Chapter 1

Introduction

1.1. Context

The academic framework of this project is the Data and Knowledge Management Group of the Research Centre on Information and Automation Technologies (CENTIA) at UDLA, Puebla, and the Database group HADAS (Networked Open Database Services) of laboratory Logiciels Systèmes Réseaux (LSR-IMAG, UMR 5526) of Grenoble, France.

1.2. Systems construction

Information systems are becoming more and more complex. This evolution is due to the democratization of distributed environments with components on multiple machines, and with notions like communication between processes, reactive systems, mobile code, multimedia data administration, etc. Therefore, the development of modern information systems cannot be done in a monolithic way but under a component (that is, autonomous services) assembly approach.

The main concern about composition is that the quality and properties of the final service depends on the properties of each component service. The structural verification of the composition guarantees that the final service does not include conflicts between the operations of its components. However, structural verification does not guarantee the correct execution of the service in the presence of an anomaly, for example, erroneous data or machine crash. Therefore, mechanisms are necessary to ensure the reliability of the service. Reliability concerns two aspects: exception handling and fault tolerance. The first one avoids and handles design anomalies, and the second aspect is the capacity to process faults.

Traditionally, there are two different approaches to build a fault tolerant distributed system [AVI1998]:

- Bottom-up: An infrastructure is designed of autonomously fault tolerant subsystems (microprocessors, memories, sensors, displays, etc), including global fault tolerant functions: reconfiguration, externally supported recovery, etc.

- Top-down: A system is build using pre-existing “off-the-shelf” subsystems that may have little or no fault tolerant at all, and to implement a global monitoring function that provides fault tolerance. This is the prevailing practice in our days [AVI1998].

The way that fault tolerance mechanisms are integrated and implemented in a top-down fashion to a target system can be classified in four categories [DQP2002]:

- Integrated: The fault tolerance mechanisms are completely dependent on the target application. The development and implementation of the target application includes the applicative code and the fault tolerance code.

- Factorized: The fault tolerance algorithms classes are factorized for providing libraries that enable the implementation of these algorithms. The developer can download part of the fault tolerance implementation. Nevertheless, the developer must have certain fault tolerance knowledge. The implementation of fault tolerance aspects is still merged within the application logic code. The target application is more modular than in the first category.

- Separated: The fault tolerance codes are separated from the target application ones. In this

case, the developer does not matter about fault tolerance concerns.

- Adaptable: Isolate the fault tolerance goes far beyond. The purpose is to separate the fault tolerance codes from the target application logical code and do it adaptable to the target application characteristics.

Figure 1.1 illustrates the four categories of the top-down approach.

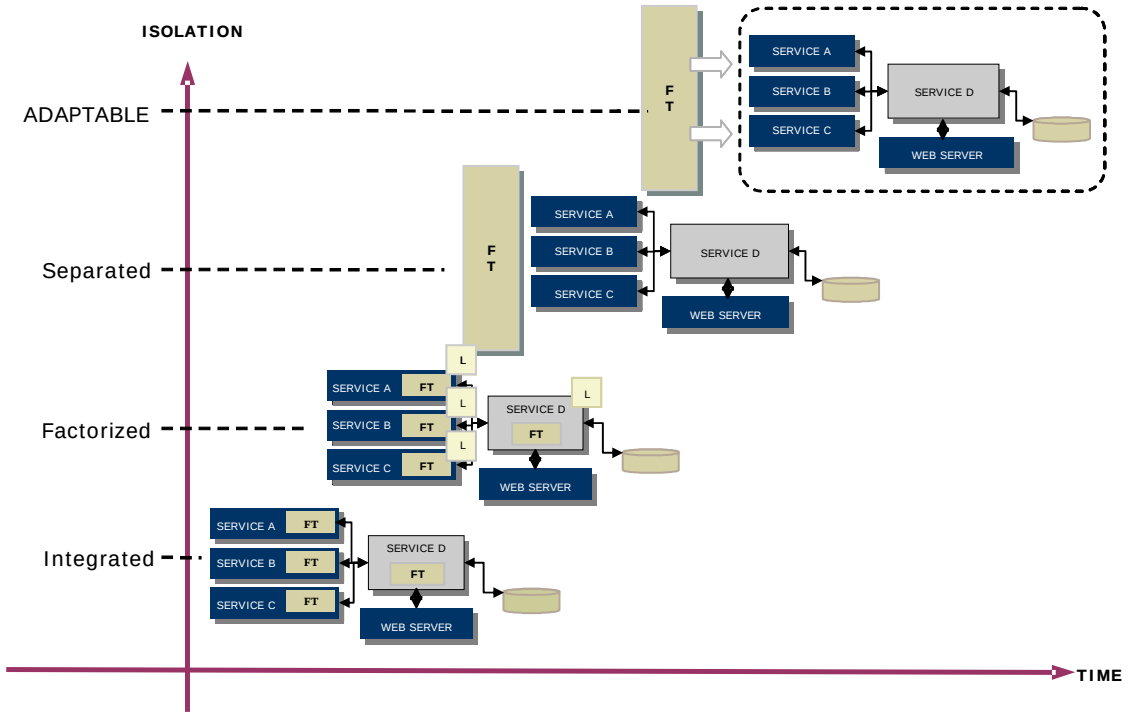

Figure 1.1 Categories of fault tolerance integration on distributed systems

The cost, size, and power requirements of microprocessors have dropped to a point where it is both feasible and desirable to embed computers into everything from people to toaster machines [KN2002]. Of course, computers embedded in people must operate correctly because they are providing what is presumably some form of medical service. But even computers embedded in toasters have to operate correctly because incorrect operation might cause a fire. A serious though less obvious issue with embedded computers, even in appliances, is the financial loss from failure. Recalling a mass-produced appliance to repair a defective embedded computer, correcting the software for example, can be financially devastating for the manufacturer.

1.3. Embedded systems

An embedded system is an electronic programmable sub-system that generally is an integrated part of a large heterogeneous system [BS2005]. Embedded systems are the union of hardware, software and mechanic pieces that have particular characteristics and limitations. For example, the computing capacity, the type of memory used, and the operating system that is used.

Embedded systems are considered as electronic systems that include a microcomputer like the Motorola 6805, 6808, or 6812 that is configured to perform a specific dedicated application. The software that controls the system is programmed or fixed into ROM and is not accessible to the user of the device. Software for embedded systems typically solves only a limited range of problems. The microcomputer is embedded or hidden inside the device. A typical automobile now contains an average of ten microcontrollers. In fact, upscale homes may contain as many as 150 microcontrollers, and the average consumer now interacts with microcontrollers up to 300 times a day [VJW2000].

Embedded systems importance has become unquestionable. Their market size is about 100 times the desktop market. Hardly any new product reaches the market without embedded systems any more. Its strong increasing penetration in products and services creates huge opportunities for all kinds of enterprises and institutions. At the same time, the fast pace of penetration poses an immense threat for most of them. It concerns enterprises and institutions in such diverse areas as agriculture, health care, environment, road construction, security, mechanics, shipbuilding, medical appliances, language products, and consumer electronics.

Embedded systems are usually designed to perform selected functions at a low cost. The system may need to be very fast for some functions, but most of its other functions will probably not need speed.

So, often many parts of an embedded system will have low performance. The slowness is not just clock speed. The whole architecture of an embedded system is often intentionally simplified to lower costs compared to general-purpose computing hardware. For example, embedded systems often use peripherals controlled by synchronous serial interfaces, which are ten to hundreds of times slower than comparable peripherals used in PCs.

Since many embedded systems are mass-produced by the millions, reducing cost is a major concern. Some embedded systems do not require great processing power or resources and this allows production costs to be minimized by using a (relatively) slow processor and a small memory size. Programs on an embedded system often run with limited hardware resources: often there is no disk drive, operating system, keyboard or screen. The software may not have anything remotely like a file system, or if one is present, a flash driver may replace rotating media. If a user interface is present, it may be a small keypad and liquid crystal display. Table 1.1 shows examples of typical microcomputer embedded system applications [VJW2000].

Consumer	Function performed by the microcomputer
General purpose:	
Exercise equipment	Measures speed, distance, calories, heart rate, logs workouts
Remote controls	Accepts key touches and sends infrared (IR) pulses to base systems
Clocks and watches	Maintains the time, alarm, and display
Communication:	
Telephone answering machines	Plays outgoing message, saves and organizes messages
Telephone system	Interactive switching and information retrieval
Cellular phones and pagers	Key pads inputs, sound I/O, and communicates with central station
Automotive:	
Automatic breaking	Optimizes stopping on slippery surfaces
Noise cancellation	Improves sound quality by removing background noise
Electronic ignition	Controls speaks and fuel injections
Military:	
Smart weapons	Recognizes friendly targets
Missile guidance systems	Directs ordnance at the desired target
Global positioning systems	Determines where you are on the planet
Industrial:	
Setback thermostats	Adjusts day/light thresholds, thus saving energy
Traffic control systems	Senses car positions and controls traffic lights
Robot systems	Input from sensors, controls the motors
Medical:	
Cardiac monitors	Measures heart functions
Cancer treatments	Controls doses of radiation, drugs, or heat
Pacemakers	Helps the heart beat regularly

Table 1.1 Typical microcomputer embedded system applications

Embedded systems reside in machines that are expected to run continuously for years without errors. Their software is usually developed and tested more carefully than software for personal computers. Most embedded systems avoid mechanical moving parts such as disk drivers, switches or buttons because these are unreliable compared to solid-state parts such as flash memory. In addition, the embedded system may be outside the reach of humans (down an oil well borehole, launched into outer space, etc.), so the embedded system must be able to restart itself even if catastrophic data corruption has taken place.

Embedded real-time systems are applied in a wide variety of industrial sectors; they require a large number of different skills, including principally [BS2005]: application domain expertise, architectural design, application software, middleware, hardware design, fault tolerant design, safety techniques, verification and testing, just to name the most important areas. Existing embedded real-time computing infrastructure often introduces formidable barriers to continuous process improvement, equipment upgrades, and agility in responding to changing markets and increased global competitions [SJ1996]. Industry needs a computing infrastructure in which upgrades are safe and predictable, with negligible down-time. Embedded real-time systems must also be predictable [HI1998].

1.4. Statement toward fault tolerant embedded real-time systems

The particular characteristics of the embedded system context determine properties like distribution, autonomy, adaptability, dynamicity, performance, and real-time execution; that must be ensured by the systems of the future.

Observation and fault tolerance mechanisms are required in order to ensure that embedded real-time systems can adapt themselves according to the states and evolution of their execution contexts even in the presence of faults. A problem is that fault tolerance techniques are not fully implemented or they are developed in an ad-hoc fashion by different industries.

Embedded real-time systems represent several challenges in the investigation area; one of these challenges corresponds to the fault tolerance mechanisms that can be applied to them and the manner in which these mechanisms must be integrated. Another problem is that fault tolerance mechanisms studied by decades for classical distributed systems[1] could not be applied in the same form for embedded systems and in particular for those that have real-time requirements. In fact, the fault classification defined for classical distributed systems could also differ for the faults that could occur in embedded systems.

1.5. Objective

The objective of my work is to specify and implement a component fault tolerance framework, understanding framework like a support structure in which another software project can be organized and developed, that is composed by a set of abstract classes; and the specification of dependencies and interactions.

The fault tolerance framework has to be adaptable to the particular characteristics of architecture, physical constraints, development platform, and real-time operation of embedded real-time systems. A fault tolerance framework must include different fault tolerance models that can be applied according to a given embedded real-time system characteristics. A fault tolerance model specifies the fault causes, the way its presence can be detected and the associated recovery strategies.

The particular objectives are:

- Define a methodology for designing fault tolerance mechanisms (observation and recovery) to guaranty the reliability and vivacity of embedded real-time systems independently of the installation platform (memory restrictions, processing capacity, communication protocol).

[1] Appendix A presents the fault tolerance basic concepts for classical distributed systems.

- Establish techniques to have fault tolerance systems that satisfy the adaptability properties (even autonomous) in a static (on the conception) and dynamic (on execution time) way, remember the four category of the fault tolerance integration on distributed systems.

- Implement a subset of the fault tolerance framework with a component based programming model and validate it on an automotive embedded systems prototype.

1.6. Methodology

In the search of fault tolerance mechanisms that can be applied on embedded real-time systems, the framework approach seems to be adapted. An instantiation of a fault tolerance framework based on components will have the following properties:

- Reusability of software at minimal cost.
- Explicit rendering of the software architecture.
- Clear separation of the fault tolerant non functional code from the application specific code.

The benefit of such an approach is to ease the distribution of applications based on components over various hardware platforms, and to significantly help the configuration, monitoring and administration tasks.

1.7. Contributions

The contributions of my work are the definition of a fault tolerance embedded real-time system framework that can be used for the construction as much of small as of great embedded real-time systems, and the demonstration that it is feasible to use a component based programming model to implement a subset of the framework.

1.8. Document organization

This work is organized in six chapters:

- Chapter 2 presents a state-of-the-art of embedded real-time systems, including their base concepts, their operation and a brief description of the principal fault focuses, which are the network, the communication, and the intra-nodes operation.

- Chapter 3 defines a fault tolerance framework for embedded real-time systems composed by three layers, which are the network layer, the communication layer, and the node layer.

- Chapter 4 presents the policies to integrate the fault tolerance layers of the framework in an embedded real-time system; it defines a set of components that implement those aspects which compose each fault tolerance layer and it defines the necessary steps for its integration.

- Chapter 5 implements the communication layer with a component based programming model and exemplifies, as validation model, the implemented layer on an embedded real-time system of an electrical vehicle.

- Chapter 6 presents the conclusion of this work.