Preface

INTRODUCTION

The central topic of this book is application-level fault-tolerance, that is, the methods, architectures, and tools that allow a fault-tolerant system to be expressed in the application software of our computers. Application-level fault-tolerance is a sub-class of software fault-tolerance that focuses on the problems of expressing the problems and solutions of fault-tolerance in the top layer of the hierarchy of virtual machines that constitutes our computers. This book shows that application-level fault-tolerance is a key ingredient to craft truly dependable computer systems—other approaches, such as hardware fault-tolerance, operating system fault-tolerance, or fault-tolerant middleware, are also important ingredients to achieve resiliency, but they are not enough. Failing to address the application layer means leaving a backdoor open to problems such as design faults, interaction faults, or malicious attacks, whose consequences on the quality of service could be as unfortunate as, for example, a physical fault affecting the system platform. In other words, in most cases it is simply not possible to achieve complete coverage against a given set of faults or erroneous conditions without embedding fault-tolerance provisions also in the application layer. In what follows the provisions for application-level fault-tolerance are called application-level fault-tolerance protocols.

As a lecturer in this area, I wrote this book as my ideal textbook for a possible course on resilient computing and for my doctoral students in software dependability at the University of Antwerp. Despite this, the main goal of this book is not just education. The main mission of this book is first of all, spreading the awareness of the necessity of application-level fault-tolerance. Another critical goal is highlighting the role of several important concepts that are often neglected or
misunderstood: The fault and the system models, that is, the assumptions on top of which our computer services are designed and constructed. Last but not the least of our goals, this book aims to provide a clear view to the state-of-the-art of application-level fault-tolerance, also highlighting in the process a number of lessons learned through hands-on experiences gathered in more than 10 years of work in the area of resilient computing.

It is my conviction that any person who wants to include dependability among the design goals of their intended software services, should have a clear understanding of concepts such as dependability, system models, failure semantics, and fault models and of their influence on their final product’s quality of experience. Such information is often scattered among research papers while it is presented here in a unitary framework and from the viewpoint of the application-level dependable software engineer.

Another aspect that makes this book unique from all others in the field is the fact that concepts are described with examples that, in some cases, reach a deep level of detail. This is not the case in all chapters, as it reflects the spectrum of working experiences that the author had during more than a decade of research in this area. Any such spectrum is inherently not uniformly distributed. As a consequence some chapters provide the reader with in-depth knowledge, down to the level of source code examples, while others just introduce the reader into the subject, explain the main concepts, and place the topic in the wider context of methods treated in this book. To increase readability, we isolated some of the most technical texts into text boxes typed in italics.

Furthermore, this book has a privileged viewpoint, which is the one of real-time, concurrent, and embedded system design. This book does not focus in particular on the design of fault-tolerance provisions for service-oriented applications, such as Web services, and does not cover fault-tolerance in the middleware layer.

In what follows the background top-level information and the structure of this book are introduced.

**BACKGROUND AND STRUCTURE**

No man conceived tool in human history has ever permeated so many aspects of human life as the computer has been doing for the last 60 years. An outstanding aspect of this success story is certainly given by an overwhelming increase in computer performance. Another one, also very evident, is the continuous decrease of costs of computer devices—a US$1,000 PC today provides its user with more performance, memory, and disk space of a US$1 million mainframe of the 1960s. Clearly performance and costs are “foreground figures”—society at large is well
aware of their evolution and of the societal consequences of the corresponding spread of computers. On the other hand this process is also characterized by “background figures”, that is, properties that are often overlooked despite their great relevance. Among such properties it is worth mentioning the growth in complexity and the crucial character of the roles nowadays assigned to computers: Human society more and more expects and relies on good quality of complex services supplied by computers. More and more, these services become vital, in the sense that lack of timely delivery ever more often can have immediate consequences on capitals, the environment, and even human lives. Strangely though it may appear, the common man is well aware that computers get ever more powerful and less expensive, but doesn’t seem to be aware or even care about computers being safe and up to their ever more challenging tasks. The turn of the century brought about this problem for the first time—the Millennium Bug, also known as Y2K, reached the masses with striking force, as a tsunami of sudden awareness that “yes, computers are powerful, but even computers can fail.”

Y2K ultimately did not show up, and the dreaded scenarios of a society simultaneously stripped by its computer services ended up in a few minor accidents.

But society had a glimpse of some of the problems that are central to this book: Why do we trust computer services? Are there modeling, design, development practices, conceptual tools, and concrete methods, to convince me that when I take a computer service, that service will be reliable, safe, secure, available? In other words, is there a science of computer dependability, such that reliance of computer systems can be measured, hence quantitatively justified? And, is there an engineer-

Figure 1. Perception of dependability in the Sixties (TM &© 2008 Marvel Characters, Inc. All Rights Reserved.)
ing of computer dependability, such that trustworthy computer services can be effectively achieved? Dependability—the discipline that studies those problems—is introduced in Chapter I.

This book in particular, focuses on fault-tolerance, which is described in Chapter I as one of the “means” for dependability: Fault-tolerance is one of the four classes of methods and techniques enabling one to provide the ability to deliver a service on which reliance can be placed, and to reach confidence in this ability (together with fault prevention, fault removal, and fault forecasting). Its core objective is “preserving the delivery of expected services despite the presence of fault-caused errors within the system itself” (Avižienis, 1985). The exact meaning of faults and errors is also given in the cited chapter, together with an introduction to fault-tolerance mainly derived from the works of Jean-Claude Laprie (Laprie, 1992, 1995, 1998, 1985). What is important to remark here is that fault-tolerance acts after faults have manifested themselves in the system: Its main assumption is that faults are inevitable, but they must be tolerated, which is fundamentally different from other approaches where, for example, faults are sought to be avoided in the first place. Why focusing on fault-tolerance, why is it so important? For the same reason referred above as a background figure in the history of the relationship between human society and computers: The growth in complexity. Systems get more and more complex, and there are no effective methods that can provide us with a zero-fault guarantee. The bulk of the research of computer scientists and engineers concentrated on methods to pack conveniently ever more complexity in computer systems. Software in particular has become a point of accumulation of complexity, and the main focus so far has been on how to express and compose complex software modules so as to tackle ever new challenging problems rather than dealing with the inevitable faults introduced by that complexity. Layered design is a classical method to deal with complexity.

Software, software fault-tolerance, and application-level software fault-tolerance, are the topics of Chapter II. It is explained what it means that a program is fault-tolerant and what are the properties expected from a fault-tolerant program. The main objective of Chapter II is introducing two sets of design assumptions that shape the way people structure their fault-tolerant software—the system and the fault models. Often misunderstood or underestimated, those models describe:

- what is expected from the execution environment in order to let our software system function correctly; and
- what are the faults that our system is going to consider. Note that a fault-tolerant program shall (try to) tolerate only those faults stated in the fault model, and will be as defenseless against all other faults as any non fault-tolerant program
Together with the system specification, the fault and system models represent the foundation on top of which our computer services are built. Not surprisingly enough, weak foundations often result in fragile constructions. To provide evidence to this, the chapter introduces three well-known accidents—the Ariane 5 flight 501 and Mariner-1 disasters and the Therac-25 accidents (Leveson, 1995). In each case it has been stressed about what went wrong, what were the biggest mistakes, and how a careful understanding of fault models and system models would have helped highlighting the path to avoid catastrophic failures that cost considerable amounts of money and even the lives of innocent people?

After this, the chapter focuses on the core topic of this book, application-level software fault-tolerance. Main questions addressed here are: How to express and achieve fault-tolerance in the mission layer? Why is application-level software fault-tolerance so important? The main reason for this is that a computer service is the result of the concurrent execution of several “virtual” and physical machines (see Fig. 2).

Some of these machines run a predefined, special-purpose service, meant to serve—unmodified—many different applications. The hardware, the operating system, the network layers, the middleware, a programming language’s run-time executive, and so forth, are common names of those machines. A key message in this book is that tolerating the faults in one machine does not protect from faults originating in another one. This includes the application layer. Now, while the machines “below” the application provide architectural (special-purpose) complexity,
the mission layer contributes to computer services with general-purpose complexity, which is intrinsically less reliable. This and other reasons that justify the need for application-level software fault-tolerance are given in that chapter. The main references here are (Randell, 1975; Lyu, 1998a, 1998b).

Chapter II also introduces what the author considers to be the three main properties of application-level software fault-tolerance: Separation of design concerns, adaptability, and syntactical adequacy (De Florio & Blondia, 2008b). In this context the key questions are: Given a certain fault-tolerance provision, is it able to guarantee an adequate separation of the functional and non-functional design concerns? Does it tolerate a fixed, predefined set of faulty scenarios, or does it dynamically change that set? And, is it flexible enough as to host a large number of different strategies, or is it a “hardwired” solution tackling a limited set of strategies?

Finally, this chapter defines a few fundamental fault-tolerance services, namely watchdog timers, exception handling, transactions, and check pointing-and-rollback.

After having described the context and the “rules of the game”, this book discusses the state-of-the-art in application-level fault-tolerance protocols. First, in Chapter III, the focus is on so-called single-version and multiple-version software fault-tolerance (Avižienis, 1985).

- Single-version protocols are methods that use a non-distributed, single task provision, running side-by-side with the functional software, often available in the form of a library and a run-time executive.
- Multiple-version protocols are methods that use actively a form of redundancy, as explained in what follows. In particular the chapter discusses recovery blocks and N-version programming.

Chapter III also features several in-depth case studies deriving from the author’s research experiences in the field of resilient computing. In particular the EFTOS fault-tolerance library (Deconinck, De Florio, Lauwereins, & Varvarigou, 1997; Deconinck, Varvarigou, et al., 1997) is introduced as an example of application-level single-version software fault-tolerance approach. In that general framework, the EFTOS tools for exception handling, distributed voting, watchdog timers, fault-tolerant communication, atomic transactions, and data stabilization, are discussed. The reader is also given a detailed description of RAFTNET (Raftnet, n.d.), a fault-tolerance library for data parallel applications.

A second large class of application-level fault-tolerance protocols is the focus of Chapter IV, namely the one that works “around” the programming language, that is to say either embedded in the compiler or via language transformations driven by translators. In that chapter it is also discussed the design of a translator support-
ing language-independent extensions called reflective and refractive variables and linguistic support for adaptively redundant data structures.

- Reflective and refractive variables (De Florio & Blondia, 2007a) are syntactical structures to express adaptive feedback loops in the application layer. This is useful to resilient computing because a feedback loop can attach error recovery strategies to error detection events.
- Redundant variables (De Florio & Blondia, 2008a) are a tool that allows designers to make use of adaptively redundant data structures with commodity programming languages such as C or Java. Designers using such tools can define redundant data structures in which the degree of redundancy is not fixed once and for all at design time, but rather it changes dynamically with respect to the disturbances experienced during the run time.

The chapter shows that by a simple translation approach, it is possible to provide sophisticated features such as adaptive fault-tolerance to programs written in any programming language.

In Chapter V the reader gets in touch with methods that work at the level of the language itself: Custom fault-tolerance programming languages. In this approach fault-tolerance is not embedded in the program, nor around the programming language, but provided through the syntactical structures and the run-time executives of fault-tolerance programming languages. Also in this case application-level complexity is enucleated from the source code and shifted to the architecture, where it is much easier and cost-effective to tame. Three classes of approaches are treated—object-oriented languages, functional languages, and hybrid languages. In the latter class, special emphasis is given to Oz (Müller, Müller, & Van Roy, 1995), a multi-paradigm programming language that achieves both transparent distribution and translucent failure handling.

A separate chapter is devoted to a large case study in fault-tolerant languages: The so-called recovery language approach (De Florio, 2000; De Florio, Deconinck, & Lauwereins, 2001). In Chapter VI the concept of recovery language is first introduced in general terms and then proposed through an implementation: the Ariel recovery language and a supporting architecture. That architecture is an evolution of the EFTOS system described in Chapter III, and targets distributed applications with non-strict real-time requirements, written in a procedural language such as C, to be executed on distributed or parallel computers consisting of a predefined set of processing nodes. Ariel and its run-time system provide the user with a fault-tolerance linguistic structure that appears to the user as a sort of second application-level especially conceived and devoted to address the error recovery concerns. This separation is very useful at design time, as it allows to bound design complexity. In
Ariel, this separation holds also at run-time, because even the executable code for error recovery is separated from the functional code. This means that, in principle, the error recovery code could change dynamically so as to match a different set of internal and environmental conditions. This can be used to avoid “hardwiring,” a fault model into the application—an important property especially when, for example, the service is embedded in a mobile terminal (De Florio & Blondia, 2005).

Chapter VII discusses fault-tolerance protocols based on aspect-oriented programming (Kiczales et al., 1997), a relatively novel structuring technique with the ambition to become the reference solution for system development, the way object-orientation did starting with the 1980s. We must remark how aspects and their currently available implementations have not yet reached a maturity comparable with that of the other techniques discussed in this book. For instance, the chapter remarks how no aspect-oriented fault-tolerance language has been proposed to date and, at least in some cases, the adequacy of aspects as a syntactical structure to host fault-tolerance provisions has been questioned. On the other hand, aspects allowed regarding the source code as a flexible web of syntactic fragments that the designer can rearrange with great ease, deriving modified source codes matching particular goals, for example, performance and, hopefully in the near future, dependability. The chapter explains how aspects allow to separate design concerns, which bounds complexity and enhances maintainability, and presents three programming languages: AspectJ (Kiczales, 2000), AspectC++ (Spinczyk, Lohmann, & Urban, 2005) and GluonJ (GluonJ, n.d.).

The following chapter, Chapter VIII, deals with failure detection protocols in the application layer. First the concept of failure detection (Chandra & Toueg, 1996), a fundamental building block to develop fault-tolerant distributed systems, is introduced. Then the relationship between failure detection and system models is highlighted—the key assumptions on which our dependable services are built, which were introduced in Chapter II. Then it is introduced as a tool for the expression of this class of protocols (De Florio & Blondia, 2007b), based on a library of objects called time-outs (V. De Florio, 2006). Finally, a case study is described in detail: The failure detection protocol employed by the so-called EFTOS DIR net (De Florio, Deconinck, & Lauwereins, 2000), a distributed “backbone” for fault-tolerance management which was introduced in Chapter III and that later evolved into the so-called Backbone discussed in Chapter VI.

Hybrid approaches are the focus of Chapter IX, that is, fault-tolerance protocols that blend two or more methods among those reported in previous chapters. In more detail ReLinda is introduced—a system coupling the recovery language approach of Chapter VI and generative communication, one of the models introduced in Chapter IV (De Florio & Deconinck, 2001). After this, the recovery language-empowered extensions of two single-version mechanisms previously introduced in Chapter III
are described, namely a distributed voting mechanism and a watchdog timer (De Florio, Donatelli, & Dondossola, 2002). The main lessons learned in this case are that the recovery language approach allows to fast-prototype complex strategies by composing a set of building blocks together and by building system-wide, recovery-time coordination strategies with the Ariel language. This allows set up sophisticated fault-tolerance systems while keeping the management of their complexity outside of the user application. Other useful properties achieved in this way are transparency of replication and transparency of location.

Chapter X provides three examples of approaches used to assess the dependability of application-level provisions. In the first case, reliability analysis is used to quantify the benefits of coupling an approach such as recovery languages to a distributed voting mechanism (De Florio, Deconinck, & Lauwereins, 1998). Then a tool is used to systematically inject faults onto the adaptively redundant data structure discussed in Chapter IV (De Florio & Blondia, 2008a). Monitoring and fault-injection are the topic of the third case, where a hypermedia application to watch and control a dependable service is introduced (De Florio, Deconinck, Truyens, Rosseel, & Lauwereins, 1998).

Chapter XI concludes the book by summarizing the main lessons learned. It also offers a view to the internals of the application-level fault-tolerance provision described in Chapter VI—the Ariel recovery language.

**SUMMARY OF CONTRIBUTIONS**

Application software development is not an easy task; writing truly dependable fault-tolerant applications is even more difficult, not only in itself for the additional complexity required by fault-tolerance but often also because of the lack of awareness which is necessary in order to master the complexity of this tricky task.

The first and foremost contribution of this book is increasing the awareness of the role and significance of application-level fault-tolerance. This has been reached by highlighting important concepts that are often neglected or misunderstood, as well as introducing the available tools and approaches that can be used to craft high-quality dependable services by working also in the application layer.

Secondly, this book summarizes the most widely-known approaches to application-level software fault-tolerance. A base of properties in which those approaches can be compared and assessed is defined.

Finally, this book features a collection of several research experiences the author had in the field of resilient computing through his participation to several research projects funded by the European Community. This large first-hand experience is reflected into the deep level of detail that is reached in some cases.
We hope that the above contributions will prove useful to the readers and intrigue them into entering the interesting arena of resilient computing research and development. Also, too many times the lack of awareness and know-how in resilient computing has brought the designers to supposedly robust systems whose failures had in some cases dreadful consequences on capitals, the environment, and even human lives—as a joke we call them sometimes “endangeneers”. We hope that this book may contribute to the spread of that awareness and know-how that should always be part of the education of dependable software engineers. This important requirement is witnessed by several organizations such as the European Workshop on Industrial Computer Systems Reliability, Safety and Security, technical committee 7, whose mission is “To promote the economical and efficient realization of programmable industrial systems through education, information exchange, and the elaboration of standards and guidelines” (EWICS, n.d.), and the ReSIST network of excellence (ReSIST, n.d.), which is developing a resilient computing curriculum recommended to all people involved in teaching dependability-related subjects.

REFERENCES


