# A Looming Fault Tolerance Software Crisis[1]?

Alexander Romanovsky
Newcastle University, UK
alexander.romanovsky@ncl.ac.uk

**Abstract**

Experience suggests that it is edifying to talk about software crises at NATO workshops. It is argued in this position paper that proper engineering of fault tolerance software has not been getting the attention it deserves. The paper outlines the difficulties in building fault tolerant systems and describes the challenges software fault tolerance is facing. The solution being advocated is to place a special emphasis on fault tolerance software engineering which would provide a set of methods, techniques, models and tools that would exactly fit application domains, fault assumptions and system requirements and support disciplined and rigorous fault tolerance throughout all phases of the life cycle. The paper finishes with an outline of some directions of work requiring special focused efforts from the R&D community.

Keywords: fault tolerance, software engineering

## Fault tolerance misuse

As reported by Flavio Cristian in the 80s [1], field experience with telephone switching systems showed that up to two thirds of system failures were due to design faults in exception handling or recovery algorithms.

Let us look into what is happening now.

- The Interim Report on Causes of the August 14th 2003 Blackout in the US and Canada [2] clearly shows that the problem was mostly caused by badly designed fault tolerance: poor diagnostics of faults, longer-than-estimated time for component recovery, failure to involve all necessary components in recovery, inconsistent system state after recovery, failures of alarm systems, etc.
- Tony Hoare reports [3] that in some MS systems more than 10% of code is dedicated to executable assertions. Yet as we all know, many customers are still unhappy with the quality of these products
- The authors of an ICSE 2006 paper [4] have experimentally found that in a 10 million LOC real-time embedded control system, misused exception handling introduced 2-3 bugs per 1 KLOS.
- Another paper, just accepted to IVNET 2006 [5], shows that in eight .NET assemblies (which represent application, library and infrastructure levels), over 90% of exceptions that the code can throw are not documented.
- Paper [6] by IBM researchers reports typical patterns of exception handling misuse and abuse in five customer and one proprietary J2EE applications, referring to them as "bad coding practice". It was found, for example, that one in ten classes swallows exceptions without doing anything about them.

*We keep making mistakes in designing fault tolerance!* The situation is deteriorating as the complexity of software and systems in general is growing, causing an increase in the complexity of fault tolerance, as computer-based systems proliferate more widely in business, society and individuals' activities.

In spite of the fact that a plethora of fault tolerance mechanisms have been developed since the 70s, that there is a good understanding of the basic principles of building fault tolerant software and that a considerable fraction of requirements analysis, run time resources, development effort and code are now dedicated to fault tolerance, we might well be on our way to a fault tolerant software crisis. At present, fault tolerance is not trustworthy as it is the least understood, documented and tested part of the system, is frequently misused or poorly designed, regularly left until too late in the development process, not typically introduced in a systematic, disciplined or rigorous way, and often not suitable for the specific situations in which it is applied.

## Fault tolerance: challenges and difficulties

Fault tolerance means can and will undermine overall system dependability if not applied properly. The following are some of the main challenges in the area:

- Fault tolerance means are *difficult* to develop or, when they are provided by some dedicated support, to use – this is because they increase system *complexity* by adding a new dimension to the reasoning about system behaviour. Their application requires a deep understanding of the intricate links between normal and abnormal behaviour and states of systems and components, as well as system state and behaviour during recovery.
- Fault tolerance (software diversity, rollback, exception handling) is *costly* as it always uses redundancy. Rather than improve fault tolerance, system developers far too often prefer to spend resources on extending functionality. We cannot and/or do not always want to put a cost on failures.
- System designers are *reluctant* to think about faults at the early phases of development. Fault tolerance is often considered to be an implementation issue. Moreover,

[1] Position paper for the 2006 NATO Workshop on Building Robust Systems with Fallible Construction (Prague, Czech Republic, 9-10 November 2006)

fault tolerance is often "added" *after* the normal part of the system is developed, which makes it less effective, may require system redesign or result in faulty fault tolerance.

- There is a lack of appropriate training for, education about or good practice in, fault tolerance:
  - We do not really know what counts as a good fault tolerant program. We usually know well only how to write programs and components that assume (unjustifiably) that nothing will go wrong
  - Developers of many applications fail to apply even the basic principles of software fault tolerance. There is no focus on clearly defining fault assumptions from the very start, early error detection, recursive system structuring for error confinement, minimising and ensuring error confinement and error recovery areas, extending component specification with a concise and complete definition of failure modes, etc.
- It is imperative that fault tolerance means *fit* the system, the types of faults (i.e. the fault assumptions), the application domain, the development paradigm, the execution environment and the system characteristics. We need suitable *fault tolerance abstractions* for a variety of particular situations.

### Fault assumptions and application fault tolerance

We believe that due to

- an increase in hardware quality and a reduction in hardware cost (e.g. hardware replication is cheap )
- a dramatic rise in software complexity and volume
- the involvement of new actors (non-professional users, multiple organisations, critical infrastructures)
- a growing complexity of the environment in which systems operate,

for many applications hardware faults are no longer the predominant threat. These applications include a wide range of safety-, life-, business- and money-critical systems – see, for example, recent studies by J.-C. Laprie [7], J. Knight [8] and by the Standish Group [9]. The *predominant types* of faults to be tolerated are

- application software faults (including design faults)
- environmental and infrastructural faults/deficiencies
- potentially damaging changes in systems, components, environments and infrastructures
- mismatches of components composed together (including mismatches of fault tolerance mechanisms [10])
- architectural and organisational mismatches and system-level inconsistencies
- degradation of services provided by components and systems
- organisational, human and socio-technical faults.

Such faults cannot be tolerated (and the system recovered) by hardware or middleware means alone, without involving application software. This is why we need to include fault tolerance measures into application system development (be it top-down or bottom-up or a mix of both).

### Fault tolerance and software development

Fault tolerance needs to be engineered in a disciplined and rigorous way. In agreement with a number of my colleagues working in fault tolerance, I see the way forward in pursuing the following directions:

- integrating fault tolerance measures (diversity, exception handling, backward error recovery, etc.) into system models starting from the early architectural design
- making fault tolerance-related decisions for each appropriate model by modelling faults, fault tolerance measures and dedicated redundant resources. In particular, we need to focus on fault tolerant software architectures
- ensuring correct transformations of those models that enrich fault tolerance measures and make models more concrete and detailed
- making fault tolerance verification and validation part of system development
- developing dedicated tool support for fault tolerance development
- providing domain-specific application-level fault tolerance mechanisms and abstractions.

Clearly, there has been some research done in these areas. Yet if we look at the proceedings of some best conferences relevant to dependability and software engineering, such as ICSE, DSN, ESEC/FSE and EDCC, we will see that these topics are at best peripheral. It is my strong belief that more focused efforts are needed to achieve fault tolerance which neither fails nor requires fault tolerance itself.

### Where to look for solutions

In this section I would like to briefly introduce some of the R&D directions which I believe are or will be contributing to the successful engineering of fault tolerance.

*Architecting fault tolerant systems* is now becoming an active research area. We need to focus on introducing specialised architectural solutions:

- supporting all main fault tolerance mechanisms (exception handling with error confinement, software diversity, atomic actions, etc.)
- introducing specific fault tolerance solutions (such as adaptors and protective wrappers for COTS component integration – [11])
- making existing and widely accepted architectures fault tolerant
- ensuring tolerance of architectural mismatches [12].

It is essential that fault tolerance is supported by a set of specialised *patterns and styles* that would assist developers at all steps of the life cycle. These should include specialised architectural, refinement, decomposition, design, implementation and model transformation patterns and styles.

Where appropriate, fault tolerance should be developed *formally* to ensure its "correctness by construction". This needs to be supported by a development environment with a set of specialised tools. We should be able to model faults and fault tolerance, to express, prove and check specific fault tolerance properties of these models and to refine them by refining both fault assumptions and fault tolerance means.

Different faults and their tolerance need to be considered *at the appropriate phases* of the life cycle and further refined and decomposed during development. This needs to start with the requirement phase.

To avoid making software more complex and introducing new faults, the fault tolerance *mechanisms and abstractions* being developed should *fit* the types of faults, the application domain, the development paradigm, the execution environment and the system characteristics and requirements.

*Fault tolerance and fault tolerant evolution.* Both system evolution and dynamic upgrade should ensure the preservation or the controlled and predictable changes of system fault tolerance. It is systems going through online modifications that are mostly vulnerable to faults, so we need specialised fault tolerance mechanisms that will ensure dependable modifications.

Some of the recent and ongoing activities that are directly related to engineering fault tolerant systems:

- RODIN - Rigorous Open Development Environment for Complex Systems, FP6 IST STREP project (2004-2007)[2]
- FME 2005 Workshop on rigorous engineering of fault tolerant systems (REFT 2005, Newcastle, July 2005) and a follow-up State of the Art LNCS collection [13]
- Workshop on engineering fault tolerant systems in Luxemburg (EFTS 2006)[3]. June 2006
- Edited collection of papers on engineering fault tolerant systems to be published in 2007
- A series of workshops on architecting dependable systems (WADS at ICSE and DSN in 2002-2006)[4] and three follow-up LNCS collections [14-16].

## Copyrights

My thanks go to many people with whom these ideas have been discussed, but first of all, to Brian Randell, Cliff Jones, Jörg Kienzle, Nicolas Guelfi and Fernando Castro Filho. This work is supported by the IST RODIN Project.

[2] http://rodin.cs.ncl.ac.uk/

[3] http://se2c.uni.lu/tiki/tiki-index.php?page=Efts2006Overview

[4] http://www.cs.kent.ac.uk/events/conf/2006/wads/

## References

[1] F. Cristian. Exception handling. In Dependability of Resilient Computers, T. Anderson (Ed.). Blackwell Scientific Publications, 1989. pp. 68-97.

[2] Interim Report: Causes of the August 14th Blackout in the United States and Canada. Canada-U.S. Power System Outage Task Force. November 2003. http://www.nrcan-rncan.gc.ca/media/docs/reports_e.htm.

[3] T. Hoare. Assertions in modern software engineering practice. Invited talk. COMPSAC 2002. Oxford, UK, 26-29 August 2002.

[4] M. Bruntink, A. van Deursen, T. Tourwé. Discovering Faults in Idiom-Based Exception Handling. ICSE 2006. 20-28 May 2006. Shanghai. China. ACM Press. pp. 242-251.

[5] P. Sacramento, B. Cabral, P. Marques. Unchecked exceptions: can the programmer be trusted to document exceptions? Accepted for the 2nd Int. Conf. on Innovative Views of .NET Technologies (IVNET 2006). 2006. Florianopolis, Brazil.

[6] D. Reimer, H. Srinivasan. Analyzing exception usage in large java applications. In Proceedings of ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems, July 2003.

[7] J.-C. Laprie. Dependability of software-based critical systems. In Dependable Network Computing. D. R. Avresky (Ed.). 1999.

[8] J. Knight. Assured Reconfiguration: An Architectural Core For System Dependability. Invited talk. ICSE 2005 Workshop on Architecting Dependable Systems. St. Louis, Missouri, USA, 17 May 2005.

[9] J. Johnson. The Other Side of Failure! DSN 2006 Industry Session. June 26. Philadelphia, USA. 2006.

[10] A. Avizienis. Infrastructure-Based Design of Fault-Tolerant Systems. In the Electronic Proceedings of the IFIP Int. Workshop on Dependable Computing and Its Applications (DCIA 98) January 12 - 14, 1998, Johannesburg, South Africa.

[11] T. Anderson, B. Randell, A. Romanovsky. Wrapping the future. In the Proceedings of the IFIP Congress Topical Sessions. Toulouse. France. 2004. pp. 165-174.

[12] R. de Lemos, C. Gacek, A. Romanovsky. Architectural Mismatch Tolerance. In Architecting Dependable Systems. LNCS 2677, 2003. pp. 175-194.

[13] M. Butler, C. Jones, A. Romanovsky, E. Troubitsyna (Eds). Rigorous development of complex fault tolerant system. LNCS 4157. 2006.

[14] R. de Lemos, C. Gacek, A. Romanovsky (Eds). Architecting Dependable Systems. LNCS 2677. 2003.

[15] R. de Lemos, C. Gacek, A. Romanovsky (Eds). Architecting Dependable Systems II. LNCS 3069, 2004.

[16] R. de Lemos, C. Gacek, A. Romanovsky (Eds). Architecting Dependable Systems III. LNCS 3549, 2005.