Chapter 8

How to Manage Failures in Air Traffic Control Software Systems

Luca Montanari, Roberto Baldoni, Fabrizio Morciano,
Marco Rizzuto and Francesca Matarese

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/51119

1. Introduction

Failure Management consists of a set of functions that enable the detection, isolation, and correction of anomalous behavior in a monitored system trying to prevent system failures. An effective failure management should monitor the system looking for errors and faults that could end up in a failure and overcome such issues when they arise.

Air Traffic Control (ATC) systems are large and complex systems supervising the aircraft trajectories from departure to destination. Such systems have hard reliability and dependability requirements. Having an effective failure management in such kind of critical systems is a must for safety and security reasons. Two main approaches have been developed in the literature to implement these failure management systems:

- Reactive Fault Management;
- Proactive Fault Management.

Due to the complexity and the strong requirements, current ATC systems adopt both of them. The Reactive Fault Management is based on the detection paradigm. A reactive fault manager get triggered at the moment in which errors occur and should have the following capability: diagnosis, symptom monitoring, correlation, testing, automated recovery, notification, online system topology update. The Proactive Fault management scheme anticipates the formation of erroneous system states before it actually materializes into a failure. Known techniques in this field are rejuvenation of system components [24], checkpointing [4], prediction mechanisms [21]: to predict a failure occurrence and thus triggering the system state recovery.

This chapter focus on failure management in ATC systems pointing out motivations that led engineers to do specific design choices. Two case studies as real implementations of the paradigms are also presented: a reactive approach deployed in a real ATC System and a novel proactive approach that has the distinctive features to be (i) *black-box*: no knowledge of applications' internals and logic of the mission critical distributed system is required (ii) *non-intrusive*: no status information of the nodes (e.g., CPU) is used; and (iii) *online*: the analysis is performed during the functioning of the monitored systems.

The chapter is organized as follows: section 2 explains the motivations of failure management in ATC. Section 3 shows the role that faults and failures have in ATC systems and the relationship with safety regulation. Section 4 presents the objectives of failure management while section 5 investigates proactive, reactive approaches and we will introduce the online failure prediction technique. The sections 6 and 7 present the two case studies, one using a reactive approach and one that use a proactive approach. Section 8 concludes the chapter.

2. Motivations

Distributed mission critical systems such ATC, battlefield or naval command and control systems consist of several applications distributed over a number of nodes connected through a LAN or WAN. The applications are constructed out of communicating software components that are deployed on those nodes and may change over time. The dynamic nature of applications is mainly due to (i) adopted policies to cope with software or hardware failures, (ii) load balancing strategies and (iii) the management of new software components joining the system. Additionally such systems have to react to input in a soft real time way, i.e., an output has to be provided after a few seconds from the input the generated it. In such complex real time systems, failures may happen with potentially catastrophic consequences for their entire functioning. The industrial trend is to face failures by using appropriate software engineering techniques at the design phase. However these techniques cannot reduce to zero the probability of failures during the operational phase due to the unpredictability and uncertainty behind a distributed systems [7], thus there is the need of supervising services that are not only capable of detecting a failure, but also predicting and preventing it through an analysis of the overall system behavior.

The literature about failures and fault management embraces several aspects: reactive approaches, proactive approaches, fault detection, failure detection, faults and failure isolation, failure prediction. Before investigating some details of these techniques, it is important to point out some definitions that will be used along this chapter [11]:

- The *system behavior* is what the system does to implement its function;
- A *failure* is an event that occurs when the delivered service deviates from correct service;
- A *fault* is the cause of an error.
- An *error* is a deviation in the sequence of the system's states.

A fault is *active* when it produces an error, it is *dormant* otherwise. The next section specializes the faults and failures in the ATC domain.

3. Faults and failures in ATC systems and relationship with safety regulation 482/2008

An ATC system is a large and complex system with several interrelated functions. It receives inputs from several heterogeneous actors like: messages from external lines (e.g. AFTN), radar information, radio communications with aircraft etc. All these information need to be integrated, processed, correlated and finally presented to an ATC system as a global operational picture of the sky. A controller looks at this picture and, according to the adopted procedures, addresses the aircraft pilot in the safest way ensuring to select the most efficient trajectory for reaching the final destination.

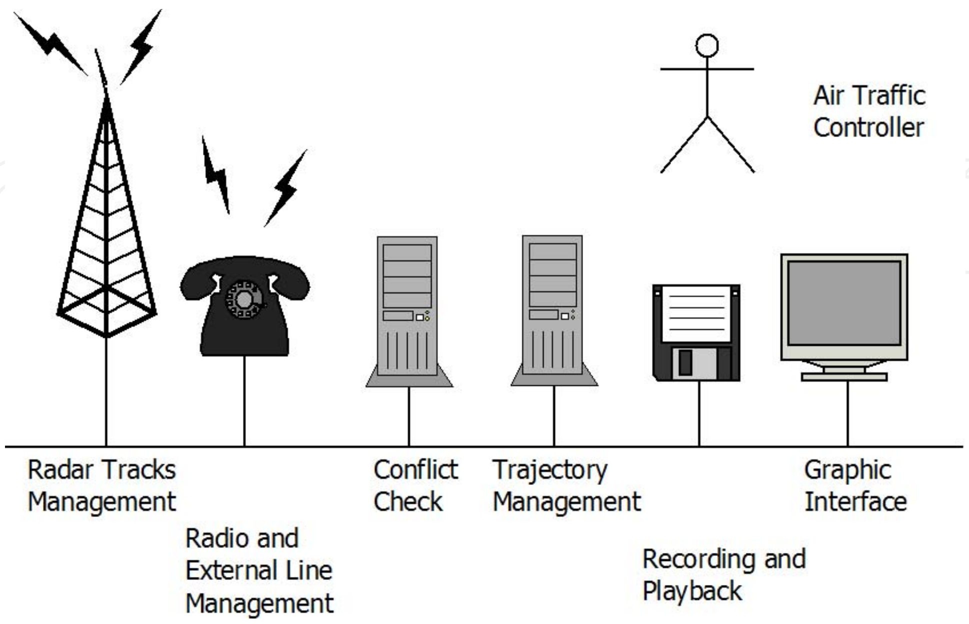

Figure 1. ATC Very High Level Architecture

A very high level architecture of an ATCs is shown in Fig. 1. The Figure highlights the needs for an ATCs in term of hardware, software and human factors. The number of components involved can change depending on the vendor, size of the system and requirements from the customers, still to give a rough idea of the order of magnitude of the size of the system, an ATC system is several million lines of code.

ATCs does not require strict real-time time of responses (the separation criteria can be around seconds) but the availability of the system should be greater than 99,99%. An ATCs architecture requires at least the following capabilities:

- discovering a fault in a predictable time;
- sharing the same data among all the components forming the system;
- maintaining the service or restore it in a predictable time.

According to the previous criteria we can identify some class of faults:

- misalignment in time (not all the system is aware of its processing capacity);
- misalignment in data (not all the system shares the same information);
- misalignment in functionalities (not all the capabilities are available).

The first class of faults implies failures related to delay in communications (inside and outside the system) and, human factor (wrong order). Faults related to the hardware are minimized by a proper configuration and tuning of the ATC system. In the worst case the entire ATC system can be replaced by a different one using a separate network and possibly employing different hardware and software components to exploit diversity argument (sometimes the previous version of the ATC system is used as fallback).

The second class of faults implies failures related to the mismatch between the output of processing server in the system; part of the system could process data no longer relevant with respect to the real status of ATC system. This impacts ATC systems as they cannot rely anymore on the information provided by the system.

The third class of faults implies failures related to degraded system usability, part of the system cannot be used and its functionality cannot be accessed by ATC systems or software components.

Safety is an essential characteristic of AirTrafficManagement/ATC functional systems. It has a dominant impact upon operational effectiveness. ATM/ATC functional systems in now evolving in a continuously growing integrated environment including automation of operational functions, formerly performed through manual procedures and massive and systematic use of software. All this has a prominent impact for the achievement of safety [5]. Moreover, regulatory compliance has become a legal and necessary extension of business continuity with an increasingly complex set of laws and regulations relating to data integrity and availability. Ensuring the integrity and availability for ATC systems bring bad and good news on regulatory compliance. The bad news is the regulations do not provide a "blueprint" for protection. The good news is high availability and continuous availability protection strategies will help you meet these regulatory requirements, minimizing the risk that under-protected systems will create breaks in the "chain of data". It is important to note that compliance is a moving target; both government and industry leaders will continue to move toward more specific regulations and standards [12]. The issue of regulatory compliance has became more acute on 1st January 2009 when the Regulation (EC) 482/2008 "establishing a software safety assurance system to be implemented by air navigation service providers" went into effect [3]. Still, laws or regulations do not set a specific process or specific requirements for an ATC system, they just describe expected outcomes. The Software Fault Management System supports business continuity and Regulation (EC) 482/2008 compliance, by identifying a set of "risk-mitigation means", defined from the risk-mitigation strategy achieving a particular safety objective. Moreover, it provides:

- "cutover or hot swapping", that is the approach of replacing European air traffic management network (EATMN) system components while the system is operational;
- "software robustness", that is the robusteness of the software in the event of unexpected inputs, hardware faults and power supply interruptions, either in the computer system itself or in connected devices; and
- "overload tolerance", that is the tolerance of the system to, inputs occurring at a greater rate than expected during normal operation of the system.

4. Failure management objectives

The objective of the failure management can be broadly divided in *Failure Detection*, *Failure Isolation*, *Failure Identification*. Failure detection and isolation has become a critical issue in the operation of high-performance ships, submarines, airplanes, space vehicles, and structures, where safety, mission satisfaction, and significant material value are at stake. [8] presents a survey on the failure detection techniques introducing some basic definition:

1. *Failure Detection* is the task to produce an indication that something is going wrong, i.e. a failure is present in the system.

2. *Failure Isolation* is the determination of the exact location of a failure.
3. *Failure Identification* is the determination of the size of the failure.

While detection and isolation are a must in any mission critical system, failure identification can be an overkill and then sometime it is not implemented.

5. Failure management techniques - reactive and proactive

5.1. Reactive approach

The reactive approach in fault management is based on the detection paradigm. A reactive fault manager gets triggered at the moment in which errors occurs. More specifically, in order to achieve it's main goals, it is necessary to have the following capabilities [10]:

- *Symptom monitoring*: Symptoms are manifestations of underlying faults and must be monitored to detect the occurrence of problems as soon as they happen. A fault manager quality is its response time to symptoms. The quicker this reaction occurs, the higher the probability to recover the system error is. This in turn raise the probability that the fault will not end up into a failure. In our case study the middleware platform used makes use of FT-CORBA (Fault tolerant CORBA) which rely on the fault detection to implement tolerance logics (see section 6)
- *Diagnosis*: identifies the root causes of "known" symptoms. A fault may originate on one component and then it could manifest on some other component. In large scale systems, there is no one-to-one mapping between faults, errors, failures. Studies on such systems have shown that typically up to 80% of the fault management effort is spent in identifying root causes after the manifestation of symptoms [23].
- *Correlation*: a correlation capability provides knowledge about root causes of "known" symptoms to the diagnosis modules. Modern systems are often richly instrumented with a large number of sensors that provide large amounts of information in the form of messages and alarms. This flow of information cannot be handled by humans in real-time as a small number of roots causes results in a huge number of messages and alarms. Therefore it is necessary to provide them with concise and aggregate notifications of underlying root causes. Correlation is the process of recognizing and organizing groups of events that are related each other.
- *Testing*: in large software systems, it is impractical (and sometime impossible) to monitor every variable. Instead key observable variables are monitored to generate symptom events. Diagnostic inference typically identifies a set of suspected root causes. A test planning facility is needed to select additional variables to be examined to isolate the root causes. The fault management application then needs to request or run these tests, and utilizes their results to complete the diagnosis. A test, as originally defined in [22], can incorporate arbitrarily complex analysis and actions, as long as it returns a true or false value.
- *Automated recovery*: identifying and automating recovery procedures facilitate rapid response to problems and allow for growth in equipment, processes, and services, without increasing the supervisory burden on system operators. The automation in recovery decreases the response time to an error and thus decreasing the probability that it may cause a proper system failure.

- *Notification*: system operators require notifications of all critical fault management activity, especially the identification of root causes, and causal explanations for alarms, tests, and repair actions in a manner that they can follow easily. Sometimes they need to distinguish between what is observed by system sensors versus what is inferred by the underlying fault management application.
- *Postmortem*: information from diagnostic problem solving is fed back to the fault management system for historic record keeping in order providing enough data for offline failure analysis to discover some of the mappings between failures and their root cause. It is important to underline that this analysis is different than the offline analysis to discover failure patterns. Failure patterns and relationships between failures and the root cause are orthogonal concepts even if some relationships between failures and faults can form a failure pattern. That's because failures are not caused just by errors or faults but also by system configurations and human interaction patterns.
- *Online system topology update*: in an ATC system the reactive fault manager should support expert systems for effective diagnosis of root causes of system errors and that the expert system uses a knowledge base to infer the right diagnosis. The knowledge base as a module can be replaced or connected to another knowledge base. Other components can be completely removed or added. All this dynamic changes need to be done at run-time. It may not be feasible indeed to take the fault management system off-line each time that there is a change in the system topology.

5.2. Proactive fault management

Using reactive schemes there are limits to increasing mission critical systems availability. Failure management started looking at proactive approaches to overcome these limitations such as rejuvenation of system components [10]. This scheme anticipates the formation of erroneous system states before it actually materializes into a failure. The listed schemes to increase system availability can be a more effective idea if applied intelligently and preventively. The question remains: when we should apply check-pointing and rejuvenation techniques? To answer this question we need a way to tell if the current state of the system is going to evolve into a failure state. We can extend this concept to include parts or the entire history of the systems state transitions. So to answer the question of the ideal trigger timing for high availability schemes we need to develop a model of the system in question which allows us to derive optimized triggering times. To increase availability of a software system during runtime basically two main concepts are involved: The method to re-initiate the system or a component to a failure free state like rejuvenation and a prediction mechanism to predict a failure occurrence and thus trigger the system state recovery.

5.3. Online failure prediction

The problem of modeling a system had always the main objective of predicting its behavior. A significant body of work has been published in this area. As far as distributed systems is concerned, a recent work [20] introduces a taxonomy that structures the manifold of approaches. The more relevant approaches for the ATC purpose are the following ones:

- Symptoms Monitoring: Manifestation of faults is not necessary a clear situation rather than a more fuzzy one. It can influence the hosted system gradually in time and space. This type of symptoms is called service degradation. A prominent example for such types of symptoms in ATC or, in general, in mission critical systems is *response-time*. The fault underlying this symptom might be a bad process priority management and consequent starvation of some other processes having lower priority. The key notion of failure prediction based on monitoring data is that faults like starvation in priority management can be detected by their side effects such as high response-time. These side-effects are called symptoms. Later on this chapter (section 7) we will show an architecture for online failure prediction in ATC that uses symptoms monitoring.

- Error Detection: Once a fault manifests itself, it becomes an error. Errors and symptoms are different: symptoms are the observation of system state over time; a symptom is a behavior that deviates from the "normal" behavior. While error is something that actually goes wrong. The fault at this stage did not develop in service failure yet but it would possibly do it. What is the probability that this error ends up in a failure? for how long since the first occurrence this probability keeps high? Error detection approaches attempt to answer these questions. The error detection usually employs online failure predictors based on rules, data-mining approaches, pattern recognition, fault trees etc.

6. Reactive approach case study: FT CORBA in a real ATC system

6.1. Motivation

"There shall be no single point of failure", this is one of the basic requirements for any ATC system. It drives alone many choices about the design, the used technologies, the verification strategies of a complex distributed system which has to provide a very high service availability : at least 99,99% i.e. downtime of about 5 minutes per month. The complexity of such systems is more and more stored in the software, which is error prone to problems injected at design or coding time as well as to unexpected scenarios due to runtime concurrency and other factors, like for example upgrading activities. Then software fault tolerance stands beside the traditional hardware based solutions and often replaces them, considering also that these systems are maintained and can evolve over a 25 years lifecycle: any chosen solution must support changes. In this context FT CORBA is widely used in ATC, but also in Naval Combat Management and other Command and Control systems. FT CORBA provides both replication and failure transparencies to the application and moreover it is standardized by the Object Management Group [18] [15].

6.2. Principle of FT CORBA

The FT CORBA specification defines an architecture and a framework for resilient, highly-available, distributed software systems suitable for a wide range of applications, from business enterprise applications to distributed, embedded, real-time applications. The basic concepts of FT CORBA are entity redundancy, fault detection and fault recovery; replicated entities are several instances of CORBA objects that implement a common interface and thus are referenced by an object group (Interoperable Object Group Reference, IOGR). IOGRs lifecycle and update are totally managed by the FT CORBA infrastructure; client applications are unaware of object replication and changes in the object group due to replica failure are transparent since their request are forwarded to the right replica. The infrastructure (see Fig. 2) provides means to monitor the replicated objects and to communicate the faults, as well as to notify the fault to other interested parties, which could contribute to recover the application. Beyond replication, object groups and complete transparency, FT CORBA relies

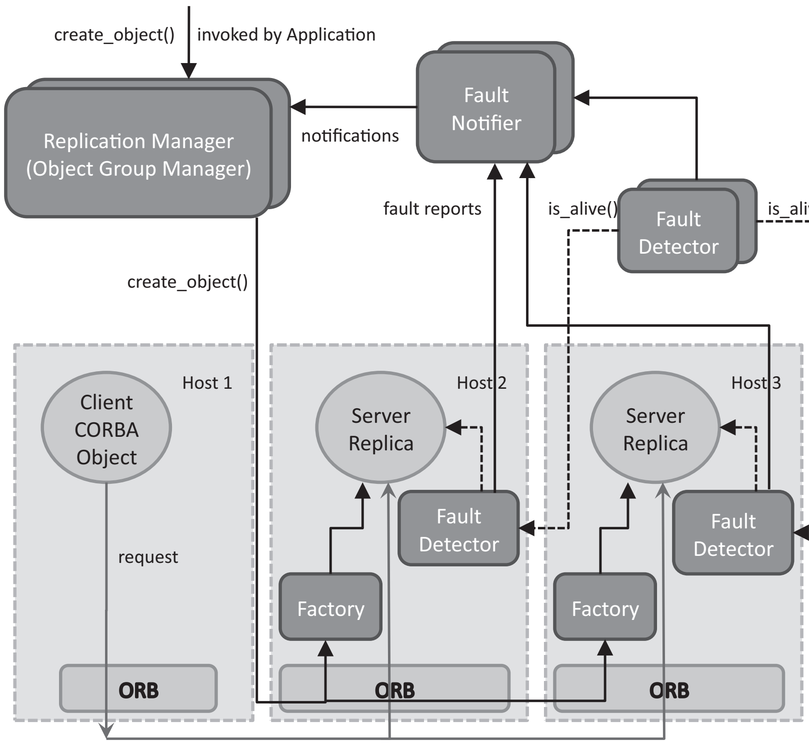

Figure 2. FT CORBA framework

also on infrastructure-controlled consistency. Strong replica consistency is enforced in order to guaranty that the sequence of requests invoked on the object group passes unaltered across the fault of one or more replicas.

6.3. Specialization of FT CORBA for safety critical systems: CARDAMOM use case

In the following we are going to focus on the design choices made for a significant piece of a real ATC system, namely CARDAMOM [2], and that is implemented in a CORBA based middleware.

Among the different replication styles, CARDAMOM adopts the warm passive approach to replicate statefull servers: during normal operation, only one member of the object group, the primary replica, executes the methods invoked on the group. The backup replicas are warm because they receive the status updates at the end of each request from the primary; this way they are always ready to process the next request, in case the primary fails. The FT infrastructure is in charge of detecting such failure and of triggering the switch to a new primary. Transferring to the backup replicas the updated status and the list of processed request ids, it is guaranteed that requests are always served exactly once as long as there are available replicas.

The software architecture is based on CORBA Component Model (see Fig. 3) and then the natural unit of redundancy is a component of the CCM; This Component is a unit of design, development and deployment realized through a collection of CORBA Objects which define attributes and interfaces, called ports [14]. In this context, the exposed ports (facets) of the server components are defined as objects of FT CORBA groups. This approach suits

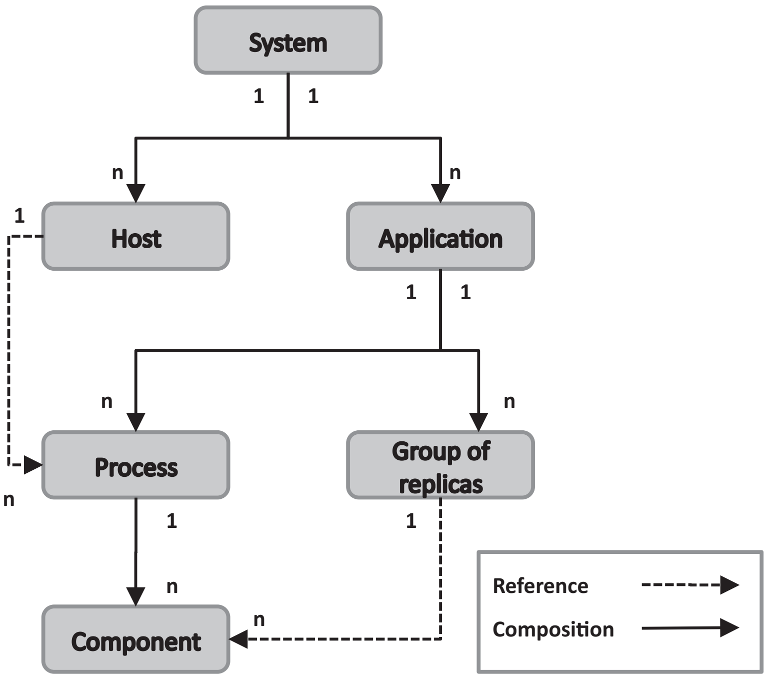

Figure 3. System decomposition in application, process, component, group and host.

well with FT CORBA specification but put in evidence an operative need: in Operating Systems that manage the process as unit of memory space and failure (e.g. POSIX process in Linux/Unix), monitoring and recovery should be done at process level. Then CARDAMOM restricts FT CORBA entity redundancy by enforcing that within the same process all replicated components play the same role, that is all primaries or all backups. This need is also tackled by an extension of FT CORBA specification, the beta OMG specification "Lightweight Fault Tolerance for Distributed RT Systems" [17].

A very important aspect of CARDAMOM is the fault detection; since the framework is tuned to react and recover from failures, namely a process crash, mechanisms are put in place to detect malfunctions like for example deadlocks or endless loops which do not lead necessarily to a crash. After the detection, most of the times the safest action to recover normal behavior is to stop or kill the faulty process in order to trigger a switch to a new replica. Normally fault detectors work with several patterns at the same time: they can use a pull model, e.g. "is alive" call, or push model, e.g. by handling OS signals to detect the death of processes or even be signaled by the application itself after a fatal error.

FT CORBA with warm-passive replication style fits well the need of statefull servers which must guarantee the processing of sequenced requests. However, an ATC system needs other components to be resilient to failures act as stateless components. Generally speaking, stateless components have to provide their services with high availability but do not need to check for "exactly once" semantics of client requests either to support the state transfer. In this case it is used the Load Balancing framework, specified at OMG by the Lightweight Load Balancing specification [16]. It reuses the object group definition of FT CORBA and allows to transparently redirect the client requests among a pool of server replicas according to predefined or user defined strategies, for example through random or round-robin policies. In this way two conflicting goals are achieved at the same time: distribute the computational load among several resources and supporting fault tolerance. because fault detectors are used to update the object group in case of failure and activate recovery mechanism. An additional and important feature is also to prevent that several replicas may crash because of the same implementation: by means of fine request identification, the framework allows to stop those requests that have caused failures, thus avoiding repetitive crashes which would result in a complete system failure.

Middleware CARDAMOM provides all the previously mentioned services (see Fig. 4): in fact it has been chosen as the foundation for a safety critical subsystem, the core part of a next generation ATC system.

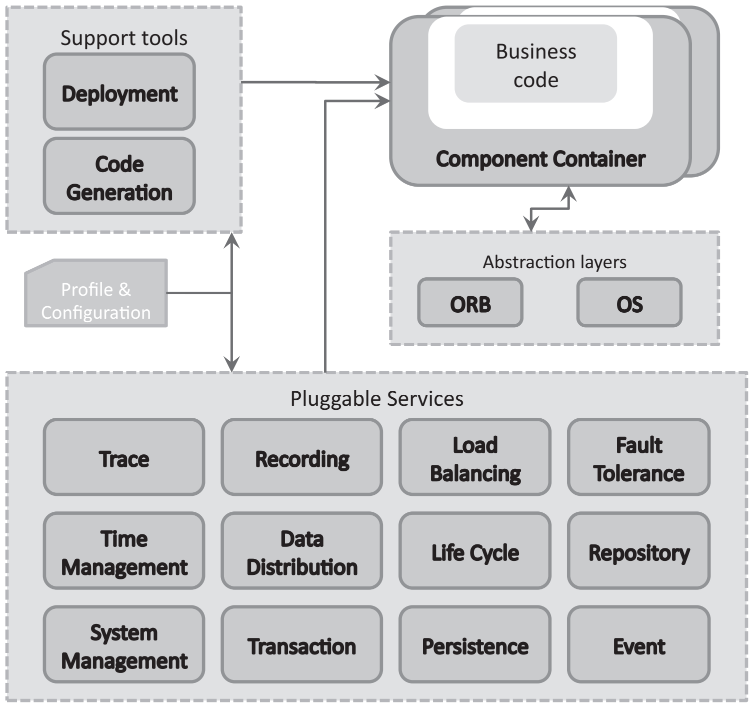

Figure 4. CCM and CORBA based middleware services.

In order to separate duties and define a clearly decoupled architecture that could support extensibility and maintainability, a three tier model has been put in place for the building blocks of the ATC system using CARDAMOM services. The first tier provides the interface to the external clients and guaranties the ordered processing of requests; it is realized by statefull components replicated with FT CORBA and warm passive replication style. The second tier executes the business logic; it is realized by stateless components replicated with LwLB supporting fault containment for killer requests; the third tier tackles the data management and persistency.

This architecture (see Fig. 5) is proven to be, at the same time, resilient to failures and highly scalable in terms of computational power, thus responding to the opposite requirements coming from availability, safety and performances. The use of FT and LB CORBA services is strongly interrelated also with System Management services, that are informed of replica crashes by the Fault Notifier. Automatic actions are put in place in order to stop or restart the replicas and contribute to the overall system availability; actions like restart and stop can be defined with different level of granularity, that is for process, application or host according to

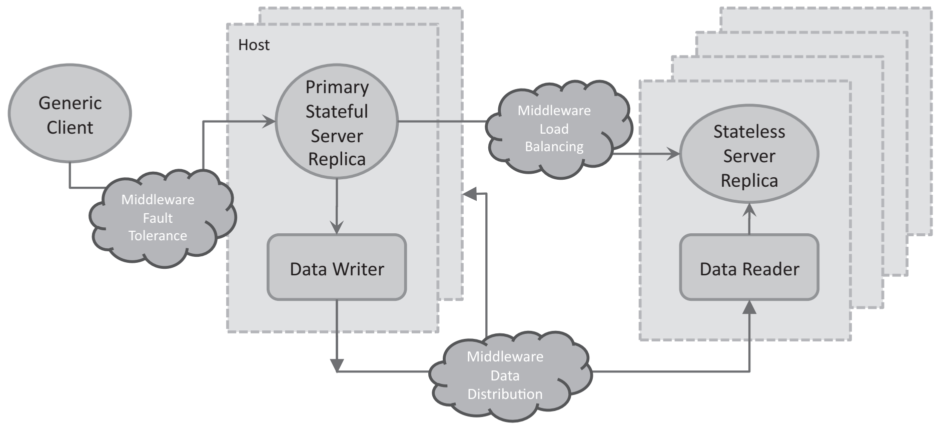
Host
Generic Client
Primary Stateful Server Replica
Middleware Fault Tolerance
Middleware Load Balancing
Stateless Server Replica
Data Writer
Data Reader
Middleware Data Distribution

Figure 5. 3-tier architecture.

the kind of failure. As final consideration it is very important to underline that the design and implementation of the middleware services that provide this fault tolerant framework have to be themselves fault tolerant.

7. Failure prediction case study: CASPER

In this section we introduce the design, implementation and experimental evaluation of a novel online, non-intrusive and black-box failure prediction architecture we named CASPER that can be used for monitoring mission critical distributed systems. CASPER is (i) *online*, as the failure prediction is carried out during the normal functioning of the monitored system, (ii) *non-intrusive*, as the failure prediction does not use any kind of information on the status of the nodes (e.g., CPU, memory) of the monitored system; only information concerning the network to which the nodes are connected is exploited as well as that regarding the specific network protocol used by the system to exchange information among the nodes (e.g., SOAP, GIOP); and (iii) *black-box*, as no knowledge of the application's internals and of the application logic of the system is analyzed. Specifically, the aim of CASPER is to recognize any deviation from normal behaviors of the monitored system by analyzing symptoms of failures that might occur in the form of anomalous conditions of specific performance metrics. In doing so, CASPER combines in a novel fashion Complex Event Processing (CEP) [13] and Hidden Markov Models (HMM) [19]. The CEP engine computes at run time the performance metrics. These are then passed to the HMM in order to recognize symptoms of an upcoming failure. Finally, the symptoms are evaluated by a failure prediction module that filters out as many false positives as possible and provides at the same time a failure prediction as early as possible. We deployed CASPER for monitoring a real ATC system. Using the network data of such a system in the presence of both steady state performance behaviors and unstable state behaviors, we first trained CASPER in order to stabilize HMM and tune the failure prediction module. Then we conducted an experimental evaluation of CASPER that aimed to show its effectiveness in timely predicting failures in the presence of memory and I/O stress conditions.

7.1. Failure and prediction model

We model the distributed system to be monitored as a set of nodes that run one or more services. Nodes exchange messages over a communication network. Nodes or services can be

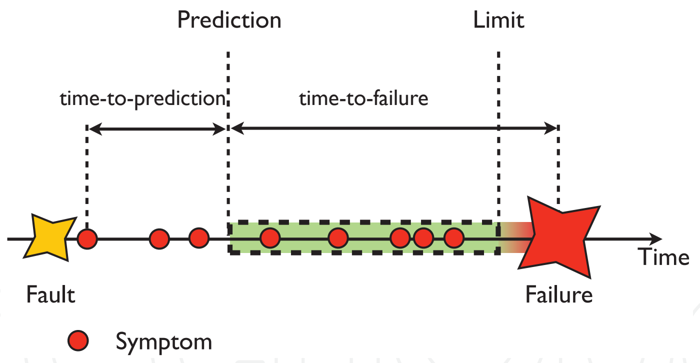

Figure 6. Fault, Symptoms, Failure and Prediction

subject to failures. A failure is an event for which the service delivered by a system deviates from its specification [11]. A failure is always preceded by a fault (e.g., I/O error, memory misusage); however, the vice versa might not be always true. i.e., a fault inside a system could not always bring to a failure as the system could tolerate, for example by design, such fault.

Faults that lead to failures, independently of the fault's root cause, affect the system in an observable and identifiable way. Thus, faults can generate side-effects in the monitored systems till the failure occurs. Our work is based on the assumptions that a fault generates increasingly unstable performance-related symptoms indicating a possible future presence of a failure, and that the system exhibits a steady-state performance behavior with a few variations when a non-faulty situation is observed [25]. In Figure 6 we define *Time-to-failure* the distance in time between the occurrence of the prediction and the software failure event. The prediction has to be raised before a time *Limit*, beyond which the prediction is not sufficiently in advance to take some effective actions before the failure occurs. We also consider the *time-to-prediction* which represents the distance between the occurrence of the first symptom of the failure and the prediction.

7.2. CASPER architecture

The architecture designed is named CASPER and is deployed in the same subnetwork as the distributed system to be monitored. Figure 7 shows the principal modules of CASPER that are described in isolation as follows.

Pre-Processing module. It is mainly responsible for capturing and decoding network data required to recognize symptoms of failures and for producing streams of events. The network data the Pre-Processing module receives as input are properly manipulated. Data manipulation consists in firstly decoding data included in the headers of network packets. The module manages TCP/UDP headers and the headers of the specific inter-process communication protocol used in the monitored system (e.g., SOAP, GIOP, etc) so as to extract from them only the information that is relevant in the detection of specific symptoms (e,g., the timestamp of a request and reply, destination and source IP addresses of two communicating nodes). Finally, the Pre-Processing module adapts the extracted network information in the form of *events* to produce streams for the use by the second CASPER's module (see below).

Symptoms detection module. The streams of events are taken as input by the Symptoms detection module and used to discover specific performance patterns through complex event processing (i.e., event correlations and aggregations). The result of this processing is a system

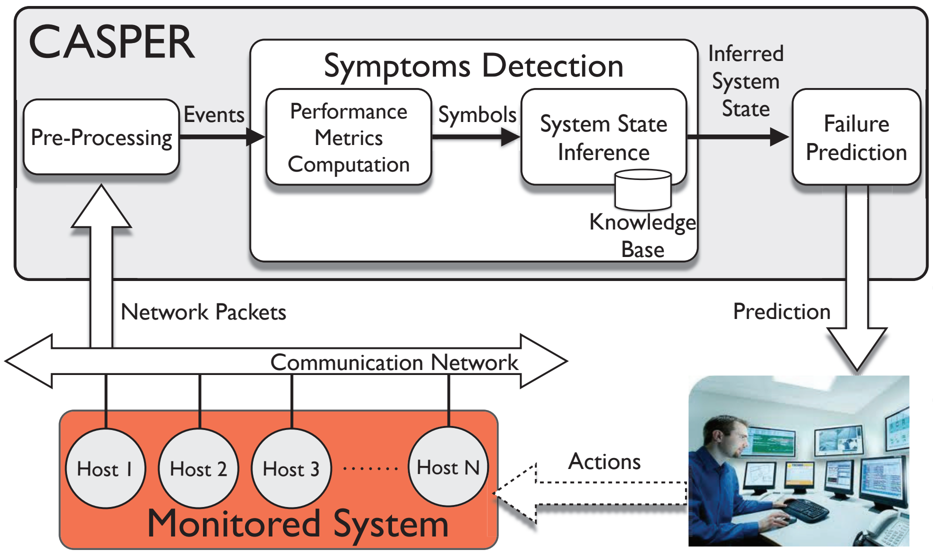

Figure 7. The modules of the CASPER failure prediction architecture

state that must be evaluated in order to detect whether it is a safe or unsafe state. To this end, we divided this module into two different components, namely a *performance metrics computation* component and a *system state inference* component.

The performance metrics computation component uses a CEP engine for correlation and aggregation purposes. It then periodically produces as output a representation of the system behavior in the form of *symbols*. Note that, CASPER requires a *clock mechanism* in order to carry out this activity at each *CASPER clock cycle*. The clock in CASPER allows it to model the system state using a discrete time Markov chain and let the performance metrics computation component coordinate with the system state inference one (see below). The representation of the system behavior at run time is obtained by computing *performance metrics*, i.e., a set of time-changing metrics whose value indicates how the system actually works (an example of network performance metric can be the round trip time). In CASPER we denote symbols as σ_m (see Figure 8), where $m = 1, \ldots, M$. Each symbol is built by the CEP engine starting from a vector of performance metrics: assuming P performance metrics, at the end of the time interval (i.e. the clock period), the CEP engine produces a symbol combining the P values. The combination of performance metrics is the result of a discretization and a normalization: each continuous variable is discretized into slots of equal lengths. The produced symbol represents the state of the system during the clock period.

The system state inference component receives a symbol from the previous component at each CASPER clock cycle and recognizes whether it is a correct or an incorrect behavior of the monitored system. To this end, the component uses the Hidden Markov Models' forward probability [19] to compute the probability that the model is in a given state using a sequence of emitted symbols and a knowledge base(see Figure 7). We model the system state to be monitored by means of the *hidden process*. We define the states of the system (see Figure 8) as *Safe*, i.e., the system behavior is correct as no active fault [11] is present; and *Unsafe*, i.e., some faults, and then symptoms of faults, are present.

Failure Prediction module It is mainly responsible for correlating the information about the state received from the system state inference component of the previous CASPER module. It takes in input the inferred state of the system at each CASPER clock-cycle. The inferred state

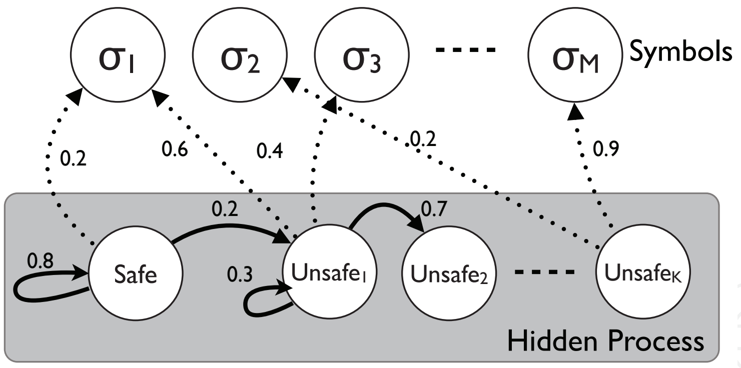

Figure 8. Hidden Markov Models graph used in the system state inference component

can be a safe state or one of the possible unsafe states. Using the CEP engine, this module counts the number of consecutive *unsafe* states and produces a failure prediction alert when that number reaches a tunable threshold (see below). We call this threshold *window size*, a parameter that is strictly related to the *time-to-prediction* shown in Figure 6.

7.2.1. Training of CASPER

The knowledge base concerning the possible safe and unsafe system states of the monitored system is composed by the parameters of the HMM. This knowledge is built during an initial training phase. Specifically, the parameters are adjusted by means of a training phase using the max likelihood state estimators of the HMM [19]. During the training, CASPER is fed concurrently by both recorded network traces and a sequence of pairs `<system-state,time>`. Each pair represents the fact that at time `<time>` the system state changed in `<system-state>`[1].

7.2.2. Tuning of CASPER parameters

CASPER architecture has three parameters to be tuned whose values influence the quality of the whole failure prediction mechanism in terms of false positives and time-to-prediction. These values are (i) the length of the CASPER *clock period*; (ii) the *number of symbols* output by the performance metrics computation module; (iii) the length of the failure prediction, i.e., *window size*.

The length of the clock period influences the performance metrics computation and the system state inference: the shorter the clock period is, the higher the frequency of produced symbols is. A longer clock period allows CASPER to minimize the effects of outliers. The number of symbols influences the system state inference: if a high number of symbols is chosen, a higher precision for each performance metrics can be obtained. The failure prediction window size corresponds to the minimum number of CASPER clock cycles required for raising a prediction alert. The greater the window size, the more the accuracy of the prediction, i.e., the probability that the prediction actually is followed by a failure (i.e. a true positive prediction). The tradeoff is that the time-to-prediction increases linearly with the windows size causing shorter time-to-failure (see Figure 6); During the training phase, CASPER automatically

[1] As the training is offline, the sequence of pairs `<system-state,time>` can be created offline by the operator using network traces and system log files.

chooses the best values for both clock period and number of symbols, leaving to the operator the responsibility to select the windows size according to the criticality of the system to be monitored.

7.3. Monitoring a real ATC system with CASPER

CASPER has been tested on the same real ATC system used in the reactive approach case study (section 6). CASPER intercepts GIOP messages produced by the CORBA middleware and extracts several information from them in order to build the representation of the system at run time. In this section we describe how the events are represented starting from the GIOP messages and how the performance metrics representing the system state are computed.

Event representation. Each GIOP message intercepted by CASPER becomes an event feeding the CEP engine of the performance metrics computation component. Each event contains (i)*Request ID*: The identifier of a request-reply interaction between two CORBA entities; (ii)*Message Type*: A field that characterizes the message and that can assume different values (e.g., Request, Reply, Close Connection) and (iii)*Reply Status*: It specifies whether there were exceptions during the request-reply interaction and, if so, the kind of the exception. In addition, we insert into the event further information related to the lower level protocols (TCP/UDP) such as source and destination IP, port, and timestamp. In order not to capture sensitive information of the ATC system (such as flight plans or routes), CASPER ignores the payload of the messages.

Performance metrics. Events sent to the CEP engine are correlated online so as to produce so-called performance metrics. After long time of observations of several metrics of the ATC CORBA-based system, we identified the following small set of metrics that well characterize the system, showing a steady behavior in case of absence of faults, and an unstable behavior in presence of faults:

- *Round Trip Time:* elapsed time between a request and the relative reply;
- *Rate of the messages carrying an exception:* the number of reply messages with exception over the number of caught messages;
- *Average message size:* the mean of the messages size in a given spatial or temporal window;
- *Percentage of Replies:* the number of replies over the number of requests in a given spatial or temporal window;
- *Number of Requests without Reply:* the number of requests expecting a reply that, in a given temporal window, do not receive the reply;
- *Messages Rate:* the number of messages exchanged in a given spatial or temporal window.

To compute the performance metrics we correlate the sniffed network data using the CEP engine ESPER [6]. This choice is motivated by its low cost of ownership compared to other similar systems (e.g. [9]) and its offered usability.

7.4. CASPER experimental evaluation

The first part of the evaluation on the field has been to collect a large amount of network traces from the ATC underlying communication network when in operation. These traces represented steady state performance behaviors. Additionally, on the testing environment of the ATC system we stressed some of the nodes till achieving software failure conditions, and we collected the relative traces. In our test field, we consider one of the nodes of the ATC system to be affected by either Memory or I/O stress (according to the experience of the ATC designers, these two stress conditions are typical of the observed system). After collecting all these traces, we trained CASPER. At end of the training phase, we deployed CASPER again on the testing environment of the ATC system in order to conduct experiments in operative conditions. Our evaluation assesses the system state inference component accuracy and the failure prediction module accuracy (see Figure 7). In particular, we evaluate the former in terms of N_{tp} (number of true positives) the system state is unsafe and the inferred state is "system unsafe"; N_{tn} (number of true negatives): the system state is safe and the inferred state is "system safe"; N_{fp} (number of false positive): the system state is safe but the inferred state is "system unsafe"; and N_{fn} (number of false negatives): the system state is unsafe but the inferred state is "system safe". Using these parameters, we compute the following metrics that define the accuracy of CASPER:

- Precision: $p = \frac{N_{tp}}{N_{tp}+N_{fp}}$
- Recall (or true positive rate): $r = \frac{N_{tp}}{N_{tp}+N_{fn}}$
- F-measure: $F = 2 \times \frac{p \times r}{p+r}$
- False Positive Rate: $f.p.r. = \frac{N_{fp}}{N_{fp}+N_{tn}}$

We evaluate the latter module in terms of N_{fp} (number of false positive): the module predicts a failure that is not actually coming and N_{fn} (number of false negatives): the module does not predict a failure that is coming. **Testbed.** We deployed CASPER in a dedicated host located in the same LAN as the ATC system to be monitored (see Figure 7). This environment is actually the testing environment of the ATC system where new solutions are tested before getting into the operational ATC system. The testing environment is composed by 8 machines, 16 cores 2.5 GHz CPU, 16 GB of RAM each one. It is important to remark that CASPER does not know the application nor the service logic nor the testbed details.

7.5. Faults and failures

The ATC testbed includes two critical servers: one of the server is responsible for disk operations (I/O) and another server is the manager of all the services. In order to induce software failures in the ATC system, we apply the following actions in such critical servers: (i)*memory stress*; that is, we start a memory-bound component co-located with the manager of all ATC services, to grab constantly increasing amount of memory resource; (ii)*I/O stress*; that is, we start an I/O-bound component co-located with the server responsible for disk operations, to grab disk resources. In both cases we brought the system to the failure of critical services. During the experiment campaign, we also considered the CPU stress; however, we discovered that due to the high computational power of the ATC nodes, the CPU stress never causes failures.

7.6. Results of CASPER

We run two types of experiments once CASPER was trained and tuned. In the first type, we injected the faults described in previous section in the ATC testing environment and we carried out 10 tests for each type of fault. In general, we obtained that in the 10 tests we carried out, the time-to-failure in case of memory stress varied in the range of [183s, 216s] and the time-to-prediction in the range of [20.8s, 27s]. In case of I/O stress, in the 10 tests, the time-to-failure varied in the rage of [353s, 402s] whereas the time-to-prediction in the range of [19.2s, 24.9s]. The time-to-failure in our evaluation has been long enough in order to trigger proper countermeasures, that can be set before the failure, to either mitigate damages or enable recovery actions. Further details can be found in [1].

8. Conclusion

This chapter presented the motivations that led the current literature to develop novel solutions to failure management in ATC systems. We analyzed the failure management objectives, what the faults and failures are and how they can be managed in a real ATC system. Some hint on the failure management reactive and proactive approaches have been described and two case studies have been presented: a reactive approach, that uses FT-CORBA, today in operation and a novel proactive approach that uses a combination of Complex Event Processing and Hidden Markov Models to predict the occurrence of failures in ATC systems.

Author details

Luca Montanari and Roberto Baldoni
"Sapienza" University of Rome, Italy

Fabrizio Morciano and Marco Rizzuto
"Selex Sistemi Integrati" a Finmeccanica Company, Italy

Francesca Matarese
"SESM" a Finmeccanica Company, Italy

9. References

[1] Baldoni, R., Lodi, G., Montanari, L., Mariotta, G. & Rizzuto, M. [2012]. Online black-box failure prediction for mission critical distributed systems, *to appear in proceedings of SAFECOMP 2012*, Springer Berlin / Heidelberg.

[2] CARDAMOM [website]. Cardamom middleware website. `http://www.cardamom.eu/`.

[3] EC482 [2008]. Commission regulation (ec) no 482/2008, *Official Journal of the European Union* pp. 5–9.

[4] Elnozahy, E. N., Alvisi, L., Wang, Y.-M. & Johnson, D. B. [2002]. A survey of rollback-recovery protocols in message-passing systems, *ACM Comput. Surv.* 34(3): 375–408.

[5] ESARR6. [2010]. *ESARR 6. EUROCONTROL Safety Regulatory Requirement. Software in ATM Systems.*, 2.0 edn, European Organisation for the Safety of Air Navigation.

[6] Esper [2012]. Esper project web page. `http://esper.codehaus.org/`.

[7] Fischer, M. J., Lynch, N. A. & Paterson, M. [1985]. Impossibility of distributed consensus with one faulty process, *J. ACM* 32(2): 374–382.

[8] Gertler, J. [1988]. Survey of model-based failure detection and isolation in complex plants, *Control Systems Magazine, IEEE* 8(6): 3 –11.

[9] IBM [2011]. System S Web Site. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html`.

[10] Kapadia, R., Stanley, G. & Walker, M. [2007]. Real world model-based fault management., *18th International Workshop on the Principles of Diagnosis Nashville TN*.

[11] Laprie, J.-C., Avizienis, A., Randell, B. & Landwehr, C. E. [2004]. Basic concepts and taxonomy of dependable and secure computing, *IEEE Trans. Dependable Sec. Comput.* 1(1): 11–33.

[12] Liebert [2005]. *Regulatory Compliance and Critical System Protection*, Liebert Corporation.

[13] Luckham, D. C. [2001]. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[14] OMG [CCM]. Corba component model (ccm), omg specification, formal/2011-11-03, part 3 - components. `http://www.omg.org/spec/CORBA/3.2/Components/PDF`.

[15] OMG [FT-CORBA]. Fault tolerant corba (ft), omg specification, formal/2010-05-07 , v1.0. `http://www.omg.org/spec/FT/1.0/PDF`.

[16] OMG [LTLOAD]. Lightweight load balancing service (ltload), omg specification, formal/2010-02-04, v1.0. `http://www.omg.org/spec/LtLOAD/1.0/PDF`.

[17] OMG [LWFT]. Lightweight fault tolerance for distributed rt systems (lwft), ptc/2011-06-05, beta 2. `http://www.omg.org/spec/LWFT/1.0/Beta2/PDF`.

[18] OMG [website]. Object management group webpage. `http://www.omg.org/`.

[19] Rabiner, L. & Juang, B. [1986]. An introduction to hidden markov models, *ASSP Magazine, IEEE* 3(1): 4 – 16.

[20] Salfner, F. [2008]. *Event-based Failure Prediction: An Extended Hidden Markov Model Approach*, PhD thesis, Department of Computer Science, Humboldt-Universität zu Berlin, Germany.

[21] Salfner, F., Lenk, M. & Malek, M. [2010]. A survey of online failure prediction methods, *ACM Computing Surveys (CSUR)* 42(3): 1–42.

[22] Simpson, W. & Sheppard, J. [1994]. *System test and diagnosis*, Kluwer Academic. URL: *http://books.google.it/books?id=Pjr93wWJMiQC*

[23] Stanley, G. M. & Vaidhyanathan, R. [1998]. A generic fault propagation modeling approach to on-line diagnosis and event correlation., *3rd IFAC Workshop on On-line Fault Detection and Supervision in the Chemical Process Industries*,.

[24] Trivedi, K. S. & Vaidyanathan, K. [2008]. Software aging and rejuvenation, *Wiley Encyclopedia of Computer Science and Engineering*.

[25] Williams, A. W., Pertet, S. M. & Narasimhan, P. [2007]. Tiresias: Black-box failure prediction in distributed systems, *Proc. of IEEE IPDPS 2007*, Los Alamitos, CA, USA.