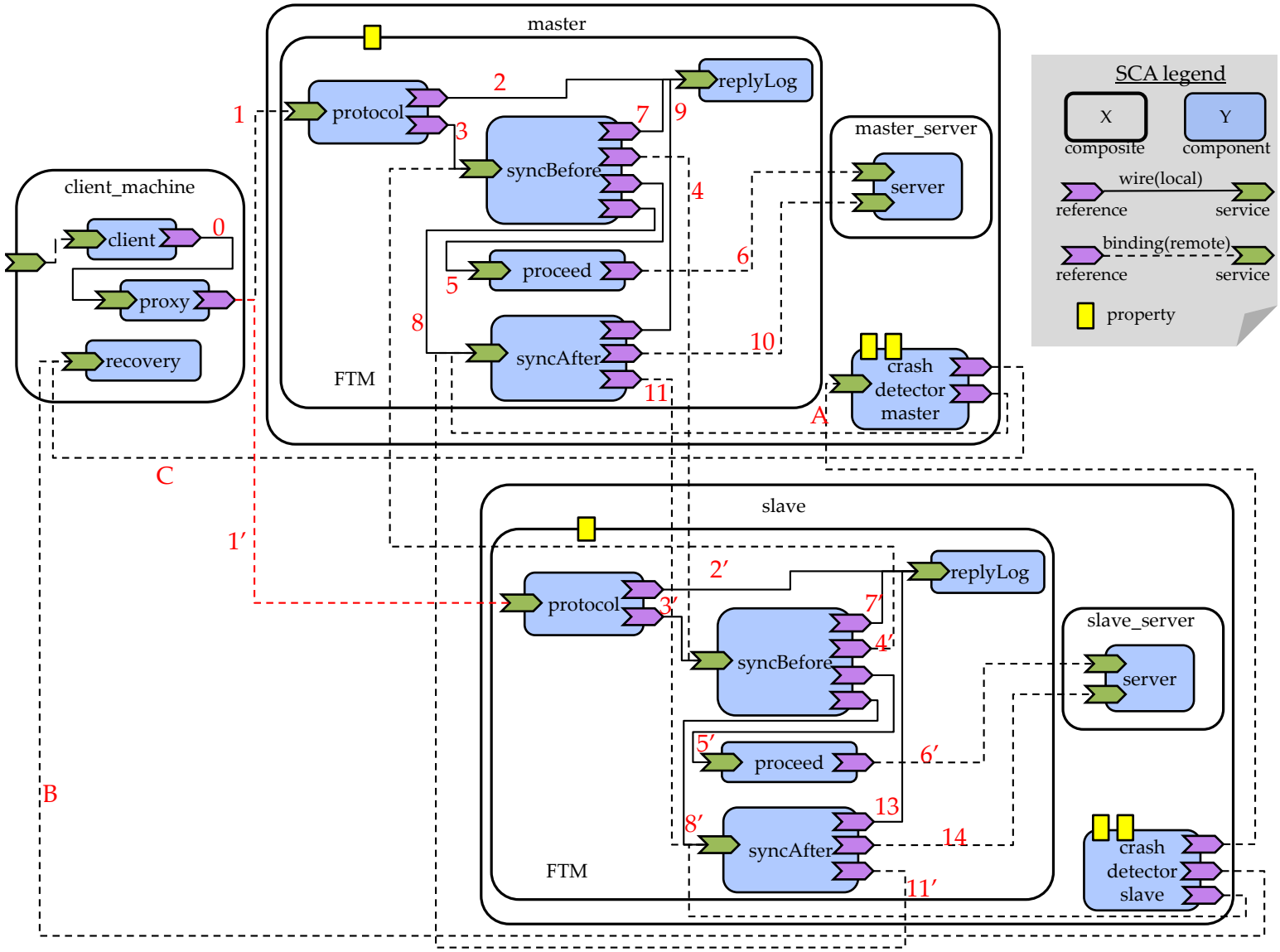


Figure 4.3: Component-Based Architecture of Primary-Backup Replication

- Wires and bindings connecting references to services, e.g., `protocol`'s reference of type `ReplyLogService` is connected to the service provided by `replyLog` through a wire, while `syncBefore` components (master-side and slave-side) are connected through a pair of bindings.

4.3.3 PBR in Action

The component-based implementation of PBR represented in Figure 4.3 operates in the following manner:

- the `client` component sends a request to the `proxy` (`wire` 0);
- the `proxy` component adds an identifier to the request and sends it on `binding` 1 from which it is received by the `protocol` component;
- the `protocol` component checks in `replyLog` (`wire` 2) whether it is a duplicate request: if so, it sends directly the stored reply, otherwise, it sends the request to `syncBefore` on `wire` 3;
- `syncBefore` first synchronizes with the other replica on `binding` 4 (in PBR there is no synchronization before the actual request processing);
- `syncBefore` sends the request for processing to `proceed` on `wire` 5;
- `proceed` calls the actual service provided by the `server` on `binding` 6 and the result is sent back to `syncBefore`;
- `syncBefore` writes the result in `replyLog` (`wire` 7) and then calls `syncAfter` (`wire` 8);
- `syncAfter` gets the last result and request identifier from the `replyLog` (`wire` 9), captures the server state by calling the state management service provided by the `server` (`binding` 10) and builds a checkpoint based on this information which it sends to `syncAfter` of the other replica (`binding` 11);
- `protocol` gets the result from the `replyLog` (`wire` 2) and sends it to the `client`;
- on the other replica, `syncAfter` writes the last result and the last request identifier in `replyLog` (`wire` 13) and maps the new server state by calling the state management service provided by the `server` (`binding` 14).

In parallel with request processing, `crash detector slave` periodically calls the `crash detector master` (`binding` A) to check its liveness. If the master does not get a call before the time interval (set by the previously mentioned *timeout* property) expires, it concludes the slave is dead and becomes master-alone, therefore it performs only request processing and no synchronization (i.e., no call on `bindings` 4 and 11). The slave can come back after the crash and start its periodic calls again, in which case master-alone becomes master and it restarts synchronizing at each request processing.

If the master does not reply to the slave before the timeout expires, the slave concludes it is dead and performs recovery by calling the `recovery` component on `client_machine` (`binding` B). This is a component written in FSCRIPT which removes `binding` 1 to the old master. Next, it creates `binding` $1'$ to the new master. From now on, all client requests are sent directly to the new master-alone (ex-slave), i.e., the server which simply processes client requests and does not have a partner to synchronize with. The master can come back after having been repaired and its presence is detected by the new master-alone (thanks to the periodic calls on `binding` A). In order not to disrupt the client (who now sends all requests to the old slave), the old master becomes slave and the new master-alone becomes master and synchronizes its state with the other replica at each request processing (i.e., on `bindings` $4'$ and $11'$). Should the new master crash, the partner detects this and performs recovery by calling the `recovery` component on `client_machine` (`binding` C). This time, `binding` $1'$ is removed and `binding` 1 is created towards the new master-alone.

This is a classic activity of Primary-Backup Replication. The novelty lies in implementing it according to the SCA programming model. This is a challenging exercise *per se* as the focus shifts from objects (as in object-oriented programming) to services. Although interesting, this exercise would be futile if it was not an essential preparing step before the fine-grained runtime reconfiguration enabling us to perform a transition towards a new mechanism.

4.3.4 Component State Management

Component state management is an important issue when developing FTMs as one of the roles of fault tolerance is to ensure that application state is not lost due to faults. The designer must decide how it is going to be performed. Several options can be envisaged, among which we cite two:

- stateful components can explicitly provide a state management service (in the form of a Java interface with `getState()` and `setState()` methods, implemented by application developers) which is called when needed;
- the designer can create a `@State` annotation and place it on the attributes representing component state and develop the corresponding component controller which interprets this annotation.

When we started developing the adaptive FT middleware, using FRASCATI v1.4, we chose the first option. Recently, in FRASCATI v1.5, a State controller has been introduced (the second option), that we plan to integrate in our mechanisms in the near future. In our current implementation, the `server` component in Figure 4.3 provides two services: the functional service (i.e., the application) and a state management service enabling us to capture and restore state for protocols demanding it, i.e., PBR and TR.

Components inside the actual FTM containing the variable features (i.e., `syncBefore, proceed, syncAfter` in Figure 4.3), which are subject to modification during transitions, are stateless, as a result of our thorough “design for adaptation" approach. The state of the FTM consists of information stored in the `replyLog` component and the state of the application, accessible through the state management interface of the `server`. These components are

unaffected by transitions. Should stateful components be manipulated on-line, it is the responsibility of the FTM developer to provide mapping functions to initialize new components from the state of the old ones. This can be done by one of the above mentioned options.

4.3.5 From Objects to Components: Design Choices

The "design for adaptation" approach previously presented provides a "differential knowledge" of the considered FTMs, by emphasizing their variable features. The system of patterns for fault tolerance described in Chapter 3 is the result of a process consisting of several design loops. Similarly, the current component-based architecture of PBR (and of the other FTMs) is the result of several design iterations. The generic *before-proceed-after* execution scheme guided us in breaking the FTM composites in (at least) three components isolating the variable features. However, some issues still needed to be considered, e.g., how to interconnect these components, how to preserve the state of FTMs in components which are not subject to modification during transitions in order to facilitate reconfiguration. The transition from an object-oriented design to a component-based architecture is not an automatic process, neither in terms of entities nor in terms of interactions between them. However, this paradigm shift from objects to services as first-class entities is a very useful exercise thanks to which we can benefit from the runtime control capabilities provided by the platform. During the process of mapping our object-oriented design to components [Stoicescu *et al.*, 12b], we observed the following:

- Relationships such as composition, association and aggregation are usually translated into a reference-service interaction;
- Translating inheritance usually needs to be tackled on a case-by-case basis;
- In some cases, 1 object maps to 1 component, e.g. FaultToleranceProtocol to `protocol`;
- In other cases, 1 method maps to 1 component, e.g. sync_before() to `syncBefore`.

Components containing variable features may be subject to modification at runtime, according to the reconfiguration required by changes in the environment. As such, they should store as little state as possible in order to reduce the number of actions to perform during a transition not only for facilitating the task but, more importantly, for reducing the window of time during which the system is executing a transition. The longer it takes to perform transitions, the longer service delivery is disrupted because the system parts affected by reconfiguration must be stopped in a quiescent state. In our component-based architecture, we chose to isolate the variable features (*before, proceed, after*) in stateless components and have them store and extract state information from a component which is not subject to modification during transitions, namely `replyLog`. This component is similar to a stable storage (stable only at runtime because in case of crash, the information is lost). Let us consider that the fault-tolerant application works for some time in a PBR configuration. When a transition towards LFR is required, components representing variable features first need to finish their internal processing, before being stopped. This means finishing processing the current request and writing the necessary state information in `replyLog`. Then, the components are replaced and if a client request arrives during reconfiguration, it is buffered. Upon completion of the transition, the

newly introduced stateless components are started and the new request can be processed. For this, the new components only need to access data in the `replyLog`.

Component-based architectures provide a great degree of flexibility if they are carefully designed and can open new perspectives for adaptation scenarios not yet considered. As shown in Figure 4.3, the `protocol` component, which is not subject to change during transitions, only delegates request processing to `syncBefore`. The latter is in charge of orchestrating the *before-proceed-after* execution logic: first, it synchronizes with the other replica (`wire` 4 in Figure 4.3), next it calls `proceed` (`wire` 5), it writes the result in `replyLog` (`wire` 7) and then calls `syncAfter` (`wire` 8). This execution logic could have been wired otherwise. One option would have been to wire all components containing variable features to the `protocol` component (that is not subject to modification during transitions as its single role is to ensure the only-once semantics). Design decisions have a strong impact on the usability and flexibility of the proposed solution and can become an impediment to adaptation. This particular component-based design has proved to be flexible enough to enable all previously discussed transition scenarios to be implemented.

4.3.6 Developing the Pieces and Putting Them Together

The SCA standard advocates technology-agnosticism, both regarding the programming language used for implementing components and the communication protocol. For dealing with the heterogeneity of these two areas, it provides the assembly language meant to "glue" components together.

The architecture of PBR in Figure 4.3 consists of a set of interconnected components written all in Java, except for `recovery` on `client_machine` that is written in FSCRIPT. Below, we present the implementation of `protocol`. Components developed in FSCRIPT are thoroughly illustrated in the reconfiguration process.

As illustrated in Figure 4.3, `protocol` provides one service and requires two services, i.e., has two references (see Listing 4.2). The service is represented by the Java interface that it implements, `ComputeServiceFTM`, illustrated in Listing 4.1. Given that this service is called remotely (by the client), it requires a binding of a specific type supported by FRASCATI (e.g., Java RMI, JSON-RPC, Web Services etc.). In all our implementation, we use bindings of type Java REST because FSCRIPT provides support for adding and removing bindings of this type at runtime. The other binding type supported by FSCRIPT is Web Services. The remaining binding types available in FRASCATI can be interactively modified at runtime through the FRASCATI Explorer GUI (see Figure 4.2). The @POST annotation on the *compute* method is specific to the use of Java REST for remote method invocation.

Listing 4.1: Java Interface Representing an SCA Service

```
@Service
public interface ComputeServiceFTM {
    @POST
    double compute(ClientRequest req);
}
```

Listing 4.2: Implementation of an SCA-Based Component in Java

```
@EagerInit
@Service (value=ComputeServiceFTM.class)
@Scope ("COMPOSITE")
public class FTProtocol implements ComputeServiceFTM {
   private SyncBeforeService syncBefore;
   private ReplyLogService rLog;

   @Reference (name="syncBeforeService")
   public void setSyncBeforeService (SyncBeforeService s) {
      this.syncBefore = s;
   }

   @Reference (name="replyLogService")
   public void setReplyLogservice(ReplyLogService s) {
      this.rLog = s;
   }

   public FTProtocol() {}

   public double compute(ClientRequest req) {
      int id = req.getId();
      double logReply = rLog.getReply(req.getId());
      double result = 0.0;
      if (Double.isNaN(logReply)) {
         this.syncBefore.triggerBefore(req);
         result = this.rLog.getReply(id);
      }
      else {
         System.out.println("Request␣id" + id + "␣already␣completed");
         result = logReply;
      }
      return result;
   }
}
```

Coming back to Listing 4.2, we can see the two references, one to `SyncBeforeService` and one to `ReplyLogService`. Upon reception of a client request, the component verifies if the request has already been processed. If so, it gets directly the result (from `replyLog`), if not, it calls the `SyncBeforeService`, which starts the whole *before-proceed-after* execution chain. At the end of the processing, `proceed` gets the result from `replyLog`. The @Scope annotation is specific to SCA and its parameter indicates to the SCA runtime how to manage instances of a component. In our case, there is one instance per composite which is used across all service calls. The default value is 'STATELESS', meaning there is one instance created per service call. The @EagerInit annotation tells the SCA runtime to override the default lazy initialization of components (upon service calls).

Composites represent the basic unit of deployment. Each composite in Figure 4.3 (i.e., `client_machine, ftm, master_server, slave_server, master, slave`) is described through a composite file written in SCDL, an XML-based language. Listing 4.3 shows an excerpt from *ftm.composite* corresponding to the `protocol` component, whose implementation in Java has already been discussed, and the `replyLog` component. We can see that the composite named `ftm` promotes the service provided by `protocol` and has a REST binding attached to it for remote invocation.

Listing 4.3: Excerpt from the *ftm.composite* File Definition

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
   xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1"
   xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
   xmlns:xs="http://www.w3.org/2001/XMLSchema" name="ftm"
   targetNamespace="http://frascati.ow2.org/lfr-rest">
   <property name="mode" type="xs:int">0</property>

   <service name="computeServiceFTM" promote="protocol/computeServiceFTM">
      <frascati:binding.rest uri="http://localhost:8080/ComputeServiceFT" />
   </service>
   <component name="protocol">
      <implementation.java class="org.ow2.frascati.examples.ftm.lib.FTProtocol"/>
      <service name="computeServiceFTM">
         <interface.java interface="org.ow2.frascati.examples.ftm.api.ComputeServiceFTM"/>
      </service>

      <reference name="syncBeforeService">
         <interface.java interface="org.ow2.frascati.examples.ftm.api.SyncBeforeService"/>
      </reference>

      <reference name="replyLogService">
         <interface.java interface="org.ow2.frascati.examples.ftm.lib.ReplyLogService"/>
      </reference>
   </component>

   <component name="replyLog">
      <implementation.java class="org.ow2.frascati.examples.ftm.lib.ReplyLog"/>
      <service name="replyLogService">
         <interface.java interface="org.ow2.frascati.examples.ftm.lib.ReplyLogService"/>
      </service>
   </component>
   <wire source="protocol/replyLogService" target="replyLog/replyLogService" />
</composite>
```

For each component, we must declare the Java class (or the FScript file) implementing it. Services and references are also declared, with their corresponding Java interfaces. In the end, we see an explicit wire connecting the reference of `protocol` to the service provided by `replyLog`. We have omitted the rest of components for the sake of clarity.

4.4 Transition Process

Our differential approach leverages the reflective component-based support provided by FRASCATI. Having decomposed our FTMs in fine-grained bricks, we tackle both reactive and proactive adaptation by replacing only components containing variable features.

Figure 4.4 represents the "big picture" of the adaptation process, consisting of several actors/entities and their interactions:

- the **System** runs a fault-tolerant application deployed on its specific hardware support;
- the **System manager** *observes* the system for detecting changes in (*FT,A,R*) parameters. The manager then *triggers* the Fault Tolerance Adaptation Controller through a secure channel using authentication, giving it as input the new FTM towards which a transition

must be executed. Changes in *R*, detected by probes, can automatically trigger transitions. Detecting changes in *FT and A* requires input from system developers or from the specifications of operational phases, as already shown in Figure 2.4;

- the **Fault Tolerance Adaptation Controller** then *accesses* the FTM&Adaptation repository to get the specific transition package (whose content is further detailed) corresponding to the transition from the current FTM to the new one, determined by the system manager; upon reception of the package (if it exists), the controller deploys the package (this process is further detailed) and *calls* the script interpreter, giving it as input the transition scripts, written in FSCRIPT;

- the **Script interpreter**, which is the FSCRIPT interpreter provided by FRASCATI, runs the transition scripts and *modifies* the system; the scripts, as shown below, consist in actions such as the ones we identified in the minimal adaptation API in Subsection 4.2.4 (add, remove, bind, unbind);

- the **FTM&Adaptation Repository** contains transition packages, which can be modified and enriched during the lifetime of the system, according to new threats and changes, by the **FTM developer**.

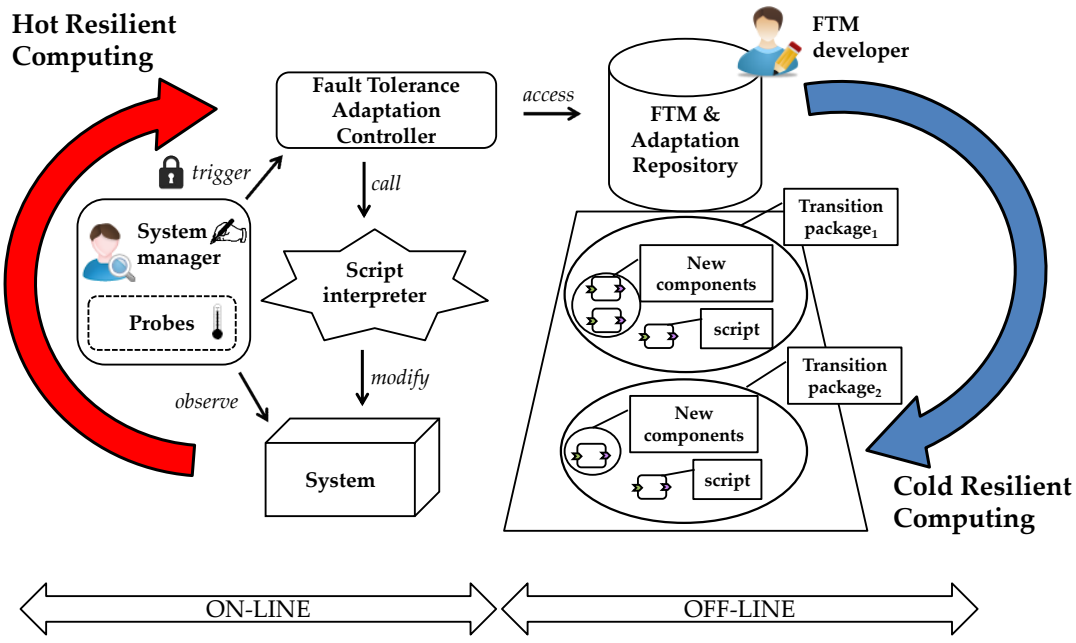


Figure 4.4: Overview of the Adaptation Process

Cold vs. Hot Resilient Computing. We consider that resilient computing encompasses two aspects: *cold resilient computing*, which covers off-line activities and preparation, such as "design for adaptation" and *hot resilient computing*, which covers on-line/runtime activities. The FTM&Adaptation repository concerns the former, while all the other actors and their interac-

tions form the latter. Together, they form a loop, as the cold one feeds transition packages to the hot one, while the hot one provides feedback for improving and enriching the cold one.

Transition Packages. Transition packages, called *contributions* in SCA vocabulary, contain the new components which must be installed to execute a transition towards a new FTM and the script component which replaces old components with new ones. For executing the transition PBR→LFR, the corresponding transition package contains the components `syncBefore` and `syncAfter` specific to LFR and the script which removes the old `syncBefore` and `syncAfter` specific to PBR and inserts the new ones. Transition packages can either transform an FTM in another one through a subtraction ⊖, when changes impact resources or application characteristics or compose ⊕ it with another one, when changes affect FT. In the case of PBR to LFR, we subtract from LFR what it has in common with PBR and the difference represents the set of components which must be inserted.

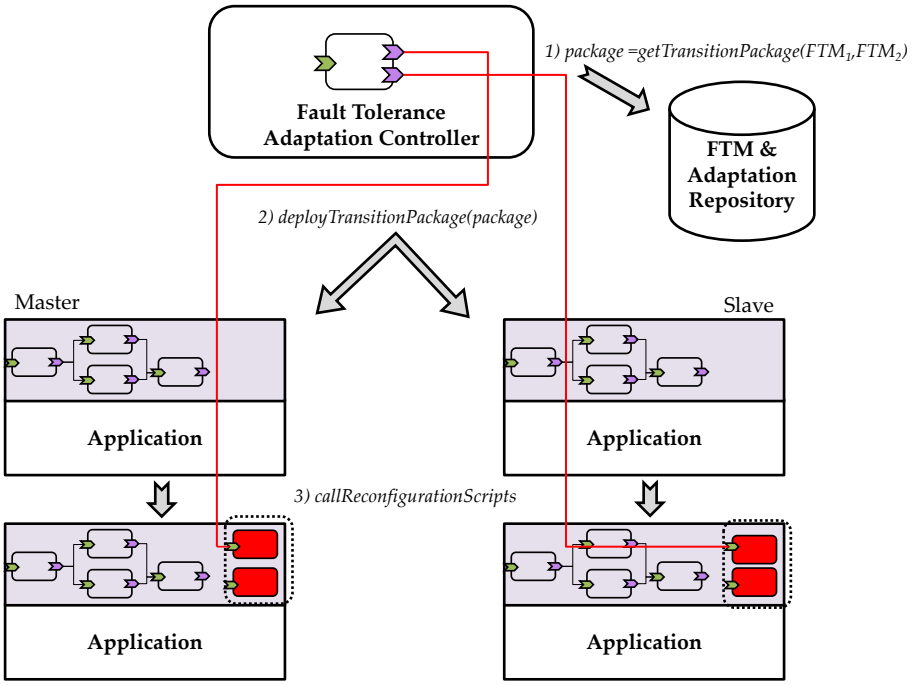


Figure 4.5: Reconfiguration Process

Reconfiguration Process. The reconfiguration process is illustrated in Figure 4.5. A script acts locally (i.e., in one process/JVM), on the site on which the FSCRIPT component is deployed, in order to access and modify the local component graph. Therefore, the script running on the master transforms the Primary in Leader and an identical script, running on the slave, transforms the Backup in Follower. Once the system manager triggers a transition between FTMs, the Fault Tolerance Controller does the following operations. It loads from the FTM&Adaptation Repository the necessary transition package (step 1) and it deploys a copy on the master and another one on the slave (step 2). As a result, the new components which

must be inserted become available on each site, together with the components containing the transition scripts. Next, the controller calls the reconfiguration services provided by the script components (step 3), which is equivalent to calling the script interpreter. Once the transition is executed, the controller removes the script components from the master and the slave.

4.5 Implementing On-line Transitions Between FTMs

In the following, we focus on the implementation of several transitions in Figure 2.3: PBR→LFR, LFR→LFR⊕TR (i.e., a composition between LFR and TR) and LFR→Assert&Duplex. In this subsection, we show both the content of the script components performing the transition mechanics and the contents of the old components and the new ones replacing them, representing the behavioral differences between FTMs. The script components and the new components which are to be introduced in the initial architecture are contained in the transition packages deployed through the process explained above. We leverage the runtime reconfiguration capabilities of FRASCATI and FSCRIPT for executing these transitions between FTMs.

4.5.1 PBR→LFR

Regarding the first transition, PBR→LFR, the variable features between PBR and LFR are `syncBefore` and `syncAfter`, according to Table 3.1. In practice, other variants of PBR and LFR exist, for which only the post-processing inter-replica synchronization (i.e., `syncAfter`) differs, which is all the more reason to perform fine-grained differential transitions between the two duplex strategies. Listing 4.4 shows the script (written in FSCRIPT) which modifies the component-based architecture of PBR from Figure 4.3 to reach LFR.

In short, the script does the following:

- disconnect the old `syncBefore` and `syncAfter` from all their services and references (lines 8 – 17 and 35 – 43);
- delete the old components (lines 18 – 20 and 44 – 46);
- add the new components (lines 21 – 24 and 47 – 49);
- connect the new `syncBefore` and `syncAfter` to all the necessary services and references (lines 25 – 33 and 50 – 60).

In order to seamlessly replace components, they must provide services described through identical interfaces. For instance, the `syncBefore` components characteristic to PBR and to LFR must implement the same `SyncBeforeService` interface so as to not affect components which have references to them and which are common to all FTMs (e.g., `protocol`, see Figure 4.3). Listing 4.5 contains the `SyncBeforeService` Java interface implemented by these components. There are two methods in it. The first one, `executeBefore(...)`, annotated with @POST and @OneWay (meaning non-blocking in SCA terms), is used for calling the synchronization method of the other replica. The second one, `triggerBefore(...)`, executes the local synchronization operations.

Listing 4.4: The Script Implementing the PBR→LFR Transition

```
action changeStrategyPbr2Lfr() {
ftm=$domain/scadescendant::ftm;
protocol=$ftm/scachild::protocol;
pbrlfr=$domain/scachild::transition_launcher/scachild::pbr2lfr;
old_sync_before=$ftm/scachild::syncBefore;
old_sync_after=$ftm/scachild::syncAfter;
--Replace syncBefore
set-state($old_sync_before,'STOPPED');
set-state($protocol,'STOPPED');
remove-scawire($protocol/scareference::syncBeforeService,$old_sync_before/scaservice::execute);
set-state($protocol,'STARTED');
remove-scawire($old_sync_before/scareference::syncAfterService,
               $old_sync_after/scaservice::execute);
remove-scawire($old_sync_before/scareference::logService,
               $ftm/scachild::replyLog/scaservice::logService);
remove-scawire($old_sync_before/scareference::proceedService,
               $ftm/scachild::proceed/scaservice::execute);
set-state($ftm,'STOPPED');
remove-scachild($ftm,$old_sync_before);
set-state($ftm,'STARTED');
add-scachild($ftm,$pbrlfr/scachild::syncBefore);
set-state($pbrlfr,'STOPPED');
remove-scachild($pbrlfr,$pbrlfr/scachild::syncBefore);
new_sync_before=$ftm/scachild::syncBefore;
add-scawire($protocol/scareference::syncBeforeService,$new_sync_before/scaservice::execute);
add-scawire($new_sync_before/scareference::proceedService,
            $ftm/scachild::proceed/scaservice::execute);
add-scawire($new_sync_before/scareference::logService,
            $ftm/scachild::replyLog/scaservice::logService);
add-rest-binding($new_sync_before/scareference::syncBeforeService,
                 "http://localhost:8081/SyncBeforeService");
add-rest-binding($new_sync_before/scaservice::execute,
                 "http://localhost:8080/SyncBeforeService");
--Replace syncAfter
set-state($old_sync_after,'STOPPED');
remove-scawire($old_sync_after/scareference::logService,
               $ftm/scachild::replyLog/scaservice::logService);
remove-scabinding($old_sync_after/scareference::stateAccessService,
                  $old_sync_after/scareference::stateAccessService/scabinding::*);
remove-scabinding($old_sync_after/scareference::synchronizeService,
                  $old_sync_after/scareference::synchronizeService/scabinding::*);
remove-scabinding($old_sync_after/scaservice::execute,
                  $old_sync_after/scaservice::execute/scabinding::*);
set-state($ftm,'STOPPED');
remove-scachild($ftm,$old_sync_after);
set-state($ftm,'STARTED');
add-scachild($ftm,$pbrlfr/scachild::syncAfter);
remove-scachild($pbrlfr,$pbrlfr/scachild::syncAfter);
new_sync_after=$ftm/scachild::syncAfter;
add-scawire($new_sync_before/scareference::syncAfterService,
            $new_sync_after/scaservice::execute);
add-scawire($new_sync_after/scareference::logService,
            $ftm/scachild::replyLog/scaservice::logService);
add-rest-binding($new_sync_after/scareference::synchronizeService,
                 "http://localhost:8081/SyncAfterService");
add-rest-binding($new_sync_after/scareference::stateAccessService,
                 "http://localhost:8080/StateAccessService");
add-rest-binding($new_sync_after/scaservice::execute,"http://localhost:8080/SyncAfterService");
set-state($new_sync_after,'STARTED');
set-state($new_sync_before,'STARTED'); }
```

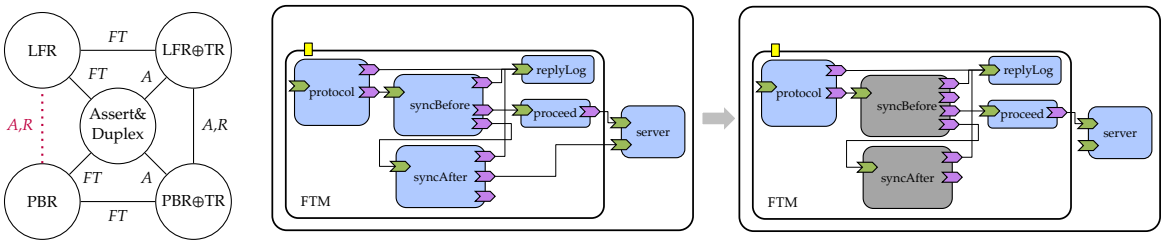


Figure 4.6: PBR→LFR Transition: Scenario (left); Initial and Final Component-Based Architectures (center and right)

Listing 4.5: SyncBeforeService Java Interface

```
@Service
public interface SyncBeforeService {
    @POST
    @OneWay
    public void executeBefore(ClientRequest syncMessage);
    public void triggerBefore(ClientRequest syncMessage);
}
```

We only manipulate stateless components, thanks to our “design for adaptation" approach. State information such as the last client request, the last reply etc. are available and accessible in `replyLog` (see Figure 4.3), which is not affected by the reconfiguration. Buffering incoming requests, putting components in a quiescent state when executing various actions such as wiring references, binding and unbinding services and references are handled by the middleware and transparent to the user, as already mentioned in paragraph 4.2.4.

Figure 4.6 illustrates the PBR→LFR transition. On the left, the scenario from Chapter 2 is recalled and the transition in question is highlighted (purple dotted line). On the right, the initial component-based architecture and the final one are represented, with a focus on the components changed during the transition (`syncBefore` and `syncAfter` represented in gray, meaning a different implementation). Besides, we can notice that the number of references is different between the initial components and the final ones.

Having detailed the mechanics behind transitions, the actual behavioral modifications corresponding to the transition between the two FTMs are now explained. Both `syncBefore` and `syncAfter` are replaced during this transition but, for the succinctness of the presentation, we only show the contents of the `syncBefore` components corresponding to PBR and LFR.

Listing 4.6: SyncBefore Java Class for PBR

```
......
@Scope("COMPOSITE")
@Service(value=SyncBeforeService.class)
@EagerInit
public class SyncBefore implements SyncBeforeService {

   private SyncAfterService syncAfter;
   private ComputeService proceed;
   private ReplyLogService rlog;

   @Reference (name="proceedService")
   public void setComputeService(ComputeService s) {
      this.proceed = s;
   }

   @Reference (name="replyLogService")
   public void setReplyLogService(ReplyLogService s) {
      this.rlog = s;
   }

   @Reference (name="synchronizeAfterService")
   public void setSyncBeforeService(SyncAfterService s) {
      this.syncAfter = s;
   }

   public SyncBefore() {}

   @Init
   public void initialize() {}

   public void executeBefore(ClientRequest syncMsg) {}

   public void triggerBefore(ClientRequest syncMsg) {
      double res = proceed.compute(syncMsg.getOp1(),syncMsg.getOp2(),syncMsg.getOp());
      this.rlog.setLog(syncMsg, res);
      this.syncAfter.triggerAfter(new SyncAfterMessage(syncMsg.getId()));
   }
}
```

Listing 4.6 and Listing 4.7 outline the contents of the `syncBefore` components of PBR and LFR, respectively. In PBR, `syncBefore` only orchestrates the computation on the master (see `triggerBefore` method in Listing 4.6). It calls `proceed`, writes the result in `replyLog` and then calls `syncAfter`. No information is sent to the slave, which does nothing during this step (the empty `executeBefore` method in Listing 4.6). In LFR, `syncBefore` on the master forwards the client request to the slave (see `triggerBefore` method in Listing 4.7) and then orchestrates its local computation. On the slave, the client request is also computed and the result is written in the local `replyLog` (see `executeBefore` method in Listing 4.7).

Listing 4.7: SyncBefore Java Class for LFR

```
......
@Scope("COMPOSITE")
@Service(value=SyncBeforeService.class)
@EagerInit
public class SyncBefore implements SyncBeforeService {
   @Controller (name="component")
   protected Component fractalComponent;
   protected SCAPropertyController propCtrl;
   private SyncBeforeService syncBefore;
   private SyncAfterService syncAfter;
   private ComputeService proceed;
   private ReplyLogService rlog;
   @Reference (name="proceedService")
   public void setComputeService(ComputeService s) {
      this.proceed = s;
   }
   @Reference (name="replyLogService")
   public void setReplyLogService(ReplyLogService s) {
      this.rlog = s;
   }
   @Reference (name="synchronizeBeforeService")
   public void setSyncBeforeService(SyncBeforeService s) {
      this.syncBefore = s;
   }
   @Reference (name="synchronizeAfterService")
   public void setSyncBeforeService(SyncAfterService s) {
      this.syncAfter = s;
   }
   public SyncBefore() {}
   @Init
   public void initialize() {}
      SuperController superCtrl = null;
      try {
         superCtrl = Fractal.getSuperController(fractalComponent);
         Component[] parents = superCtrl.getFcSuperComponents();
         Component ftm = parents[0];
         propCtrl = (SCAPropertyController)ftm.getFcInterface(SCAPropertyController.NAME);
      }
      catch (NoSuchInterfaceException e1) {
         e1.printStackTrace();
      }
   }
   public void executeBefore(ClientRequest syncMsg) {
      double res = proceed.compute(syncMsg.getOp1(),syncMsg.getOp2(),syncMsg.getOp());
      this.rlog.setLog(syncMsg, res);
   }
   public void triggerBefore(ClientRequest syncMsg) {
      int mode = Integer.parseInt(this.propCtrl.getValue("mode").toString());
      if(mode==1) {
         this.syncBefore.executeBefore(syncMsg);
      }
      double res = proceed.compute(syncMsg.getOp1(),syncMsg.getOp2(),syncMsg.getOp());
      this.rlog.setLog(syncMsg, res);
      this.syncAfter.triggerAfter(new SyncAfterMessage(syncMsg.getId()));
   }
}
```

4.5.2 LFR→LFR⊕TR

The second transition we present is LFR→LFR⊕TR, which is a composition of FTMs. The *before-proceed-after* execution scheme of TR is presented in Table 3.1. In order to facilitate the mapping to components and to minimize the number of components to introduce in the architecture of LFR (hence, the reconfiguration time, as further shown), we decided to combine the execution steps of TR in a single `proceed` component. In short, the `syncBefore` and `syncAfter` components of LFR are left untouched, as inter-replica synchronization is specific to a duplex strategy, while the initial `proceed` component, which simply consisted in a service call on the `server` in LFR (see Listing 4.8), is replaced with a more complex component containing the specificities of TR (see Listing 4.9):

- state capture on the server prior to computation;
- multiple service invocations and result storage;
- state restoration on the server between consecutive service invocations;
- comparison of results.

Figure 4.7 illustrates this transition. On the left, the transition is highlighted in the context of the scenario from Chapter 2 (purple dotted line). On the right, the initial component-based architecture and the final one are represented, with a focus on the components changed during the transition (`proceed` represented in gray, meaning a different implementation). We can also notice that the final component has an additional reference, towards the state management service provided by the `server`.

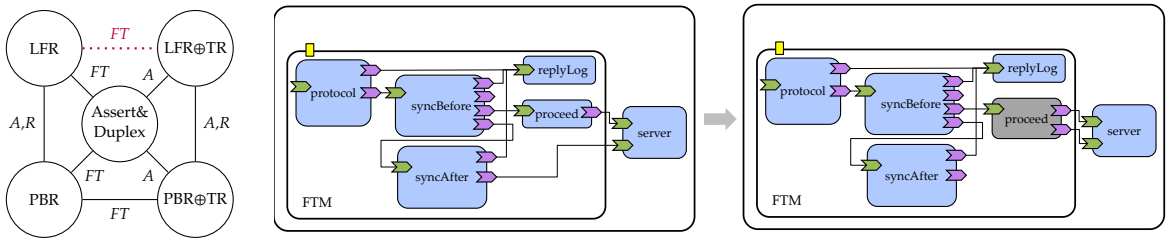


Figure 4.7: LFR→LFR⊕TR Transition: Scenario (left); Initial and Final Component-Based Architectures (center and right)

The LFR→LFR⊕TR transition package contains the new `proceed` component which must be introduced and the component written in FSCRIPT which performs the reconfiguration. The script follows the same pattern as the previous one: it disconnects the old `proceed` from all its services and references, deletes the old component, adds the new `proceed` and connects it to all necessary services and references.

Listing 4.8: Proceed Java Class for PBR and LFR

```
@Scope("COMPOSITE")
@Service(value=ComputeService.class)
@EagerInit
public class Proceed implements ComputeService {

   @Reference (name="compute")
   private ComputeService server;

   public Proceed() {}

   public double compute(double op1, double op2, char op) {
      return server.compute(op1, op2, op);
   }
}
```

Listing 4.9: Proceed Java Class for Composing PBR or LFR and TR

```
@Scope("COMPOSITE")
@Service(value=ComputeService.class)
@EagerInit
public class Proceed implements ComputeService {

   @Reference (name="compute")
   private ComputeService server;

   @Reference (name = "stateAccessService")
   private ServerStateAccessService stateAccess;

   private ServerState state;

   public Proceed() {}

   public double compute(double op1, double op2, char op) {
      this.state = this.stateAccess.captureState();
      Double reply1 = new Double(this.server.compute(op1, op2, op));
      this.stateAccess.mapState(this.state);
      Double reply2 = new Double(this.server.compute(op1, op2, op));
      if (reply1.compareTo(reply2) == 0) {
         return reply1;
      }
      else {
         this.stateAccess.mapState(this.state);
         Double reply3 = new Double(this.server.compute(op1, op2, op));
         if ((reply1.compareTo(reply3) == 0) || (reply2.compareTo(reply3) == 0)) {
            return reply3;
         }
         else {
            System.out.println("error detected; crashing now");
            System.exit(0);
         }
      }
      return Double.NaN;
   }
}
```

It is worth noting that the same transition package is used for executing the PBR→PBR⊕TR transition from Figure 4.7. This is possible because the `proceed` component is the same in PBR and LFR (Listing 4.8), consisting in a service call on the `server`, and the transition script is identical to the one used for composing LFR with TR.

4.5.3 LFR→Assert&Duplex

As previously mentioned, several versions of Assert&Duplex can be designed. In the toolbox of FT design patterns described in Chapter 3, two versions were developed, one based on PBR and one based on LFR (PBR_A and LFR_A, respectively in Figure 3.6). Both these versions were mapped on the component-based architecture.

Their execution leverages the fact that the client can issue a request up to three times, if it does not receive a reply. In the `syncAfter` step of these mechanisms, an assertion is applied to the result obtained by the master. If the assertion succeeds, the master sends either a checkpoint (in Assert&PBR) or a notification (in Assert&LFR) to the slave and the reply to the client. If the assertion fails, the master crashes, to enforce the fail-silent assumption. The slave detects its crash and becomes master-alone. When the client reissues the request, it arrives this time on the ex-slave. This one either already has the reply (in Assert&LFR, the slave computes the request in parallel with the master) or computes the request as a new one (in Assert&PBR).

The LFR→Assert&Duplex transition is executed by replacing the `syncAfter` component of LFR with a new one implementing the specificities of Assertion. Figure 4.8 illustrates this transition. On the left, the transition is highlighted in the context of the scenario from Chapter 2 (purple dotted line). On the right, the initial component-based architecture and the final one are represented, with a focus on the components changed during the transition (`syncAfter` represented in gray, meaning a different implementation).

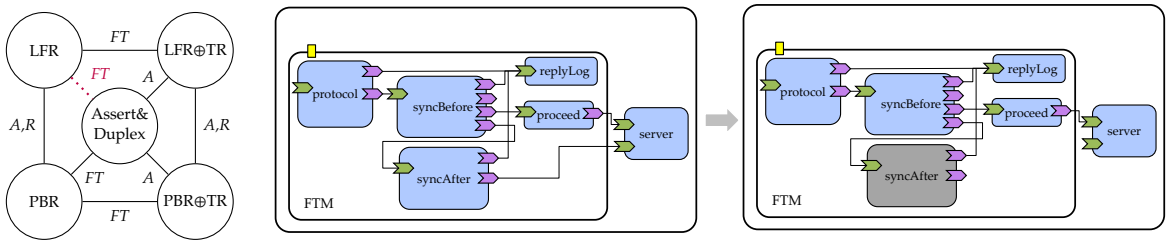


Figure 4.8: LFR→Assert&Duplex Transition: Scenario (left); Initial and Final Component-Based Architectures (center and right)

Listing 4.10 shows the contents of the `syncAfter` component characteristic to LFR. Listing 4.11 outlines the contents of the `syncAfter` component corresponding to Assert&LFR. The behavior corresponding to LFR is maintained (i.e., the master sends the id of the last request processed to the slave) and is enriched with the application of an assertion on the master (`triggerAfter` method).

Listing 4.10: SyncAfter Java Class for LFR

```
.......
@Scope("COMPOSITE")
@Service(value=SyncAfterService.class)
@EagerInit
public class SyncAfter implements SyncAfterService {
   @Controller(name = "component")
   protected Component fractalComponent;
   protected SCAPropertyController propCtrl;
   private ReplyLogService log;
   private SyncAfterService syncAfter;
   @Reference (name="synchronizeService")
   public void setSyncAfterService (SyncAfterService s) {
      this.syncAfter = s;
   }
   @Reference (name="replyLogService")
   public void setReplyLogService (ReplyLogService s) {
      this.log = s;
   }
   public SyncAfter() {}
   @Init
   public void initialize() {
      SuperController superCtrl = null;
      try {
         superCtrl = Fractal.getSuperController(fractalComponent);
         Component[] parents = superCtrl.getFcSuperComponents();
         Component ftm = parents[0];
         propCtrl = (SCAPropertyController)ftm.getFcInterface(SCAPropertyController.NAME);
      }
      catch (NoSuchInterfaceException e1) {
         e1.printStackTrace();
      }
   }
   public void triggerAfter(SyncAfterMessage syncMessage) {
      if(syncMessage.getRecovery()) {
         int id = this.log.getId();
         syncMessage.setRequest_id(id);
         syncMessage.setReply(this.log.getReply(id));
         this.syncAfter.executeAfter(syncMessage);
      }
      else {
         int mode = Integer.parseInt(this.propCtrl.getValue("mode").toString());
         if(mode==1) {
            this.syncAfter.executeAfter(syncMessage);
         }
      }
   }
   public void executeAfter(SyncAfterMessage syncMessage) {
      int id = syncMessage.getRequest_id();
      if (syncMessage.getRecovery()) {
         this.log.setLog(id, syncMessage.getReply());
      }
      else {
         System.out.println("received from Master id: " + id);
      }
   }
}
```

Listing 4.11: SyncAfter Java Class for Assert&LFR

```
......
@Scope("COMPOSITE")
@Service(value=SyncAfterService.class)
@EagerInit
public class SyncAfter implements SyncAfterService {
   @Controller (name = "component")
   protected Component fractalComponent;
   protected SCAPropertyController propCtrl;
   private ReplyLogService log;
   private SyncAfterService syncAfter;
   private Assertion assertion;
   @Reference (name="synchronizeService")
   public void setSyncAfterService (SyncAfterService s) {
      this.syncAfter = s;
   }
   @Reference (name="replyLogService")
   public void setReplyLogService (ReplyLogService s) {
      this.log = s;
   }
   public SyncAfter() {
      this.assertion = new Assertion();
   }
   @Init
   public void initialize() {
      SuperController superCtrl = null;
      try {
         superCtrl = Fractal.getSuperController(fractalComponent);
         Component[] parents = superCtrl.getFcSuperComponents();
         Component ftm = parents[0];
         propCtrl = (SCAPropertyController)ftm.getFcInterface(SCAPropertyController.NAME);
      }
      catch (NoSuchInterfaceException e1) {
         e1.printStackTrace();
      }
   }
   public void triggerAfter(SyncAfterMessage syncMessage) {
      int mode = Integer.parseInt(this.propCtrl.getValue("mode").toString());
      if(syncMessage.getRecovery()) {
         int id = this.log.getId();
         syncMessage.setRequest_id(id);
         syncMessage.setReply(this.log.getReply(id));
         this.syncAfter.executeAfter(syncMessage);
      }
      else {
         int id = syncMessage.getRequest_id();
         ClientRequest req = this.log.getRequest();
         assertion.applyAssertion(log.getReply(id),req.getOp1(),req.getOp2(),req.getOp());
         if (assertionField == 0) {
            if(mode ==1) {
               this.syncAfter.executeAfter(syncMessage);
            }
         }
         else {
            System.out.println("assertion failed! crashing now");
            System.exit(0);
         }
      }
   }
   public void executeAfter(SyncAfterMessage syncMessage) {
      int id = syncMessage.getRequest_id();
```

```
      if (syncMessage.getRecovery()) {
         this.log.setLog(id, syncMessage.getReply());
      }
      else {
         System.out.println("received␣from␣Master␣id:␣" + id);
      }
   }
}
```

4.6 Consistency of Distributed Adaptation

A crucial aspect of transitions between FTMs is ensuring their consistency because evolvability must not reduce the reliability of fault-tolerant applications. Thanks to the support provided by the underlying component-based middleware and to the manner in which transitions are performed, our approach tackles key aspects such as local and distributed consistency and recovery in case of crash during adaptation.

4.6.1 Local Consistency

FRASCATI and its integrated FSCRIPT engine guarantee the consistency of local reconfigurations performed using scripts. FSCRIPT enforces an all-or-nothing semantics [Léger *et al.*, 10]. The reliability of the reconfiguration process is achieved by using a model of configurations (i.e., component-based architectures) and reconfigurations (i.e., transitions between two consistent configurations), verifying integrity constraints (i.e., configuration invariants and pre/post conditions on elementary reconfiguration operations) and performing reconfigurations in a transactional manner [Léger *et al.*, 10]. In case of constraint violation during the reconfiguration process, a `ScriptException` is thrown, the transaction is rolled back and the component-based architecture remains in its initial configuration.

4.6.2 Consistency of Request Processing

Stopping a component implies waiting for all its internal processing to finish, blocking its inputs and buffering them. Therefore, the components which must be replaced during transitions can only be stopped when their processing is finished. This means that, if a client request is received just before adaptation is triggered, the request is processed and the client receives the reply before the subsequent requests are blocked and buffered and the interaction between replicas is locked (see Figure 4.9). When the lock is released, the buffered client requests are processed in the new configuration of the FTM.

4.6.3 Distributed Consistency

The FTMs discussed in this work consist of two replicas and a specific inter-replica protocol, therefore transitions between FTMs must be performed on two different sites. On each site, a script component performs the required reconfiguration, which can either terminate successfully or with a `ScriptException`, in case integrity constraints are violated. The script component on each site is wrapped in a Java class which kills the local replica if an exception is

thrown, to enforce the fail-silent assumption and to prevent the overall FTM from reaching an inconsistent hybrid configuration. Therefore, a local reconfiguration can either terminate successfully (see Figure 4.9) or with a crash. The failure detection mechanism incorporated in the considered duplex strategies detects the crash and informs the remaining replica which, if the reconfiguration has succeeded on it (i.e., it has not crashed during reconfiguration), becomes master-alone.

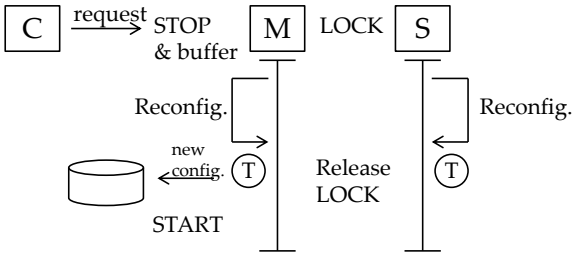


Figure 4.9: Transition Process

4.6.4 Recovery of Adaptation

We consider that replicated computing units can crash during transitions. Therefore, upon successful completion of the reconfiguration on one replica (either master or slave), the current configuration (i.e., the target FTM) is logged on a stable storage (see Figure 4.9). If the other replica crashes (either because of a `ScriptException` or because of an ordinary crash fault) before completing its own reconfiguration, it is normally restarted/replaced by the system manager to comply with the fault tolerance requirements. However, it must not be restarted in its old configuration (i.e., with the initial FTM), but in the same one as its counterpart, which has successfully completed the transition. This information is recovered from the stable storage which keeps track of the currently active configuration. This is a form of forward recovery: the FTM is re-established in its final configuration not in the one before adaptation.

Let us assume that the PBR→LFR transition is triggered and the Backup crashes before becoming Follower, while the Primary successfully becomes Leader and logs the new status. The Leader detects the crash of the slave and becomes Leader-only. If the slave is repaired/rebooted, it must not be deployed as Backup but as Follower, according to the new status. For this, it must first read the stable storage to identify which entity must be deployed, to complete the current FTM configuration, namely LFR.

4.7 Summary

This chapter focused on the mapping of the previously described fault tolerance design patterns on a reflective component-based middleware and on the agile on-line differential transitions between FTMs.

First of all, we detailed the necessary infrastructure for developing component-based FTMs and explained our choice of tools by identifying a set of functionalities that must be provided by the runtime support.

Next, we described the result of mapping the *before-proceed-after* generic execution scheme on a component-based architecture. Afterwards, the transition process leveraging the support provided by the chosen reflective component-based middleware was explained.

Three examples of transitions between FTMs were presented, each one requiring a different set of modifications in terms of variable features: PBR→LFR requires the replacement of `syncBefore` and `syncAfter`, LFR→LFR⊕TR requires the replacement of `proceed` and LFR→Assert&LFR requires the replacement of `syncAfter`.

Last but not least, the consistency of distributed adaptation of replicas involved in FTMs was discussed, with an emphasis on the strategy for making the adaptation process tolerant to faults that might occur during transitions.

Chapter 5

Evaluation, Integration, Application

> "His way had therefore come full circle, or rather had taken the form of an ellipse or a spiral, following as ever no straight unbroken line, for the rectilinear belongs only to Geometry and not to Nature and Life."
>
> — Hermann Hesse, *The Glass Bead Game*

5.1 Introduction

This chapter consists of two main parts. In the first part, we assess the performance of our adaptive fault tolerance middleware. This is done, firstly, by providing quantitative measurements of transition time between FTMs and, secondly, in terms of the agility of modifications.

In the second part, we illustrate the usability and interest of this framework by describing its integration in two different platforms. In the first example, our notion of adaptive fault tolerance is incorporated in a design-driven development methodology - named DiaSuite - for the design, development, and deployment of resilient applications. This methodology aims to ensure the traceability of dependability requirements along the application life cycle, including runtime adaptation. In the second example, we illustrate the interest of adaptive fault tolerance in a scenario based on a parking structure management application. The proposed adaptive fault tolerance mechanisms are integrated in Srijan, a toolkit for enabling application development for WSNs based on data-driven macroprogramming. These two examples prove that the framework presented in this work can be readily integrated and exploited in different contexts where the need for adaptive fault tolerance arises. This is facilitated by a clear separation between the different concerns and roles of the stakeholders.

5.2 Evaluation

In this section, we analyze the performance and agility of our differential approach and associated on-line transitions between FTMs. Firstly, we assess the advantages of performing differential modifications as opposed to monolithic replacements of FTMs by providing a set of quantitative measurements. Next, the agility of transitions is discussed.

5.2.1 Performance

In order to assess the performance of agile fine-grained transitions between FTMs, we mapped all the protocols designed in Chapter 3 (see Figure 3.6) on a component-based architecture. More precisely, we developed PBR, LFR, PBR⊕TR, LFR⊕TR, Assert&PBR and Assert&LFR as stand-alone FTMs that can be directly deployed and all the differential transitions between them, both direct and inverse. In Chapter 4, in order to illustrate the approach, we described in detail three transitions, which are part of this implementation.

All the FTMs were validated through fault injection. The duplex mechanisms were tested by random crash faults. The mechanisms tolerating value faults were tested by injecting random value faults. All the composed mechanisms were validated through software implemented fault injection using multiple faults. Before presenting the comparison between differential transitions and monolithic deployments of FTMs, we must emphasize that these are simplified implementations of existing FTMs, serving as proof-of-concept. This explains the low total number of components in one FTM (7) shown below.

Table 5.1 shows that, while the architecture of a full FTM (on one replica) contains 7 components (first line), transitions between FTMs affect a much smaller number of components, between 1 (e.g., PBR→PBR⊕TR, PBR→Assert&PBR) and 3 (e.g., for the complex PBR→LFR⊕TR). It is worth noting that these components are stateless, thanks to our "design for adaptation" step, which means that state transfer issues inherent in monolithic transitions are completely eliminated. This shows that by modifying tiny bricks which contain variable features, we obtain other FTMs, in a flexible and reversible manner. The underlying variable features affected by transitions are also shown in Table 5.1 (B=Before, P=Proceed, A=After).

Initial FTM \ Final FTM	PBR	LFR	PBR⊕TR	LFR⊕TR	Assert&PBR	Assert&LFR
∅	7	7	7	7	7	7
PBR	-	2 (B,A)	1 (P)	3 (B,P,A)	1 (A)	2 (B,A)
LFR	2 (B,A)	-	3 (B,P,A)	1 (P)	2 (B,A)	1 (A)
PBR⊕TR	1 (P)	3 (B,P,A)	-	2 (B,A)	2 (P,A)	3 (B,P,A)
LFR⊕TR	3 (B,P,A)	1 (P)	2 (B,A)	-	3 (B,P,A)	2 (P,A)
Assert&PBR	1 (A)	2 (B,A)	2 (P,A)	3 (B,P,A)	-	2 (B,A)
Assert&LFR	2 (B,A)	1 (A)	3 (B,P,A)	2 (P,A)	2 (B,A)	-

Table 5.1: Number of Components & Variable Features Replaced During Transitions

In Table 5.2, we compare the time necessary for deploying full FTMs (first line) and the time necessary for executing differential transitions between them. The functional application

used for testing the FTMs consists in a single component whose deployment has a negligible effect on the overall deployment time. These results represent averages over 100 tests for each cell of the table. All tests have been performed on a Dell Latitude E6420 with an Intel Core i5-2410M CPU at 2.30 GHz, having 8 GB of RAM and running under Windows 7 Professional. As deployment of FTMs and transitions are performed in parallel on two replicas, in this table we show the time corresponding to one replica. We can notice that our differential approach not only eliminates state transfer issues inherent to monolithic replacements of FTMs but is also more efficient in terms of execution time, making the system more reactive to changes. For instance, we can see that performing PBR→PBR⊕TR in a differential manner takes 840 ms, while deploying PBR⊕TR from scratch takes 3.8 s.

Initial FTM \ Final FTM	PBR	LFR	PBR⊕TR	LFR⊕TR	Assert&PBR	Assert&LFR
∅	3819	3751	3852	3783	3824	3786
PBR	-	1003	840	1146	856	1090
LFR	1011	-	1151	838	1085	840
PBR⊕TR	836	1148	-	1012	937	1191
LFR⊕TR	1145	830	1019	-	1186	930
Assert&PBR	851	1081	938	1184	-	1007
Assert&LFR	1085	834	1186	932	1005	-

Table 5.2: FTM Deployment from Scratch vs. Duration of Execution of Transitions (ms)

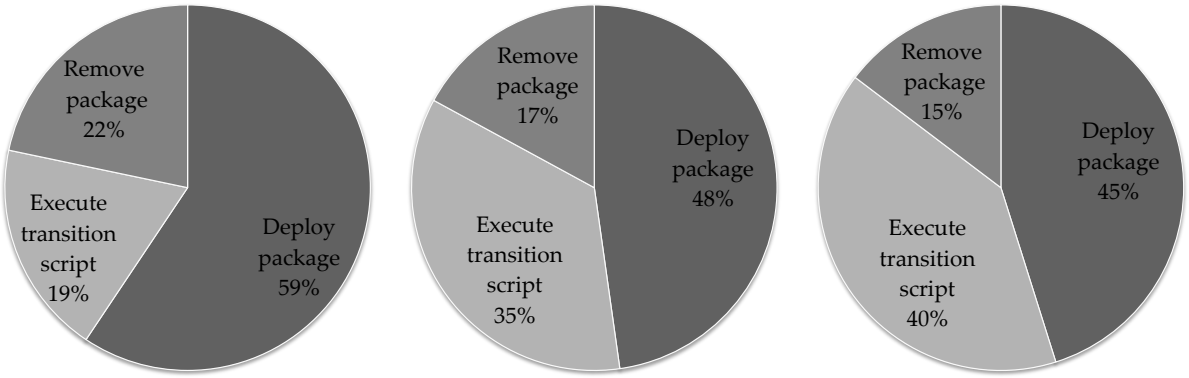
Obviously, the ratio between the transition time and the deployment time is more relevant than absolute values. These values are based on a simplified implementation of the FTMs but the ratio should remain constant if the implementation becomes more complex because both the stand-alone FTM and the transition packages would be affected.

As we have explained earlier, transitions consist in three main steps: deployment of transition packages, execution of reconfiguration scripts and removal of script components. This process is orchestrated by the Fault Tolerance Adaptation Controller (see Figure 4.4). Figure 5.1 shows the contribution of each step to the overall transition execution time for three transitions affecting different number of components (i.e., from one variable features to all three). We see that the actual execution of the reconfiguration script, which can be considered the most complex step, takes 19% of the total time for the replacement of one component, 35% for two components and 40% for three components.

This means that even in the most complex transitions, affecting all three variable features, the execution of the transition script (i.e., of the reconfiguration mechanics) takes less then half of the total transition time. These quantitative measurements also give us indications as to what could speed up the transition process, namely the optimization of the deployment step, which currently takes approximately half of the total transition time.

Figure 5.2 shows the correlation between the number of components which are replaced during transitions and the different steps of the reconfiguration process (deployment, script execution, removal), on the one hand, and the total transition time, on the other hand. The removal step takes almost the same amount of time in all three cases. The time necessary for the deployment of transition packages depends on the number of components (that are

inside them). As scripts consist of basic mechanical operations of wiring and unwiring, their execution is expected to have a roughly linear correlation with the number of components which are being replaced.



(a) LFR→LFR⊕TR (1) (b) PBR→LFR (2) (c) PBR→LFR⊕TR (3)

Figure 5.1: Distribution of Duration of Transitions w.r.t. Number of Components Replaced

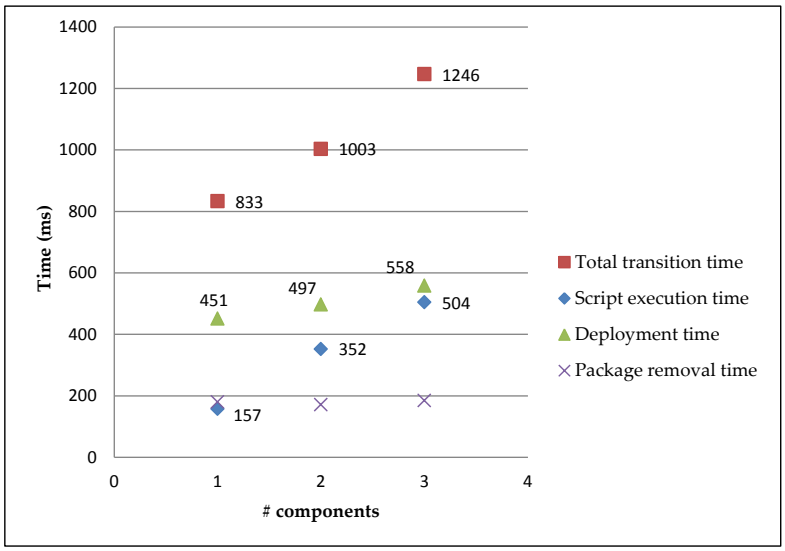


Figure 5.2: Duration of Execution of Transitions w.r.t. Number of Components Replaced

5.2.2 Agility

It is worth noting that, beyond the quantitative evaluation of the approach, the capacity to perform evolution of FTMs using an agile fine-grained approach is a key feature for long-lived

autonomous systems. The proposed approach aims to equip an application only with the FTMs which are necessary and thus operational at a certain point in time and to provide system developers and managers with the means to replace or augment these FTMs only if/when necessary in order to accommodate change. This flexibility is expected to have an impact on the time necessary for executing a transition. An agile transition consisting in the deployment of new components and removal of old ones is expected to be more time consuming than a preprogrammed one which can be visualized as the choice of a branch in an already programmed switch statement.

The current implementation of our framework proves the feasibility of agile adaptation of FTMs (as opposed to the preprogrammed one, be it fine-grained or coarse-grained). As illustrated in Figure 4.4, transitions which were either unknown or considered unnecessary at design time can be performed during the lifetime of the system by developing off-line the necessary differential package and integrating it on-line. All the transitions between FTMs are performed by minimizing the impact on the overall component-based architecture, i.e., by replacing only the variable features. Furthermore, at no point in time is the system loaded with unnecessary FTMs or parts of FTMs (resulting in inactive code) as in the case of preprogrammed adaptations.

In our investigation of related work on adaptive fault tolerance, we found some examples of quantitative evaluations of transitions between FTMs. In [Marin *et al.*, 01], the switch from active (equivalent to our LFR) to passive strategy (equivalent to our PBR) takes 4.5ms. The stabilization between passive and active replication takes 360ms and the reverse takes 390ms in [Lung *et al.*, 06]. In both cases adaptation is preprogrammed, i.e., the supported fault tolerance strategies are known at initial design time and hard-coded at system deployment. In [Fraga *et al.*, 03], it takes 260ms to alternate between passive and active strategy. Although the authors leverage a component model, reconfiguration does not appear to be performed agilely at runtime and adaptation has a coarse grain compared to ours: there is a replication coordinator component encapsulating all the fault tolerance logic. While in our case the transition from passive to active replication takes 1003ms in total (i.e., package deployment, script execution and package removal combined), the substantial difference lies in the fact that this is an agile adaptation, not a preprogrammed one. As expected, agility comes with an additional cost in terms of deployment time. However, compared to [Lung *et al.*, 06, Fraga *et al.*, 03], this cost does not appear to be excessive, given that our approach brings flexibility and the ability to accommodate changes unforeseen at design time.

5.3 Integrating AFT in the Development Process

5.3.1 Motivation and Context

The integration of fault tolerance strategies in applications is often performed in an ad-hoc manner, leading to code which is difficult to understand, reuse and maintain. When runtime adaptation of these strategies must also be taken into account, resulting systems become even more complex. Design-driven development approaches provide a solution to this growing complexity by enabling a clear separation of the various cross-cutting concerns.

In the case of resilient systems, general-purpose approaches have often proven to be insufficient in terms of guiding the development process and providing traceability of requirements from design to deployment (and, further on, to runtime evolution). When targeting pervasive computing systems that, by definition, operate in the immediate vicinity of humans, resilience is mandatory. This calls for an integrated development process centered around a conceptual framework that allows to guide the development process of a resilient application in a systematic manner [Enard *et al.*, 13]. The fine-grained adaptive FTMs developed in this work represent a brick in this integrated development process architected in the context of the "Software Engineering for Resilient Ubiquitous Systems" Collaborative Research Action (ARC SERUS), with two INRIA projects (PHOENIX Project from Bordeaux and ADAM Project from Lille).

5.3.2 The Sense-Compute-Control Paradigm

The overall approach, described in [Enard *et al.*, 13], relies on a design language that is extended with fault tolerance declarations. DiaSuite [Cassou *et al.*, 11b] is a design-driven methodology for developing applications based on the SCC paradigm. The SCC paradigm originates from the *Sense/Compute/Control* architectural pattern [Taylor *et al.*, 09], which is extremely appropriate for applications interacting with their external environment, e.g. domotics, automotive, avionics etc.

Figure 5.3 illustrates this paradigm. The architectural pattern encompasses three types of components:

- *entities*, corresponding to devices (both hardware and software, e.g., a surveillance camera and its device driver) that interact with the external environment through its sensing and actuating capabilities;
- *context components* that refine (i.e., filter, aggregate and interpret) raw data sensed by entities;
- *controller components* that trigger actions on entities, based on refined data.

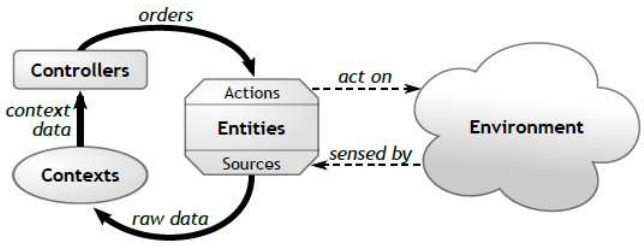


Figure 5.3: The Sense-Compute-Control Paradigm

Several similarities between this pattern and other existing paradigms can be easily identified, undoubtedly because of their common target: systems having a strong interaction with their environment and subject to dynamics. Two examples of such paradigms are the MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) autonomic loop [Computing, 06,

Huebscher and McCann, 08] and the architecture-based adaptation illustrated in RAINBOW [Garlan *et al.*, 01], using two types of low-level entities: probes, similar to entities and gauges, similar to context components.

The particularity of DiaSuite lies in the fact that it leverages the SCC paradigm to support each stage of the development process, from design to deployment. In the course of the SERUS research project, the approach was extended to incorporate resilience.

5.3.3 An Illustrative Example

In order to illustrate the overall approach, an anti-intrusion application was chosen. The application secures a room with video cameras and alarms. To detect an intrusion, the application controls a video camera and periodically analyzes the pictures it has taken of the environment. When an intrusion is detected, an alarm is triggered and the pictures are recorded in a database that can be consulted by an operator for examining the situation and/or identifying the intruder. Since the application is critical for the security of the building and its inhabitants, intrusion detection should be guaranteed in the event of hardware defects.

This typical example of pervasive computing application illustrates two of the main goals of the approach: ensuring traceability of requirements and ensuring separation of concerns.

Traceability

As stated throughout this thesis, in order for an application to be resilient, it must first be dependable and, secondly, it must be able to preserve this attribute in the presence of changes. The anti-intrusion application is critical and, as such, it must be augmented with an appropriate FTM. The choice of FTMs is based on the frame of reference presented in Chapter 2, i.e., targeted fault model, application characteristics and resources. In this particular example, the fault model to tolerate is the crash of the camera. For this, our duplex mechanisms can be used and the application requires two cameras. The values of the parameters are fixed throughout the development process but may vary at runtime. Therefore, dependability requirements must be traceable at every step of the life-cycle, from design to development and, further on, to runtime evolution. This approach aims at providing the necessary support to developers for ensuring the conformance of the application with the requirements throughout the application life-cycle.

Separation of Concerns

In this example, we consider the following adaptation scenario. The two cameras used for tolerating the crash can work either on battery or connected to the main power supply. An intruder may switch off the main power supply in order to deactivate the surveillance system. In this case, the system must rely on alternate power sources (batteries) and, as a result, the application must adapt to lower its power consumption. FTMs have a strong impact on power consumption and, in this example, the application needs to adapt its corresponding FTM to the current energy consumption requirements. When running on the main power supply, the system uses LFR for reducing recovery time in case of crash and optimizing network bandwidth. However, LFR comes with high computation and energy consumption. Therefore, when running on battery, PBR is used to reduce energy consumption.

Runtime adaptation requires the developers to deal with the functional application, fault tolerance and adaptation concerns while implementing the application logic. In this example, the developers must tackle the monitoring of the power supply, the use of PBR and LFR, and the transition between them. Separation of concerns is essential in this process. The design-driven development approach ensures the separation of concerns at design time, by clearly identifying the layer in charge of fault tolerance and the one in charge of runtime adaptation. Through generation of programming support, the separation of concerns is preserved along the application life-cycle.

5.3.4 Overall Approach

Figure 5.4 shows an overview of the DiaSuite design-driven methodology enriched with support for taking into account resilience. At the design stage, the DiaSpec language provides SCC-specific declarations (stage ①) [Cassou *et al.*, 11a]. A component is defined by its interaction contract. For integrating the new requirements in the methodology, the DiaSpec language has been extended in the course of the project, allowing a safety expert to refine the interaction contract of an SCC component with fault tolerance declarations (stage ②). The declarations reflect the fault tolerance taxonomy presented in Chapter 2. For the time being, we focus on physical faults and the safety expert can annotate the critical components either with **require availability** (i.e., the fault model is crash-only) or with **require correctness** (i.e., value faults must be tolerated). The design language can be easily extended to take into account other fault models.

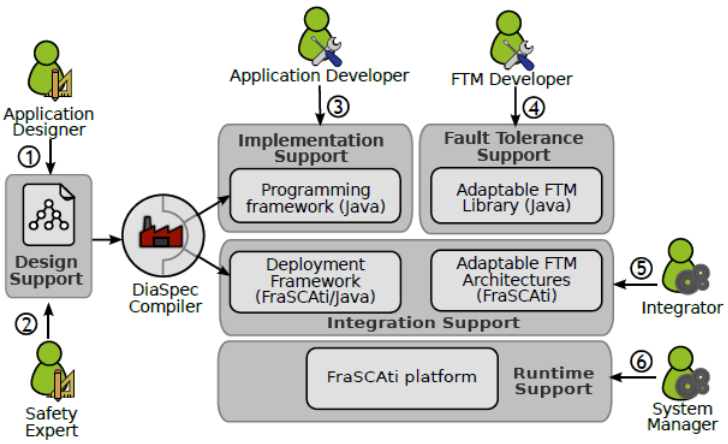


Figure 5.4: Overview of the Approach

With respect to runtime adaptation strategies, the design of a resilient application is layered into application logic and a supervisory layer in charge of monitoring and triggering transitions between FTMs. It is worth noting that both layers are described through the SCC paradigm, in order to ensure separation of concerns without introducing additional concepts in the design language. Figure 5.5 shows the two layers of the crash-tolerant anti-intrusion application, using SCC notation. On the left, the application logic is illustrated: there are four entities, a `Timer`, a `Camera`, an `Alarm` and a `Database`, two context components, `ImageProcessing`

and `Intrusion`, and an `AlarmController`. When the `ImageProcessing` context component receives a `signal` from the `Timer`, it accesses the images provided by the `Camera`. The image represents the raw data that is further processed by the `Intrusion` context to determine whether there is an intrusion. In case there is one, the `AlarmController` uses the `Trigger` action on `Alarm` entities and the `Log` action on the `Database` entity. As the diagram shows, the safety expert has considered that the `Camera` and `Alarm` entities are required to tolerate crash faults. On the right, the supervisory layer is depicted. The `power` source is used by the `FTStrategy` context component to determine whether the camera is on main power supply or on battery and triggers a transition between FTMs based on this information. The transition is operated by the `AdaptationController` component that triggers the `AdaptFT` action on `Camera`.

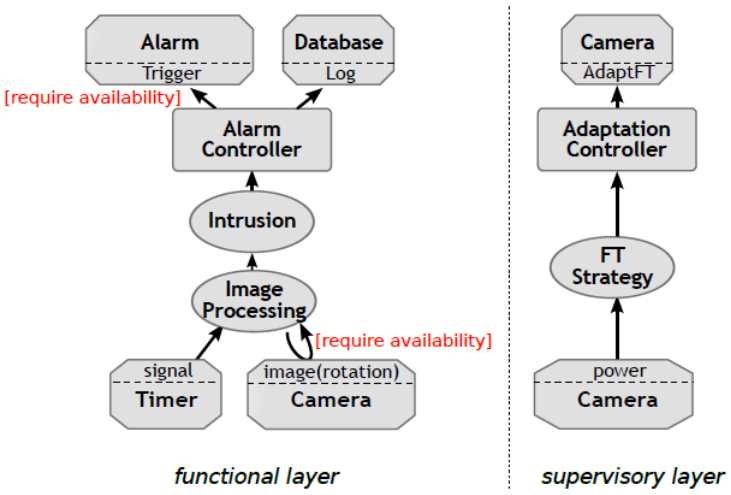


Figure 5.5: Functional and Supervisory Layers of the Anti-Intrusion Application

The FT declarations **require availability** are compiled into programming constraints to ensure the traceability of the requirements. From a DiaSpec description, a programming framework is generated to guide and support the application developer (step ③ in Figure 5.4). The design is leveraged to automatically generate a deployment framework that guides the integrator to combine the functional and non-functional developments step ⑤ in Figure 5.4. The generative approach [Cassou *et al.*, 11a], ensuring the conformance between the design, the implementation and the deployment, is out of the scope of this thesis. The methodology and toolkit are designed and developed by the Phoenix research group from INRIA Bordeaux, the initiators of the SERUS project.

As the reader has surely intuited, our role in this project is determinant in steps ② and ④ in Figure 5.4. Our contribution consists in providing the frame of reference for describing existing FTMs, the fault taxonomy and, most importantly, the library of component-based FTMs. Figure 5.6 shows the result of integrating the component-based architecture of LFR in the intrusion-detection SCC-based application. The server component from the initial architecture that only served as a dummy-component for testing the FTM is replaced by the `Camera` entity. The client from the initial architecture is, in this example, the `ImageProcessing` con-

text component that requests data from the crash-tolerant master-camera.

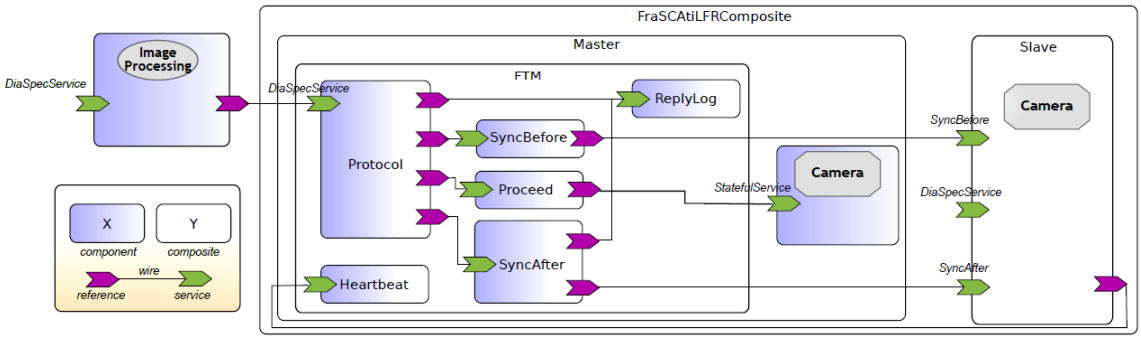


Figure 5.6: FRASCATI-Based Architecture of a Crash-Tolerant Camera Using LFR

5.3.5 Lessons Learned

Our participation in the SERUS Collaborative Research Action was very fruitful. On the one hand, it provided the context for us to become familiar with the FraSCAti middleware developed and maintained by the ADAM Project from Lille and illustrated the interest and usability of this platform in a new context, that of dependable computing. On the other hand, the integration of our adaptive fault tolerance mechanisms in DiaSuite demonstrates that they can be easily reused in an overall development process. Furthermore, DiaSuite was enhanced with fault tolerance abstractions and mechanisms, which represents an asset. Although only one transition was illustrated for the proof of concept, several interesting perspectives exist. First, the integration of other transitions triggered by changes in resources and fault models should consolidate the approach. Next, by further developing the considered example (i.e., the anti-intrusion application), changes in the business logic can be introduced, e.g., storing a video instead of a picture, which would constitute new adaptation triggers.

5.4 Integrating AFT in WSN-Based Applications

5.4.1 Motivation and Context

During the past few years, Wireless Sensor Networks (WSNs) have attracted a significant amount of interest, both from industry and academia. They are used in a plethora of applications of pervasive/ubiquitous computing for detecting and measuring physical properties of interest. The actual devices range from basic nodes sending a boolean value (e.g., metal detectors) to more sophisticated nodes, embedding several different sensors and processing capabilities (e.g. Sun SPOTs[2]). In general, WSNs have a heterogeneous structure. Despite their various domains of applicability, they share common challenges such as resource constraints, scalability and dynamics of the environment. Given their presence in the immediate vicinity

[2] `http://www.sunspotworld.com`

of humans and the economic stake inherent in many WSNs-based applications, fault tolerance must also be carefully considered [Chetan *et al.*, 05].

WSNs share common failure causes with traditional wired and wireless distributed systems and also introduce new ones [Paradis and Han, 07], especially related to the environment in which they are deployed and to their limited resources. Failures occur at different layers of the system [de Souza *et al.*, 07], ranging from node hardware failures, to sink failures (sinks are nodes with more resources that collect data from several elementary sensors and forward them, possibly after some intermediate processing), to failures of the actual application that receives input from sensors, due to software faults. As many networks use multi-hop routing, a lot of effort has been devoted to designing algorithms for fault-tolerant routing, in order to avoid network partitioning in case of node failure. Tolerance to crash faults at node level is intrinsic to networks in which several sensors provide the same functionality. In case one of them crashes, readings (and, possibly, alternate routing) will be provided by its neighbors and the node will be either rebooted, repaired or replaced. To tolerate faults on more complex nodes, that run stateful applications based on input from sensors, more complex fault tolerance mechanisms such as the ones presented in this work can be readily used.

In this section, we illustrate the applicability of our adaptive FTMs in a scenario based on a parking structure management application. Secondly, we describe the integration of our library of adaptive FTMs in Srijan [Pathak and Prasanna, 08], a toolkit for sensor network macroprogramming having previously no notion of fault tolerance. These contributions were developed in the context of the ANR MURPHY research project that included partners from academia (INRIA, CNAM) and from industry (SmartGrains[3]).

5.4.2 Application Scenario

The scenario is based on a parking structure management application deployed by SmartGrains. The role of this application is to reduce traffic, fuel consumption and pollution in a multi-floor parking structure. Figure 5.7 shows the hierarchical structure of the entities deployed for this purpose. Each parking slot contains a sensor that monitors its occupancy status. Parking slots are grouped in rows and the status of each row is monitored and stored by a *totem* that receives inputs from the elementary parking slot sensors. Each totem displays the number of available slots in its row. All totems on one floor communicate their status (i.e., number of available slots and other useful data such as the duration of occupancy) to a floor manager that displays the total number of available slots on its floor. All floor managers forward information to the parking manager, who is the person in charge of monitoring and maintenance operations. The total number of available parking slots is displayed at the entrance of the parking. The status of the parking (open/closed, number of available slots) is available through a web service that car drivers can access through their smartphone.

Parking slot sensors are resource-constrained entities that send a boolean representing their occupancy status. According to their position, they either communicate their status directly to their corresponding totem or they forward their information to their neighbors (i.e., multi-hop routing). They have a limited battery life. Furthermore, they can be subject to interference in

[3] `www.smartgrains.com`

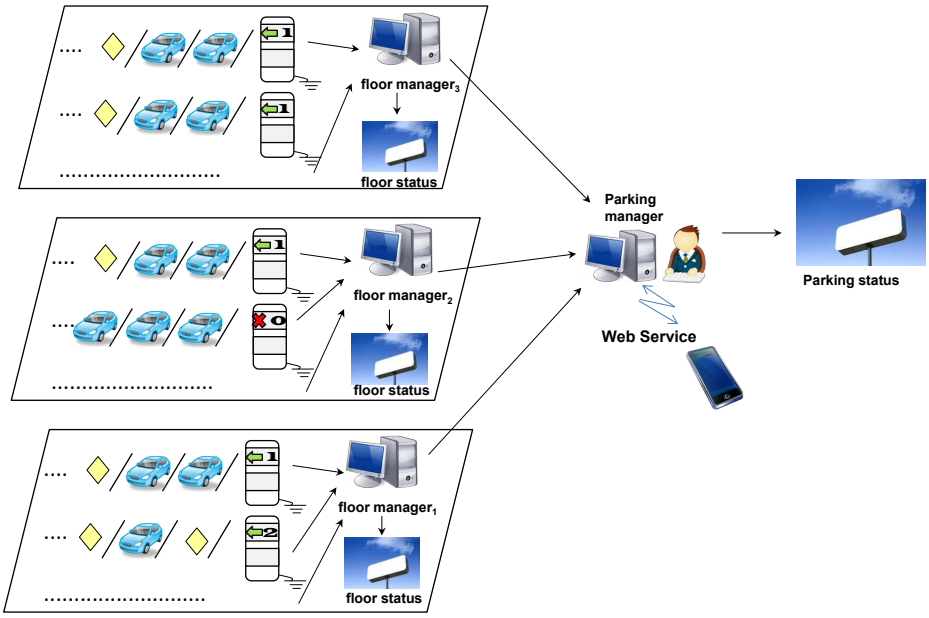


Figure 5.7: Scenario of the Multi-Floor Parking Structure Management Application

the presence of large vehicles and can be affected by high temperatures during the day. Totems are more resourceful and are capable of directly requesting the status of any sensor in their row. They are connected to the main power supply but, in case of power outage, they run on battery. Totems feature a main display and a secondary one.

5.4.3 Adaptive FTMs in the Application Scenario

Totems are stateful as they store and display the status of the rows they manage. In case they exhibit incorrect behavior, such as displaying incorrect values and forwarding them to the floor manager, the application requirements will be violated. Anomalous behavior such as displaying false positives (displaying available slots when there are none) or false negatives (declaring a row full even if there are available slots) leads to financial losses and customer dissatisfaction. For all these reasons, totems should be fault-tolerant. The main fault model to tolerate is crash: a totem that becomes non-responsive due to crash cannot account for the slots in its row. This has a high impact on the global status displayed at the entrance of the parking and on the floor status. To tolerate crash faults, totems work by pairs: a totem has two roles, acting as master for its row and as slave for the adjacent one (we assume there is an even number of rows on each floor). If a totem crashes, its partner becomes in charge of both rows, as shown in Figure 5.8 (IRP stands for Inter-Replica Protocol). On the main display, it shows the number of available slots in the row it is normally in charge of. On the secondary display, it shows the number of available slots in the row of its crashed partner. This is a degraded mode of operation: the totem may exhibit latency in updating information related to the secondary row as it is farther placed from the corresponding slot sensors.

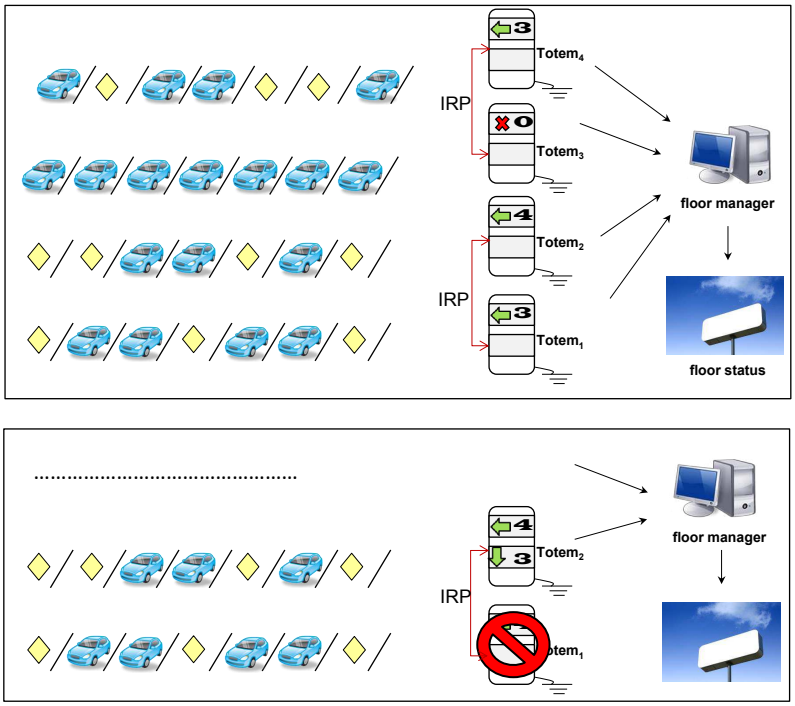


Figure 5.8: Integrating Fault Tolerance in the Application Scenario

Based on this duplex configuration, the FTMs presented in this work can be readily integrated. In the initial configuration, the pairs of totems work in Primary-Backup mode. A checkpoint consists in the occupancy vector of the row in question and information such as duration of occupancy of the slots for computing statistics.

Several adaptation triggers can be illustrated through this application scenario. If there is a bandwidth drop on the connection between the two partners, checkpoints can be sent less frequently or the configuration can execute a transition to Leader-Follower mode. Conversely, if there is a power outage and the totems must run on battery, the configuration must switch back to Primary-Backup.

Furthermore, changes in the fault model can require the composition of the initial FTM with another one that tackles transient value faults. These changes have physical causes. On the one hand, sensor hardware aging, exhaustion of the sensor battery and electromagnetic interferences result in erroneous readings. On the other hand, high temperatures have a negative impact on the reliability of the sensor readings. Totems can detect a faulty sensor by comparing its readings on a time interval. Furthermore, a totem can directly request the status from each of its sensors for mechanisms such as Time Redundancy.

The proposed approach for fine-grained transitions between FTMs consisting of three steps (i.e., download package, execute script, remove residual components) can be used as already shown. However, in this scenario, the totems being numerous and expected to react quickly to change, it may be more convenient to equip them, from the beginning, with the capacity to operate in various FTM configurations. This eliminates the downloading and removal step from the process, at least for predictable transitions, for which the totem is prepared from the

beginning. Obviously, other transitions that were not included in the initial set can be operated as well, using the initial process. In the following, we discuss this alternate adaptation process that is to be implemented on totems.

Figure 5.9 illustrates the alternate adaptation process. FTMs are equivalent to sets of interconnected components. In the alternate approach, the union of several FTMs is deployed from the beginning on each totem. Thanks to the clear identification of common parts and variable features between FTMs, the union is much smaller (in terms of number of components) than a simple juxtaposition of FTMs, as already demonstrated. At any point in time, a single FTM is "active", which is equivalent to a certain component-based configuration.

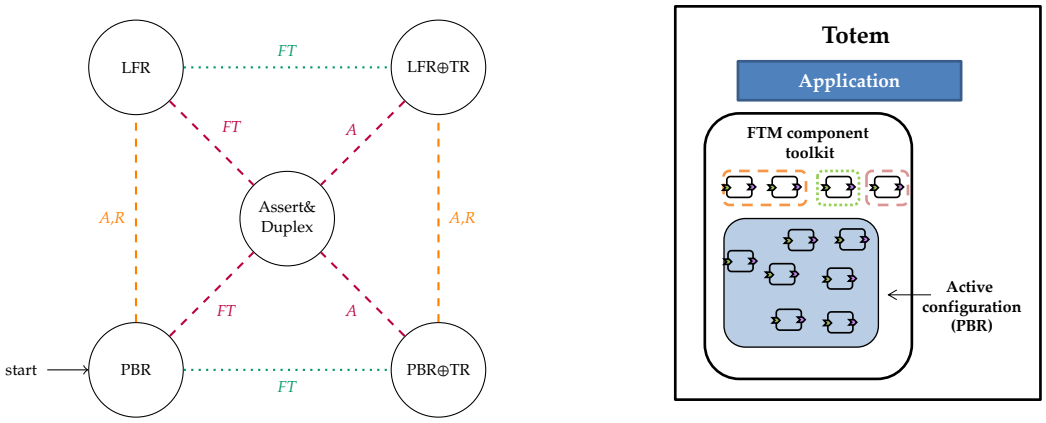


Figure 5.9: Transition Scenario (left) and FTM Component Toolkit Deployed on Totems (right)

Let us consider, for instance, that the totem is expected to switch during its service life between PBR, LFR, PBR⊕TR, LFR⊕TR and Assert&Duplex. The initial configuration is PBR. The union of all these FTMs that must be loaded contains their intersection and the variable features, namely: $\texttt{syncBefore}_{PBR}$, $\texttt{syncBefore}_{LFR}$, proceed, $\texttt{proceed}_{TR}$, $\texttt{syncAfter}_{PBR}$, $\texttt{syncAfter}_{LFR}$, $\texttt{syncAfter}_{Assert}$. In Figure 5.9, the active configuration is PBR, which contains the intersection of the requested FTMs and the variable features corresponding to PBR ($\texttt{syncBefore}_{PBR}$, proceed, $\texttt{syncAfter}_{PBR}$). The proceed component is common to PBR, LFR and Assert&Duplex, therefore it has no specific index, as opposed to $\texttt{proceed}_{TR}$. Although not specifically shown, the FTM toolkit must also contain the components implemented in FScript that contain the actual transition scripts. The transitions on the left of Figure 5.9 are highlighted in the same color as their corresponding components on the right. For example, PBR→PBR⊕TR and LFR→LFR⊕TR are represented in green dotted lines, that correspond to one component in the toolkit, namely $\texttt{proceed}_{TR}$.

5.4.4 Macroprogramming Toolkit

In order to map our adaptive FTMs on target systems such as the totems in the application scenario, we integrate them in a programming model and accompanying toolkit dedicated to WSNs. In the context of the MURPHY research project, the FTMs presented in this work were integrated in a data-driven macroprogramming language called Abstract Task Graph

(ATaG) [Pathak and Prasanna, 11] and its Srijan toolkit [Pathak and Prasanna, 08]. In the following, we briefly present ATaG and Srijan and discuss the integration process.

Programming Model

In the macroprogramming philosophy, WSN-based applications are designed and developed at system level, as opposed to node level. As a result, the application developer can abstract away the intricacies of the underlying heterogeneous distributed system and focus on business logic. This enables domain experts such as biologists and city planners to write their WSN-based applications. Applications written in ATaG feature three main types of entities that interact in order to detect and measure physical properties of interest, compute decisions based on this data and act on the environment accordingly. This so-called *sense-compute-actuate* interaction pattern is identical to the Sense/Compute/Control architectural pattern [Taylor *et al.*, 09] mentioned earlier in this chapter.

To illustrate the use of ATaG, we show in Figure 5.10 a fragment of the parking management application, namely the interaction between slot sensors on one parking row/alley and their totem. The three types of actors in the *sense-compute-actuate* pattern are:

- **Abstract Data Items** are the main currency of information in an ATaG program (data-driven language). They represent the information in its various stages of processing inside a WSN. In this application, slot sensors produce the VehiclePresence data item. Based on their inputs, totems then produce the AlleyTotal data item;
- **Abstract Tasks** represent the processing performed on the abstract data items in the system. In this application, there are three types of tasks: VehicleSampler (i.e., a slot sensor), TotalCalculator (i.e., the application running on the totem) and AlleyTotalDisplayer (i.e., the display of the totem). Tasks are annotated with *instantiation rules*, specifying where they can be located (e.g., VehicleSampler is deployed on every node that has a magnetometer sensor attached for detecting the presence of a vehicle; TotalCalculator must be deployed "oncein(Alley)", i.e., there is one such task per parking row, regardless of the number of hardware devices). Furthermore, tasks are annotated with *firing rules*, specifying whether the task is triggered periodically (e.g. "periodic:10") or due to the production of certain data item(s) ("anydata", i.e., whenever data is available);
- **Abstract Channels** connect tasks to the data items consumed (i.e., ascending channel) or produced (i.e., descending channel) by them. Channels are annotated with *logical scopes*, expressing the interest of a task in a data item. For instance, "(0,Alley)" means that data should only be collected from the row/alley where the TotalCalculator is placed.

DiaSpec and ATaG

Admittedly, application description in ATaG is very similar to the one in DiaSpec (see Section 5.3), as the two languages describe similar interaction patterns. However, there are some key differences between the two approaches. DiaSpec and its DiaSuite toolkit focus on the development process, on the traceability of functional and non-functional requirements throughout the application lifecycle. ATaG and its Srijan toolkit present a particular concern for scale,

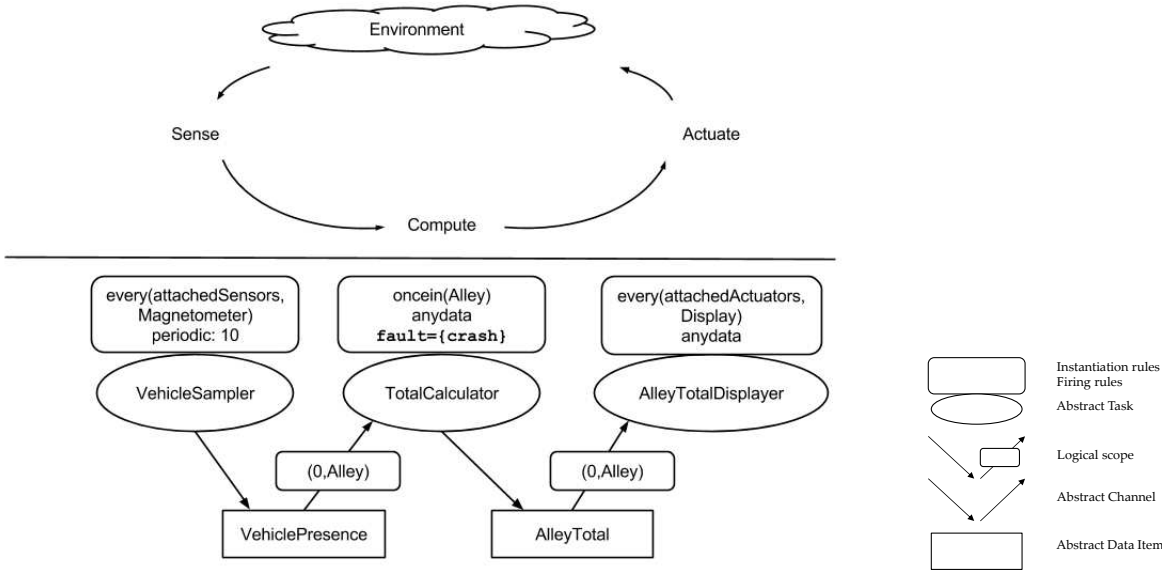


Figure 5.10: Task Graph Representing a Fragment of the ATaG Program for the Parking Management Application

an issue inherent in WSN-based applications, and for mapping tasks on physical devices. As a result, integrating our adaptive FTMs in the two approaches is not redundant but illustrates their usability in two different contexts and how they can enhance already existing toolkits that target different goals. While DiaSuite shows how adaptive fault tolerance can be integrated in a "disciplined" development process, Srijan brings us closer to performing real experimentation on already deployed systems.

Application Development Using ATaG and Srijan

Application development using ATaG is performed in two main phases through the Srijan toolkit:

- The application is first specified through a task graph (as the one in Figure 5.10). From this description, code templates are generated (abstract Java classes) and the application developer/domain expert complements each task with imperative code detailing the actions performed by the hosting device when the task is fired.

- A *network description* is provided to the ATaG compiler, consisting of the properties of each device in the target deployment. Based on this description and on input from the first phase, the compiler instantiates copies of the abstract tasks and assigns them to the devices on that target deployment.

Figure 5.11 shows this process. The first phase is illustrated on the left, and the second phase on the right. Clear arrows show inputs, while dark arrows show the output of each step in the process.

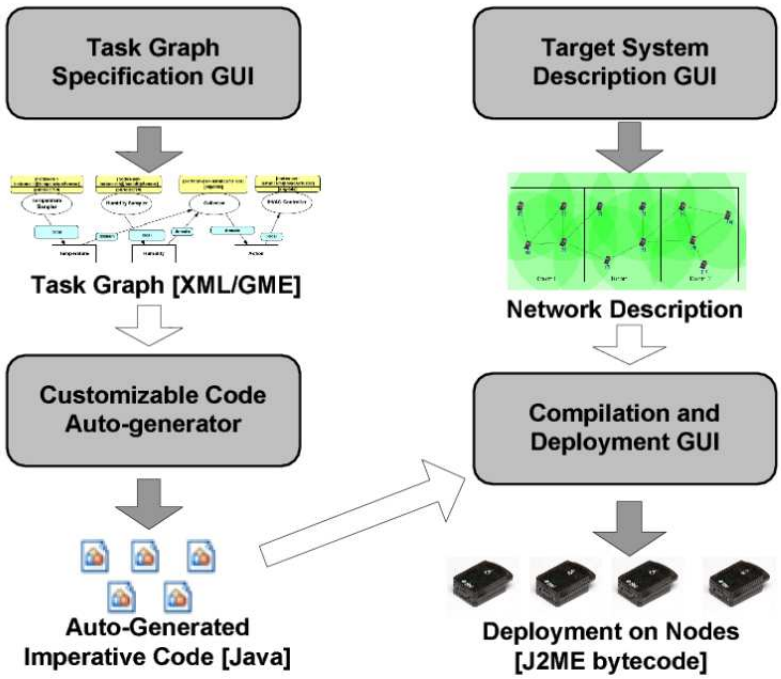


Figure 5.11: Overview of ATaG-Based Application Development Using Srijan

Incorporating (A)FT in ATaG Macroprograms

As already mentioned, there was no previous notion of fault tolerance integrated in ATaG. Similarly to the DiaSuite approach, the goal was to integrate adaptive fault tolerance while ensuring a clear separation of concerns. As developers of applications in ATaG are not expected to tackle the intricacies of the underlying distributed physical system, the addition of fault tolerance should not demand extra effort from them either.

To make a task fault-tolerant, an optional parameter whose value represents the fault model to tolerate was added to the abstract task in the application description stage. In Figure 5.10, the TotalCalculator task (running on the totem) has the "fault" parameter set to "crash". Both crash faults and value faults are tackled thanks to our FTMs. New annotations were introduced to the imperative code for the developer to identify the variables storing task state. The ATaG runtime was modified in order to intercept task firing and to redirect execution to our FTMs. The compilation process was also enhanced in order to tackle task replication. When it is not possible to instantiate extra copies (i.e., the description of the target system does not contain the extra physical node necessary for replication), the compiler issues a warning during the task-mapping process.

Figure 5.12 illustrates the process of developing fault-tolerant WSN-based applications with ATaG and Srijan:

1. We denote by F the overall set of possible faults tolerated by ATaG programs. The FTM developer provides to a central database details about his mechanism m_i, including: the subset of F that it can help tolerate, the number of physical nodes needed for it to work, specific implementation needed from application developers, dependencies or conflicts

with other FTMs, and path to the binary implementation of the FTM.

2. When a WSN application developer specifies the application in ATaG, he optionally specifies for each task T the requirements $R(T) \subseteq F$ of the faults that all instances of T must tolerate.

3. The Srijan toolkit provides feedback to the developer in case some of the faults in $R(T)$ need more information from T (e.g., annotation of state variables) in order to be tolerated.

4. The task graph is then instantiated on the network description provided by the deployment engineer. For each device in the deployment, the runtime system is customized according to the requirement of the task graph as well as the fault-tolerance protocols needed to satisfy the application requirements. In case of a mismatch (e.g., insufficient number of suitable nodes available in a certain region), the compiler notifies the deployment engineer.

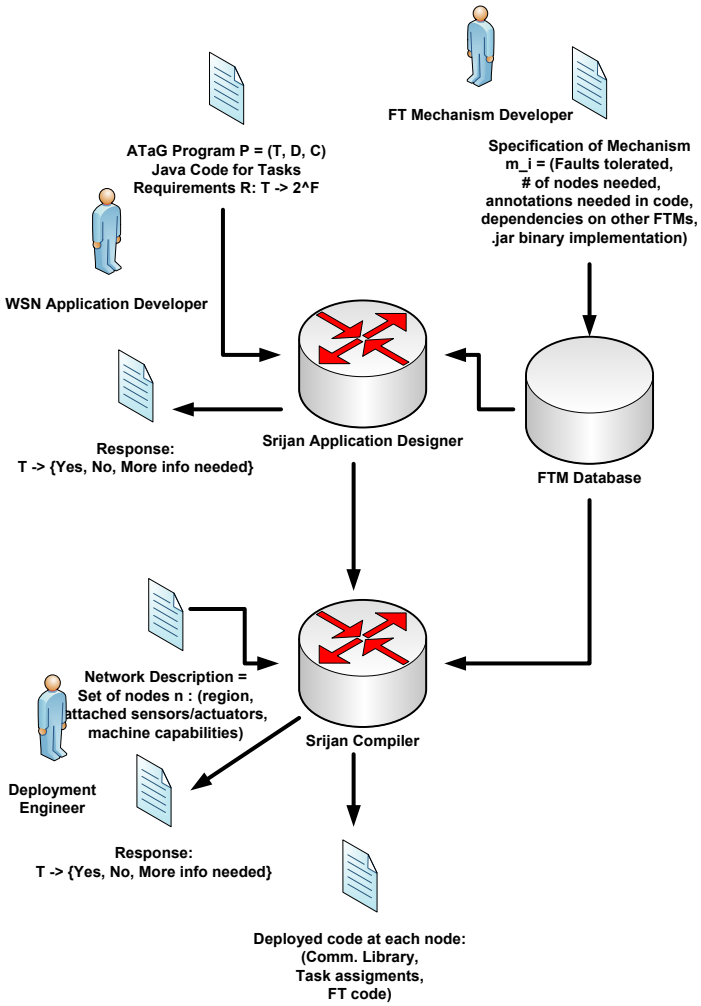


Figure 5.12: Overview of the Development Process of Fault-Tolerant WSN-Based Applications

As the reader must have intuited, one of our roles in this process was to provide the FTMs to the FTM database. We have also contributed to the identification of the elements featured in the specification of FTMs. Based on this description, other FTM developers may integrate their own solutions to the FTM database.

5.4.5 Lessons Learned & Work in Progress

In the context of the MURPHY research project, we examined a real WSN-based parking management system. Based on this data, we designed an application scenario that illustrates the interest and applicability of A(FT) for such targets. In order to map the presented fine-grained adaptive FTMs on these particular systems, we integrated them in Srijan, a macroprogramming toolkit. The FTMs are generic enough to be easily attached to WSN-based applications. Furthermore, we assisted the developers of Srijan in designing a specification of FTMs to ensure that other FT developers can integrate their own solutions (be they adaptive or not) in their repository. So far, Srijan and ATaG do not specify adaptation triggers, as opposed to DiaSpec that uses the Sense/Compute/Control paradigm both for application description and for the supervisory layer that triggers adaptation. On the other hand, Srijan enables us to perform real experimentation on physical systems, with many resource constraints. We are currently finalizing the integration of the adaptive FTMs in Srijan. The next step is to explicitly describe adaptation triggers and finally to develop the full application scenario.

5.5 Summary

This chapter focused on two fundamental aspects. On the one hand, we assessed the performance of our approach, firstly, through quantitative measurements of transitions between FTMs and, secondly, in terms of the agility of the modifications. This led us to pinpoint a linear relationship between a transition execution time and the number of modified components. More importantly, this assessment showed the overall feasibility of a performance-friendly and agile adaptation strategy. Comparisons to other existing approaches were also provided.

On the other hand, we illustrated in two different development projects the usability of our framework. During this process, we showed how to map the concepts related to adaptive fault tolerance in two different scenarios. First, we described the integration of the adaptive FTMs in DiaSuite, a design-driven development approach. Interestingly, the resulting methodology ensures the full traceability of dependability requirements along the application lifecycle, from its design up to the ability of handling explicit runtime adaptation. Next, we illustrated the benefits of adaptive fault tolerance in wireless sensor networks, a radically different development environment. We described the different steps that have been performed to integrate (adaptive) fault tolerance in a development environment tailored for resource-constrained wireless sensor networks, highlighting similarities and differences between such environments and DiaSuite-based development. We finally illustrated the resulting environment on the the scenario of a parking management application deployed using WSNs, enriched with adaptive fault tolerance capabilities. To bridge the gap between our component-based FTMs and this target system, we described the currently carried integration of our framework in the Srijan macroprogramming toolkit.

Conclusion & Future Work

> "Each of us is merely one human being, merely an experiment, a way station. But each of us should be on the way toward perfection, should be striving to reach the center, not the periphery."
>
> — Hermann Hesse, *The Glass Bead Game*

1 Conclusion

In computing, resilience denotes the capacity of a system to maintain its dependability properties despite various types of changes. In this thesis, an approach was proposed for tackling a key aspect of resilient computing, namely the systematic on-line adaptation of fault tolerance mechanisms (FTMs). Although adaptive fault tolerance (AFT) has attracted research efforts for some time now, existing solutions perform modifications of FTMs in a preprogrammed manner. This is not satisfactory for long-lived systems that operate in highly dynamic environments because it is impossible to predict all changes that might occur during their service life. As adaptivity has long been a prerogative of the business logic, a substantial body of knowledge exists in the field of software engineering. Consequently, we proposed an approach that leverages state-of-the-art tools and standards from this field. The proposed approach comprises four milestones: a change model frame of reference, a "design for adaptation" approach, mapping of the design on a reflective component-based middleware, and experiments to validate and illustrate the approach.

First of all, we identified three classes of parameters, namely fault tolerance requirements, application characteristics and resources, whose values dictate the choice of an appropriate FTM for a given application. The variation of these parameters can invalidate the initial choice and may require a transition towards a new FTM consistent with the new values. A change model frame of reference was established for visualizing the evolution of FTMs during the service life of a system. Based on this frame of reference, we proposed a classification of well-known FTMs and illustrated the goal of our work throughout several transition scenarios.

Next, we thoroughly analyzed a subset of FTMs with various underlying characteristics during a "design for adaptation" process. Through this analysis that consisted of several iterations, we revealed a generic protocol execution scheme capturing the variable features of FTMs. Inspired from aspect-oriented programming, we named it the *before-proceed-after* scheme. A second outcome of the "design for adaptation" process was a toolbox of fault tolerance design

patterns that can serve as a starting point for real-world applications in which new FTMs can be easily developed and integrated off-line. The generic execution scheme represented a cornerstone of our overall approach as it facilitated the composition of FTMs and prepared the ground for on-line fine-grained transitions.

Afterwards, we focused on the necessary infrastructure for performing on-line agile transitions between FTMs. To this aim, we identified a minimal API for runtime adaptation, representing the set of functionalities that must be provided by a component-based middleware. Then, we mapped the generic protocol execution scheme on FraSCAti, a reflective component-based middleware providing the required runtime control capabilities. During this development stage, the subset of FTMs were implemented as proof-of-concept and the fine-grained transitions in the illustrative scenario were performed. The fine-grained design and transition algorithms are reproducible on other platforms providing the minimal API.

Next, we assessed the performance of the approach, firstly, through quantitative measurements of transitions between FTMs and, secondly, in terms of the agility of modifications. Comparisons to already existing works were also provided. The main benefit of the proposed approach lies in the agile fine-grained nature of runtime transitions between FTMs as opposed to preprogrammed monolithic solutions. Furthermore, we illustrated the usability of our framework and, more generally, of the concepts related to adaptive fault tolerance in two different contexts, first in a design-driven development process and, second, in a scenario based on a parking management application using WSNs. The successful integration of our adaptive FTMs in two toolkits having no previous explicit notion of fault tolerance (let alone adaptive fault tolerance) demonstrates the genericity of the proposed approach.

The main lesson learned during this thesis is that fine-grained agile adaptation of FTMs can be achieved by combining the strengths of detailed design, performed with adaptivity in mind, and reflective component-based middleware. This can obviously be extrapolated to other non-functional concerns as well as to functional ones. The result is a development process for enabling the systematic on-line adaptation of FTMs through fine-grained modifications. The work presented in this thesis is essentially a contribution to the field of resilient computing. Given the significant overlap between the goals of resilience and the goals of self-healing systems, the proposed approach is also part of the broader spectrum of autonomic computing. Last but not least, our methodology illustrates the benefits brought by software engineering tools and concepts in the context of fault-tolerant systems.

2 Future work

Situated at the intersection of several areas, primarily, dependability and software architectures and, secondly, pervasive systems and autonomic computing, this thesis opens various research perspectives for achieving medium/long-term goals.

A medium-term goal is to map the library of adaptive FTMs to another component-based middleware providing the required minimal API. This would support the claim that our approach is reproducible on another support. Furthermore, it would provide the means to compare different implementations and to evaluate the effort necessary for mapping the approach to another support, for future reference.

Another medium-term goal is to push further the two examples of integration of the adaptive FTMs presented in Chapter 5. In the case of DiaSuite, this implies integrating the entire set of transitions currently available and developing applications that illustrate various adaptation triggers. In the case of Srijan, the next step is to develop the parking management application scenario.

A long-term goal is to develop the adaptation logic, more specifically, a monitoring framework and an adaptation manager which triggers adaptations between FTMs (with or without human intervention). This requires a thorough analysis of adaptation triggers, specifically for proactive transitions discussed in Chapter 2. The possible high-level triggers we have identified so far are the following:

- anticipation of operational phases;
- hardware aging that increases the number of transient faults and that can affect the software running on it resulting in a high number of exceptions;
- decrease in software reliability due to the introduction of a new software version (i.e., containing residual bugs).

The monitoring framework and the adaptation logic should then be plugged in the adaptive fault tolerance architecture presented in this thesis and illustrated on a complex running example, such as a simulated satellite or any autonomous system with strong dependability requirements.

Résumé

1 Introduction & Problématique

1.1 Concepts fondamentaux

L'évolution des systèmes informatiques pendant leur vie opérationnelle est incontournable. Les systèmes doivent évoluer pour s'adapter aux changements internes et externes, provenant de l'environnement ou de l'interaction avec les utilisateurs. En particulier, les systèmes sûrs de fonctionnement, c'est-à-dire dans le service desquels les utilisateurs peuvent placer une confiance justifiée [Laprie, 04], doivent évoluer pour s'adapter à des changements comme, par exemple, de nouveaux types de fautes ou la perte de ressources. Le défi de l'évolution est plus important dans ce contexte car une modification ne doit pas porter atteinte aux propriétés de sûreté de fonctionnement. Dans ce contexte, la *résilience informatique* a été définie [Laprie, 08] comme la persistance de la sûreté de fonctionnement en dépit des changements. La notion de résilience n'est pas propre à l'informatique mais figure dans divers domaines comme la psychologie, la science des matériaux ou l'écologie. Dans tous les domaines concernés, elle représente la capacité d'une entité de se remettre après avoir subi une perturbation.

D'une part, la résilience informatique couvre plusieurs aspects, dont l'adaptivité, c'est-à-dire la possibilité de faire évoluer un système pendant sa vie opérationnelle. D'autre part, la sûreté de fonctionnement informatique est basée sur quatre moyens principaux [Laprie *et al.*, 96] :

- la prévention des fautes vise à empêcher l'occurence ou l'introduction de fautes;
- la tolérance aux fautes vise à fournir un service à même de remplir la fonction du système en dépit des fautes;
- l'élimination des fautes vise à réduire la présence (nombre, sévérité) des fautes;
- la prévision des fautes vise à estimer la présence, la création et les conséquences des fautes.

Par rapport aux trois autres moyens, la tolérance aux fautes, qui s'appuie sur des mécanismes de détection des fautes et de recouvrement attachés à l'application, est une activité qui accompagne les systèmes tout au long de leur vie opérationnelle. À ce titre, l'adaptation des mécanismes de tolérance aux fautes (FTMs) à l'exécution s'avère un défi essentiel pour développer des systèmes résilients. Dans la plupart des travaux de recherche existants, l'adaptation des FTMs à l'exécution est réalisée de manière préprogrammé [Fraga *et al.*, 03, Lung *et al.*, 06,

Marin *et al.*, 01] ou se limite à faire varier quelques paramètres. Tous les FTMs envisageables doivent être connus dès la conception du système, déployés et attachés à l'application dès le début. Pourtant, les changements ont des origines variées et équiper un système a priori pour le pire scénario qui pourrait survenir pendant la vie opérationnelle est souvent impossible. Selon les observations pendant la vie opérationnelle, de nouveaux FTMs peuvent être développés hors-ligne, mais intégrés en-ligne, c'est-à-dire pendant que le système est en train de s'exécuter. Nous appelons cette capacité *adaptation agile*, contrairement à l'adaptation préprogrammée qui est plus répandue et qui confine le système à un scénario d'adaptation figé.

Dans cette thèse, nous présentons une approche pour développer des systèmes résilients flexibles dont les FTMs peuvent s'adapter pendant l'exécution de manière agile par des modifications à grain fin pour minimiser l'impact sur l'architecture logicielle. L'approche proposée met à profit des outils et des concepts issus du domaine du génie logiciel, comme les architectures orientées services [Chappell, 07, Marino and Rowley, 09, Margolis and Sharpe, 07], la programmation orientée aspect [Kiczales *et al.*, 97] et les intergiciels réflexifs à base de composants [Szyperski, 02], qui permettent l'observation, le contrôle et la reconfiguration de l'architecture logicielle pendant la vie opérationnelle des systèmes.

1.2 Un exemple illustratif

Les satellites sont des objets autonomes assurant des fonctions complexes avec des ressources limitées. La plupart des fonctions fournies sont critiques, au moins du point de vue économique. Comme décrit dans [James *et al.*, 10], en dépit d'amples campagnes de test, des bogues peuvent rester non-détectés dans le logiciel. Ces bogues, ainsi que des fautes internes et externes (des effets de rayonnement, des vibrations ou des fautes des opérateurs) peuvent compromettre les systèmes, par exemple lors des missions spatiales. Nous allons illustrer notre propos avec ce domaine applicatif dans lequel les systèmes sont autonomes, non réparables, à ressources contraintes et avec des moyens de communication limités. La tolérance aux fautes est utilisée depuis les années 60 à bord des véhicules spatiaux et elle est essentielle pour des systèmes avec une connexion au sol limitée, tels que les sondes autonomes, afin de contenir les éventuelles erreurs.

Un satellite a une durée de vie contractuelle pouvant aller jusqu'à plusieurs années (le plus souvent atteinte, voire largement dépassée dans la pratique). Une caractéristique importante en ce qui concerne sa maintenance et son évolutivité, c'est que la réparation sur place après le lancement est extrêmement difficile et coûteuse, si des problèmes surviennent. Il est impossible de prédire tous les problèmes auxquels un tel dispositif autonome pourrait se confronter durant sa vie opérationnelle, par exemple les pertes accidentelles de ressources, l'impact des fautes, le vieillissement du matériel. On ne peut pas anticiper des stratégies de tolérance aux fautes pour pallier des problèmes qui ne sont pas encore connus. L'évolution du logiciel embarqué, de la configuration et de l'intégrité des ressources doit aussi être prise en compte pendant la vie du satellite et peut changer certaines hypothèses faites lors du choix de la stratégie de tolérance aux fautes. Dans ce contexte, lorsque le besoin se présente, l'adaptation des mécanismes de tolérance aux fautes (ainsi que de la couche fonctionnelle), est exécutée de manière ad-hoc. Selon le cas, l'adaptation prend différentes formes, allant de l'application d'un patch qui met à jour une région de la mémoire, jusqu'au remplacement complet des procédures embarquées.

Les bénéfices apportés par notre approche peuvent être illustrés à travers l'exemple ci-dessus. Définir a priori tous les mécanismes de tolérance aux fautes (FTMs) envisageables et leurs combinaisons selon les défaillances et les menaces qui peuvent survenir pendant la vie du système est impossible (indécidable a priori, un problème NP-complet). D'ailleurs, embarquer un nombre trop élevé de mécanismes sur un équipement spatial serait inacceptable en raison des ressources de mémoire limitées. Notre approche vise à concevoir les FTMs comme un assemblage "Lego", en s'appuyant sur des concepts et des outils du génie logiciel tels que les architectures à base de composants et des supports d'exécution reconfigurables. Les applications sont initialement équipées de certains FTMs selon les spécifications non-fonctionnelles initiales. Le monitoring effectué à bord du satellite informe les ingénieurs au sol sur l'état actuel du satellite. L'approche proposée permet aux ingénieurs de concevoir une solution de tolérance aux fautes hors-ligne à tout moment au cours de la vie opérationnelle du satellite, de télécharger les "briques de Lego" nécessaires pour modifier le FTM initial et de les attacher à celui-ci pour mettre en place un nouveau FTM qui soit cohérent avec les nouvelles conditions opérationnelles. En résumé, selon le monitoring effectué à bord (ciblant les caractéristiques des fonctions, les ressources disponibles, la priorité entre les fonctions, etc), nous proposons une mise à jour différentielle des FTMs qui rend la gestion de l'ensemble du logiciel embarqué plus facile que s'il était représenté comme un seul bloc. Par rapport aux modifications ad-hoc des FTMs (et, plus généralement du logiciel embarqué), notre approche permet une adaptation méthodique. En s'appuyant sur les outils du génie logiciel, l'approche proposée fournit un moyen de développer systématiquement les briques nécessaires et de les intégrer dans l'architecture du logiciel embarqué existant.

Les bénéfices de l'approche proposée vont bien au-delà des missions spatiales de longue durée. La résilience est un défi aussi bien pour les petits appareils autonomes qui doivent opérer sans intervention humaine. C'est le cas, par exemple, des sondes sous-marines qui détectent les tremblements de terre ou des sondes de détection d'incendie dans les forêts. Ce travail est donc une contribution aux techniques de self-healing nécessaires à l'autonomic computing [Kephart and Chess, 03], ciblant de nombreux équipements autonomes non réparables qui feront partie de notre vie quotidienne dans un avenir proche .

1.3 Description générale de l'approche

La séparation des préoccupations [Dijkstra, 82] (en anglais, "separation of concerns"), est un principe généralement adopté pour la mise en œuvre des logiciels à haute modularité [Parnas, 72] qui permet le développement indépendant de la logique fonctionnelle et des préoccupations transversales [McKinley *et al.*, 04] telles que la qualité de service, la sécurité, la fiabilité, etc. Dans le cas des applications tolérantes aux fautes, les deux préoccupations évidentes sont les services fonctionnels et non-fonctionnels, à savoir le(s) mécanisme(s) de tolérance aux fautes.

La *réflexivité* [Maes, 87] a servi de support à la séparation des préoccupations. Cette propriété représente la capacité d'un programme à examiner sa structure (réflexion structurelle) et son comportement (réflexion comportementale) et, éventuellement, les modifier. La réflexivité se compose de deux activités: l'introspection, qui permet à l'application de s'observer, et l'intercession, qui permet à l'application d'agir sur elle-même.

Les architectures logicielles fondées sur ces principes se composent de deux niveaux d'abstraction où le niveau de base fournit les fonctionnalités requises (i.e., la logique métier) et le niveau supérieur, ou le méta-niveau, contient les mécanismes non-fonctionnels (par exemple, le FTM(s) [Fabre, 09]). Comme nous ciblons l'adaptation des FTMs, nous devons gérer la dynamique du niveau supérieur, qui peut avoir deux causes:

- Le niveau applicatif reste inchangé, mais le FTM doit être modifié soit à cause de l'évolution des demandes en tolérance aux fautes *FT* (par exemple, l'évolution du modèle de faute en raison de perturbations physiques) soit à cause des fluctuations des ressources disponibles *R* qui le rendent inadéquat ou, au moins, sous-optimal du point de vue de la performance;

- Les variations du niveau supérieur sont indirectement déclenchées par des modifications du niveau applicatif qui rendent le mécanisme de tolérance aux fautes inadéquat (i.e., des changements dans les caractéristiques de l'application *A*). Dans ce cas, les deux niveaux doivent exécuter une transition vers une nouvelle configuration.

L'adaptation des FTMs peut être considérée comme une troisième préoccupation, nécessitant ainsi un troisième niveau d'abstraction. Cette séparation présente les mêmes avantages que la séparation entre le niveau applicatif et le niveau non-fonctionnel, assurant la flexibilité et, par conséquent, la réutilisabilité et l'évolutivité des mécanismes d'adaptation [Redmond and Cahill, 02].

Notre approche pour mettre en œuvre cet élément essentiel de la résilience, à savoir l'adaptation en-ligne des FTMs, comprend plusieurs étapes. Tout d'abord, nous avons identifié les paramètres responsables du choix d'un FTM particulier parmi un ensemble de mécanismes. Les valeurs de ces paramètres indiquent, au moment de la conception, quel est le FTM le plus approprié à joindre à une application, si un tel FTM existe. La variation de ces paramètres peut invalider le choix initial et déclencher une transition vers un nouveau FTM, cohérent avec les nouvelles valeurs. Ces paramètres forment un référentiel qui permet la visualisation de la dynamique des FTMs. Une classification de FTMs classiques a été obtenue à partir de ce référentiel. En outre, il nous a fourni la base pour une série de scénarios de transition entre FTMs . Cette étape est résumée dans la Section 2.

Ensuite, nous avons analysé en détail un ensemble de FTMs selon leur structure et leur fonctionnement. Le but de cette analyse a été d'identifier leurs éléments communs et leurs points de variabilité. Inutile de dire, l'adaptation en-ligne à grain fin est extrêmement difficile, voire impossible, sans ce genre de préparation au préalable et sans les outils appropriés. Cette étape de préparation est ce que nous appelons la "conception pour l'adaptation", ce qui signifie que les FTMs considérés doivent être conçus de manière à isoler leurs caractéristiques communes et leurs points variables pour pouvoir les modifier en-ligne avec un impact minimal sur l'architecture logicielle. Cette approche présente des avantages non seulement pour l'adaptation à l'exécution, mais aussi pour le développement de FTMs statiques car elle fournit des briques de base qui peuvent être réutilisés et customisés par la suite. Le résultat de cette étape est un système de patrons de conception (en anglais, "design patterns") pour la tolérance aux fautes. L'efficacité de la méthode a été évaluée en mesurant l'impact du raffinement de

la conception/du design sur le temps de développement nécessaire à la construction de nouveaux FTMs. Ce travail est résumé dans la Section 3.

Une fois les éléments de variabilité entre FTMs clairement identifiés, l'étape suivante consiste à exploiter des outils spécifiques pour l'adaptation à l'exécution, à savoir un intergiciel réflexif à base de composants, et projeter les patrons de conception pour la tolérance aux fautes évoqués sur une architecture à base de composants. À ce stade aussi, les choix de conception sont extrêmement importantes car ils déterminent la granularité des modifications en-ligne : les caractéristiques variables doivent être projetées sur des entités/des composants individuels qui puissent être identifiés et modifiés à l'exécution, sinon la "conception pour l'adaptation" n'est pas mise à profit. Nous avons mis en œuvre nos scénarios de transition mentionnés précédemment afin de prouver la faisabilité de notre approche et d'illustrer la flexibilité apportée à nos FTM adaptatifs par ces outils provenant du génie logiciel. Les avantages de notre approche ont été évalués en mesurant l'impact des modifications différentielles sur l'architecture du FTM initial quant au nombre de composants remplacés et au temps de reconfiguration. La cohérence du processus d'adaptation, un aspect crucial car les transitions en-ligne doivent modifier les FTMs de manière atomique, a aussi été prise en compte. Cette étape est briévement décrite dans la Section 4.

Enfin, nous avons évalué la facilité d'utilisation de nos FTMs adaptatifs à travers deux exemples d'intégration. Dans le premier, nous avons intégré nos FTMs dans un processus de développement dirigé par la conception, dédié aux applications Sense-Compute-Control (et la suite logicielle sous-jacente nommée DiaSuite) qui ne contenait précédemment pas d'élément de tolérance aux fautes. Cet exemple prouve que nos FTMs adaptatifs sont génériques et peuvent être attachés à des applications externes fournies par différentes parties prenantes et peuvent être mis à profit dans différentes suites de développement logiciel. La méthodologie dirigée par la conception assure la traçabilité des exigences de tolérance aux fautes, de la conception jusqu'au déploiement et vise à aider les développeurs d'applications Sense-Compute-Control dans le processus d'intégration de la tolérance aux fautes dans leurs applications en la rendant la plus transparente possible. Dans le deuxième exemple, les FTMs adaptatifs ont été intégrés dans Srijan, un outil qui sert à développer des applications pour les réseaux de capteurs sans fil, basé sur le macroprogramming. Comme dans le premier cas, la plate-forme ne contenait initialement pas de notion de tolérance aux fautes. L'objectif a été, d'une part, de développer un langage de spécification des exigences de tolérance aux fautes et de décrire les FTMs à l'aide de ce langage et, d'autre part, d'intégrer les FTMs dans Srijan et de les spécialiser pour les applications ciblées. Ce travail est résumé dans la Section 5.

2 Adaptation des FTMs

2.1 Du modèle de faute au modèle de changement

Nous avons identifié trois classes de paramètres responsables du choix d'un FTM adéquat parmi un ensemble de mécanismes :

- les exigences de sûreté de fonctionnement de l'application et, plus particulièrement, les propriéés de tolérance aux fautes (*FT*) demandées;

- les caractéristiques de l'application (*A*);
- les ressources disponibles (*R*).

Dans nos travaux, *FT* couvre un ou plusieurs FTM qui sont évidemment fortement liés au modèle de faute et aux propriétés de sûreté de fonctionnement requises (fiabilité, disponibilité, intégrité, confidentialité). Les caractéristiques de l'application *A* ont une incidence sur la validié des FTM sélectionnés. Pour exécuter les mécanismes sélectionnés, un ensemble de ressources *R* est nécessaire et devrait être disponible. Le choix d'un FTM approprié pour une application donnée est basée sur les valeurs des paramètres *(FT, A, R)* et, à tout moment de la vie opérationnelle du système, les valeurs de ce triplet doivent être cohérentes avec le FTM courant afin de satisfaire les propriétés de sûreté de fonctionnement. Toute modification dans les valeurs de ces paramètres peut invalider le choix initial et exiger un nouveau FTM. Les transitions entre les FTM sont nécessaires lorsqu'il y a une incohérence entre le FTM actuel et les conditions opérationnelles.

À partir de ces trois classes de paramètres, nous avons proposé un système de référence [Stoicescu *et al.*, 11b], appelé le référentiel du modèle de changement, dont chaque axe correspond à une classe. Ce réfrentiel, illustré dans la Figure 2.1 à la page 30, nous permet de visualiser l'évolution d'une application tolérante aux fautes durant sa vie opérationnelle. Les trois axes du référentiel sont des variables multivariées:

1. *FT* : nous nous concentrons ici sur le modèle de faute
2. *A* : les caractéristiques de l'application peuvent invalider certains FTMs qui tolèrent le même modèle de faute :
   - le déterminisme
   - l'existence d'un état de l'application à restaurer en cas de défaillance
   - l'accès à l'état de l'application
3. *R* : les ressources nécessaires au FTM :
   - bande passante
   - énergie
   - ressources de calcul (CPU)

En détaillant les critères liés aux modèle de faute (pour lequel on se base sur [Avižienis *et al.*, 04]) et aux caractéristiques des applications, nous avons établi une classification d'un ensemble de FTMs bien connus [Stoicescu *et al.*, 11a]. Cette classification, représentée dans la Figure 2.2, à la page 32, peut servir dans le processus de sélection d'un FTM adéquat, au cours de la conception d'une application tolérante aux fautes.

2.2 Ensemble de FTMs considérés

Pour illustrer notre approche, nous nous sommes concentrés sur un ensemble réduit de FTMs parmi ceux qui sont représentés dans la figure 2.2 : deux variantes de duplex pour tolérer les

fautes matérielles par *crash* et deux mécanismes pour tolérer les fautes matérielles par valeur, à savoir *Time Redundancy* (redondance temporelle) et *Assertion&Duplex*. Dans ce qui suit, nous discutons à la fois la façon dont ils fonctionnent et leurs caractéristiques sous-jacentes quant aux paramètres (*FT,A,R*). La discussion est limitée à ce sous-ensemble de FTMs car ils représentent la base pratique pour le reste de ce travail.

La tolérance aux fautes matérielles par *crash*

Les fautes par *crash* dans les systèmes client-serveur peuvent être tolérées par des protocoles duplex en reproduisant le serveur sur deux ou plusieurs machines (maître et esclave(s)). Le *crash* du maître est détecté par une entité de type *heartbeat* ou *watchdog* et déclenche une action de recouvrement par laquelle l'esclave devient maître (ou l'un des esclaves est élu maître).

Il existe deux principaux types de protocoles duplex : passifs et semi-actifs (parfois aussi appelés actifs). *Primary-Backup Replication (PBR)* [Speirs and Barrett, 89] est une stratégie passive : le maître (*primary*) est le seul à traiter les requêtes du client et il envoie des points de reprise (*checkpoints*) contenant son état à l'esclave. Les points de reprise peuvent être envoyés à chaque requête ou périodiquement ou de manière différentielle. *Leader- Follower Replication (LFR)* [Barret *et al.*, 90] est une stratégie semi-active : toutes les répliques traitent les requêtes du client, mais seulement le maître (*leader*) lui envoie la réponse.

La tolérance aux fautes matérielles par valeur

Les fautes en valeur sont tolérées en répétant le traitement de la requête sur une machine ou en utilisant une stratégie duplex accompagnée par des tests d'acceptation/assertions ou par des exécutions parallèles, suivies par un vote sur les résultats. Plusieurs stratégies existent (comme le montre notre classification de la Figure 2.2 de la page 32) dont nous analysons deux. *Time Redundancy (TR)* (la redondance temporelle) tolère les fautes en valeur transitoires et nécessite un seul ordinateur hôte. Une requête est traitée deux fois et les résultats sont comparés. Si les résultats diffèrent, cela signifie qu'une faute transitoire a été détectée. La requête est traitée à nouveau et si deux résultats sur trois sont identiques cette réponse est envoyée au client. *Assertion&Duplex (A&Duplex)* tolère les fautes en valeur transitoires et permanentes. Comme son nom l'indique, il nécessite deux ordinateurs hôte et les fautes sont détectées en utilisant une assertion.

Caractéristiques (*FT,A,R*) des FTMs considérés

Les caractéristiques sous-jacentes de ces FTM, quant aux paramètres (*FT , A, R*) sont présentées dans le Tableau 1. Nous pouvons voir que, bien que PBR et LFR tolèrent le même modèle de faute, leurs caractéristiques *A* et *R* sont considérablement différentes : LFR exige le déterminisme du comportement de l'application (c'est-à-dire que des données d'entré identiques doivent produire des résultats identiques parce que les requêtes sont traitées par toutes les répliques), mais ne nécessite pas d'accès à l'état de l'application parce que chaque réplique construit son état local au fur et à mesure en traitant les requêtes. Inversement, PBR n'exige pas le déterminisme du comportement de l'application (car le maître les modifications de son état

aux esclaves à travers les points de reprise) mais impose la capture de l'état de l'application qui peut être une action très complexe. PBR consomme plus de bande passante que LFR en fonctionnement normal (c'est-à-dire, en absence des fautes) en raison des points de reprise. LFR consomme plus de ressources de calcul que PBR (et, par conséquent, plus d'énergie) car toutes les répliques traitent les requêtes, comme indiqué par le symbole ★dans le Tableau 1. TR nécessite l'accès à l'état de l'application pour le sauver avant le premier traitement de la requête et le restaurer entre deux calculs consécutifs. Comme TR fonctionne sur une seule machine hôte, ce FTM ne peut pas tolérer le *crash* et il ne consomme pas de bande passante. A&Duplex peut tolérer le *crash* et les fautes en valeur car deux machines sont utilisées pour exécuter les répliques. Contrairement à TR, ce FTM ne nécessite pas l'accès à l'état. Les mécanismes TR et A&Duplex nécessitent plus de puissance de calcul (donc plus d'énergie) que PBR à cause du traitement multiple des requêtes. Cependant, avec A&Duplex, la couverture de détection des fautes transitoires est inférieure à celle de TR. Cela illustre le genre de compromis qui doit être analysé au moment de choisir un FTM pour une application donnée.

Caractéristiques \ FTM		PBR	LFR	TR	A&Duplex
Modèle de faute	Arrêt (*crash*)	✓	✓		✓
	Valeur transitoire			✓	✓
	Valeur permanente				✓
Caractéristiques de l'application	Deterministe	✓	✓	✓	✓
	Non-deterministe	✓			✓
	Nécessite accès à l'état	✓		✓	
Ressources	Bande passante	★★	★	n/a	★
	CPU	★	★★	★★	★★

Légende

PBR=Primary-Backup Replication
LFR=Leader-Follower Replication
TR=Time Redundancy
A&Duplex=Assertion&Duplex

Table 1: Caractéristiques (*FT,A,R*) des FTMs considérés

Scénario de transition entre les FTMs

Pendant la vie opérationnelle du système, les valeurs des paramètres énumérés dans le Tableau 1 peuvent changer. Une application peut devenir non-déterministe lorsqu'une nouvelle version est installée. Le modèle de faute peut devenir plus complexe passant, par exemple, de *crash* à *crash* plus faute en valeur en raison du vieillissement du matériel ou de perturbations physiques dans l'environnement. Les ressources disponibles peuvent également varier : par exemple, la bande passante et/ou l'énergie peuvent baisser. La Figure 1 montre un graphe de transitions possibles entre les FTMs évoqués précédemment. Les sommets représentent les FTMs et les arêtes sont marquées avec le paramètre (*FT,A,R*) dont la variation déclenche la transition. La transition LFR→LFR est déclenchée par un changement dans les caractéristiques de l'application ou des variations dans les ressources disponibles, alors que la transition PBR→A&Duplex est déclenchée par un changement dans le modèle de faute considéré. Les transitions peuvent se produire dans les deux directions, selon la variation d'un paramètre. Dans le Chapitre 2, nous élaborons sur la différence subtile qui existe entre une transition et son inverse.

Le modèle de faute peut changer en devenant plus ou moins complexe. Sa variation nécessite, le plus souvent, une *composition de FTMs*. Par exemple, si au début nous tolérons seulement

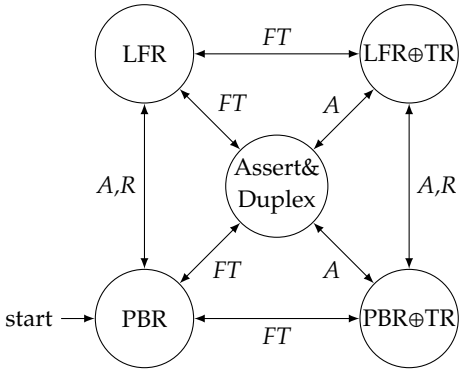


Figure 1: Transitions possibles entre les FTMs

les fautes par *crash* avec PBR et nous voulons ajouter la tolérance aux fautes en valeur transitoires, nous allons composer PBR avec TR et obtenir un FTM avec le comportement de PBR et de TR.

Dans ce travail, l'opérateur $\oplus$ désigne la composition de FTMs (par exemple, PBR⊕TR dans la Figure 1). Généralement, deux FTMs sont composés afin de tolérer les deux types de faute qu'ils ciblent individuellement.

3 Conception pour l'adaptation des FTMs

La conception pour l'adaptation, aussi nommée conception pour le changement ou pour l'évolution, est un problème fondamental lorsque l'on conçoit l'architecture d'un logiciel car les changements ne devraient pas affecter les fonctionnalités de base, sinon le logiciel sera difficile à maintenir et la prise en compte des changements dans les besoins sera très coûteuse ([Buschmann *et al.*, 96, p. 169]). Dans notre cas, l'objectif est de concevoir les FTMs de manière à faciliter leur évolution en ligne (à l'exécution) en minimisant le nombre de modifications structurelles et/ou comportementales à apporter lors des transitions entre différentes stratégies. La conception pour l'adaptation est un procédé itératif qui représente une étape de préparation avant les transitions à l'exécution proprement dites. Le procédé consiste en plusieurs boucles de conception et a comme résultats un schéma d'exécution générique des FTMs, reproductible tant dans la programmation orientée object (comme présenté dans cette section) que dans la programmation orientée composant (comme présenté dans la section suivante), et un système de patrons de conception pour la tolérance aux fautes.

3.1 Architecture initiale des FTMs

Pour illustrer notre approche, nous prenons comme point de départ le design (réalisé en UML) et l'implémentation (en C++) d'une stratégie PBR. Les spécifications initiales dont résulte cette implémentation étaient de développer un framework de FTMs qui tolèrent les fautes par *crash*. Le noyau de cette implémentation réside dans une classe qui englobe des fonctionnalités générales de la tolérance aux fautes, le comportement des mécanismes duplex et aussi

la logique spécifique de PBR. Il va sans dire que ce design est très difficile à faire évoluer de manière efficace. En commençant l'étape de conception pour l'adaptation, notre objectif a été d'arriver à une séparation nette entre toutes ces préoccupations afin de maximiser la réutilisation lors du développement de nouvelles variantes de duplex et d'autres FTMs.

3.2 Première itération de la conception pour l'adaptation

En analysant les FTM décrits ci-dessus, nous avons identifié un schéma d'exécution générique qui capte leurs caractéristiques communes et leurs caractéristiques variables. Á la réception d'une requête de la part du client, un serveur tolérant aux fautes exécute certaines actions *avant* le traitement (telles que la synchronisation avec une réplique). Ensuite, il *poursuit* avec le traitement de la requête. *Après* le traitement, il exécute certaines actions (telles que la synchronisation avec une réplique) et, enfin, il envoie la réponse au client. Nous appelons cela le schéma générique d'exécution *before-proceed-after* (avant-traitement-après), inspiré de la programmation orientée aspect [Kiczales *et al.*, 97].

Le Tableau 2 décrit le contenu de chaque étape du schéma d'exécution pour tous les FTMs considérés. Il est à noter que le fonctionnement de chaque FTM peut être décrit de manière très intuitive selon notre schéma. Lors de la conception d'une stratégie duplex, ce schéma peut être

FTM	Before	Proceed	After
PBR (Primary)	Rien	Traite requête	Checkpoint au Backup
PBR (Backup)	Rien	Rien	Traite checkpoint
LFR (Leader)	Envoie requête au Follower	Traite requête	Notifie Follower
LFR (Follower)	Reçoit requête	Traite requête	Traite notification
TR	Sauve état appli	Traite requête	Restaure état appli
A&Duplex	Rien	Traite requête	Vérifie assertion

Table 2: Schéma générique d'exécution des FTMs

traduit en *sync_before-proceed-sync_after*, car une synchronisation inter-répliques a lieu avant le traitement des requêtes et une autre après. Ce schéma d'exécution nous a permis de factoriser dans une classe mère ce qui est commun à tous les protocoles duplex, `DuplexProtocol` et ensuite spécialiser, par héritage, les FTMs proprement dits, `PBR` et `LFR` (voir le diagramme de classes de la Figure 2). D'autres variantes de duplex peuvent être ajoutées au framework, soit en héritant de la classe de base abstraite `DuplexProtocol` ou de l'une des classes concrètes.

Caractéristiques variables des FTMs

Les étapes de notre schéma générique d'exécution représentent les caractéristiques variables/-points de variation des FTMs. En comparant, par exemple, le fonctionnement de PBR avec celui de LFR (voir le Tableau 2), nous avons l'intuition que si on divisait le protocole inter-répliques dans des briques isolées qui puissent être identifiées et remplacées en-ligne, nous pourrions exécuter une transition *différentielle* entre PBR et LFR. De cette façon, nous ne remplacerions que les composants qui contiennent les caractéristiques variables entre les deux FTMs, sans

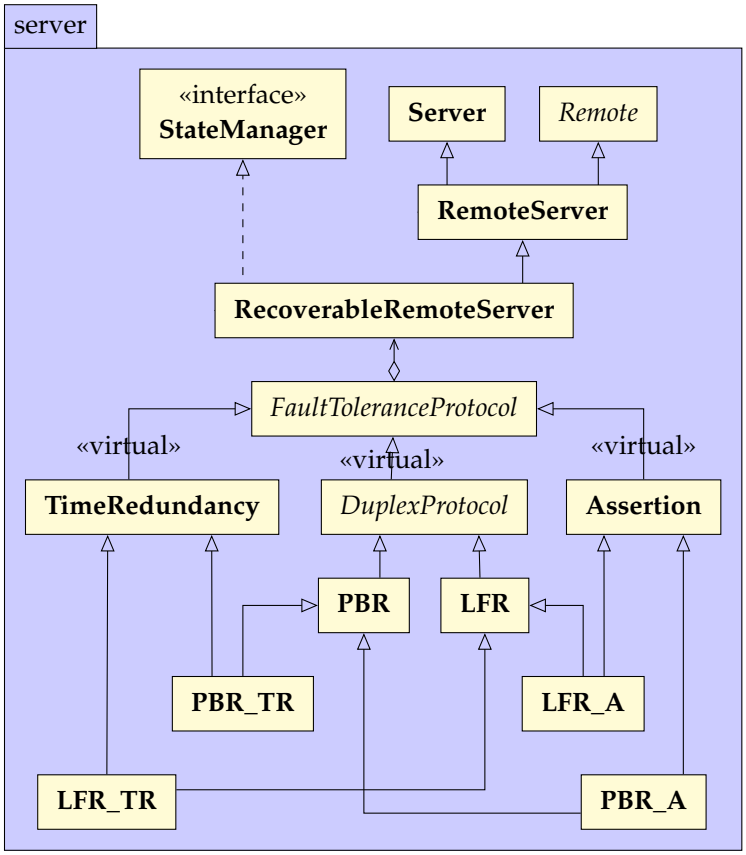


Figure 2: Extrait du framework de patrons de conception pour la tolérance aux fautes (paquet `serveur`)

modifier le reste du système (par exemple, la communication avec le client, le traitement de la requête). En identifiant les caractéristiques variables des FTMs, nous pouvons facilement développer hors-ligne des variantes à partir des FTMs existants et, lorsque l'on utilise un intergiciel reconfigurable à base de composants (comme montré dans la section suivante), exécuter les transitions entre FTMs en-ligne avec un minimum de modifications apportées au logiciel.

3.3 Deuxième itération de la conception pour l'adaptation

Une deuxième séparation des préoccupations peut être faite entre ce qui est commun à tous les FTMs et ce qui est propre aux stratégies duplex. La communication avec le client et le transfert des requêtes au service fonctionnel proprement dit (l'étape *proceed*) sont encapsulés dans une classe, `FaultToleranceProtocol` sur la Figure 2. Cette deuxième factorisation nous a permis d'introduire dans notre framework des protocoles non-duplex ciblant d'autres modèles de fautes que le *crash*, plus précisément les fautes en valeur (transitoires et permanentes) : `TimeRedundancy` et `Assertion` sur la Figure 2, qui suivent le même schéma d'exécution générique, comme déjà indiqué dans le Tableau 2.

De nouveaux protocoles peuvent être facilement ajoutés au framework soit en héritant de la classe de base abstraite `FaultToleranceProtocol` ou de l'une des classes concrètes. Ceci

est particulièrement utile dans le cas de `Assertion`, qui est dépend fortement de l'application.

3.4 Composition de FTMs

Comme conséquence directe des deux itérations du processus de conception pour l'adaptation, la composition des FTMs est intuitive et presque immédiate. En héritant d'un protocole duplex (qui tolère les fautes par *crash*) et d'un mécanisme de tolérance aux fautes en valeur, nous obtenons quatre FTMs composés (voir la Figure 2) : `PBR_TR` et `LFR_TR` qui correspondent à PBR⊕TR et LFR⊕TR respectivement du graphe des transitions de la Figure 1 et `PBR_A` et `LFR_A` qui sont deux variantes de A&Duplex. La Figure 2 montre un extrait de la version finale du framework de patrons de conception pour la tolérance aux fautes résultant des deux boucles de conception, à savoir le paquet `serveur`. Une évaluation des bénéfices apportés par la conception pour l'adaptation quant à la facilité de développer de nouveaux protocoles se trouve à la fin du Chapitre 2.

4 Architectures à base de composants des FTMs

Dans cette section, nous discutons l'implémentation à base de composants des FTMs adaptatifs. Cette étape fondamentale de notre travail prouve la faisabilité de l'approche en rassemblant les enseignements tirés du processus de conception pour l'adaptation et le support apportés par des outils modernes du génie logiciel.

4.1 Standards et outils

Les progrès récents dans le domaine du génie logiciel représentent l'un des catalyseurs de ce travail de recherche. Ces progrès prennent différentes formes allant de normes et spécifications jusqu'à des supports d'exécutions/intergiciels (en anglais, *middleware*) complexes offrant une pléthore de fonctionnalités. Dans ce qui suit, nous présentons les outils issus du génie logiciel que nous avons trouvés particulièrement utiles pour la réalisation de notre objectif.

Notre choix d'outils n'a pas été dicté par la nouveauté des solutions proposées. Nous nous sommes principalement intéressés à trouver une solution offrant les fonctionnalités que nous avions déjà identifiées comme primordiales pour nos fins, résumées dans ce que nous appelons "l'API minimale pour l'adaptation en-ligne des mécanismes de tolérance aux fautes".

Le standard SCA

Service Component Architecture (SCA) [Chappell, 07, Marino and Rowley, 09] est un ensemble de spécifications pour la construction d'applications distribuées faiblement couplées et facilement modifiables et personnalisables. Ces applications s'appuient sur les principes de la Service-oriented Architecture (SOA) et du Component-Based Software Engineering (CBSE) et utilisent un large spectre de technologies.

L'idée principale à la base de SCA est que les applications ont une structure hiérarchique basée sur des *composants* qui peuvent être inclus dans des *composites*. Les composants et les composites peuvent exhiber des *propriétés*, c'est-à-dire des paramètres de configuration,

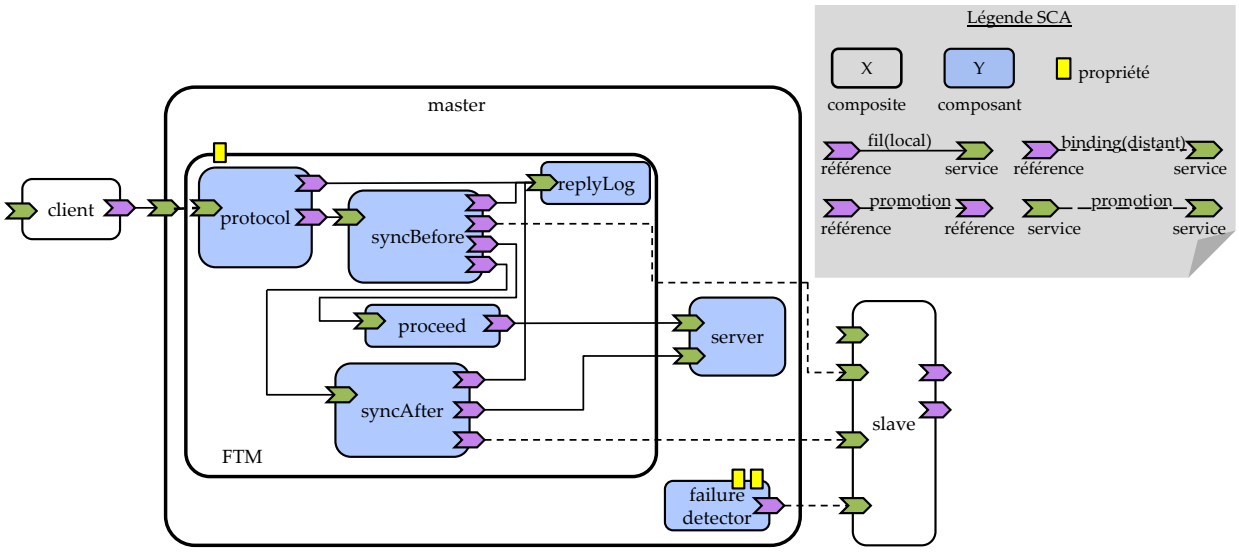


Figure 3: Architecture à base de composants de PBR

représentés sous forme de paires nom-valeur lues à l'exécution de l'application. Les composants fournissent et requièrent des *services*. Les services requis sont appelés des *références* et ils sont connectés au fournisseur du service effectif par des *fils*, *wires* en anglais, (dans le cas de connexions locales) ou par des *bindings* (dans le cas des connexions entre machines différentes). Les références et les services appartenant à un composant peuvent être rendus visibles au niveau du composite qui contient le composant par des liens de promotion. Les composites peuvent également agir en tant que composants à l'intérieur de composites plus grands (c'est-à-dire, un composant peut être soit un composant primitif soit un composant composite). La Figure 3 illustre ces notions sur le cas spécifique de l'implémentation à base de composants de PBR, qui sera discutée plus en détail. Les composites représentent l'unité de déploiement des applications et sont contenus dans une structure appelée *domaine*.

En séparant les interfaces (services et références) de l'implémentation proprement dite [Szyperski, 02], cette approche facilite la réutilisation, l'évolution et la combinaison de diverses technologies, puisque les composants consomment des services fournis par d'autres composants sans être "conscients" de la façon dont ils sont mis en œuvre et si l'implémentation change au fil du temps. Le modèle de programmation promu par SCA permet aux développeurs de construire des applications distribuées plus facilement. En s'abstrayant de la complexité du système distribué et des spécificités des technologies de communication sous-jacentes, le développeur peut se concentrer sur la logique des applications et les structurer dans des briques réutilisables et personnalisables.

FRASCATI et le support d'exécution pour la reconfiguration

Étant donné que SCA ne fournit qu'un ensemble de spécifications, les applications développées selon ses principes nécessitent un support approprié pour le déploiement et l'exécution. Il existe plusieurs plate-formes de ce type, commerciales ou bien open-source. FRASCATI est une plate-forme open-source d'origine académique qui fournit non seulement le support

du standard SCA mais va plus loin, en y rajoutant des fonctionnalités essentielles pour nos travaux. Selon [Seinturier *et al.*, 11], une omission fondamentale de la norme SCA est la gestion à l'exécution de l'application et de la plate-forme sous-jacente (le support adéquat), puisque la description de l'assemblage des composants est utilisée seulement pour instancier et déployer l'application. Étant donné que nos FTMs adaptatifs requièrent des capacités de reconfiguration après le déploiement de la part du support d'exécution, une simple implémentation de la norme SCA serait insuffisant pour nos besoins. FRASCATI gère la reconfiguration à l'exécution des applications à base de composants grâce à l'intégration des capacités de réflexivité de FRACTAL [Bruneton *et al.*, 06]. FRACTAL est un modèle à composants indépendant du langage de programmation conçu pour la construction de systèmes logiciels hautement configurables. En conséquence de cette intégration, les composants à l'intérieur des applications SCA qui s'exécutent au-dessus de l'intergiciel FRASCATI sont également dotés des CONTRÔLEURS spécifiques à FRACTAL pour la gestion du cycle de vie, l'introspection et la reconfiguration à l'exécution de l'architecture à base de composants. Il existe plusieurs moyens d'effectuer ces opérations sous FRASCATI. Pour nos expérimentations, nous utilisons son interpréteur de scripts de reconfiguration écrits en FSCRIPT [David *et al.*, 09]. Le langage de script FSCRIPT fournit du support pour les modifications atomiques sûres [Léger *et al.*, 10] de l'architecture des applications à base de composants, garantissant ainsi que l'opération de reconfiguration soit aboutit dans un état cohèrent (du point de vue des invariants architecturaux) soit laisse l'architecture non modifiée, dans son état initial, si une exception est levée. C'est une propriété essentielle pour nos transitions puisque l'ajout de cette dimension dynamique aux FTMs ne doit pas réduire la fiabilité des mécanismes.

L'API minimale pour l'adaptation en-ligne des FTMs

Avant même de choisir les outils et de réaliser nos expérimentations, nous avons identifié un ensemble de fonctionnalités qui doivent être fournies par le support d'exécution pour qu'il soit adéquat à nos objectifs. Toutes ces fonctionnalités sont liées au degré d'observabilité et de contrôle de l'architecture à base de composants fourni par la plate-forme d'exécution:

- *contrôle* sur le *cycle de vie* des composants à l'exécution (pour ajouter ou supprimer des composants, les démarrer ou les arrêter);
- *contrôle* sur les *interactions* entre composants à l'exécution (pour créer ou supprimer des fils ou des bindings entre les références et les services).

Ensuite, pour assurer la cohérence avant, pendant et après la reconfiguration, ces problèmes doivent aussi être pris en compte:

- les composants doivent être arrêtés dans un état de "quiescence" ou dormant, quand il n'y a plus d'opération en cours d'exécution à l'intérieur;
- les requêtes entrantes sur les composants arrêtés doivent être bufférisées jusqu'au redémmarage des composants en question pour assurer la cohérence du traitement.

Pour nos travaux, nous avons choisi FRASCATI (avec son interpréteur FSCRIPT) parce que cette plate-forme implémente l'API minimale que nous venons d'introduire. Cependant, il

est important de souligner que notre approche est parfaitement reproductible sur tout autre intergiciel avec ces fonctionnalités.

4.2 Architecture à base de composants de PBR

La Figure 3 montre l'architecture à base de composants d'une version simplifiée de PBR [Speirs and Barrett, 89]. Cette preuve de concept résulte de la projection de notre schéma d'exécution générique (*before-proceed-after*) présenté dans la Section 3 sur FRASCATI, selon les spécifications de SCA. Nous pouvons observer la séparation des préoccupations entre la logique applicative (le composant `server`) et le niveau non-fonctionnel. Ce dernier est composé du protocole inter-répliques (le composite `ftm` qui implémente, dans ce cas, la logique de PBR) et du mécanisme de détection des *crash* (le composant `failure detector`). Tous les composants représentés dans la Figure 3 sont implémentés en Java. L'aspect essentiel de cette architecture à base de composants est le fait que les étapes du schéma générique d'exécution sont contenues dans des composants individuels, bien déterminés (`syncBefore, proceed, syncAfter`) qui peuvent être modifiés à l'exécution, en faisant appel aux fonctionnalités fournies par FRASCATI.

4.3 Transitions entre FTMs à l'exécution

Ayant décomposé les FTMs décrits dans la Section 2 en "briques" élémentaires, nous nous appuyons sur les fonctionnalités de FRASCATI afin d'exécuter des transitions différentielles (à grain fin) entre les FTMs, en remplaçant uniquement les composants qui contiennent les caractéristiques variables entre le FTM final et le FTM initial. La Figure 4 présente la vue d'ensemble du processus de transition qui regroupe plusieurs entités et leurs interactions:

- le **Système** sur lequel s'exécute une application tolérante aux fautes, attachée à un FTM adéquat;
- le **Manager du système** qui observe le système afin de détecter des changements dans les valeurs des paramètres (*FT,A,R*). Lorsque le FTM courant n'est plus cohérent avec les nouvelles valeurs des paramètres, le manager appelle le Contrôleur d'adaptation en lui donnant comme paramètres d'entrée le nouveau FTM vers lequel une transition doit être exécutée;
- le **Contrôleur d'adaptation** appelle le Dépôt de FTMs adaptatifs pour récupérer le paquet de la transition en question. Ce paquet contient les nouveaux composants qui doivent être introduits dans l'architecture actuelle afin d'arriver, de manière différentielle, au FTM désiré et le script (en FSCRIPT) dans lequel s'exécute la mécanique de reconfiguration (suppression des composants qui ne sont plus nécessaires et introduction des nouveaux);
- l'**Interpréteur de scripts**, qui est en fait l'interpréteur FSCRIPT de FRASCATI, interprète le script contenu dans le paquet de la transition désirée et modifie le système. Les scripts sont construits à partir des actions élémentaires identifiées dans l'API minimale pour l'adaptation des FTMs;

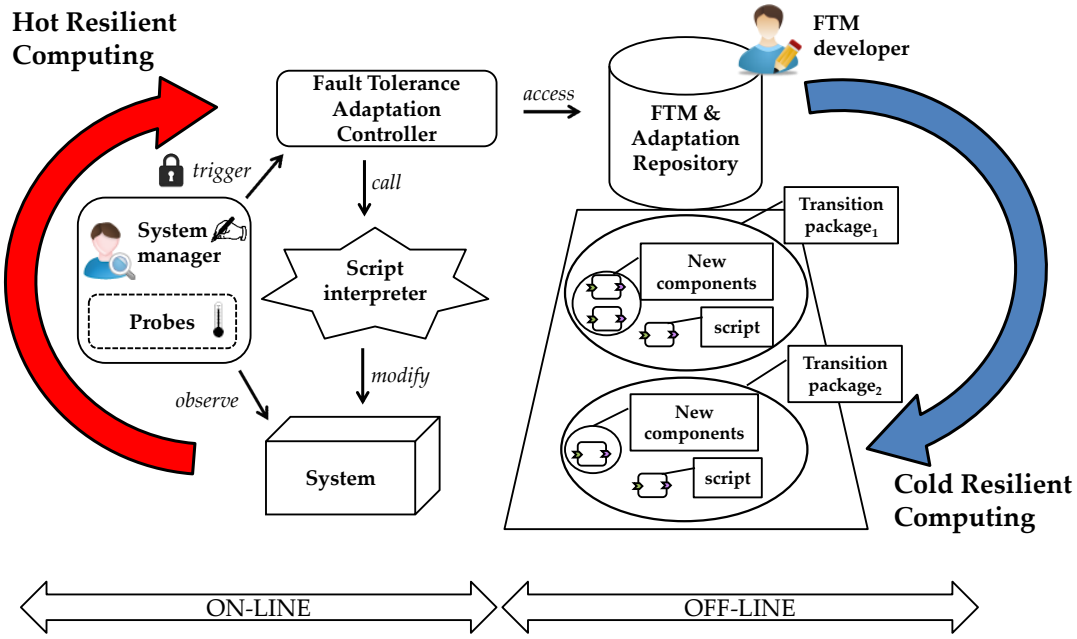


Figure 4: Aperçu du processus d'adaptation des FTMs

- le **Dépôt de FTMs adaptatifs** contient les paquets de transition. L'agilité promue par notre approche réside dans le fait que de nouveaux paquets peuvent être développés hors-ligne à tout moment pendant la vie opérationnelle du système et introduits en-ligne lorsque le besoin se présente avec un impact minimal sur l'architecture existante pour exécuter de nouvelles transitions, pas prévues au design.

Le Chapitre 4 présente en détail l'implémentation à base de composants de PBR et plusieurs transitions parmi celles du graphe de la Figure 1. Par exemple, la transition PBR→LFR requièrt, d'après l'analyse des caractéristiques variables du Tableau 2, le remplacement des composants `syncBefore` et `syncAfter` caractéristiques à PBR avec ceux de LFR.

5 Evaluation, intégration et application

Dans le Chapitre 5 de cette thèse, nous avons discuté deux aspects fondamentaux. Dans la première partie du chapitre, nous avons analysé les performances de notre approche, d'une part, à travers des mesures du temps nécessaire pour exécuter une transition et du nombre de composants modifiés et, d'autre part, en discutant l'agilité des modifications. Cette campagne de mesures nous a permis de trouver une dépendance linéaire entre le temps d'exécution d'une transition et le nombre de composants remplacés et, plus important, à travers cette analyse nous avons démontré la faisabilité de l'adaptation agile des FTMs. Nous avons également comparé les performances avec celles d'autres approches existantes.

Dans la deuxième partie, nous avons illustré l'intérêt et l'utilisabilité de notre approche. Dans ce but, nous avons montré comment appliquer les concepts liés à la tolérance aux fautes adaptatives dans deux scénarios différents . Tout d'abord, nous avons décrit l'intégration des FTMs adaptatifs dans DiaSuite [Cassou *et al.*, 11b], une approche de développement dirigée par

la conception. La méthodologie résultante [Enard *et al.*, 13] assure la traçabilité des exigences de sûreté de fonctionnement le long du cycle de vie de l'application, de sa conception jusqu'à la capacité de gérer l'adaptation à l'exécution. Ensuite, nous avons illustré les avantages de la tolérance aux fautes adaptative dans les réseaux de capteurs sans fil, un environnement de développement radicalement différent. Nous avons décrit les différentes étapes qui ont été effectuées pour intégrer la tolérance aux fautes (classique et adaptative) dans un environnement de développement dédié aux réseaux de capteurs sans fil, en soulignant les similitudes et les différences entre cet environnement et celui basé sur DiaSuite. Nous avons enfin illustré l'environnement résultant sur le scénario d'une application de gestion de parking déployée en utilisant des réseaux de capteurs et enrichie avec des capacités de tolérance aux fautes adaptative. Pour combler l'écart entre nos FTMs à base de composants et le système cible, nous avons décrit l'intégration de notre framework dans Srijan [Pathak and Prasanna, 08] un toolkit de macroprogrammation. Cette intégration est en train d'être finalisée.

6 Conclusion & Perspectives

6.1 Conclusion

La résilience informatique désigne la capacité d'un système à maintenir ses propriétés de sûreté de fonctionnement en dépit des changements. Dans cette thèse, une approche a été proposée pour s'attaquer à un aspect fondamental de la résilience informatique, à savoir l'adaptation en-ligne des mécanismes de tolérance aux fautes (FTMs). Bien que la tolérance aux fautes adaptative (AFT) ait déjà attiré l'intérêt depuis un certain temps, dans les solutions existantes, l'adaptation des FTMs est faite de manière préprogrammée. Dans ces approches, tous les FTMs qui représentent de potentiels candidats pour une application doivent être connus dès la conception du système et déployés dès le début. Cette méthode n'est pas satisfaisante pour les systèmes avec une longue vie opérationnelle dans des environnements très dynamiques, car il est impossible de prévoir tous les changements qui pourraient survenir. Comme l'adaptivité a longtemps été l'apanage du niveau fonctionnel des applications (qui implémente la logique métier), une quantité considérable de travaux existe dans le domaine du génie logiciel sur ce sujet. Par conséquent, nous avons proposé une approche qui s'appuie sur des outils et des standards provenant de ce domaine. L'approche proposée comporte quatre étapes : un référentiel pour illustrer la dynamique des FTMs et les paramètres qui en sont responsables, un processus itératif de "conception pour l'adaptation" des FTMs, le développement des FTMs sur un intergiciel réflexif à base de composants, et des expérimentations pour valider et illustrer la démarche .

6.2 Perspectives

Située à l'intersection de plusieurs domaines, à savoir la sûreté de fonctionnement et le génie logiciel en premier lieu, l'informatique ubiquitaire et les systèmes autonomiques dans une moindre mesure, cette thèse ouvre plusieurs perspectives de recherche.

Un objectif à moyen terme est de développer l'ensemble de FTMs adaptatifs sur un autre intergiciel à base de composants fournissant l'API minimale requise. Ce travail nous fournirait

les moyens de comparer différentes implémentations et d'évaluer les efforts nécessaires pour instancier l'approche sur un autre support. Un deuxième objectif à moyen terme est de pousser plus loin les deux exemples d'intégration des FTMs adaptatifs (dans DiaSuite et Srijan).

Un objectif à long terme est de développer la logique d'adaptation, plus particulièrement, un moniteur et un manager d'adaptation qui déclenche les transitions entre FTMs (avec ou sans intervention humaine).

Appendix

FTM	Fault model	Nb. of nodes	Application characteristics
PBR	Crash	2	Requires state access
LFR	Crash	2	Requires determinism
TR	Transient	1	Requires state access
A&Duplex	Crash Transient Permanent	2	May require determinism or state access
TMR	Crash Transient Permanent	3	Requires determinism
Comp&DD	Crash Transient Permanent	4	Requires determinism
RB	Transient Software	1	Requires state access, determinism, design diversity
NVP	Crash Transient Permanent Software	3,5,...	Requires determinism, design diversity
NSCP	Crash Transient Permanent Software	4	Requires determinism, design diversity

Legend
PBR=Primary-Backup Replication
LFR=Leader-Follower Replication
TR=Time Redundancy
A&Duplex=Assertion&Duplex
TMR=Triple Modular Redundancy
Comp&DD=Comparison&Double Duplex
RB=Recovery Blocks
NVP=N-Version Programming
NSCP=N-Self-Checking Programming

Table 3: Underlying Characteristics of the Entire Set of FTMs in Figure 2.2

Bibliography

[com] Component Object Model Technologies. http://www.microsoft.com/com/default.mspx.

[ejb] Enterprise Java Beans® 3.1. http://www.oracle.com/technetwork/java/javaee/ejb/index.html.

[OSG] OSGi Alliance. http://www.osgi.org.

[ftc, 02] OMG Fault-Tolerant CORBA. 2002. http://www.omg.org/cgi-bin/doc?formal/02-12-02.

[ccm, 11] OMG CORBA Component Model v4.0. 2011. http://www.omg.org/spec/CCM/4.0/.

[André *et al.*, 11] Pierre André, Jean-Charles Fabre, Camille Fayollas, Jérémie Guiochet, Matthieu Roy, and Miruna Stoicescu. A Simple Primary-Backup Replication Design Pattern: Development in UML and application. LAAS Report 11476, LAAS, Sep 2011. http://dbserver.laas.fr/pls/LAAS/publis.rech_doc?langage=FR&clef=118731.

[Armstrong, 94] L. T. Armstrong. ADAPTIVE FAULT TOLERANCE. Technical report, GE Aerospace Advanced Technology Laboratories, 1994. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA281251&Location=U2&doc=GetTRDoc.pdf.

[Avižienis, 67] Algirdas Avižienis. Design of fault-tolerant computers. *In Proceedings of the November 14-16, 1967, fall joint computer conference*, AFIPS '67 (Fall), pages 733–743, New York, NY, USA. ACM, 1967. http://doi.acm.org/10.1145/1465611.1465708.

[Avižienis, 85] Algirdas Avižienis. The *N*-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, (12):1491–1501, IEEE, 1985. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1701972&userType=inst.

[Avižienis *et al.*, 04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1:11–33, IEEE Computer Society Press, Los Alamitos, CA, USA, January 2004. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1335465.

[Barret *et al.*, 90] P.A. Barret, A.M. Hilborne, P.G. Bond, D.T. Seaton, P. Verissimo, L. Rodrigues, and N.A. Speirs. The Delta-4 extra performance architecture (XPA). *In Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 481–488. IEEE, 1990. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=89386.

[Batista *et al.*, 05] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. *In Proceedings of the 2nd European conference on Software Architecture*, EWSA'05, pages 1–17, Berlin, Heidelberg. Springer-Verlag, 2005. http://dx.doi.org/10.1007/11494713_1.

[Bharath *et al.*, 01] Roger Bharath, Melanie Dumas, and Mevlut Erdem Kurul. Adaptive Fault Tolerance in Distributed Systems. *Department of Computer Science University of California, San Diego La Jolla, CA*, 2001. http://charlotte.ucsd.edu/classes/wi01/cse221/OSSurveyW01/papers/rbharath,mdumas,mkurul.adaptive_fault_tolerance_in_distributed_systems.pdf.

[Bhatti *et al.*, 97] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. COYOTE: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16:321–366, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.3037.

[Bruneton *et al.*, 06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, Wiley Online Library, 2006. http://www.cs.colostate.edu/saxs/se/FractalComponent.pdf.

[Budhiraja *et al.*, 93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. *Distributed systems*, 2:199–216, Addison-Wesley, 1993. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.5896.

[Buschmann *et al.*, 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture*, volume 1: A System of Patterns. Wiley Chichester, United Kingdom, 1996. https://wiki.sch.bme.hu/images/9/98/Sznikak_jegyzet_Pattern-Oriented-SA_vol1.pdf.

[Cassou *et al.*, 11a] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. *In Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 431–440, New York, NY, USA. ACM, 2011. http://doi.acm.org/10.1145/1985793.1985852.

[Cassou *et al.*, 11b] Damien Cassou, Julien Bruneau, Charles Consel, and Emilie Balland. Towards a Tool-based Development Methodology for Pervasive Computing Applications. *IEEE TSE: Transactions on Software Engineering*, 2011. http://arxiv.org/abs/1203.6459.

[Chan *et al.*, 07] Pat. P.W. Chan, Michael R. Lyu, and Miroslaw Malek. Reliable Web Services: Methodology, Experiment and Modeling. *In Web Services, 2007. ICWS 2007. IEEE International*

Conference on, pages 679–686. IEEE, 2007. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4279659&tag=1.

[Chang *et al.*, 98] Ilwoo Chang, Matti A. Hiltunen, and Richard D. Schlichting. Affordable Fault Tolerance through Adaptation. *In Parallel and Distributed Processing*, pages 585–603. Springer, 1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.7628.

[Chappell, 07] David Chappell. INTRODUCING SCA. 2007. http://www.davidchappell.com/articles/Introducing_SCA.pdf.

[Cheng and Garlan, 12] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems*, 85(12), December 2012. http://www.sciencedirect.com/science/article/pii/S0164121212000714#.

[Chérèque *et al.*, 92] Marc Chérèque, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. Active Replication in Delta-4. *In The Twenty-Second Annual International Symposium on Fault-Tolerant Computing (Digest of Papers: FTCS-22)*, pages 28–37. IEEE, 1992. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=243618&tag=1.

[Chetan *et al.*, 05] Shiva Chetan, Anand Ranganathan, and Roy Campbell. Towards Fault Tolerant Pervasive Computing. *Technology and Society Magazine, IEEE*, 24(1):38–44, IEEE, 2005. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1407746.

[Computing, 06] Autonomic Computing. An architectural blueprint for autonomic computing. *IBM White Paper*, IBM, 2006. http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf.

[Coulson, 01] Geoff Coulson. What is Reflective Middleware? *IEEE Distributed Systems Online*, 2(8):165–169, 2001. http://www.comp.lancs.ac.uk/~geoff/Publications/RMARTICLE1.pdf.

[Coulson *et al.*, 08] Geoff Coulson, Gordon Blair, Paul Grace, François Taïani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A Generic Component Model for Building Systems Software. *ACM Trans. Comput. Syst.*, 26(1):1:1–1:42, ACM, New York, NY, USA, March 2008. http://dl.acm.org/citation.cfm?id=1328672.

[Crnković *et al.*, 11] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering*, 37(5):593–615, IEEE, 2011. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5587419&tag=1.

[Daniels *et al.*, 97] Fonda Daniels, Kalhee Kim, and Mladen A. Vouk. The Reliable Hybrid Pattern A Generalized Software Fault Tolerant Design Pattern. *In Int. Conf. PloP*, pages 1–9. 1997. http://130.203.133.150/viewdoc/summary;jsessionid=5F290EAC59945420527DD66FCB518994?doi=10.1.1.52.3351.

[David *et al.*, 09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, 64(1-2):45–63, 2009. http://link.springer.com/article/10.1007%2Fs12243-008-0073-y.

[de Souza *et al.*, 07] Luciana Moreira Sá de Souza, Harald Vogt, and Michael Beigl. A Survey on Fault Tolerance in Wireless Sensor Networks. *Interner Bericht. Fakultät für Informatik, Universität Karlsruhe*, 2007. http://ahvaz.ist.unomaha.edu/azad/temp/multipath/09-souza-wireles-sensor-networks-fault-tolerance-survey.pdf.

[Dijkstra, 82] Edsger W. Dijkstra. On the role of scientific thought. *In Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF.

[Enard *et al.*, 13] Quentin Enard, Miruna Stoicescu, Emilie Balland, Charles Consel, Laurence Duchien, Jean-Charles Fabre, and Matthieu Roy. Design-Driven Development Methodology for Resilient Computing. *In CBSE'13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*. 2013. http://dl.acm.org/citation.cfm?id=2465458.

[Fabre, 09] Jean-Charles Fabre. Architecting Dependable Systems Using Reflective Computing: Lessons Learnt and Some Challenges. *In WADS'09*, pages 273–296. 2009. http://link.springer.com/chapter/10.1007%2F978-3-642-17245-8_12.

[Feljan *et al.*, 09] Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petričić, and Ivica Crnković. Classification and survey of component models. Technical report, 2009. http://www.fer.unizg.hr/_download/repository/classification_dices_tech_report_1.0.pdf.

[Ferreira and Rubira, 98] Luciane Lamour Ferreira and Cecília Mary Fischer Rubira. Reflective Design Patterns to Implement Fault Tolerance. *In OOPSLA Workshop on Reflective Programming*. 1998. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.9709&rep=rep1&type=pdf.

[Fraga *et al.*, 03] Joni Fraga, Frank Siqueira, and Fábio Favarim. An Adaptive Fault-Tolerant Component Model. *In Proceedings of the Ninth IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*, pages 179–186. IEEE, 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.1124.

[Gamma *et al.*, 93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *In ECOOP '93*, pages 406–431. Springer-Verlag, 1993. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.2555.

[Gamma *et al.*, 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Garlan *et al.*, 00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural Description of Component-Based Systems. *In* Gary T. Leavens and Murali Sitaraman, editors. *Foundations of Component-Based Systems*, pages 47–67. Cambridge University Press, New York, NY, USA, 2000. http://www.cs.cmu.edu/afs/cs/project/able/ftp/acme-fcbs/acme-fcbs.pdf.

[Garlan and Schmerl, 02] David Garlan and Bradley Schmerl. Model-based Adaptation for Self-Healing Systems. *In Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 27–32, New York, NY, USA. ACM, 2002. http://dl.acm.org/citation.cfm?id=582134.

[Garlan *et al.*, 01] David Garlan, Bradley Schmerl, and Jichuan Chang. Using Gauges for Architecture-Based Monitoring and Adaptation. *In Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*. 12-14 December 2001. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.5333&rep=rep1&type=pdf.

[Gibert *et al.*, 12] Victor Gibert, Mathilde Machin, Jean-Charles Fabre, and Miruna Stoicescu. Design for Adaptation of Fault Tolerance Strategies. LAAS Report 12198, LAAS, Apr 2012. http://dbserver.laas.fr/pls/LAAS/publis.rech_doc?langage=FR&clef=120092.

[Goldberg *et al.*, 93] Jack Goldberg, Ira Greenberg, and Thomas F. Lawrence. Adaptive Fault Tolerance. *In Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–132. IEEE, 1993. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00588861.

[Gong and Goldberg, 94] Li Gong and Jack Goldberg. Implementing Adaptive Fault-Tolerant Services for Hybrid Faults. Technical report, 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.312.

[Gouda and Herman, 91] Mohamed G. Gouda and Ted Herman. Adaptive Programming. *IEEE Transactions on Software Engineering*, 17(9):911–921, IEEE, 1991. https://webspace.utexas.edu/goudamg/IEEE/00092911.pdf.

[Hecht *et al.*, 00] Myron Hecht, Herbert Hecht, and Eltefaat Shokri. Adaptive Fault Tolerance for Spacecraft. *In Aerospace Conference Proceedings, 2000 IEEE*, volume 5, pages 521–533. IEEE, 2000. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=878526.

[Heineman and Councill, 01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*, volume 17. Addison-Wesley Reading, 2001.

[Hiltunen and Schlichting, 93] Matti A. Hiltunen and Richard D. Schlichting. An Approach to Constructing Modular Fault-Tolerant Protocols. *In SRDS, Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 105–114. IEEE, 1993. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=393468.

[Hiltunen and Schlichting, 94] Matti A. Hiltunen and Richard D. Schlichting. A Model for Adaptive Fault-Tolerant Systems. *In EDCC, Proceedings of the 1st European Dependable*

Computing Conference, Lecture Notes in Computer Science, pages 3–20. Springer-Verlag, 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.3108.

[Hiltunen and Schlichting, 96] Matti A. Hiltunen and Richard D. Schlichting. Adaptive Distributed and Fault-Tolerant Systems. *Computer Systems Science and Engineering*, 11(5):275–285, 1996. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.3907.

[Hošek *et al.*, 10] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka, and Michal Malohlava. Comparison of Component Frameworks for Real-Time Embedded Systems. *In CBSE, Proceedings of the 13th International Symposium on Component-Based Software Engineering, Prague, Czech Republic*, pages 21–36. Springer, 2010. http://link.springer.com/chapter/10.1007%2F978-3-642-13238-4_2.

[Huebscher and McCann, 08] Markus C. Huebscher and Julie A. McCann. A survey of Autonomic Computing–Degrees, Models, and Applications. *ACM Computing Surveys (CSUR)*, 40(3), ACM, 2008. http://dl.acm.org/citation.cfm?id=1380585.

[James *et al.*, 10] Mark James, Paul Springer, and Hans Zima. Adaptive Fault Tolerance for Many-Core Based Space-Borne Computing. *In Euro-Par Parallel Processing, Proceedings of the 16th International Euro-Par Conference, Ischia, Italy*. Springer, 2010. http://link.springer.com/chapter/10.1007%2F978-3-642-15291-7_25.

[Kalbarczyk *et al.*, 99] Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Saurabh Bagchi, and Keith Whisnant. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, IEEE, 1999. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=774907.

[Kephart and Chess, 03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, IEEE, 2003. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055.

[Kiczales *et al.*, 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. *ECOOP'97 – Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, Springer, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.115.8660.

[Kim *et al.*, 97] K. H. (Kane) Kim, Jack Goldberg, Thomas F. Lawrence, and C. Subbaraman. The Adaptable Distributed Recovery Block Scheme and a Modular Implementation Model. *In Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems*, PRFTS '97, pages 131–, Washington, DC, USA. IEEE Computer Society, 1997. http://dl.acm.org/citation.cfm?id=826040.827004.

[Kim and Lawrence, 90] K. H. (Kane) Kim and Thomas F. Lawrence. Adaptive Fault Tolerance: Issues and Approaches. *In Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 38–46. IEEE, 1990. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=138292.

[Kim and Welch, 89] K. H. (Kane) Kim and Howard O. Welch. Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Transactions on Computers*, 38(5):626–636, IEEE Computer Society, Washington, DC, USA, May 1989. http://dx.doi.org/10.1109/12.24266.

[Knight, 12] John Knight. *Fundamentals of Dependable Computing for Software Engineers*. CRC Press, 2012.

[Laprie, 04] Jean-Claude Laprie. Sûreté de fonctionnement des systèmes: concepts de base et terminologie. *REE. Revue de l'électricité et de l'électronique*, (11):95–105, Société de l'Electricité, de l'Electronique et des Technologies de l'Information et de la Communication (SEE), 2004.

[Laprie, 08] Jean-Claude Laprie. From Dependability to Resilience. *In International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, AK, USA*, volume 8. 2008. http://www.ece.cmu.edu/~koopman/dsn08/fastabs/dsn08fastabs_laprie.pdf.

[Laprie *et al.*, 90] Jean-Claude Laprie, Jean Arlat, Christian Béounes, and Karama Kanoun. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *Computer*, 23(7):39–51, IEEE, 1990. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=56851.

[Laprie *et al.*, 96] Jean-Claude Laprie, Jean Arlat, Jean-Paul Blanquart, Alain Costes, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Hubert Guillermain, Mohamed Kaâniche, Karama Kanoun, Corinne Mazet, David Powell, Christophe Rabéjac, and Pascale Thévenod. *Guide de la Sûreté de Fonctionnement*. Cépaduès, 1996. ISBN 978-2-854-28382-2.

[Laprie *et al.*, 92] Jean-Claude Laprie, A. Avižienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag Wien New York, Inc., 1992.

[Lardner, 34] Dionysius Lardner. Babbage's Calculating Engine. *Edinburgh Review*, 59(120):263–327, 1834.

[Laws *et al.*, 11] Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod, and Simon Nash. *Tuscany SCA in Action*. Manning Publications Co., 2011.

[Lee and Anderson, 90] Peter A. Lee and Thomas Anderson. *Fault Tolerance*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag Wien New York, Inc., 1990.

[Léger *et al.*, 10] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable Dynamic Reconfigurations in a Reflective Component Model. *In Proceedings of the 13th international conference on Component-Based Software Engineering*, CBSE'10, pages 74–92, Berlin, Heidelberg. Springer-Verlag, 2010. http://dx.doi.org/10.1007/978-3-642-13238-4_5.

[Lung *et al.*, 06] Lau Cheuk Lung, Fabio Favarim, Giuliana Teixeira Santos, and Miguel Correia. An Infrastructure for Adaptive Fault Tolerance on FT-CORBA. *In Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '06)*. IEEE, 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.6352.

[Lyons and Vanderkulk, 62] R.E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, IBM, 1962. http://www.ccs.neu.edu/course/csg712/resources/Lyons-Vanderkulk-62.pdf.

[Maes, 87] Pattie Maes. Concepts and Experiments in Computational Reflection. *In Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155, New York, NY, USA. ACM, 1987. http://dl.acm.org/citation.cfm?id=38821.

[Margolis and Sharpe, 07] Ben Margolis and Joseph L. Sharpe. *SOA for the Business Developer: Concepts, BPEL, and SCA*. MC Press, 2007.

[Marin *et al.*, 01] Olivier Marin, Pierre Sens, Jean-Pierre Briot, and Zahia Guessoum. Towards Adaptive Fault-Tolerance for Distributed Multi-Agent Systems. *In Proceedings of The Fourth European Research Seminar on Advances in Distributed Systems*, ERSADS '01, pages 195–201. 2001. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.6131&rep=rep1&type=pdf.

[Marino and Rowley, 09] Jim Marino and Michael Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 2009.

[McIlroy, 68] M. D. McIlroy. Mass produced software components. *In First NATO Software Engineering Conference, Garmisch Pattenkirchen, Germany*, pages 138–157. 1968. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[McKinley *et al.*, 04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, IEEE, 2004. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1310241.

[Moghimi, 00] Reza Moghimi. Curing Comparator Instability with Hysteresis. *Analog Dialogue*, 34(7), 2000. http://www.analog.com/library/analogDialogue/archives/34-07/comparators/comparators.pdf.

[Narasimhan *et al.*, 05] P. Narasimhan, T. A. Dumitraş, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: support for Real-Time Fault-Tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, Wiley Online Library, 2005. http://www.gm.ece.cmu.edu/~mead/ccpe-2005.pdf.

[Paradis and Han, 07] Lilia Paradis and Qi Han. A Survey of Fault Management in Wireless Sensor Networks. *Journal of Network and Systems Management*, 15(2):171–190, Plenum Press, New York, NY, USA, June 2007. http://dx.doi.org/10.1007/s10922-007-9062-0.

[Pareaud *et al.*, 08] Thomas Pareaud, Jean-Charles Fabre, and Marc-Olivier Killijian. Componentization of Fault Tolerance Software for Fine-Grain Adaptation. *In Proceedings of The 14th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '08, pages 248–255. IEEE, 2008. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4725303.

[Parnas, 72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, ACM, 1972. http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf.

[Pathak and Prasanna, 08] Animesh Pathak and Viktor K. Prasanna. Srijan: A Graphical Toolkit for WSN Application Development. *In ProSenSe Workshop in the 4th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS), Santorini, Greece*. June 2008. https://who.rocq.inria.fr/Animesh.Pathak/papers/pathakprosense08.pdf.

[Pathak and Prasanna, 11] Animesh Pathak and Viktor K. Prasanna. High-Level Application Development for Sensor Networks: Data-Driven Approach. *In Theoretical Aspects of Distributed Computing in Sensor Networks*, pages 865–891. Springer, 2011. http://link.springer.com/chapter/10.1007%2F978-3-642-14849-1_26.

[Randell, 75] Brian Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, (2):220–232, IEEE, 1975. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6312842.

[Ranganathan and Campbell, 04] Arnand Ranganathan and Roy H. Campbell. Autonomic Pervasive Computing based on Planning. *In Proceedings of the First International Conference on Autonomic Computing*, ICAC '04, pages 80–87, Washington, DC, USA. IEEE Computer Society, 2004. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1301350f.

[Redmond and Cahill, 02] Barry Redmond and Vinny Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. *In Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 205–230. Springer, 2002. http://link.springer.com/chapter/10.1007%2F3-540-47993-7_9.

[Román *et al.*, 02] Manuel Román, Christopher Hess, Renato Cerqueira, Arnand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1158281.

[Sadjadi and McKinley, 04] S. M. Sadjadi and P. K. McKinley. ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation. *In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 74–83, Washington, DC, USA. IEEE Computer Society, 2004. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1281570.

[Salatge and Fabre, 07] Nicolas Salatge and Jean-Charles Fabre. Fault Tolerance Connectors for Unreliable Web Services. *In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 51–60, Washington, DC, USA. IEEE, 2007. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4272955.

[Saridakis, 02] Titos Saridakis. A System of Patterns for Fault Tolerance. *In Proceedings of the 7th European Conference on Pattern Languages of Programs*, EuroPLoP '02, pages 535–582. 2002. http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2002/2002_Saridakis_ASystemOfPatternsForFaultTolerance.pdf.

[Schneider, 90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, ACM, 1990. http://www.cs.cornell.edu/fbs/publications/smsurvey.pdf.

[Seinturier *et al.*, 11] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, Wiley, 2011. http://hal.inria.fr/docs/00/56/74/42/PDF/frascati.pdf.

[Shokri *et al.*, 97] Eltefaat Shokri, Herbert Hecht, Patrick Crane, Jerry Dussault, and K. H. (Kane) Kim. An Approach for Adaptive Fault-Tolerance in Object-Oriented Open Distributed Systems. *In Proceedings of the Third IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '97)*, pages 298–305. IEEE, 1997. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00609971.

[Speirs and Barrett, 89] N.A. Speirs and P.A. Barrett. Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing. *In Digest of Papers., Nineteenth International Symposium on Fault-Tolerant Computing*, FTCS '89, pages 184–190. IEEE, 1989. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00105564.

[Sterritt, 05] Roy Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, Springer, 2005. http://eprints.ulster.ac.uk/8226/1/17-2005-NASA-J-U7335H136027G642.pdf.

[Sterritt and Bustard, 03] Roy Sterritt and Dave Bustard. Autonomic Computing–a Means of Achieving Dependability? *In Proceedings of the Tenth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. 2003. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1194805.

[Stoicescu *et al.*, 11a] Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy. Architecting Resilient Computing Systems: Overall Approach and Open Issues. *In Software Engineering for Resilient Systems*, volume LNCS 6968 of *SERENE '11*, pages 48–62. 2011. http://link.springer.com/chapter/10.1007%2F978-3-642-24124-6_5.

[Stoicescu *et al.*, 11b] Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy. Towards a System Architecture for Resilient Computing. *In Actes de la Journée Sécurité des Systèmes & Sûreté des Logiciels (3SL)*, pages 17–19, Saint-Malo (France). 2011. http://www.univ-orleans.fr/lifo/evenements/3SL/actes/3sl.pdf.

[Stoicescu *et al.*, 12a] Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy. Experimenting with Component-Based Middleware for Adaptive Fault Tolerant Computing (fast abstract). European Dependable Computing Conference, 2012. http://arxiv.org/pdf/1204.1232v1.pdf.

[Stoicescu *et al.*, 12b] Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy. From Design for Adaptation to Component-Based Resilient Computing. *In 18th Pacific Rim International Symposium on Dependable Computing*, PRDC, pages 1 –10. IEEE, nov. 2012. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6385065.

[Szyperski, 02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[Taylor *et al.*, 09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[Wang *et al.*, 01] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Overview of the CORBA Component model. *In* George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., 2001. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.717&rep=rep1&type=pdf.

[Wensley *et al.*, 78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, 66(10):1240–1255, IEEE, 1978. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.66.8167&rep=rep1&type=pdf.

[Zheng and Lyu, 10] Zibin Zheng and Michael R. Lyu. An adaptive QoS-aware fault tolerance strategy for web services. *Empirical Software Engineering*, 15(4):323–345, Springer, 2010. http://link.springer.com/article/10.1007%2Fs10664-009-9126-8.

Abstract

Evolution during service life is mandatory, particularly for long-lived systems. Dependable systems, which continuously deliver trustworthy services, must evolve to accommodate changes e.g., new fault tolerance requirements or variations in available resources. The addition of this evolutionary dimension to dependability leads to the notion of resilient computing. Among the various aspects of resilience, we focus on adaptivity. Dependability relies on fault-tolerant computing at runtime, applications being augmented with fault tolerance mechanisms (FTMs). As such, on-line adaptation of FTMs is a key challenge towards resilience. In related work, on-line adaption of FTMs is most often performed in a preprogrammed manner or consists in tuning some parameters. Besides, FTMs are replaced monolithically. All the envisaged FTMs must be known at design time and deployed from the beginning. However, dynamics occurs along multiple dimensions and developing a system for the worst-case scenario is impossible. According to runtime observations, new FTMs can be developed off-line but integrated on-line. We denote this ability as agile adaption, as opposed to the preprogrammed one. In this thesis, we present an approach for developing flexible fault-tolerant systems in which FTMs can be adapted at runtime in an agile manner through fine-grained modifications for minimizing impact on the initial architecture. We first propose a classification of a set of existing FTMs based on criteria such as fault model, application characteristics and necessary resources. Next, we analyze these FTMs and extract a generic execution scheme which pinpoints the common parts and the variable features between them. Then, we demonstrate the use of state-of-the-art tools and concepts from the field of software engineering, such as component-based software engineering and reflective component-based middleware, for developing a library of fine-grained adaptive FTMs. We evaluate the agility of the approach and illustrate its usability throughout two examples of integration of the library: first, in a design-driven development process for applications in pervasive computing and, second, in a toolkit for developing applications for WSNs.

Keywords: resilience, runtime adaptation, fault tolerance mechanisms, component-based architecture

Résumé

L'évolution des systèmes pendant leur vie opérationnelle est incontournable. Les systèmes sûrs de fonctionnement doivent évoluer pour s'adapter à des changements comme la confrontation à de nouveaux types de fautes ou la perte de ressources. L'ajout de cette dimension évolutive à la fiabilité conduit à la notion de résilience informatique. Parmi les différents aspects de la résilience, nous nous concentrons sur l'adaptativité. La sûreté de fonctionnement informatique est basée sur plusieurs moyens, dont la tolérance aux fautes à l'exécution, où l'on attache des mécanismes spécifiques (Fault Tolerance Mechanisms, FTMs) à l'application. A ce titre, l'adaptation des FTMs à l'exécution s'avère un défi pour développer des systèmes résilients. Dans la plupart des travaux de recherche existants, l'adaptation des FTMs à l'exécution est réalisée de manière préprogrammée ou se limite à faire varier quelques paramètres. Tous les FTMs envisageables doivent être connus dès le design du système et déployés et attachés à l'application dès le début. Pourtant, les changements ont des origines variées et, donc, vouloir équiper un système pour le pire scénario est impossible. Selon les observations pendant la vie opérationnelle, de nouveaux FTMs peuvent être développés hors-ligne, mais intégrés pendant l'exécution. On dénote cette capacité comme adaptation agile, par opposition à l'adaptation préprogrammée. Dans cette thèse, nous présentons une approche pour développer des systèmes sûrs de fonctionnement flexibles dont les FTMs peuvent s'adapter à l'exécution de manière agile par des modifications à grain fin pour minimiser l'impact sur l'architecture initiale. D'abord, nous proposons une classification d'un ensemble de FTMs existants basée sur des critères comme le modèle de faute, les caractéristiques de l'application et les ressources nécessaires. Ensuite, nous analysons ces FTMs et extrayons un schéma d'exécution générique identifiant leurs parties communes et leurs points de variabilité. Après, nous démontrons les bénéfices apportés par les outils et les concepts issus du domaine du génie logiciel, comme les intergiciels réflexifs à base de composants, pour développer une librairie de FTMs adaptatifs à grain fin. Nous évaluons l'agilité de l'approche et illustrons son utilité à travers deux exemples d'intégration : premièrement, dans un processus de développement dirigé par le design pour les systèmes ubiquitaires et, deuxièmement, dans un environnement pour le développement d'applications pour des réseaux de capteurs.

Mots-clés: résilience, adaptation, mécanismes de tolérance aux fautes, architectures logicielles à base de composants