INTRODUCTION AND OBJECTIVES

This chapter discusses two large classes of fault-tolerance protocols:

• Single-version protocols, that is, 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, which are methods that use actively a form of redundancy, as explained in what follows. In particular recovery blocks and N-version programming will be discussed.

The two families have been grouped together in this chapter because of the several similarities they share.
FAULT-TOLERANT PROTOCOLS USING SINGLE- AND MULTIPLE-VERSION SOFTWARE FAULT-TOLERANCE

A key requirement for the development of fault-tolerant systems is the availability of replicated resources, in hardware or software. A fundamental method employed to attain fault-tolerance is multiple computation, i.e., N-fold (N > 1) replications in three domains:

- **Time** That is, repetition of computations.
- **Space** I.e., the adoption of multiple hardware channels (also called “lanes”).
- **Information** That is, the adoption of multiple versions of software.

Following Avižienis (Avižienis, 1985), it is possible to characterize at least some of the approaches towards fault-tolerance by means of a notation resembling the one used to classify queuing systems models (Kleinrock, 1975):

\[ nT/mH/pS, \]

the meaning of which is “n executions, on m hardware channels, of p programs”. The non-fault-tolerant system, or 1T/1H/1S, is called simplex in the cited paper.


Single-version software fault-tolerance (SV) is basically the embedding into the user application of a simplex system of error *detection* or recovery features, e.g., atomic actions (Jalote & Campbell, 1985), checkpoint-and-rollback (Deconinck, 1996), or exception handling (Cristian, 1995). The adoption of SV in the application layer requires the designer to concentrate in one physical location, namely, the source code of the application, both the specification of what to do in order to carry on some user computation and the strategy such that faults are tolerated when they occur. As a result, the size of the problem addressed is increased. A fortiori, this translates into increasing the size of the user application. This induces loss of transparency, maintainability, and portability while increasing development times and costs.

A partial solution to this loss in portability and these higher costs is given by the development of libraries and frameworks created under strict software engineering processes. In the following, three examples of this approach are presented—the EFTOS library and the SwIFT system. Special emphasis is reserved in particular to the first system, for which the author of this book designed a number of contributions.
Related Content

Legacy Systems towards Aspect-Oriented Systems
www.igi-global.com/chapter/legacy-systems-towards-aspect-oriented-systems/135231?camid=4v1a

Generic Model of the Business Model and Its Formalization in Object-Z
www.igi-global.com/chapter/generic-model-business-model-its/30756?camid=4v1a

www.igi-global.com/article/semantic-framework-for-energy-aware-resource-management-of-iot-in-business-processes/207348?camid=4v1a
Preventing the Increasing Resistance to Change through a Multi-Model Environment as a Reference Model in Software Process Improvement
Mirna Muñoz and Jezreel Mejia (2014). *Agile Estimation Techniques and Innovative Approaches to Software Process Improvement* (pp. 97-118).
www.igi-global.com/chapter/preventing-the-increasing-resistance-to-change-through-a-multi-model-environment-as-a-reference-model-in-software-process-improvement/100274?camid=4v1a