Sapienza University of Rome

Ph.D. program in Computer Engineering

XXV Cycle - 2012

Online Failure Prediction in Air Traffic Control Systems

Luca Montanari

Sapienza University of Rome

Ph.D. program in Computer Engineering

XXV Cycle - 2012

Luca Montanari

Online Failure Prediction in Air Traffic Control Systems

Thesis Committee	**Reviewers**
Prof. Roberto Baldoni (Advisor)	Prof. Domenico Cotroneo
Prof. Luigi Laura	Prof. Priya Narasimhan

Author's address:
Luca Montanari
Dipartimento di Ingegneria Informatica Automatica e Gestionale
"A. Ruberti"
Sapienza Università di Roma
Via Ariosto 25, 00185 Roma, Italy
e-mail: montanari@dis.uniroma1.it
www: http://www.dis.uniroma1.it/~montanari

Contents

1 Introduction **1**
1.1 Software Dependability . . . 2
1.1.1 Basic concepts of Dependability . . . 4
1.1.2 Faults, Errors, Failures . . . 5
1.2 Fault Management: reactive and proactive approaches . . . 8
1.3 Online Failure Prediction . . . 10
1.4 Taking Actions and reaction methods . . . 12
1.5 Faults and Failures in Mission Critical Systems . . . 12
1.6 Motivation and Contribution . . . 16
1.6.1 Motivation . . . 16
1.6.2 Novelty. . . . 18
1.6.3 Contributions . . . 19
1.7 Outline of the Thesis . . . 20

2 Related Work **21**
2.1 Online Failure Prediction . . . 21
2.2 Black Box Monitoring and Non-intrusiveness . . . 29
2.3 Classification . . . 31
2.3.1 Bayesian Network Classifier . . . 31
2.3.2 Decision Trees . . . 32
2.3.3 Neural Networks . . . 32
2.3.4 Hidden Markov Models . . . 32
2.4 Events, Event Processing, Event Based Programming . . . 34
2.4.1 Complex Event Processing . . . 35
2.4.2 Complex Event Processing applications . . . 38
2.4.3 Complex Event Processing engines . . . 39

3 Model and Basic Techniques **43**
3.1 Failure and Prediction Model . . . 44
3.2 Data Pre-Processing . . . 45
3.3 Classification . . . 48

3.4 Aggregator 54

4 Architecture **57**

4.1 Assumptions 58

4.2 CASPER Architecture 58

4.2.1 Pre-Processing module. 59

4.2.2 Symptoms detection module. 60

4.2.3 Hidden Markov Model as a state recognizer 62

4.2.4 Hosts Activity Detection Module 64

4.2.5 Failure Prediction module 68

4.3 Training and Tuning of CASPER 68

4.3.1 Training of the model 68

4.3.2 Tuning of CASPER parameters 69

5 Evaluations and results on a real ATC system **71**

5.1 Evaluation Metrics 72

5.2 Monitored System 73

5.2.1 Principle of FT CORBA 73

5.2.2 Testbed 77

5.2.3 Faults Injection 78

5.2.4 Training Data 78

5.3 Performance Metrics 79

5.4 Results 81

6 Conclusion **93**

Glossary **96**

Bibliography **97**

List of Figures

1.1 The propagation chain or “chain of threats” introduced in [72] 6
1.2 ATC Very High Level Architecture 14

2.1 A taxonomy for online failure prediction approaches found in [97]. 22
2.2 The stages in the operation of Tiresias, [109]. 25
2.3 The [59] feature space. 26
2.4 The [104] predictive anomaly management for distributed hosting infrastructures. 27
2.5 The [104] anomaly prediction using triple-state decision tree classifier. 27
2.6 The pre-component diagnosis scheme of [40]. 34
2.7 Conceptual difference between DBMS on the left and DSMS on the right. 37
2.8 Esper architecture 40

3.1 Online Failure Prediction chain. 43
3.2 Fault, Symptoms, Failure and Prediction 44
3.3 A graphical representation of a generic Hidden Markov Model. The $\omega_1, \dots, \omega_N$ vertices are the hidden states of the Markov process. The observable symbols are $\sigma_1, \dots, \sigma_M$. Some elements of A and B matrices are represented as edges. 50
3.4 Aggregator. N input lines receive a vector of numeric values. The output will be a symbol belonging to an ordered set of integers. The cardinality of the set has to be given. 55
3.5 Example of square grid in $\mathbb{R}^2$ with $D = 4$. 55
3.6 Example of aggregator behavior with $N = 2$ and $D = 4$. 56

4.1 The modules of the CASPER failure prediction architecture . . 59
4.2 Performance Metrics Computation component 60
4.3 Hidden Markov Models graph used in the system state inference component 63
4.4 Unsafe and safe zones of a two-performance metrics square grid. 64

4.5 An example of graph representing a 9-nodes ATC system. The black nodes are inactive nodes while the white node is a source node (with no inner edges). The dotted edges are inactive network links, the number on the edges is the port number. . . . 66

4.6 An example of decreased repeatability: two near values, a circle and a triangle, in the (a) case will corresponds to the same symbol 11 while in the (b) case will correspond to two different symbols, 37 and 46. 70

5.1 FT CORBA framework 74

5.2 System decomposition in application, process, component, group and host. 75

5.3 CCM and CORBA based middleware services. 76

5.4 3-tier architecture. 77

5.5 Symbols emitted by the performance metrics computation component in case of a recorded trace that exhibits stress of the memory. 82

5.6 Symptoms detection module: F-Measure varying the number of symbols and clock period in case of a recorded trace subject to memory stress 83

5.7 Performance of the symptoms detection module varying the number of possible symbols in case of a recorded trace subject to memory stress. CASPER clock period 800 ms 84

5.8 Performance of the symptoms detection module varying the number of possible symbols in case of a recorded trace subject to I/O stress. CASPER clock period 800 ms 85

5.9 False positives varying the window size feeding. CASPER is fed with a recorded trace behaving in steady-state. Its clock period is 800 ms 86

5.10 Failure prediction in case of memory stress starting at second 105. Window size 16s, clock period 800ms, time-to-prediction 23s, time-to-failure 207s 87

5.11 Failure prediction in case of I/O stress starting at second 408. Window size 16s, clock period 800ms, time-to-prediction 21s, time-to-failure 376s. 88

5.12 Performance of CASPER in terms of time-to-failure. 88

5.13 400 seconds of a steady-state run of the ATC system in operation. 89

5.14 Hosts ranking behavior. At second 191 a memory stress starts on host number 102. Starting from second 191 will have always the higher mark. 90

5.15 Hosts ranking behavior. At second 230 a memory stress starts on host number 105. After some seconds the vote of the host 105 grows rapidly, despite the host remained alive. 91

Chapter 1

Introduction

A plenty of fields in today's life and organizations depend on the correct functioning of the computer systems. The dependability, defined as *the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers*, is nowadays crucial also in terms of safety, according to the kind of applications that are treated. Despite the cruciality of the dependability and the safety of the actual computer systems, the complexity level reached by those precludes the possibility to develop a system that is completely correct. Complexity means that the systems are built starting by several commercial Off The Shelf (OTS) components (i.e. hardware and software), each of those needs to be correctly integrated to create the system. Millions of lines of code and millions of transistors imply millions of fault sources. The occurrence of failures cannot be completely avoided but its likelihood should be always minimized.

Over the last years, computers are enabling crucial human everyday activities, such as public economy, large scale critical infrastructures management (e.g., for water and power supply plants and energy production), and air traffic control. Given the growing dependence on computers in these life- and cost-critical applications, dependability becomes an essential demand: a failure, indeed, can be catastrophic in terms of business or, even worse, human losses. Business critical systems, e.g., for e-commerce or e-government applications, have to maximize system availability and service reliability in order to maximize customers' satisfaction and survive today's competition. Since these systems are widely distributed to users with different and unknown usage patterns, developing dependability strategies becomes quite complicated. This holds also for everyday critical systems, e.g., hospitals or banks, which rely on databases management systems whose failures would deeply affect individuals or groups. Conversely, mission and safety critical systems have a narrow set of target users, and usage patterns are predictable in many cases. For these systems, the dependability level is regulated by standard specifications, carried out by international bodies, to which commercial products have to be compliant (e.g., the DO-178B standard for avionics software certification [1]). The availability of those systems, for these reasons, is typically at least *five-nines*[2], meaning that on average the system must not be down for more then 5.26 minutes per year. In the worst case a failure occurs once a year: a human being should analyze, diagnose and repair the complex system having less then six minutes. This is impossible even if we consider a failure every three years! This yields to the need for the system to react to a failure in a nearly automated way. Several times, react means restart the system. Even if the reaction is completely automated, complex systems like Air Traffic Control systems, rarely restart themselves within 5 minutes. A broader approach to the classical *detection-diagnosis-reaction* is required in order to try the possible to completely avoid the failure. This requires some short-term anticipation of upcoming failure, based on a continuous evaluation of the state of the system, followed in certain conditions, by a proactive mechanism that tries to avoid the upcoming failure or to alert the human operators timely to minimize their effect. The proactive failure management approach has to live parallel to the classic approach.

This thesis focuses on *online failure prediction* for *complex distributed systems* in order to achieve a proactive fault management. In literature can be found several typology of failure prediction, here we consider short-term failure prediction. The need for a short-term failure prediction for computer systems has been demonstrated by Liang et al. [74].

1.1 Software Dependability

Dependability is a complex attribute whose definition changed several times in the last decade. Indeed, the increasing complexity of systems has caused dependability to become a major concern, encompassing several aspects, from safety to security. Focus is on software dependability into which current research efforts are striven to face the problem of transient manifestation of software faults. In this section particular attention is devoted to the classification of software faults and the ways they can manifest. Software and hardware behave differently in term of their dependability. Hardware reliability is dominated by random physical factors affecting the components on which there is

[1]http://www.lynuxworks.com/solutions/milaero/do-178b.php3

[2]"five nines" equals 0.99999 (or 99.999%) availability

engineering knowledge enough to prevent failures. This is demonstrated by the several reliability theories that have been developed so far for the realization of highly dependable hardware systems, as well as for hardware reliability evaluation and assessment. Note that the discipline of failure prediction is born in the hardware failure management. Software unreliability is only due to design faults, i.e., to the consequences of human failures. The modern systems show a replacement in the fields of safety and mission critical applications of the older hardware-based technologies by software modules. Examples are air traffic control, railroad interlocking, nuclear plants management, that gains also more efficiency and capability by integrating software system (modern traffic control systems and railroad control systems handle much more traffic than in the past). Additionally, software is being used to solve novel problems for which there is a lack of evidence from the past history as well as to perform novel actions otherwise impossible (think about stability systems for the modern aircrafts control, speed controls for the trains, cars bricking systems or cruise control). The advantages of the introduction of software is straightforward in reducing human efforts but in the other hand the introduction of a novel kind of faults i.e. software faults, led to a greater probability of mistakes which can even results on catastrophes: several examples are given from the space industry which has always been pioneer in the introduction of software systems e.g.:

- 1962 - Mariner I space probe: A bug in the flight control software causes the Mariner I rocket to calculate the incorrect trajectory. The rocket was destroyed by Mission Control over the Atlantic.

- 1996 - Ariane 5 Flight 501: the rocket self-destructing 37 seconds after launch because of a malfunction in the control software. A data conversion from 64-bit floating point value to 16-bit signed integer value to be stored in a variable representing horizontal bias caused a processor trap because the floating point value was too large to be represented by a 16-bit signed integer.

- 1999 - Mars Polar Lander: the probe hit the Mars ground destroying itself. A software error that incorrectly identified vibrations, caused by the deployment of the stowed legs, as surface touchdown. The resulting action by the spacecraft was the shutdown of the descent engines, while still likely 40 meters above the surface. Although it was known that leg deployment could create the false indication, the software's design instructions did not account for that eventuality.

- 1999 - Mars Climate Orbiter: the spacecraft encountered Mars at an improperly low altitude, causing it to incorrectly enter the upper atmosphere and disintegrate. The flight system software on the Mars Climate Orbiter was written to take thrust instructions using the metric unit newtons (N), while the software on the ground that generated those instructions used the Imperial measure pound-force (lbf).

Other examples can be made in civil applications:

- 1982 - Soviet gas pipeline: a bug in the Soviet gas pipeline software controls caused the largest non-nuclear, man-made explosion in history

- 1985-1987- Therac-25 medical accelerator: a therapeutic device that utilizes radiation has a bug which can lead to a race condition. If that condition occurs then the patient receives multiple times the recommend dosage of radiation. The failure directly caused the deaths of five patients and harmed many more.

- 1990 - AT&T Network Outage. A bug in a new release of code causes the switches of AT&T to crash. Over 60 thousand New Yorkers were left without phone service for nine hours.

- 2000 - National Cancer Institute, Panama City: the software of a therapeutic device that utilizes radiation for treatment delivers twice the recommended dosage. Eight patients die and 20 more will undoubtedly be permanently disabled.

- 2004 Mercedes-Benz - "Sensotronic" braking system - Mercedes-Benz has to recall 680,000 cars due to a failure of its Sensotronic breaking system.

1.1.1 Basic concepts of Dependability

Even if the effort on the definition of the basic concepts and terminology for computer systems dependability dates back to 1980s, the milestone paper in the field of dependable systems is [71] which was published in 1985. Here dependability was defined as *the quality of the delivered service such that reliance can justifiably be placed on this service,* but the notion has evolved over the years. Recent efforts from the same community define the dependability as *the ability to avoid service failures that are more frequent and more severe than is acceptable* [72]. This last definition has been introduced since it does not stress the need for justification of reliance. The dependability is a composed quality attribute, that encompasses the following sub-attributes:

- Availability: readiness for correct service;

- Reliability: continuity of correct service;
- Safety: absence of catastrophic consequences on the user(s) and the environment;
- Confidentiality: absence of improper system alterations;
- Maintainability: ability to undergo modifications and repairs.

1.1.2 Faults, Errors, Failures

The causes that lead a system to deliver an incorrect service, i.e., a service deviating from its function, are manifold and can manifest at any phase of its life-cycle. Hardware faults and design errors are just an example of the possible sources of failure. These causes, along with the manifestation of incorrect service, are recognized in the literature as dependability threats, and are commonly categorized as *failures*, *errors*, and *faults* [72].

A **failure** is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. A service failure is a transition from correct service to incorrect service, i.e., to not implementing the system function. The period of delivery of incorrect service is a service outage. The transition from incorrect service to correct service is a service recovery or repair. The deviation from correct service may assume different forms that are called *service failure modes* and are ranked according to *failure severities*.

An **error** can be regarded as the part of a system's total state that may lead to a failure. In other words, a failure occurs when the error causes the delivered service to deviate from the correct service. The adjudged or hypothesized cause of an error is called a **fault**. Faults can be either internal or external of a system, and they can be classified in several ways (e.g., basing on their nature, or the way they manifest in errors).
Failures, errors, and faults are related each other in the form of a *chain of threats* [72], as sketched in figure 1.1. A fault is *active* when it produces an error; otherwise, it is *dormant*. An active fault is either i) an internal fault that was previously dormant and that has been activated, or ii) an external fault. A failure occurs when an error is *propagated* to the service interface and causes the service delivered by the system to deviate from correct service. An error which does not lead the system to failure is said to be a *latent* error. A failure of a system component *causes* an internal fault of the system that contains such a component, or causes an external fault for the other system(s) that receive service from the given system.
The dependability attributes can be formalized mathematically, and basic

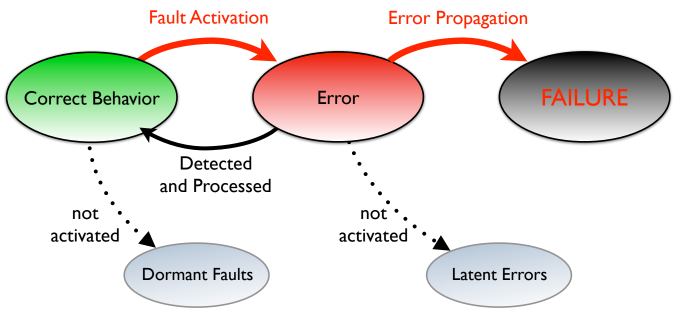


Figure 1.1: The propagation chain or "chain of threats" introduced in [72]

measures have been introduced in charge of quantifying them.

The reliability, $R(t)$, was the only dependability measure of interest to early designers of dependable computer systems. It is the the conditional probability of delivering a correct service in the interval $[0, t]$, given that the service was correct at the reference time 0 [95]:

$$R(0, t) = P(no\ failures\ in\ [0, t] | correct\ service\ in\ 0)$$

Let us call $F(t)$ the unreliability function, i.e., the cumulative distribution function of the failure time. The reliability function can thus be written as:

$$R(t){=}1{-}F(t)$$

Since reliability is a function of the mission duration T, mean time to failure (MTTF) is often used as a single numeric indicator of system reliability [84]. In particular, the time to failure (TTF) of a system is defined as the interval of time between a system recovery and the consecutive failure.

As for availability, they say a system to be available at a the time t if it is able to provide a correct service at that instant of time. The availability can thus be thought as the expected value $E(A(t))$ of the following $A(t)$ function:

$$A(t) = \begin{cases} 1 & if\ proper\ service\ at\ t \\ 0 & otherwise \end{cases}$$

In other terms, the availability is the fraction of time that the system is operational. The measuring of the availability became important with the advent of time-sharing systems. These systems brought with it an issue for the continuity of computer service and thus minimizing the "down time" became a prime concern. Availability is a function not only of how rarely a system fails but also of how soon it can be repaired upon failure. Clearly, a synthetic availability indicator can be computed as:

$$Av = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}$$

where MTBF = MTTF + MTTR is the mean time between failures. The time between failures (TBF) is the time interval between two consecutive failures. Obviously, this measure makes sense only for the so-called repairable systems. A complete dissertation about dependability fundamentals can be found in [72], along with a description of dependability measures. Again according to [72], the *means* of the dependability can be grouped into four major categories:

- **Fault prevention**, to prevent the occurrence or introduction of faults. It is enforced during the design phase of a system, both for software (e.g., information hiding, modularization, use of strongly-typed programming languages) and hardware (e.g., design rules).

- **Fault tolerance**, to avoid service failures in the presence of faults. It takes place during the operational life of the system. A widely used method of achieving fault tolerance is redundancy, either temporal or spatial. Temporal redundancy attempts to reestablish proper operation by bringing the system in a error-free state and by repeating the operation which caused the failure, while spatial redundancy exploits the computation performed by multiple system's replicas. The former is adequate for transient faults, whereas the latter can be effective only under the assumption that the replicas are not affected by the same permanent faults. This can be achieved through design diversity [21].
  Both temporal and spatial redundancy requires error detection and recovery techniques to be in place: upon error detection (i.e., the ability to identify that an error occurred in the system), a recovery action is performed.

- **Fault removal**, to reduce the number and severity of faults. The removal activity is usually performed during the verification and validation phases of the system development, by means of testing and/or fault injection [23]. However, fault removal can also be done during the operational phase, in terms of corrective and perfective maintenance.

- **Fault forecasting**[3], to estimate the present number, the future incidence, and the likely consequences of faults. Fault forecasting is conducted by evaluating the system behavior with respect to fault occurrence or activation. Evaluation can be (i) qualitative, aiming at iden-

[3]Not to be confused with failure forecasting!

tifying, classifying, and ranking the failure modes that would lead to system failures and (ii), quantitative evaluation, aiming at evaluating the extent to which some of the attributes are satisfied in terms of probabilities; those attributes are then viewed as measures. The quantitative evaluation can be performed at different phases of the system's life cycle: the design phase, the prototype phase and the operational phase [92]. In the design phase, the dependability can be evaluated via modeling and simulation, including simulated fault injection. During the operational phase, field failure data analysis (FFDA) can be performed, aiming at measuring the dependability attributes of a system according to the failures that naturally manifest during system operation. When using FFDA, several issues arise related to data collection, filtering and analysis, which are extensively addressed in [92].

The *Online Failure Prediction* that this thesis investigates is a discipline belonging to the **Fault tolerance** which aim is to provide a *short-term* assessments that allow to decide if the current situation is going to bring the system to a failure.

1.2 Fault Management: reactive and proactive approaches

Fault 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 fault management should monitor the system looking for errors and faults that could end up in a failure and overcome such issues when they arise. Fault management techniques are *reactive* in nature: the faults have to be detected before of taking actions. In order to do this, a reactive fault manager needs some capabilities identified in [67], and particularly interesting:

- *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.

- *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 [101].

- *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 [100], 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*: 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.

In order to overcome the limitations that reactive approaches have, the fault management start to investigate the so called *proactive approaches*: techniques that trigger suitable corrective actions to *prevent* a fault *before* the system experiences a failure. Check-pointing [51] and Software rejuvenation [64, 106] are examples: specific forms of proactive fault management that can be performed at suitable times, such as when there is no load on the system, and thus typically results in less downtime and cost than the reactive approaches [36]. The proactive scheme anticipates the formation of erroneous system states before it actually materializes into a fault and to a failure by consequence. Since proactive fault management incurs some overhead, 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, i.e. a *prediction mechanism* to predict failure occurrences and thus trigger the system state recovery. Proactive fault management can be greatly enhanced by the ability to predict the failure enough in advance that one can take actions to avoid or mitigate its effects. The *online failure prediction* aim is exactly this.

1.3 Online Failure Prediction

The term “failure prediction” is widely used, e.g., for *reliability prediction* where the goal is to assess future reliability of a system from its design or specification [97]. Examples of *reliability prediction* can be found in [78], [32], [43], [31]. Reliability prediction is concerned with long-term prediction, based on input data evinced by architectural properties or the number of bugs that have been fixed. In [65] can be found a survey on reliability and availability prediction methods. Although architectural properties such as software and hardware interdependencies play a role in some online failure prediction methods, online failure prediction aim is to provide a short-term assessments that allow to decide if the current situation is going to bring the system to a failure or not. The goals of online failure prediction, as intended in this work, is to identify such failure-prone situations. The evaluation is based on the results of a runtime monitoring process. Note that the output of online failure prediction can either be (i) an alert to allow some (human or automatic) recovery mechanism to timely trigger or (ii) continuous measures in order to allow the prediction mechanism to tune its judgment on the current situation. In [97] can be found the following definition, characterizing Online Failure Prediction:

Online Failure Prediction is the task to identify ***during runtime*** *whether a failure will occur in the* ***near future*** *based on an assessment of the monitored current system state.*

Origin of Online Failure Prediction

Proactive fault management and online failure prediction belongs to the Fault Tolerance research discipline. Fault tolerance was born together the computing itself, where the methods developed have had to deal with the problem of the unreliability of the hardware components. Various variants of fault tolerance mechanisms, employing static and dynamic fault tolerance techniques, have been developed (see [98] for an overview). As the complexity of the computing systems increased over the years, fault tolerance became more dynamic. Examples are the Self-Testing And Repairing computer, otherwise known as STAR computer [20]. As the software became more and more complex, the software became also the principal point of failure, hence, the software failures management and software fault management have gained more interest in literature. The N-version programming [19] or recovery block [91] are examples of this. In the early 1990's the software failures are predominant with respect to hardware failures [103]. Until the 1990s, however, fault tolerance were intended as a passive and static technique, the problems had to be detected before any kind of reaction and the reaction had to be well defined during the system design. In the 1995 a new approach was introduced and rapidly became well-known in the community. This approach was introduced by Huang et. al. [64] and is called Software Rejuvenation (see [106] for a survey), a proactive fault management method. It consists in restarting part of the system even if there are still no faults. The objective is to deal with another well-known problem: software-aging [85]. Software aging refers to progressive performance degradation or a sudden hang/crash of a software system due to exhaustion of operating system resources, fragmentation and accumulation of errors. Once that the proactive techniques were introduced, quickly several ways to cope the software failures and fault proactively were presented: Horn introduced the concept of autonomic computing [62]; Munfie et al. introduced the trustworthy computing [42]; Coleman et al. [39] introduced the adaptive enterprise; Brown et al. [33] the recovery-oriented computing and so on. Particular importance deserves the well-known technique of checkpointing (see [51] for a survey and [94] for a scalable solution combined with failure prediction techniques). All these techniques belong to the proactive fault management.

1.4 Taking Actions and reaction methods

There are several kind of actions that can be undertaken if a failure is predicted reasonably in advance. The objectives, ordered by importance, are:

- avoid the failure;
- control the shut down;
- minimize the downtime;

The reaction methods can be (i)automated, some recovery mechanism is triggered after the prediction and the diagnosis of the fault (e.g. a checkpoint is triggered), or (ii)performed by humans, an operator takes actions as soon as is alerted. Note that a downtime avoidance improves Mean Time To Failure while a downtime minimization reduces Mean Time To Repair. However, In case of frequent false positive and false negative predictions, proactive fault management can also reduce availability since the system has to restore itself without any necessity. The recovery can assume the form of rollback (the system is brought back to a saved state that existed prior the occurrence of the error; it needs to periodically save the system state, via checkpointing techniques [51, 107]), rollforward (the system is brought to a new, error-free state), and compensation (a deep knowledge of the erroneous state is available to enable error to be masked).

The actions and the reaction methods, the recovery mechanisms and the smooth shut-down techniques are out of the scope of this dissertation.

1.5 Faults and Failures in Mission Critical Systems

Distributed *mission critical systems* such Air Traffic Control, 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 system [54], 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 Air Traffic Control (ATC) mission critical systems are large and complex systems supervising the aircraft trajectories from departure to destination. Having an effective failure management in such kind of critical systems is a must for safety and security reasons. Due to the complexity and the strong requirements, current ATC systems adopt both the reactive and proactive fault management approaches. In this work an online failure prediction framework has been designed, developed and the tested over a real ATC system. A real implementation of the reactive paradigm is also presented. This implementation is already deployed in the real ATC System while the online failure prediction framework developed is going to be deployed in the same system.

The next section specializes the faults and failures in the ATC domain with their relationship with safety regulation 482/2008.

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. A very high level architecture of an ATCs is shown in Fig. 1.2. The Figure highlights the needs

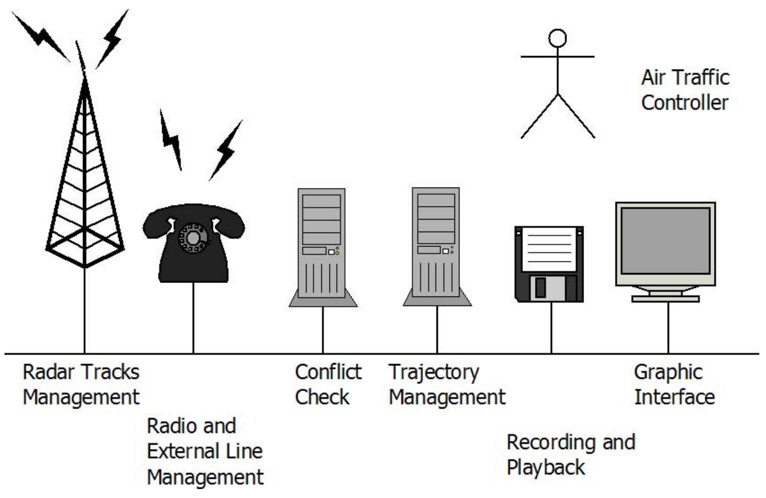


Figure 1.2: ATC Very High Level Architecture

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 do 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 are 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 [52]. 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 brings 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 [75]. 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 [49]. 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.

1.6 Motivation and Contribution

1.6.1 Motivation

The work is born thanks to a collaboration between *University of Rome "Sapienza"* and *Selex - Sistemi Integrati,* a *Finmeccanica* company. The company presented to our research team a complex problem regarding their complex Air Traffic and Naval control systems. We classified this problem in the research fields of pattern recognition, Complex Event Processing and online failure prediction. The final objective was identified in developing an online failure prediction framework that triggers alerts as soon as dangerous conditions are recognized in the observed system. The alerts should trigger proactive actions on the observed system in order to improve the dependability. The actions to be undertaken/triggered were beyond the scope of the work.

Among the requirements obtained and identified, one deserves particular attention: the observed system has to be leaved "as is". This led to a completely *non-intrusive* approach: one of the key points of the work. The observation has to be performed only using network data, without install any kind of software on the hosts involved in the computations of the ATC system. This has been one of the most challenging task but at the end it led to a framework that is "ready-to-use" in almost all middleware-based distributed systems. Non-intrusiveness also differentiates our work from other approaches, that require applications or middleware modifications. The works in literature usually use nodes functional data (e.g. cpu consumption, free memory, number of context switches), errors and failures logs and so on. Collecting this kind of information requires to log-in to the hosts of the system and install software on it [4]. Many mission-critical distributed systems do not allow this approach, hence a non-intrusive monitoring is mandatory and result very attractive for industrial purposes.

[4] In this thesis this kind of approach will be called *intrusive.*

Non-Instrusive and Black-Box monitoring

The motivations to adopt a *non-intrusive* and *black-box* approach is twofold. Firstly, applications change and evolve over time: grounding failure prediction on the semantic of the applications' communications would require a deep knowledge of the specific system design, a proven field experience, and a non-negligible effort to keep the supervision service aligned to the controlled system. Secondly, interactions between the service and system to be monitored might lead to unexpected behaviors, hardly manageable as fully unknown and unpredictable. Mission critical distributed systems are system with strong requirements of availability and stringent Quality of Service (QoS) requirements, providing a variety of services. The monitoring services shall have a minimum impact on the supervised system and possibly no interaction with the operational applications. Such complex distributed environments are composed of several software components of several different vendors. Administrators and operators very often don't know every component and don't have access to source code of these. Another very common situation is the restricted access to the hosts due to policy of the companies and due to technical issues: operators and administrators access the system using consoles and graphical user interfaces (GUIs) and do not have privileges to log-in directly to the hosts of the network or, nonetheless, access to files or file system proc[5]. In these circumstances think about installing software in an already-deployed system becomes difficult or impossible when not forbidden. In such an environment, with these strict requirements, is born the architecture presented in this dissertation, which must be as non-intrusive as possible. The idea comes after a feasibility study conducted in cooperation with the administrators of a real ATC system to be monitored, and consists in using only sniffed network traffic to evince the state of the system. The goal is to plug-in a "ready-to-use observer" that acts at run time and can also be used in several contexts without particular modifications.

How to observe in a non-intrusive way

In physics, the term *observer effect* refers to changes that the act of observation will make on the phenomenon being observed. In information technology, the *observer effect* is the potential impact of the act of observing a process output while the process is running. Even if theoretically impossible, we tried to find a way to observe complex mission critical distributed systems without changing them. The idea reached is to consider the network hosts as black-

[5]The great part of the monitoring mechanism are based on the analysis of the file system proc.

boxes and monitor the network traffic produced by the boxes. In a system composed by nodes exchanging information, a software failure inside one of the nodes, or a critical condition like a bottleneck, changes the way in which the node behaves: this misbehavior can be discovered observing the network traffic related to the node. Think about a simple web-browser accessing to a remote streaming: the message rate exchanged between the computer and the server varies depending on the load of the server, if the server is busy the message rate during the streaming will decrease, sometimes halting the stream. This is only an example, there are several information that can be captured from the network. The task is also aided by the fact that these mission critical systems are middleware-based (more precisely Fault-Tolerant middleware-based) and the middleware uses, almost always, standard network protocols. We quickly discovered that by analyzing the network packed sniffed from the network, and dissecting the headers belonging to standard protocols (e.g. TCP/IP, UDP, GIOP, SOAP), without any software installation on the monitored hosts, a plenty of information can be captured. Very predictably situations can be recognized in advance. Among the motivating works found, [47] presents an empirical evidence that the unpredictability inherent in such systems arises from merely 1% of the remote invocations: by filtering the 1% of the raw observations (i.e. the outliers) of metrics like end-to-end latency and throughput, the performance can be bounded, easy to understand and control. Such kind of metrics can be easily computed at run-time using network data. The observation remains completely application-agnostic i.e. (i)considering all the network hosts as black-box and (ii) without recognizing causal paths (i.e. the difference between this work and the Aguilera et al. [10] black-box definition);

During these years a number of prototypes have been designed, developed and tested over several environments and over a real Air Traffic Control testing environment. Important results have been obtained for both industrial and academic purposes. Several lessons have been learnt, a nowadays complex architecture for this target has been designed which has strength and weaknesses points. A version of this architecture is going to be deployed in a real ATC supervision system.

1.6.2 Novelty.

The approach proposed in this dissertation differs from the other work presented for some important aspects. Firstly, this approach is both black-box and non-intrusive. In contrast, other works do not seem to satisfy these characteristics together for failure prediction purposes. Secondly, the techniques we employ to the monitoring are different from those used by them. We combine in a novel fashion both Complex Event Processing for network performance metrics computation and Hidden Markov Models to infer the system state. Note that typically Hidden Markov Models are widely used in failure detection, to build a components' state diagnosis [40]. In the other hand the architecture introduced in this work models the entire system state (not the individual components state) as a hidden state, thus inferred using an Hidden Markov Model classifier. Consider the whole set of components as a unique coherent system has several pros and cons, but is an approach that is rarely present in literature. In the field of online failure prediction, the Markov Models, Hidden Markov Models and Hidden Semi-Markov Models have been used in error detection based techniques but not in the symptoms-based techniques as we have done. At the best of our knowledge, the non-intrusive monitoring is also a novelty for Air Traffic Control systems which usually rely on consolidated fault-tolerant middleware and on complex monitoring subsystems designed ad-hoc. The non-intrusiveness and black-box approach followed implies a certain degree of interoperability of the monitoring system and this is a clear novelty in the field of mission critical fault management.

1.6.3 Contributions

The main contribution of this work is the development of a novel approach to symptoms-based failure prediction that is able to satisfy together the non-intrusiveness, black-box and online properties required by mission critical contexts. Other contributions can be grouped in:

- the design, implementation and experimental evaluation of a novel online, non-intrusive and black-box failure prediction architecture named CASPER, that can be used for monitoring mission critical distributed systems.
- the idea of using only information concerning the network to build a real-time representation of the system state. This allows (i)to consider the hosts as black-box, (ii)to not incur in privacy problems and (iii) to leave the system untouched (no new software is installed on the hosts).
- the combination of Complex Event Processing techniques and Hidden Markov Models in order to represent a complex system state via real-time performance metrics and to recognize the dangerous system states i.e. states the usually lead to services failure.
- the introduction of an interoperable approach to monitor Air Traffic Control systems. The interoperability comes from the fact that each component is considered as a black box and analyzed at network level,

without trying to infer the relation among the hosts and ignoring any application level logic.

- the introduction of a tunable aggregation mechanism to simplify the representation of the system state.

1.7 Outline of the Thesis

The proposed thesis is organized as follows. Chapter 2 reports the related work with particular emphasis on the on-line failure prediction state of the art with some interesting systems, it defines Black Box Monitoring and Non-intrusiveness and it presents the classification problem, with particular interest given to the Hidden Markov Model applications. After, it reports some basic concepts and definition on Event Computing, Event Processing and Complex Event Processing in particular, with some implementations.

Chapter 3 introduces the failure and prediction model followed in this dissertation, defining important concepts: time-to-failure, time-to-prediction and limit time. It presents Esper [3] as Complex Event Processing engine with some useful details about its paradigms and presents the Hidden Markov Model used with important algorithms defined on it. It introduces also the Aggregator, a component that allows to simplify the representation of the system state.

Chapter 4 presents the architecture of the framework devised, namely CASPER. A description of each component is provided and details on the training and tuning of the CASPER parameters are given. It also presents the performance metrics identified to monitor middleware based systems.

Chapter 5 presents the air traffic control system over which the experimental evaluation has been conducted: the principle of FT-CORBA[81] and the specialization of FT-CORBA for safety critical system, i.e. the CARDAMOM [35] middleware. The chapter also presents the queries to compute at runtime the performance metrics used in the campaign of experiments in the native event processing language. After that, the results obtained during the experimental evaluation is presented: the characterization of the training and tuning of the CASPER parameters and the results of the failure prediction task when the system is suffering stress conditions.

Chapter 6 concludes the thesis describing the future work and some weakness and open problems discovered during the work.

Chapter 2

Related Work

The Related Work is organized as follows: The first section investigates the *online failure prediction* state of the art and it presents some interesting systems. The second section defines Black Box Monitoring and Non-intrusiveness. The third section is dedicated to the *classification* with particular interest to the *Hidden Markov Model* applications. The last section is about Event Computing, Event Processing and Complex Event Processing in particular with some implementations.

2.1 Online Failure Prediction

A significant amount of work can be found in the online failure prediction area; Salfer [97] shows a taxonomy that structures the manifold of approaches (see Figure 2.1).

This taxonomy differentiates two different approaches: one that rely on the current system state which is *data-driven* and another approach that is based on quantitative and qualitative mathematical description of the system model that is called *analytical approach to online failure prediction.* Salfner [97] collected all different methods developed in the data-driven modeling field, broad disciplines like econo-physics, artificial intelligence and game theory are the main producers of these approaches to prediction. Although this taxonomy has been introduced in 2008, it is still very valid. Three main branches can be found: Failure Observation, Symptoms Monitoring and Error Detection.

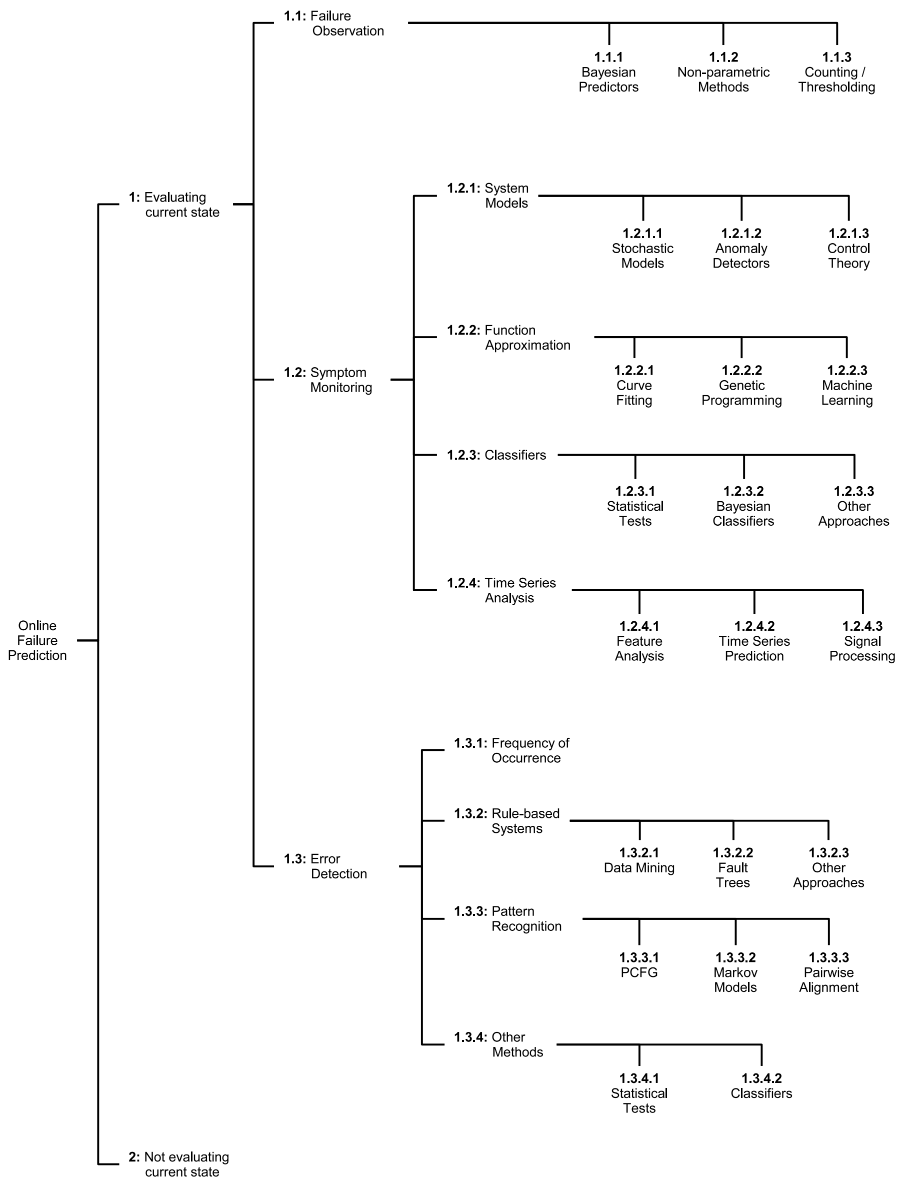

Online Failure Prediction
1: Evaluating current state
1.1: Failure Observation
1.1.1 Bayesian Predictors
1.1.2 Non-parametric Methods
1.1.3 Counting / Thresholding
1.2: Symptom Monitoring
1.2.1: System Models
1.2.1.1 Stochastic Models
1.2.1.2 Anomaly Detectors
1.2.1.3 Control Theory
1.2.2: Function Approximation
1.2.2.1 Curve Fitting
1.2.2.2 Genetic Programming
1.2.2.3 Machine Learning
1.2.3: Classifiers
1.2.3.1 Statistical Tests
1.2.3.2 Bayesian Classifiers
1.2.3.3 Other Approaches
1.2.4: Time Series Analysis
1.2.4.1 Feature Analysis
1.2.4.2 Time Series Prediction
1.2.4.3 Signal Processing
1.3: Error Detection
1.3.1: Frequency of Occurrence
1.3.2: Rule-based Systems
1.3.2.1 Data Mining
1.3.2.2 Fault Trees
1.3.2.3 Other Approaches
1.3.3: Pattern Recognition
1.3.3.1 PCFG
1.3.3.2 Markov Models
1.3.3.3 Pairwise Alignment
1.3.4: Other Methods
1.3.4.1 Statistical Tests
1.3.4.2 Classifiers
2: Not evaluating current state

Figure 2.1: A taxonomy for online failure prediction approaches found in [97].

Failure Observation, Symptoms Monitoring and Error Detection

Literature on online failure prediction is broadly divided by these three approaches. The idea of the first one, **Failure Observation**, is to extract information from past failure occurrences. The information extracted can be qualitative or quantitative. Quantitative information would for example be given by a statistical analysis as the probability distribution of failures and can be used to directly predict the probability of a particular failure occurrence or given as prior knowledge for Bayesian predictors. On the other hand qualitative information can lead to custom heuristics to particular failures prediction. Among failure observation techniques, [7] apply a Bayesian predictive approach [11] to the Jelinski-Moranda software reliability model [66] in order to yield an improved estimate of the next time to failure probability distribution. A Bayesian predictive approach is exploited also to the some Italian underground trains system to check the actual reliability of the door opening systems [8]. They carry out a Bayesian inference via a Monte Carlo simulation, obtaining prediction intervals for the expected number of failures during periods of desired length. [56] analyzes the correlation in time and space of failure events and implements a long-term failure prediction framework named hPrefects for such a purpose. hPrefects, based on the correlations among failures, forecasts the time-between-failure of future instances. The approach of [56] is based on long-term prediction (from week to years). In this work the system is analyzed in realtime predicting failures from minutes to seconds before. We also not observe failures since we perform symptoms monitoring (see later). Among the Failure Observation approaches, Zhang & Fu, [115] propose a framework for autonomic failure management with hierarchical failure prediction functionality for coalition clusters and compute grids. It analyzes node, cluster and system wide failure behaviors and forecasts the prospective failure occurrences based on quantified failure dynamics. Failure correlations are inspected by the predictor: in a compute node, an event sensor scrutinizes local event logs, extracts failure records and creates formatted failure reports for the failure predictor. The failure predictor calculates the estimations of future failure occurrences based on information collected by the event sensor.

Manifestation of faults is not necessary a crisp situation, it can influence the system gradually in time and space. The **Symptoms Monitoring** aim it to recognize this type of *service degradation* and it is in contrast with the *Error Monitoring*. Some examples for such types of faults can be done. Memory leaks, for instance, imply that some part of the system consume more and more memory. As long as there is still memory available, there will be neither failure or errors. In case of distributed systems, if one host suffers for a bottleneck, the services involved will endure some degradation but the failure can be also remote. Symptoms monitoring has to recognize these conditions in order to timely trigger actions to avoid the failure.

Among the symptoms monitoring based works, Berenji et al. [30] build a system model in a hierarchical two step approach: First, they build component simulation models that try to mimic the input/output behavior of system components. These models are used to train component diagnostic models by combining input data with component outputs obtained from the component simulation models. The target output values of the diagnostic models are binary where a value of one corresponds to faulty component behavior and zero to non-faulty behavior. The same approach is then applied on the next hierarchical level to obtain a system-wide diagnostic models. The authors use a clustering method to obtain a radial basis function rule base. In the context of symptoms monitoring mechanisms, there exist also research works that use black-box approaches, i.e., no knowledge of the applications of the system is required. Narasimhan et. all [109] introduce Tiresias, a black-box failure prediction system that considers symptoms generated by faults in distributed systems. There is a terminology difference between [109] and this work: Black Box Monitoring in this work case means that we don't access not even nodes functional data while [109] is Black Box in terms that applications running in the hosting infrastructure are opaque to the infrastructure provider. In order to identify symptoms, Tiresias uses a set of performance metrics that are system-level metrics, (e.g., file system proc metrics) and network traffic metrics. In addition, Tiresias uses specific heuristics in order to predict the failure. Figure 2.2 depict clearly the collected time-series data on the system's actors. The framework presented in this dissertation uses only one of them i.e. "network traffic" by means of the libpcap library. [58] uses the online failure prediction to achieve an efficient proactive failure management in the field of distributed stream processing systems. System-level and application-level metrics (e.g., available memory, free CPU time, virtual memory page-in/page-out rates, tuple processing time, buffer queue length) are periodically collected and classified using decision trees. It perform just-in-time failure prevention (short-term) of the faulty components only. It has a similarity with our work that is the tuning mechanism: a desired tradeoff between failure penalty reduction and prevention cost based on a user-defined reward function. In our work the user can define the desired accuracy of the failure prediction but loosing prediction time.

Gu et. all [59] present a stream-based mining algorithm for online anomaly prediction. The field of application is the large-scale cluster systems, vulnerable to various software and hardware problems. The objective of [59] is to diagnose bottleneck anomalies. In order to do this, the work achieves classification on future data by employing Markov models to capture the changing patterns of different measurement metrics that are used as features by the Bayesian classifiers. They predict the values of each metric for the next time

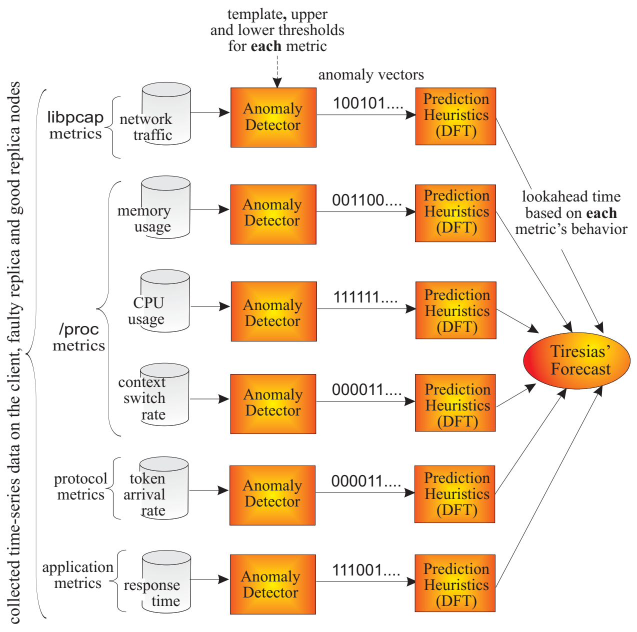


Figure 2.2: The stages in the operation of Tiresias, [109].

units. The Bayesian classifier is then used to predict the probability of different anomaly symptoms by combining the metric values. The combination of the metrics values is interesting and is similar to the approach used in this thesis. A *feature space* is created (see Figure 2.3), in order to create a bottleneck classifier that incorporates multiple metrics, called features, that can collectively capture the distinctive symptoms of different bottlenecks. Figure 2.3 shows a two-dimensional feature space where anomaly symptoms of three different causes (bottlenecks caused by insufficient CPU, insufficient memory, or both insufficient CPU and insufficient memory) form three clusters.

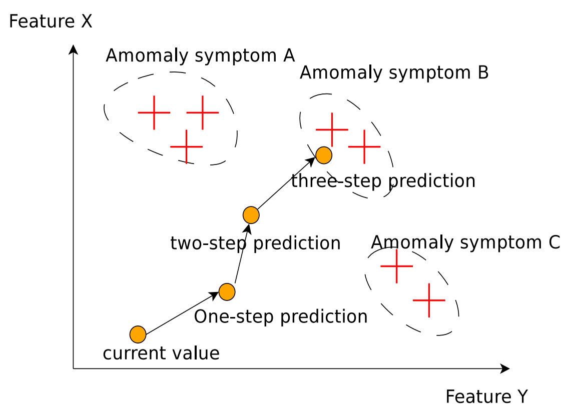


Figure 2.3: The [59] feature space.

In Figure 2.3, the position of a point in the feature space is predicted in one, two and three time steps. As it shows in the figure, one possible outcome is that, in three time steps, the measurement point falls into the cluster representing anomaly symptom B. If that outcome has a large probability, the system should raise alert that an anomaly with symptom B will occur after three time steps. To predict feature values, the system needs to model the statistical changing patterns of different feature values. Combining both anomaly symptom classification and feature value prediction, the system can perform online anomaly prediction, that is, performing anomaly symptom classification on future data.

[104] presents ALERT an anomaly prediction system that considers the hosts of the monitored system as black-boxes, given the applications running in the hosting infrastructure are opaque to infrastructure provider. ALERT embodies a fully decentralized monitoring and learning architecture, in order to cope with large-scale hosting infrastructures (see Figure 2.4). Specifically, it uses sensors deployed in all the hosts of the controlled infrastructure to continuously monitor a set of metrics concerning CPU consumption, memory usage, input/output data rate. Note that the use of sensor deployed in the hosts makes this work *intrusive* and not feasible for mission critical distributed systems. A set of pre-defined *anomaly predicates* based on user's service level objective (SLO), allows to detect the symptoms of anomalies. To classify component states ALERT uses triple-state decision tree classifier that can produce rules with direct, intuitive interpretation by non-experts (see Figure 2.5). Figure 2.5 illustrates a simple case of classification using two metrics. For state classification, the decision tree essentially applies a sequence of threshold

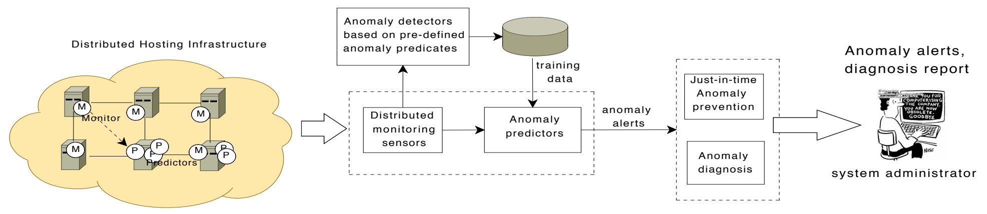

Distributed Hosting Infrastructure
Monitor
Predictors
M
P
Anomaly detectors based on pre-defined anomaly predicates
training data
Distributed monitoring sensors
Anomaly predictors
anomaly alerts
Just-in-time Anomaly prevention
Anomaly diagnosis
Anomaly alerts, diagnosis report
system administrator

Figure 2.4: The [104] predictive anomaly management for distributed hosting infrastructures.

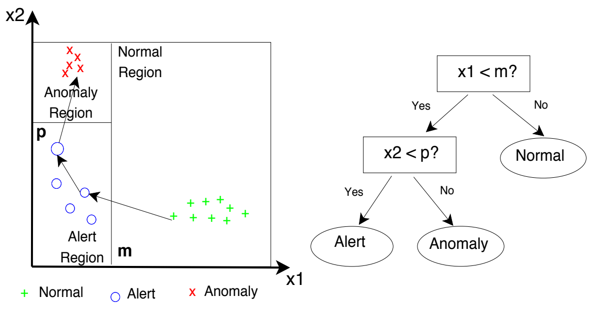


Figure 2.5: The [104] anomaly prediction using triple-state decision tree classifier.

tests on the metrics. Similar to the approach used in this dissertation, the combination of performance metrics form a space. According to the values of the metrics, the regions represent normal or anomaly conditions.

Guan et al. [88] apply proactive failure management techniques in Cloud Computing Systems. In particular present a (symptoms monitoring) failure prediction mechanism exploiting both unsupervised and semi-supervised learning techniques for building dependable cloud computing systems. Unsupervised failure detection method uses an ensemble of Bayesian models, it characterizes normal execution states of the system and detects anomalous behaviors. After the anomalies are verified by system administrators, labeled data are available, then apply supervised learning based on decision tree classifiers to predict future failure occurrences in the cloud. The data are collected by health monitoring tools when cloud servers perform normally. At the best of my knowledge this is the only failure prediction work that uses unsupervised learning.

Rood et. all in [94] cope the problem of failure prediction and scalable checkpointing in large-scale grid computing. They state that the primary weapons against the problem of node unavailability and cluster failure i.e. checkpointing, migration, replication, and effective scheduling, do not scale well enough to be effective for the largest, most important grid and cluster applications. They propose to address this issue at a variety of levels, including: (i) low level mechanisms that will predict individual processor failures by observing and reacting to low-level indicators in their chip state; (ii) scalable cluster-level checkpointing solutions that do not require centralized storage for replicated checkpoints; (iii) grid-level efforts to differentiate between different node unavailability states, to characterize the behavior of nodes, to predict their near-future unavailability, and to make better grid scheduling decisions based on this information, and on characteristics and capabilities of applications.

The **Error Detection** approach has several differences with the symptoms monitoring. The main difference between them is that symptoms are the observation of system state over time; a symptom is a behavior that deviates from the "average" behavior. While error is something actually goes wrong. The fault at this stage did not develop in service failure yet but would. A good question based on detecting an error is: how probable this error will develop in a failure? for which time window since occurrence this probability is high? The Rule-Based systems Rules are logic expression that represents knowledge representation. A very common rule pattern is the IF-THEN pattern where condition can be any logical expression that represent the premise and expression any logical expression to represent the conclusion.

Malek et. all in [97] present an error monitoring-based failure prediction technique that uses Hidden Semi-Markov Model (HSMM) in order to recognize error patterns that can lead to failures. The idea of HSMM approach is to predict failures by recognizing specific patterns of error events that report an imminent failure. The approach in [97] is event-driven as no time intervals are defined: the errors are events that can be triggered anytime.

[60] describes two non-intrusive data driven modeling approaches to error monitoring (it uses error logs and logs of the operating system): the first approach relies on a Discrete Time Markov Model and the second approach is based on function approximation. In general [60] performs short-term predictions in order to anticipate failures. This work uses existing error logs served as source for event and logs of the operating system level monitoring tools. The difference between this thesis approach and these works is twofold: firstly we propose a symptoms monitoring system in contrast to error monitoring for predicting software failures. In addition, our approach is not event-based as proposed in [97]; rather, our solution can be defined period-based [73] as we

use Hidden Markov Models (discrete time) to recognize, in the short term, patterns of specific performance metrics that show the evidence of symptoms of failures.

[34] comes up with an optimal checkpointing algorithm that reduces overhead caused due to checkpointing, using failure prediction algorithm. The proposed system uses Job replication to ensure completion of work and dynamic load balancing is used to avoid overload in any resources and to achieve maximum resource utilization and maximize throughput. The system is applied in a Grid computing infrastructure prone to failures.

An important problem affect both traditional reactive approaches and proactive approach that is based on continuous application performance evaluation. The problem is the fact that a workload change can be recognized as an anomaly, since the performance of the observed system can strongly vary when applications change. It is important to distinguish between performance anomaly and workload change. A performance anomaly is indicative of abnormal situation that needs to be investigated and resolved. On the contrary, a workload change (i.e., variations in transaction mix and load) is typical for several applications. Therefore, it is highly desirable to avoid false alarms raised by the algorithm due to workload changes, though information on observed workload changes can be made available to the service provider. The issue is highlighted and analyzed by Cherkasova et. all in [38] that also propose a new integrated framework of measurement and system modeling techniques for anomaly detection and analysis of essential performance changes in application behavior.

This dissertation copes the online failure prediction problem using a symptoms monitoring approach. In order to recognize the system state a Hidden Markov Model classifier is implemented. The performance metrics representing the runtime system state are computed using Complex Event Processing techniques, considering as input data only network data in a completely non-intrusive fashion. According to the taxonomy introduced by [97] (see Figure 2.1) should be positioned in the "1.2.3" set: we use HMM as a classifier in order to discover if the inferred state is failure-prone or not.

2.2 Black Box Monitoring and Non-intrusiveness

There are several definitions that can be found in literature about non-intrusiveness and black-box monitoring. In order to clarify these concepts, we report our definitions.

*A **non-intrusive monitoring system** is a monitoring mechanism that does not require to install software nor to log in on the monitored system's* *hosts.*

At the best of our knowledge, no pre-existent online failure prediction mechanism respect this definition.

Our definition of black-box monitoring follows:

*A monitoring can be considered as **Black-Box** if considers the monitored system components as black box (i.e. no knowledge of the application's internals and of the application logic of the system is analyzed) and if does not try to recognize causality paths among the boxes.*

The non-intrusiveness and black-box approach yields to the following advantages:

- privacy and policy issues are avoided;
- reusability of the framework requires only few implementation modifies;
- a Plug and Play framework can be obtained.

While the drawbacks are:

- Network performance can be varied by several factors that can trick the monitoring;
- A (application-level) workload change can heavily affect the hosts (network-level) behavior ([38] analyzes this problem);
- A network-level observer cannot distinguish from the several possible kind of inactivities, while a monitoring software installed on the host can easily do.

Some works in literature consider the problem of non-intrusiveness and of black-box monitoring. Among these, is particularly important Aguilera et. all [10] that considers the problem of discovering performance bottlenecks in large scale distributed systems consisting of black-box software components: systems usually without source code available and not accessible due to vendors restrictions. The system introduced in [10] solves the problem by using message-level traces related to the activity of the monitored system in a non-intrusive fashion (passively and without any knowledge of node internals or semantics of messages). In [10] the tools developed are aimed at the debugging phase, not providing real-time results, hence their willing to use offline analysis. It is an interesting work because has requirements near to mission critical system analyzed in this dissertation. It also uses network traffic in order to infer the dominant causal paths through the distributed system violating our Black Box Monitoring definition.

2.3 Classification

Classification is one of the most frequently encountered decision making tasks of human activity. A classification problem occurs when an object needs to be assigned into a predefined group or class based on a number of observed attributes related to that object [113]. Many problems in business, science, industry, and medicine can be treated as classification problems. Examples include bankruptcy prediction, credit scoring, medical diagnosis, quality control, handwritten character recognition, and speech recognition. The classification is a particularly known problem in machine learning and statistics. It is a basic task in data analysis and pattern recognition that requires the construction of a *classifier*: a function that assigns a *class* label to instances described by a set of attributes [55]. In general each algorithm that implements classification is known as *classifier.* The term classifier sometimes also refers to the mathematical function, implemented by a classification algorithm, that maps input data to a category. In the machine learning terminology, classification is considered an instance of supervised learning, i.e. learning where a training set of correctly-identified observations is available. Several classifiers can be found in literature, from the *naive Bayes classifier* to more complex approaches like neural networks. Particular interesting are a set of classifier that automatically build their ruleset, called *Learning Classifier Systems* (see [99] for a survey). More details on classifiers are out the scope of this dissertation, but some details on some of the classification methods are provided below.

2.3.1 Bayesian Network Classifier

Bayesian networks are directed acyclic graphs whose nodes represent random variables in the Bayesian sense: they may be observable quantities, latent variables, unknown parameters or hypotheses. Edges represent conditional dependencies; nodes which are not connected represent variables which are conditionally independent of each other. Each node is associated with a probability function that takes as input a particular set of values for the node's parent variables and gives the probability of the variable represented by the node. Similar ideas may be applied to undirected, and possibly cyclic, graphs; such are called *Markov networks.* Several efficient algorithms are present that perform inference and learning in Bayesian Networks, see [55] and [77] for a comprehensive survey. A generalization of Bayesian networks that can represent and solve decision problems under uncertainty are called *influence diagrams* while Bayesian networks that model sequences of variables (e.g. speech signals or protein sequences) are called dynamic Bayesian networks. The hidden Markov model can be considered as a simple dynamic Bayesian network.

2.3.2 Decision Trees

The *Decision Trees* is a decision support tool that uses a tree graph and is famous as a understandable way to graphically represent an algorithm. Are used to identify the strategy to more likely reach a goal. The scope is to transform a complex decision-making process into a collection of simpler decisions. A decision tree is a graph that consists of 3 types of nodes: *Decision nodes* commonly represented by squares; *Chance nodes* represented by circles; *End nodes* represented by triangles. Please refer to[96] for a survey on *Decision Tree Classifiers*.

2.3.3 Neural Networks

Artificial Neural Networks usually called *neural network* is a mathematical model or computational model that is inspired by the structure and the functional aspects of biological neural networks. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation. In most cases an Artificial Neural Networks is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs or to find patterns in data. There are three major learning paradigms on neural networks, each corresponding to a particular abstract learning task. These are supervised learning, unsupervised learning and reinforcement learning. Tasks that fall within the paradigm of supervised learning are pattern recognition (i.e. the classification) and regression (also known as function approximation). The supervised learning paradigm is also applicable to sequential data (e.g., for speech and gesture recognition). See [113] for a survey on neural networks classifiers.

2.3.4 Hidden Markov Models

A hidden markov models is a statistical model (i.e. a formalization of relationships between variables in the form of mathematical equations) in which the system modeled is assumed to be a Markov process with hidden states. Can be consider as the simplest dynamic Bayesian network. While in a regular Markov model the state is known to the observer, in a hidden Markov Model the state is not visible but the output of the states, according to a given probability, is visible. The sequences of output can hence gives information about the sequences of hidden states. This model allow to recognize the state of the modeled system starting from observation of it output, building in this way a classification of the output symbols. This is the approach we followed. Mathematical details of this model are given in section 3.3.

Hidden Markov Models applications

The first work in voice recognition [89], has launched the very large number of today's applications like clustering in time-series data [13], intrusion detection [68] and many others in temporal pattern recognition such as, handwriting, gesture recognition, part-of-speech tagging, speech recognition, musical score following, partial discharges. A good number of works can be found in the field of computational biology e.g.[50] and [70]. In [110] is presented a human action recognition method based on a Hidden Markov Models. To apply HMMs, one set of time-sequential images is transformed into an image feature vector sequence, and the sequence is converted into a symbol sequence by vector quantization. The predicting capabilities of the HMMs are also used by Dockstader et. all in [44]: the problem of the detection and prediction of motion tracking failures with application in human motion and gait analysis is presented. The approach defines a failure as an event and uses the output probability of a trained HMM to detect and a logarithmically transformed probability to predict such events. The vector observations for the model are derived from the time-varying noise covariance matrices of a Kalman filter that tracks the parameters of a structural model of the human body. A medical application of HMM is presented in [69] where the hidden Markov models are used to model ECG signals, [18] presents an original HMM approach for online beat segmentation and classification of electrocardiograms. The HMM framework has been visited because of its ability of beat detection, segmentation and classification, highly suitable to the electrocardiogram problem. The same author published [15] which originally combines HMM and wavelets providing new insights on the ECG segmentation problem. A P2P-TV traffic has been modeled using HMM in [57], by proposing a simple traffic model that can be representative of P2P- TV applications. HMM is often used for deviation detection and state diagnosis; [40] uses the Hidden Markov Model formalism considering three aspects involved in component's state diagnosis: the monitored component, the deviation detection mechanism and the state diagnosis mechanism. In [40] the use of hidden Markov models is similar to this work: the monitored components' state cannot be detected, hence an estimation of these have to be done. Using a monitoring mechanism and the forward algorithm of the HMM (see later) the components state can be inferred. A deviation detection mechanism is also included in the model. This approach can also be used for failure prediction as if a component state is detected as faulty, a failure prone situation is present. Figure 2.6 depicts

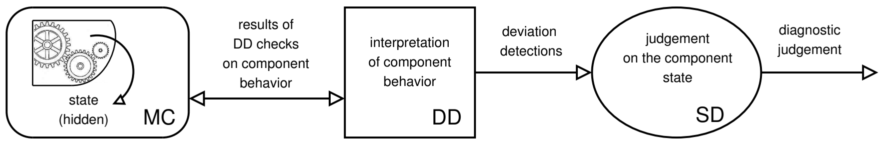


Figure 2.6: The pre-component diagnosis scheme of [40].

the per-component diagnosis scheme. The approach used in this dissertation is the same, with some important differences: in this thesis we model the state of the entire system as a hidden state that is unknown, not considering the components independently for the prediction purposes. Our approach embodies the deviation detection of the state of the system and use it to predict future failures, using the same forward algorithm.

2.4 Events, Event Processing, Event Based Programming

There are several definition of "event": Mani Chandy [37] defines an event as "a significant change in a state" where a significant state change is a change in the "real" state that deviates from the "expected" state and a "deviation" is significant enough when it causes a change in plans. Opher Etzion (IBM, [53]) defines an event as:

- An occurrence within a particular system or domain; that is something that has happened or is contemplated as having happened in that domain.
- In a computing system, event means a programming entity that represents such an occurrence.

Opher Etzion also defines event processing as follows:

Event Processing *is computing that performs operations on events. Common Event Processing operations include reading, creating, transforming and deleting events.*

and event based programming as follows:

Designing and coding applications that make use of events, directly or indirectly.

Note that according to this definition, it is possible to write event-based programs without using Event Processing. In [53] can be found the following “reasons for using event-driven computing”:

- real-Time operational behavior: a common reason for using event-driven computing systems is to be able to change the behavior of the system dynamically in order react to react to incoming events. Matching auction buyers and sellers is an example of this type, the result of the match then determines the subsequent flow of the system. Another example is automatic re-routing of luggage when a passenger’s itinerary changes.

- Observation: another reason to use event-driven computing systems is to look for exceptional behavior and generate alerts when such behavior occurs. In such cases the reaction, if any, is left to the consumer. The job of the event processing application is just to produce the alerts. Examples of observation are regulation compliance systems, as well as the patient monitoring system described above.

- Information dissemination:a third reason for using event-driven computing systems is to deliver the right information to the right consumer in the right granularity at the right time in other words personalized information delivery. Examples of this type are personalized alerts from banking systems, and the emergency system sending alerts to first responders.

- Active diagnostics: here the goal of the event processing application is to diagnose a problem, based on observed symptoms. The mechanical failure case is such an example; a help-desk system is another example.

- Predictive processing: Here the goal is to identify events before they have happened, so that they can be eliminated or at least have their affects mitigated. The fraud detection example is of this kind.

2.4.1 Complex Event Processing

Complex Event Processing (CEP) is important to this work as well as Hidden Markov Models. CEP is an active research field recently used for the development of monitoring and sense-and-respond applications [76]. It addresses two crucial prerequisites in building highly scalable and dynamic systems: mediation of the information in the form of events and detection of relationships among them, i.e. temporal relationships that can be identified by defining correlation rules (often called Event Patterns). In CEP, three fundamental concepts are defined: event streams, correlation rules, and event engine. An event is a representation of a set of conditions in a given time instant. Events belong to streams: events of the same type are in the same event stream [53]. Correlation rules are commonly SQL-like queries (but also other types of queries are possible [45]) used by the event engine in order to correlate events: i.e.,in order to discover temporal and spatial relationships between events of possibly different streams (to filter only the relevant ones, or to perform calculation among them). The engine takes in input the streams and correlates events belonging to both different streams and the same stream, according to the correlation rules. The concept of correlation can be very articulated since events can be correlated spatially and temporarily; it can be joined and filtered; new streams can be created at runtime; also new events can be created composing other events, creating in this way what is called *complex events.*

Common Complex Event Processing application fields are:

- Business: process management and automation (process monitoring, BAM, reporting exceptions, operational intelligence)
- Finance : algorithmic trading, fraud detection, risk management.
- Distributed systems: Network and application monitoring (intrusion detection, SLA monitoring;
- Sensor network applications: RFID reading, scheduling and control of fabrication lines, air traffic.

The reason for which it is being adopted in all the previous fields is that it takes much of the complexity out of developing applications that detect patterns among events, filter events, aggregate time or length windows of events, join event streams, trigger based on absence of events etc. A primary difference with system relying on classical SQL databases is that CEP engines do not query a repository for events matching some conditions, but instead trigger customized actions as the flow of events come in matching event conditions - hence drastically reducing the latency. Technically there are other important differences between the way in which the data are processed in the classical DBMS and the Data Stream Management Systems. Figure 2.7 and Table 2.1 emphasize these differences.

There are several commercial products that implement the CEP paradigm (e.g. Esper [3], JBoss Drools Fusion [87], IBM System-S [4], Sec. 2.4.3 describes some of these). Among those currently available, in the experimental evaluation and in the prototype developed in this work, has been chosen the well-known open source event engine Esper [3]. The use of Esper is motivated by both its low cost of ownership compared to other similar systems (e.g. IBM System S [4]), its offered usability, and the ability of dynamically adapting the complex event processing logic by adding at run time new queries.

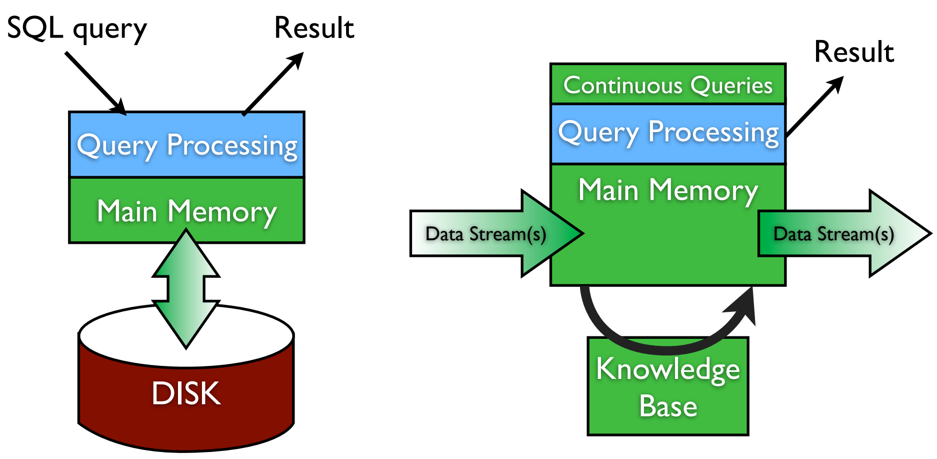


Figure 2.7: Conceptual difference between DBMS on the left and DSMS on the right.

Table 2.1: Main differences between DBMS and DSMS.

DBMS	DSMS
Persistent relations	Transient streams
One-time queries	Continuos queries
Random access	Sequential access
“Unbounded” disk store	Bounded main memory
Only current state matters	Historical data counts
No real-time services	Real time requirements
Relatively low update rate	Multi arrival and variable rate
Data at any granularity	Data at fine granularity
Assume precise data	Data imprecise

2.4.2 Complex Event Processing applications

Detecting event patterns, sometime referred to as situations, and reacting to them are in the core of Complex Event Processing (CEP) and Stream Processing (SP), both of which play an important role in the IT technologies. Business intelligence, air traffic control, collaborative security, complex system software management are examples of such applications. A very interesting work is [53] where the major concepts of event-driven architectures are introduced and how to use, design, and build event processing systems and applications is shown. A definition of CEP follows:

Complex Event Processing *is an event processing that combines data from multiple sources to infer events or patterns that suggest more complicated circumstances*

Several uses of the CEP paradigm can be found in literature, applications can be found in the fields. In particular, IBM System S[4] has been used by market makers in processing high-volume market data and obtaining low latency results as reported in [14, 114]. In the financial sector, as overviewed in [9], several patterns can be recognized. IBM System S as other CEP/SP systems, e.g. [12, 86], are based on event detection across distributed event sources. Detection of particular cyber attacks and cyber security in general is a recent application of the CEP concept as overviewed in [17]. A recent application of CEP is in CoMiFin European Project. The aim of CoMiFin is to develop a middleware for the protection of networked financial players from cyber attacks by analyzing network data sources coming from several financial institutions. In [16] and [93] the idea of the project and the *Semantic Room* abstraction is presented. Treating events introduces the problem of data dissemination supporting complex event processing: basic events, potentially occurred at different sites, are correlated in order to detect complex event patterns formed by basic events that could have temporal and spatial relationships among them. A fundamental functionality is the data dissemination that brings events from event producers to event consumers where complex event patterns are detected. In [25] the characteristics that a Data Dissemination service should have in order to support complex event pattern detection are discussed. The issue of using massive complex event processing among heterogeneous organizations for detecting network anomalies and failures has been suggested and evaluated in [63], who proposes using network-wide analysis of routing information to diagnose (i.e., detect and identify) network disruptions.

2.4.3 Complex Event Processing engines

Almost all the research centers of the major companies have developed their CEP implementation. From IBM to Google inc., several CEP systems are available in the market. In general, these systems are quite complex and expensive systems and sometimes hard to test. There are no several differences between the various implementations. However, all of them share the same concept: the data flow through the system; the system consists of a principal engine which detects temporal and spatial relationships between the data. These relationships are identified by means of long running queries issued by the engine on the data. The queries are generally specified in SQL-like languages or by means of ad-hoc mechanisms. Each implementation has their programming language in order to define the set of the rules, in [53] can be found a brief description of the languages used by *ALERI, APAMA, Esper, ETAILS, RULECORE, STREAMBASE.* All of this are Complex Event Processing engine, or Stream Event Processing engine. The CEP systems are also equipped with a number of adapters: the adapters analyze miscellaneous information, coming from heterogeneous human and computer-generated sources (e.g., audio and video data, sensors data, text, e-mail, IM, web logs). The adapter takes the raw data generated by those sources and transforms the data in a format that can be read by the main correlation engine. To this end, standard adapters that come with the main CEP technologies can be used for such purposes. However, for complex specialized systems, ad-hoc adapters can be created that take in input the specific data that are to be analyzed.

Esper

As a case study we consider Esper (Event Stream and Complex Event Processing). Esper is an Event Stream Processing (ESP) and Complex Event Processing engine written in Java. It provides an event stream processing module, including event representations and event pattern matching, supporting the extraction of meaningful information from the large amount of data within a stream. In ESPER, while the events pass through memory, the query engine continuously sieves for the relevant events that may satisfy one of the correlation queries; it performs calculations and it trigger a listener if a query is satisfied. This technique is also known as *continuous query.* Figure 2.8 depicts the archi tecture of esper. The queries are defined using an SQL-like query language named Event Processing Language (EPL). EPL can thus support all SQL's conventional constructs such as Group By, Having, Order By, Sum, etc. However it adds further constructs (e.g., the `pattern`) that allows it to perform complex correlations among events. The main difference and addition to SQL is the ability of EPL to detect *Patterns* among events (i.e.,

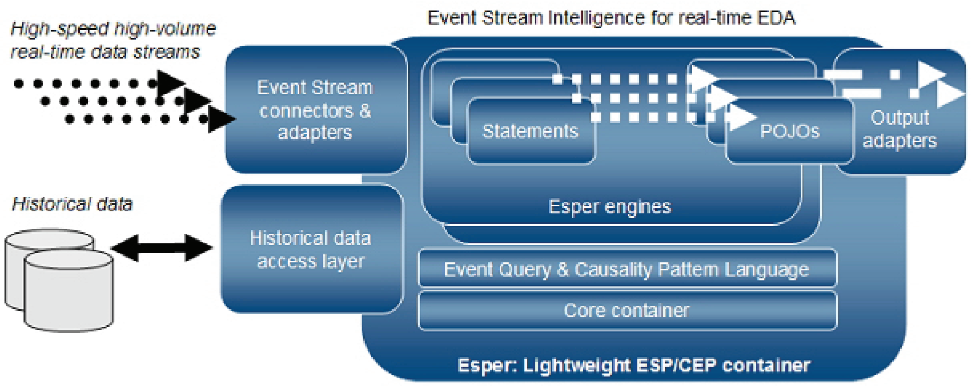

Event Stream Intelligence for real-time EDA
High-speed high-volume real-time data streams
Event Stream connectors & adapters
Statements
POJOs
Output adapters
Historical data
Historical data access layer
Esper engines
Event Query & Causality Pattern Language
Core container
Esper: Lightweight ESP/CEP container

Figure 2.8: Esper architecture

the ability to perform event correlations). A pattern may appear anywhere in the from clause of an EPL statement including joins and subqueries. The performance of this CEP engine are very interesting since can easily process more than 100.000 events per seconds even over an off-the-shelf laptop. This means that can easily cope with the real time requirements that we have. In section 3.2 provide some details about EPL and how has been used to represent the system state.

System S

The IBM System S [4] is the CEP engine developed by the IBM and is interesting in several ways. Define a new abstraction of *Processing Element*(PE), a computing node that can be a single node, a process or more in general an entity that performs their functionality. The event streams or the streams (having a adapting phase supports several kinds of data) is constricted to pass through a given set of these PE and depending on this set the wanted result can be achieved. Has event several other properties, some of them are:

- Supports structured and unstructured datastream processing;
- Can be scaled from one to thousands of computer nodes;
- At runtime can execute a large number of jobs

While all the CEP implementations found uses a SQL-like or, more generally, a logical language in order to specify the rules, System S uses a language that allows to create the rules only specifying which are the interconnection among the several available processing elements. So, what we have is a graph for each query (just to remain in a SQL-like fashion), and not a statement. Obviously allows the users to create their processing element. Since is thought for all kind of user, a processing element can be created in several ways: using a programming language, if the users has computer science skills or even using quasi-natural language, starting from ontologies [45]. System S is one of the more complex and powerful system of this family but is not affordable and this limit their use for research purpose.

Drools Fusion

JBoss Drools Fusion [87] is a module responsible for enabling event processing capabilities in unified behavioral modeling platform. Support asynchronous multi-thread streams: Events may arrive at any time and from multiple sources (or streams). They can also be stored in cloud-like structures. Drools Fusion supports both work with streams and clouds of events. In case of streams it supports asynchronous, multi-thread feeding of events. Support for temporal reasoning: events usually have strong temporal relationships and constraints. Drools Fusion adds a complete set of temporal operators to allow modelling and reasoning over temporal relationships between events. Support events garbage collection: events grow old, quickly or slow, but they do grow old. Drools Fusion is able to identify the events that are no longer needed and dispose them as a way of freeing resources and scaling well on growing volumes. Support reasoning over absence of events: the same way in that it is necessary to model rules and processes that react to the presence of events, it is necessary to model rules and processes that react to the absence of events. Example: ”If the temperature goes over the threshold and no contention measure is taken within 10 seconds, then sound the alarm”. Drools Fusion leverages on the capabilities of the Drools Expert engine, allowing it complete and flexible reasoning over the absence of events, including the transparent delaying of rules in case of events that require a waiting period before firing the absence. Support to Sliding Windows: a especially common scenario on Event Processing applications is the requirement of doing calculations on moving windows of interest, be it temporal or length-based windows. Drools Fusion has complete support for Sliding Windows, providing out of the box aggregation functions as well as leveraging the plugable function framework to allow for the use of users defined custom functions.

RTM Analyzer One of the problems that you find approaching the develop of a CEP engine is that the costumer wants to use it, taking advantage of all the functionalities offered, without ”adapt” the events that already travel in his existing system. One of the CEP systems that cope this problem is *RTM Anayzer* [1]. This offer their services having two important adapting modules, one for the input data and one for the output. It use SQL as rules programming language and can do

- filtering
- correlation
- aggregation

and continuous detection of

- trends
- patterns

Even if RTM comes with an explicit adapter module, the others systems support and implement this concept of adapter in a more implicit fashion.

TIBCO Business events

The TIBCO develops *Business Events* [2]. The key features are about the modeling, the rule engine and the way in witch it captures the events. Business Event requires an UML-based state model to describe how applications and services interact as part of activities and processes. Regarding the Rule Engine, in this system is based on the industry-standard RETE protocol for familiarity and stability, the BusinessEvents rules engine has been recompiled and tuned to support simultaneous application of thousands of rules to millions of events. The events are routed across TIBCO's integration and messaging infrastructure as well as other vendors' implementations of JMS and other integration platforms including IBM's MQSeries messaging software. Actually the modeling portion is not required from the others systems and this leads TIBCO to be one of the less scalable and usable CEP engine in an existing system.

PROGRESS—APAMA

Relevant differences between this approach and the others is for sure the several IDE available, as well as an integration framework (works like the adapters in RTM Analyzer). But is interesting even the tools dedicated to a sophisticated backtesting and analysis for the Progress Apama environment, called APAMA Data Player. They enable Apama users to investigate the likely behavior of Apama Scenarios prior to deployment, as well as analyze the actual performance of those Scenarios already in production.

Chapter 3

Model and Basic Techniques

Symptoms based Online Failure Prediction requires a monitoring phase in which the system is observed. In this phase are identified which are the basic information used to represent the runtime system state. We call them *input data* and must capture the symptoms of faults. The input data need to pass through a pre-processing: a computation phase which aim is to provide, starting from basic information, complex informations representing the state of the system. The representation of the system has to be classified as failure-prone (i.e. symptoms of faults have been recognized and an upcoming failure is likely occurring) or safe. The classification phase, starting from a representation of the state, provides an estimation about its safety. An adaptation of the state representation, in order to make it feasible for the mathematical model used by the classification phase, is often required. As soon as a failure-prone estimation is recognized, a failure prediction has to be triggered by means of alerts. We call this sequence of steps *Online Failure Prediction chain*, depicted in Figure 3.1.

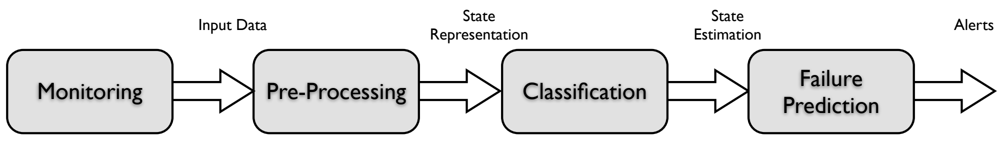


Figure 3.1: Online Failure Prediction chain.

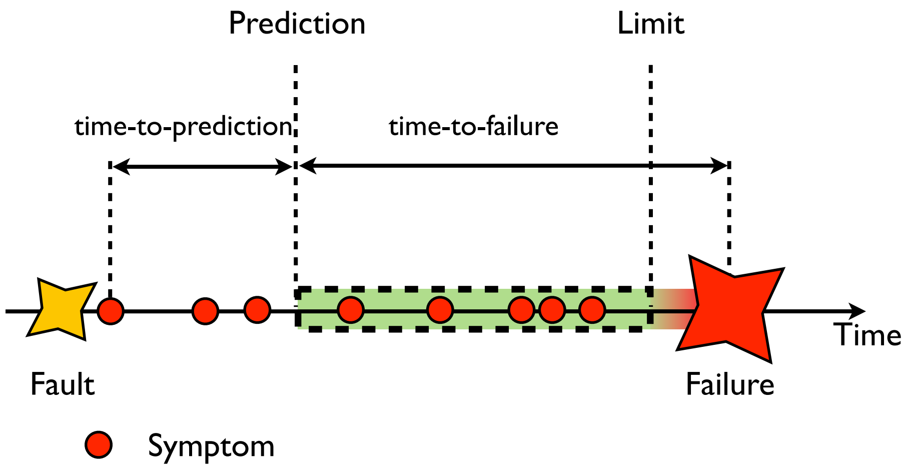


Figure 3.2: Fault, Symptoms, Failure and Prediction

In this chapter is presented how we characterized these phases using Complex Event Processing in the pre-processing phase and Hidden Markov Models as classifier of the system state, bonded by means of an aggregator; This chapter also introduces the failure and prediction model followed in this dissertation.

3.1 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 subject to failures. A failure is an event for which the service delivered by a system deviates from its specification [22]. 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 (e.g., an application-level problem or a network-level fault), affect the system in an observable and identifiable way. Thus, faults can generate side-effects in the monitored systems till the failure occurs. This 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. This assumptions are not new in literature, [61, 105, 109] use similar ones.

A failure prediction mechanism consists in monitoring the behavior of the distributed system looking for possible symptoms generated by faults, and in raising timely alerts regarding software failures if symptoms become severe. Proper countermeasures can be set before the failure to either mitigate damages or enable recovery actions. The following definitions introduce some important aspects in the task of prediction that are also represented in Figure 3.2.

The ***time to prediction*** *is the time from the instant of the fault and the instant in which a prediction is triggered.*

The ***time to failure*** *the distance in time between the occurrence of the prediction and the software failure event (i.e. a deviation from a correct behavior).*

The prediction has to be raised before a *limit time*, beyond which the prediction is not sufficiently in advance to take some effective actions before the failure occurs.

The ***limit time*** *is the time beyond which a failure prediction is not sufficiently in advance to take effective actions.*

As a matter of fact, an ideal failure prediction mechanism should produce zero false positives, i.e., zero errors in raising alerts regarding failures, and exhibit the maximum prediction time (i.e., it is able to produce an alert indicating the prediction of a failure when the first symptom is observed).

3.2 Data Pre-Processing

We modeled the input data as events which are correlated and combined in order to achieve complex information representing the state of the observed system. In order to do this, we used a Complex Event Processing engine called ESPER. Input data in real time are information that can be modeled as events. The event streams representing the input data need to be correlated, aggregated, and used to compute values representing metrics of performance. Since the criticality of the speed in which a failure-prone situation needs to be recognized (in order to maximize the *time-to-failure*) these calculation among events have to be performed in real time, as soon as the information arrive. Continuous queries techniques, and complex event processing in general, aim is exactly this. Here we explain how a representation of the system state can be built using a Complex Event Processing engine called ESPER, along with its language to specify rules, called Event Processing Language.

ESPER

The CEP engine chosen is ESPER that implements the CEP paradigms and provide a specific Event Processig Language in order to specify the continuous queries.

Event Processing Language In the 1980's, active databases technology were implemented to satisfy the real-time processing need, stimulating the creation and developing of languages to express rules and event patterns [46]. Today, we can find many computer languages dedicated to event processing well-founded in some kind of a SQL-like language, such as Event Processing Language from Esper project [3]. EPL was inspired by many of the ideas that have come out of the research and industrial CEP communities [3, 24, 102, 108]. An example of an EPL query is the following one:

```
INSERT INTO replies
    SELECT *
     FROM events
        WHERE
       events.type == 'reply'
```

The example demonstrates a number of the basic features of EPL. The From clause references the input stream of a given *event type*. Event types are mapped to Java classes. In the example, each event in the *events* stream has *type*, *size*, *timestamp* and other properties which must also be present on the associated Java class. The Where clause contains a simple expression that evaluates to true whenever the property *type* is "reply". The Select clause selects all the properties from the *events* stream that satisfy the Where clause. The 'Insert Into' clauses is an example of EPL extension to the SQL language. The Insert Into clause references a second event type, "replies", which defines the type of the events that are output by the example query. Actually a new stream of this event type is created at runtime. In the example, the mapping is very simple, but EPL does allow for much more complex mapping between event types. Another important EPL extension w.r.t. SQL is the possibility to define *input windows*. An input window is conceptually near to a table in a relational database. An input window can be temporal or spatial depending on the fact that the number of events inside the window is decided secondly a time-based rule or on a rule based on the number of events. In particular can be defined four types of window:

- batch-space: wait for n events, calculate among the n events arrived in a given stream. This kind of window is fixed in size and variable in time. There is one calculation each n events, among exactly n events.

- batch-time: wait for n seconds, calculate among the events arrived within the n seconds. This kind of window is fixed in time but variable in size. There is one calculation each n seconds.
- sliding-space: wait for the firsts n events and after calculate among the last n events as soon a new event arrives. Moving, fixed size window. There is one calculation per event, among the last n events.
- sliding-time: when a new event arrives, calculate among the events arrived in the last n seconds. Moving window, fixed time. There is one calculation per event among a variable number of events.

The windows are needed to the event processing languages in order to embody the concept of time, that is actually lacking in the SQL language. Among the most useful constructs present in EPL, *pattern* allow to capture the causality relation among events. Examples are:

- an A event followed by an A event;
- every A event followed by a B event;
- every A event followed by a B event which is followed by a C event.

Below we report an example of the use of the pattern.

```
INSERT INTO correlated_stream

    SELECT *

     FROM pattern(every A-> B).win:length(10)
```

In this example, the engine will detect an A event followed by a B event. When B occurs the pattern matches; after the match, the engine will look for the next event A that is followed by a B event. The pattern matcher restarts and looks for the next A event. The Pattern clause is very powerful and can be used in very different manner in order to cover practically all kinds of event correlation. Note that according to the window defined, the relations among events captured can be spatial or temporal: e.g. an event A followed by a B event may not satisfy the query if the B event is not inside the window, in the example inside 10 events batch-window. The "every" operator is very important: the logic of the query varies according to the brackets and of the position of it. Examples are:

- *every* $(A \rightarrow B)$ detects an A event followed by a B event.

- *every* $A \rightarrow B$ the pattern fires every A event followed by a B event.
- ($A \rightarrow$*every* B) the pattern fires for an A event followed by every B event.
- *every* $A \rightarrow$ *every* B the pattern fires every A event followed by every B event.

Patterns may be used in combination with the where clause, group by clause, having clause as well as output rate limiting and insert into.

In any case a query is satisfied Esper allows three possible way to handle the events that have satisfied it. In particular those events can be inserted into another event stream (via the *insert into* operator), can be given to a java method executed as soon as the query is satisfied (listeners and subscribers) or both of the them. *insert into* is very powerful since can be used to create a network of streams separating and joining them.

3.3 Classification

The classification is the heart of the online failure prediction. Given a representation of the state of the system, a classifier has to provide, according to its knowledge about the system, an estimation about the safety of the state. This classification yields to a straightforward decision: if the system is recognized as unsafe, trigger a failure prediction, don't trigger it otherwise. As stated before we chosen Hidden Markov Models to classify and recognize the state of the system. Here we report the mathematical formulation of it and the classical algorithms defined on the HMM. Also details about the training of the model are provided, which allow to build the knowledge base of the classifier.

Hidden Markov Models

Rabiner [89] with "An Introduction to Hidden Markov Model" illustrates very well the model and how this model can be used in several way. Interested readers can refer to [89] for further details on the model. Here some basic concepts are reported.

HMM consists of a hidden stochastic process, i.e. a Markov chain whose state is not observable (*hidden*) and a set of symbols. The hidden process emits an observable symbol each time it changes state according to a probability distribution. In particular, the HMM is identified by the transition probabilities of the hidden process and by the *emission* probabilities: the probability to emit a specific observable symbol being the Markov process in a given state. Formally an HMM consist of four elements:

1. a homogeneous first order Markov chain: a set $\{s_0, s_1, \dots, s_t, \dots\}$ of the process states at time $t \geq 0$. The elements s_t assume values in $\Omega = \{\omega_1, \dots, \omega_N\}$ where Ω is the set of the N possible states of the model. The Markov chain is characterized by a $N \times N$ matrix A, whose elements are

$$a_{i,j} := p(s_t = \omega_j | s_{t-1} = \omega_i), \qquad \forall i, j = 1, \dots, N$$

and represent the probabilities to transit from the state ω_i to ω_j at time $t \geq 2$. Moreover the following property holds:

$$\sum_{j=1}^{N} a_{ij} = 1, \quad \forall i = 1, \dots, N$$

2. a set $\{o_0, o_1, \dots, o_t, \dots\}$ of the observable symbols at time $t \geq 0$. This is an identically distributed and discrete time stochastic process assuming values in $\Sigma = \{\sigma_1, \dots, \sigma_M\}$, where Σ is the set of the M possible observable symbols, namely, the alphabet of the model.

3. a $N \times M$ matrix B whose elements are

$$b_k(\sigma_j) := p(o_t = \sigma_j | s_t = \omega_k), \quad \forall k = 1, \dots, N, \, \forall j = 1, \dots, M$$

and represent the probabilities to emit a symbol σ_j at time t, given that the state of the Markov chain is ω_k. Each symbol of the alphabet Σ can be emitted according to a given probability if the Markov model is in state ω_k.

4. a vector $\pi(0)$ having the generic element $\pi_i(0)$ defined as follows:

$$\pi_i(0) := p(s_0 = \omega_i), \quad \forall i = 1, \dots, N.$$

$\pi_i(0)$ represents the probability that the Markov chain is in the state $s_0 = \omega_i$ at the initial time $t = 0$. The following property holds:

$$\sum_{i=1}^{N} \pi_i(0) = 1.$$

A generic HMM results completely defined by $\lambda := (A, B, \pi(0))$. Figure 3.3 is a graphical representation of a generic HMM.

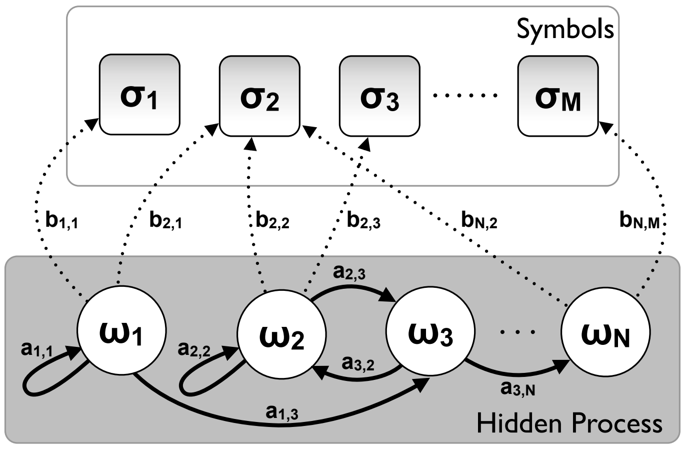


Figure 3.3: A graphical representation of a generic Hidden Markov Model. The $\omega_1, \ldots, \omega_N$ vertices are the hidden states of the Markov process. The observable symbols are $\sigma_1, \ldots, \sigma_M$. Some elements of A and B matrices are represented as edges.

In literature [89] three fundamental problems on HMM are defined:

- *evaluation problem*: consider a sequence of observable symbols $O = \{o_0, o_1, \ldots, o_L\}$ and a HMM . What is the probability that the given sequence O can be generated by the HMM? This probability is called sequence likelihood. The *Forward algorithm* (sec. 3.3) provides an efficient solution to this problem: given a sequence of observations, it computes the probability that the HMM could emit the sequence O.

- *decoding problem*: given a sequence and a HMM, what is the most probable sequence of hidden states the process has travelled through while producing the given observation sequence? The *Forward-Backward* and *Viterbi* algorithms (described later) provide solutions to this problem. Formally given a sequence of observations $O = \{o_0, o_1, \ldots, o_L\}$, discover the most probable hidden state sequence $P_\star = \{s_0, s_1, \ldots, s_L\}$ able to generate the observed sequence O.

- *training problem*: given a set of sequences of observations $\{O_i\}_{\{i=1,\ldots,n\}}$, what are the optimal HMM parameters (i.e. A, B, $\pi(0)$) to maximize the probability of emitting each O_i? The *Baum-Welch* training algorithm (described later) yields a solution by iteratively converging to at least a local maximum.

Forward-Backward Algorithm

The Forward-Backward Algorithm is composed by two parts. The forward part provides a solution for the computation of sequence likelihood (evaluation problem), while the combination of the two part solves the decoding problem. Given an observation sequence $O = \{o_t\}_{\{t=0,\dots,L\}}$ and a HMM with parameters $\lambda = (A, B, \pi(0))$, the likelihood of O is denoted with $P(O|\lambda)$. If we assume that the sequence $S = \{s_t\}_{\{t=0,\dots,L\}}$ of hidden states is known, P can be computed by:

$$P(O|\lambda) = \pi(0) b_{s_0}(o_0) \prod_{t=1}^{L} a_{s_{t-1},s_t} b_{s_t}(o_t),$$

Since only O is known, all the possible state sequences S have to be considered and summed:

$$P(O|\lambda) = \sum_{S} \pi(0) b_{s_0}(o_0) \prod_{t=1}^{L} a_{s_{t-1},s_t} b_{s_t}(o_t).$$

Starting from this, an efficient reformulation exploiting the Markov condition (transition probabilities are time homogeneous and depend only on the current state) yields to the*Forward algorithm.*

Forward Algorithm. Assuming that the stochastic process is in state i at time t, the "forward variable" $\alpha_t(i)$ is defined:

$$\alpha_t(i) = P(o_o o_1 \dots o_t, s_t = \omega_i|\lambda).$$

$\alpha_t(i)$ can be computed recursively:

$$\begin{cases} \alpha_0(i) = \pi_i(0) b_{s_i}(o_0) \\ \alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_{s_j}(o_t) \quad 1 < t \leq L \end{cases} \tag{3.1}$$

As $\alpha_L(i)$ is the probability of the entire sequence having the stochastic process in the state i at the end of the sequence, we can compute the sequence likelyhood i.e. the solution of the evaluation problem:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_L(i).$$

Backward Algorithm. The backward variable is defined as:

$$\beta_t(i) = P(o_{t+1} \dots o_L | s_t = \omega_i, \lambda)$$

and denotes the probability of the rest of the sequence $o_{t+1} \dots o_L$ having the process in the state ω_i at time t. $\beta_t(i)$ can be computed recursively in a similar way:

$$\begin{cases} \beta_L(i) = 1 \\ \beta_t(i) = \sum_{j=1}^{N} a_{ij} b_{s_j}(o_{t+1}) \beta_{t+1}(j) \quad 1 < t \leq L-1. \end{cases} \tag{3.2}$$

Forward-Backward Algorithm. Combining the forward and backward variables can be obtained the probability that the process is in the state ω_i at time t given an observation sequence O:

$$\gamma_t(i) = P(s_t = \omega_i | O, \lambda).$$

after some computation (please refer to [97] or [89]), can be obtained:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)}.$$

Note that the implementation of this algorithms is not straightforward due to some underflow problems. [112] and [111] cope this and several other practical problems in implementing a forward-backward algorithm for an explicit-duration hidden Markov model.

Viterbi Algorithm

Viterbi Algorithm slightly modify the Forward-Backward algorithm by introducing the probability of the most probable state sequence for the sub-sequence of observations $o_0 \dots o_t$ that ends in the state ω_i. Defined modifying the $\alpha_t(i)$ variable, that probability is the following:

$$\delta_t(i) = \max_{s_0 \dots s_{t-1}} P(o_0 \dots o_t, s_0 \dots s_{t-1}, s_t = \omega_i | \lambda).$$

$\delta_t(i)$ can be computed by substituting the sum in the 3.1 with the maximum operator:

$$\begin{cases} \delta_0(i) = \pi_i b_{s_i}(o_0) \\ \delta_t(j) = \max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij} b_{s_j}(o_t) \quad 1 < t \leq L \end{cases} \tag{3.3}$$

Training of Hidden Markov Model

Among the features of HMMs, the possibility to estimates the λ parameters is of paramount relevance. The solution of training problem allows to estimates the parameters, training in this way the model, using for instance recorded sample data. In terms of HMMs, the goal of training is to maximize sequence likelihood for training sequences. More precisely, the λ parameters have to be set such that $P(O|\lambda)$ is maximized. This is the approach followed in the experimental evaluation in order to tune the HMM parameters of the architecture designed. The most important HMM training algorithm is known as the *Baum-Welch Algorithm.*

The Baum-Welch Algorithm. In the forward-backward algorithm, the HMM parameters (A and B matrices) were assumed to be fixed and known. For convenience of explanation consider a simpler case where the sequence of hidden states is known. The parameters of the HMM can be optimized by maximum likelihood estimates:

- The probabilities $\hat{\pi}_i$ are determined by the relative frequency of sequences starting in state ω_i

$$\hat{\pi}_i = \frac{\text{number sequences starting in } \omega_i}{\text{total number of sequences}}$$

- The elements of the A matrix i.e. transition probabilities are determined by the number of transitions from ω_i to ω_j divided by the total number of transition from the ω_i state:

$$\hat{a}_{ij} = \frac{\text{number of transitions}(\omega_i, \omega_j)}{\text{number of transitions}(\omega_i, \omega_k)\forall \omega_k \in \Sigma}$$

- The elements of the B matrix i.e. emission probabilities are determined by the number of times the process has generated symbol σ_j (i.e. an emission of σ_j) being in the state ω_i divided by the number of time the process has been in the state ω_i:

$$\hat{b}_i(o_j) = \frac{\text{number of } \sigma_j \text{ emissions}}{\text{number of times the process has been in state } \omega_i}$$

Note that if the training is “supervised”, the sequences of states is known and what described is sufficient to build the A and B matrices and the π_i as well. This is the approach followed in this work. However, a description of the Baum-Welch algorithm follows. The objective of the algorithm is to provide expectation values for the unknown quantities and consist in a two-step procedure. Based on an existing model λ' (possibly obtained randomly), the first step transforms the objective function $P(O|\lambda)$ into a new function $Q(\lambda', \lambda)$ that essentially measures a divergence between the initial model λ' and the updated model λ. The second step consist in the maximization of the $Q(\lambda', \lambda)$, since that $Q(\lambda', \lambda) \geq Q(\lambda', \lambda)$ implies that $P(O|\lambda) \geq P(O|\lambda')$. The algorithms continues by replacing λ' and , λ and repeating the two steps until some stopping criterion is met. The algorithm is of a general hill-climbing type and is only guaranteed to produce fixed-point solutions, although in practice the lack of global optimality does not seem to cause serious problems in recognition performance [41]. More details regarding this algorithm can be found in [90]. The Baum-Welch algorithm is not the only one solving the estimation problem. Interested reader can refer to [90] to have other alternatives offering different modeling perspectives.

3.4 Aggregator

Often the information required by mathematical or probabilistic models are very simple, while a representation of a system state is something quite complex. A software component that simplifies complex information may be useful if not mandatory in such kind of application. This component can be particularly useful between the *pre-processing* phase and the *classification* phase (see Figure 3.4) of the Online Failure Prediction chain. This section introduce the *Aggregator*, a software component that simplifies the information by aggregating them.

The aggregator is a software component that takes in input a vector $\mathbf{V} \in \mathbb{R}^N$ and gives in output a single value in an interval $A \subset \mathbb{N}$, with $|A| = M$ (see Figure 3.4). The aggregator works using a fix square grid of $\mathbb{R}^N$ constituted by $M = D^N$ $N-$dimensional intervals, where D is the number of 1-dimensional intervals per each component of $\mathbb{R}^N$. The N-dimensional intervals are numbered from 1 to M. Figure 3.5 represents an example of the defined square grid in $\mathbb{R}^2$ with $M = 16$.

In order to perform its task, the aggregator needs to know a-priori estimations of the input vectors. In particular a maximum and a minimum values of their components have to be provided during a configuration phase:

$$v_i \in [v_i^{Min}, v_i^{Max}], \quad \forall i = 1, \dots, N,$$

for each component v_i of the input vectors $\mathbf{V}$. Starting from $\mathbf{V} = (v_1, v_2, \dots, v_N)$ the aggregator defines a new vector $\mathbf{V}_{normalized} = (v'_1, v'_2, \dots, v'_N)$ as follows:

$$v'_i := \frac{D(v_i - v_i^{Min})}{v_i^{Max} - v_i^{Min}}, \quad \forall i = 1, \dots, N. \tag{3.4}$$

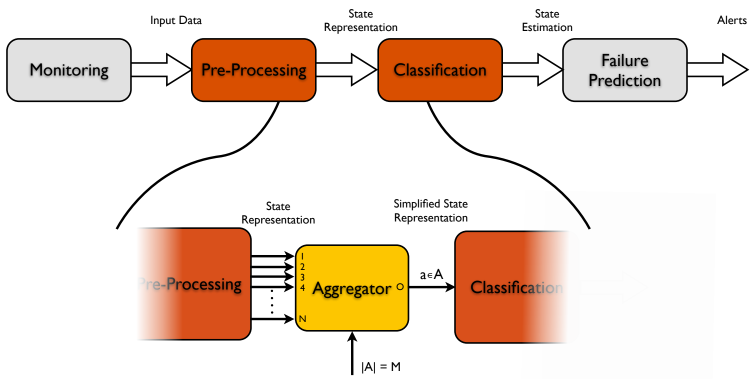


Figure 3.4: Aggregator. N input lines receive a vector of numeric values. The output will be a symbol belonging to an ordered set of integers. The cardinality of the set has to be given.

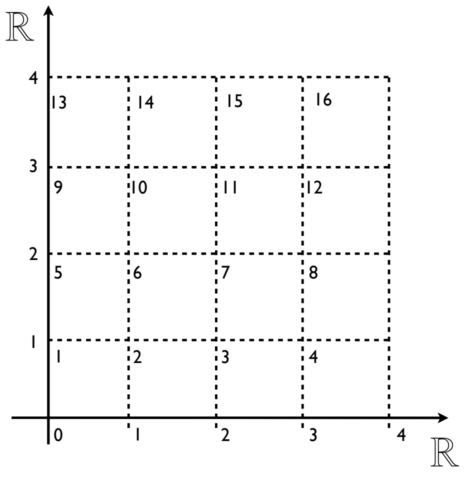


Figure 3.5: Example of square grid in $\mathbb{R}^2$ with $D = 4$.

The 3.4 simply normalizes each component v_i (according to its maximum and minimum values) in order to have $v_i' \in [0, D]$, for all i. This means that $\mathbf{V}_{normalized}$ identifies a point in the square grid defined before. Each component of $\mathbf{V}_{normalized}$ is now compared with the intervals of the grid in order to identify the $N-$dimensional interval $\mathbf{V}_{normalized}$ belongs to. Consider the following example to clarify the aggregator behavior.

Example

Assume an aggregator in $\mathbb{R}^2$ tuned with a square grid of $M = 16$ 2-dimensional intervals. This means that there are $D = 4$ 1-dimensional intervals per component. Assume also that $v_1^{Min} = v_2^{Min} = 0$ and $v_1^{Max} = v_2^{Max} = 100$ Consider for instance an input vector $\mathbf{V} = [51.34, 58.22]$. Applying the 3.4 formula, the normalized vector $\mathbf{V}_{normalized}$ can be computed:

$$\mathbf{V}_{normalized} = (\frac{4(51.34 - 0)}{(100 - 0)}, \frac{4(58.22 - 0)}{100 - 0}) = (2.05, 2.034).$$

The vector $\mathbf{V}_{normalized}$ obtained can be represented in the square grid, see Figure 3.6. A comparison with the intervals boundary can easily identify the

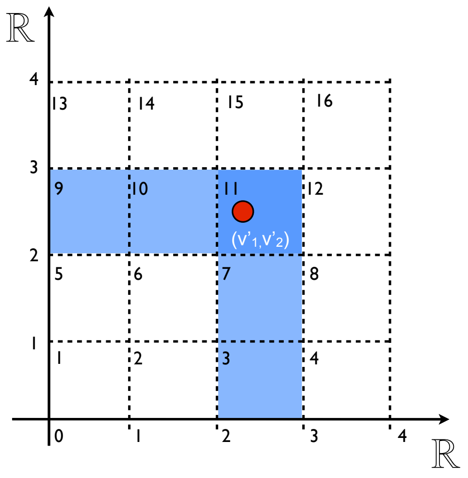


Figure 3.6: Example of aggregator behavior with $N = 2$ and $D = 4$.

2-dimensional interval which the point belongs to. In this case the interval is the number 11.

Chapter 4

Architecture

In this chapter the architecture designed for monitoring mission critical distributed systems, e.g. Air Traffic Control Systems, is presented. The architecture has been named CASPER and implements all the techniques presented in chapter 3. CASPER is characterized by the following features:

- *online*, as the failure prediction is carried out during the normal functioning of the monitored system,
- *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
- *black-box*, as no knowledge of the application's internals and of the application logic of the system is analyzed.

Before introducing the architecture, some assumptions has to be done.

4.1 Assumptions

Not every failure can be predicted efficiently, some can not be predicted at all. The architecture presented has been developed using a strong assumption: faults that lead to failures, independently of the fault's root cause (e.g., an application-level problem or a network-level fault), affect the system in an observable and identifiable way. Thus, faults can generate side-effects in the monitored systems till the failure occurs. Other assumptions are:

- A fault generates increasingly unstable performance-related symptoms indicating a possible future presence of a failure.
- The system exhibits a steady-state performance behavior with a few variations when a non-faulty situation is observed.

These assumptions are not new in literature ([61, 105, 109]) and are representative of a number of real situations experienced in real systems. We also recall that faults that are *dormant* cannot be identified since, by definition [72], have not been activated (see Figure 1.1). Errors that instantly lead the system to a failure cannot be detected by the architecture presented. Instant failures caused by hardware misbehaviors (e.g. power issues) cannot be predicted for obvious reasons.

4.2 CASPER Architecture

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) [76] and Hidden Markov Models (HMM) [89]. 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. CASPER also provides hints about the identity of the faulty hosts, using a regularity-based activity detection mechanism, along with the prediction. Note that we use HMM rather than other more complex dynamic bayesian networks [77] since it provides us with high accuracy, with respect to the problem we wish to address, through simple and low complexity algorithms. Figure 4.1 shows the principal modules of CASPER that are described in isolation as follows.

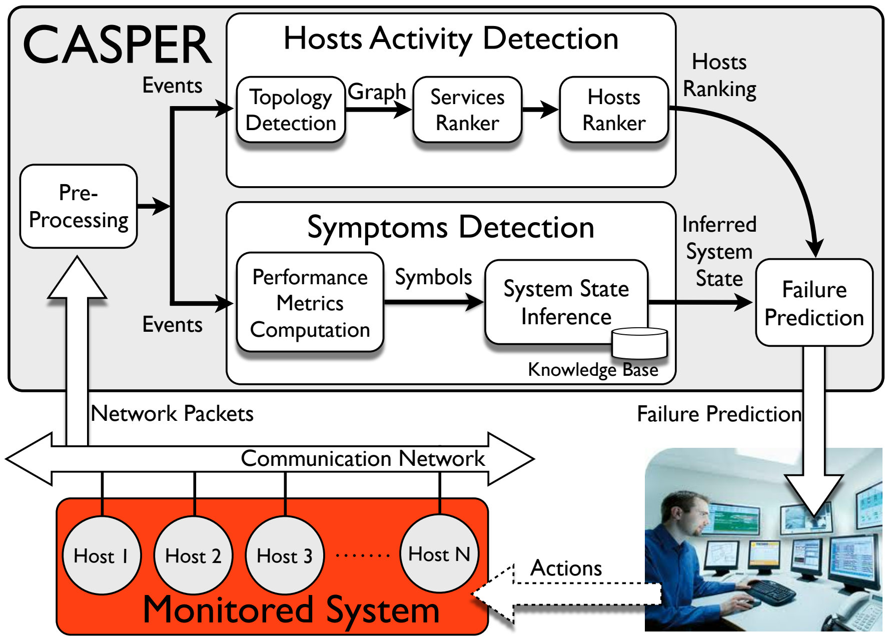


Figure 4.1: The modules of the CASPER failure prediction architecture

4.2.1 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 streams of events carry a large amount of information that is obtained from the entire set of network packets exchanged among the interconnected nodes of the monitored distributed system (see Figure 4.1). Considering all the packets allows CASPER to own a larger view of what is happening on the network, thus augmenting the chance of discovering specific performance patterns that show the evidence of possible symptoms of failures. 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 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 two main CASPER's modules (see below).

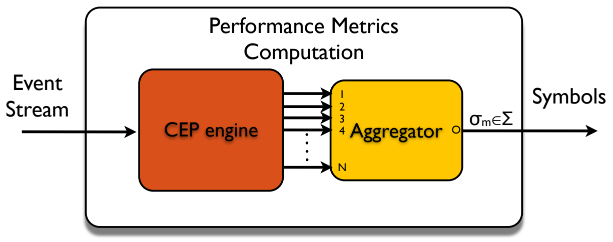


Figure 4.2: Performance Metrics Computation component

Events in CASPER

As a completely non-intrusive framework, CASPER considers as input event *the fact that a network packet has been captured*. Each network packet embodies a plenty of information and, if the system to observe is middleware-based, standard communication protocols are used. Each event will contain application-agnostic information such as timestamp, source and destination host and port number, and middleware information captured using a dissector, as state of the services, exceptions, requests and replies id. CASPER uses all these information to build an event for each network packet. All these events will be processed and correlated using Complex Event Processing techniques. The objective is to represent the system state using only network data captured at runtime from the system's infrastructure.

4.2.2 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 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

A CEP engine and an Aggregator (defined in sec 3.4) compose in the performance metric computation component (see Figure 4.2). The module periodically produces as output a representation of the system behavior in the form of *symbols* (see Figure 4.1). Note that, CASPER requires a *clock mech-*

anism 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 *P performance metrics*, i.e., a set of time-changing metrics whose value indicates how the system actually works (see later). The output of this module is a symbol σ_m, where $m = 1, \ldots, M$ per each clock cycle. Each symbol is built by the Aggregator starting from a vector $\mathbf{V}$ of $\mathbb{R}^P$. The generic entry v_i of $\mathbf{V}$ is the mean value the performance metric i assumes during the past time interval. The aggregator takes in input this vector of $\mathbb{R}^P$ and returns an integer value σ_m belonging to the finite alphabet Σ i.e. the alphabet of the observations of HMM.

Real-time performance metrics

After long time of observations of several metrics of a real mission critical distributed systems, we identified the following 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 replies 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 that, in a given temporal window, do not receive a reply;
- *Messages Rate:* the number of messages exchanged in a given spatial or temporal window.

All of these metrics are computed by the CEP engine starting from a stream of events. The basic event considered is the the fact that a network packet has been captured from the network. The motivation of this are the non-intrusiveness and the black box observation we performed.

The system state inference component

This module 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 Hidden Markov Models.

We recall that HMM consists of a *hidden stochastic process*, a set of *symbols* Σ and two probability matrices A and B as defined in Section 3.3. Figure 4.3 shows how we instantiated HMM in our architecture. The knowledge about the possible systems state is embodied in the two matrices A and B. Each time interval, one of the M symbols (i.e. $\sigma m, \forall m = 1 \ldots M$) composing the alphabet Σ, will be emitted by the performance metric computation component (more precisely by the aggregator). CASPER uses the forward algorithm (3.3) to recognize the most likely hidden state. If we examine the recursive formulation of the forward probability calculation, we can see that it is assumed to have an observation $O = (o_0, \ldots o_L)$ of L symbols when the algorithm begins. The base step of the recursive formula starts from the first symbol o_0 and all the involved operations are commutative. Therefore, we modify the algorithm in order to have an online version where a new instance of the vector $\alpha_t(i)$ is computed for each new symbol, using only $\alpha_{t-1}(i)$. The main advantage of this approach is that we can use an arbitrary, even infinite, length of the observation L since we do not need to store all the $L-1$ symbols but only the last one. The computational complexity of the online version saves an L multiplicative factor with respect to the recursive formulation on each run. Each clock cycle this module produces in output the most likely hidden state.

4.2.3 Hidden Markov Model as a state recognizer

We model the state of the system to be monitored by means of the *hidden process*. We define the states of the system (see Figure 4.3) as:

- *Safe*: the system behavior is correct as no "active fault" [22] is present;
- *Unsafe*: some faults, and then symptoms of faults, are present. There will be an unsafe state per each kind of possible fault. We assume a finite number of k typologies of faults (e.g., memory stress, disk stress).

The number of states N is the sum of the k unsafe states and the *safe state*: $N = k + 1$. We assume that initially the system is in the *safe* state: if we call it ω_1, then

$$\pi_1(0) = p(s_0 = \omega_1) = 1$$

Since the state of the system is not known a priori, we can observe it only looking at the *emissions* of symbols. Figure 4.3 represents the emitted symbols as the set of $\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_M\}$. In addition, Figure 4.3 shows labeled

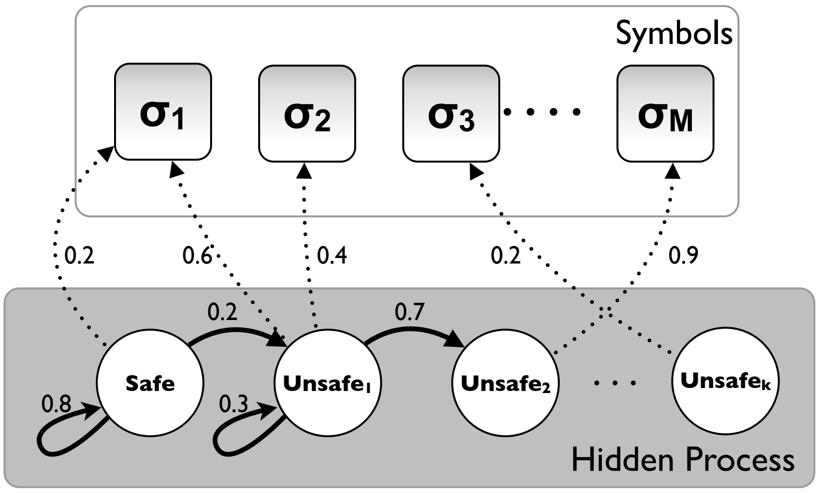


Figure 4.3: Hidden Markov Models graph used in the system state inference component

edges among the vertices of the hidden process; these represent the values of the A matrix, i.e., the matrix of the transition probabilities. In contrast, the edges that connect the states of the hidden process to the symbols are the probabilities to emit a given symbol σ_m, that is, the B matrix of HMM. CASPER considers as emissions a combination of the performance metrics described before, that the aggregator gives in output periodically. The idea of the aggregation is similar to the approach found in [104] and [59] presented in Chapter 2 (Figure 2.5 and Figure 2.3 respectively), but with several differences. The HMM classifies each observable symbol σ_m in one of the N hidden states. According to our definition of σ_m, HMM actually classifies each N-dimensional interval created by the aggregator. Figure 4.4 clarifies the concept representing a situation with 2 performance metrics $p1$ and $p2$ using the aggregator configured with $M = 16$. Each of the 2-dimensional intervals is classified as *unsafe* or *safe* by the HMM. The red circle in Figure 4.4 represent a symbol $\sigma_{11} = 11$.

The aggregator mechanism, combined with the HMM, allows to classify each N-Dimensional interval produced by the aggregator as *safe* or *unsafe*[1] in a completely automated way. The system can build its knowledge base (actually constituted by the A & B matrices) without knowing in advance the critical values (i.e. values assumed if symptoms of fault are present) for the

[1]Anyone of the states of the Hidden Markov Process.

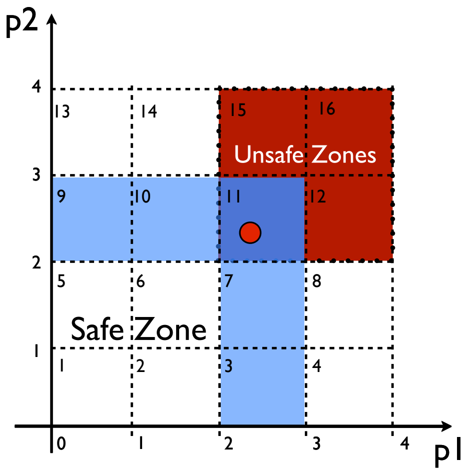


Figure 4.4: Unsafe and safe zones of a two-performance metrics square grid.

performance metrics. With respect to other classification mechanisms that require human intervention in order to recognized the thresholds, to build the decision tree, to build the bayesian network and so on, this is an advantage. Another advantages that comes from the use of the HMM is the lightweight of the knowledge base. The whole architecture can be trained building the $A\&B$ matrices and tuning the parameters in a testing environment and after sent via mail to the field where has to be used. The drawbacks are (i) that and the choice of the number of symbols M is critical and (ii) the maximum values of the performance metrics has to be known in advance[2].

4.2.4 Hosts Activity Detection Module

The symptoms detection module analyzes the observed distributed system behavior as a single component. Therefore the inferred system state represents the state of the whole distributed system: nothing can be stated regarding the single hosts. The Host Activity Detection Module (HADM) aim is to cope this problem, providing a periodic snapshot of the hosts status. In particular the HADM allows to:

- disclose the network topology of the observed system in a completely non-intrusive fashion;

[2]Know in advance e.g. the maximum Round Trip Time that will arrive can be a problem but some automated solutions have been developed.

- create a ranking of the network level services based on their regularity in term of network activity, i.e. the mean number of message produced in a given temporal window;
- create a ranking of the network hosts based on the *per-host* services ranking.

In order to perform these actions, the HADM uses exactly the same event stream produced by the pre-processing module. The HADM architecture embodies three components namely *Topology Detector Component, Service Ranker* and *Hosts Ranker*. A description of these components is now provided.

Topology Detector Component

Since each event received by HADM represents a network packet exchanged by two hosts in a given port, it is easy to represent all the interactions among the hosts using a graph. The aim of this submodule is to provide a representation of the network topology of the system observed that is updated in real time. Each hosts is represented as a vertex of a graph. Each logical link between two hosts is represented as a directed edge between two vertices. A logical link (edge) represents a connection between two hosts using a source port and a destination port. If there are more connections among two hosts there are more edges, obtaining in this way a *directed multi-graph*, otherwise known as *pseudograph* (see Figure 4.5). The idea behind this submodule is not only to represent the topology, but also to provide some real-time information about the hosts and about the links among them. In order to do this, among the several off-the-shelf libraries, we chose JUNG [6]. JUNG allows to graphically represents the topology in real time, but also to use vertices/edges colors and shapes to represent several information in a quick and understandable way: e.g. if the message rate of a given link is too high, the relative edge can be drawn thicker, if an host is inactive i.e. it does not send messages, the relative vertex can be drawn black and so on. Figure 4.5 shows an example of JUNG output. We also invite to view a video[3] to have an idea of the output produced in real time by this module and of the whole framework during its functioning.

[3]link to the video: www.cis.uniroma1.it/projects/casper.php

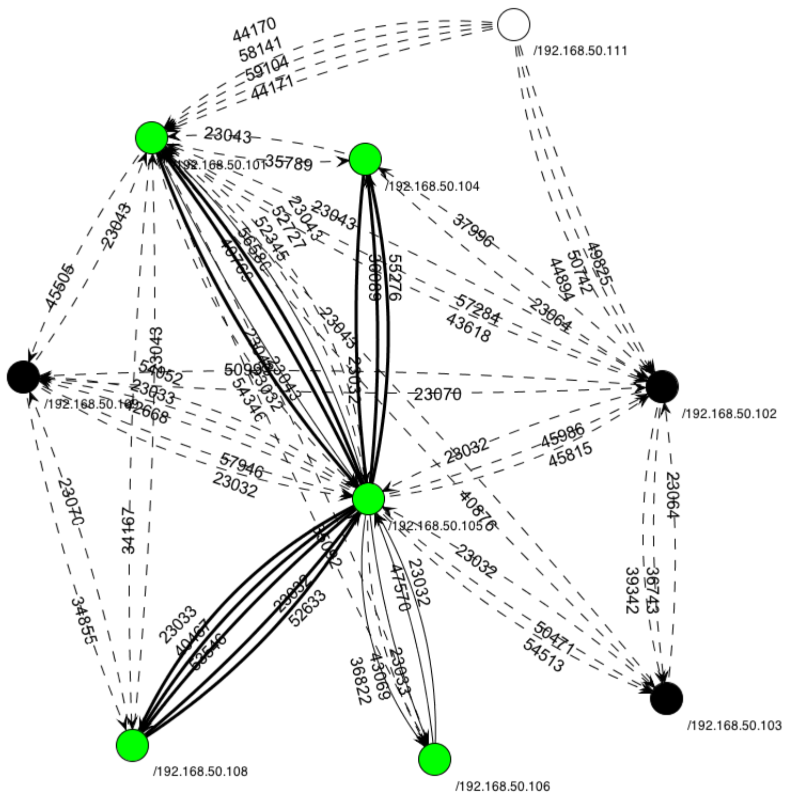


Figure 4.5: An example of graph representing a 9-nodes ATC system. The black nodes are inactive nodes while the white node is a source node (with no inner edges). The dotted edges are inactive network links, the number on the edges is the port number.

Services Ranker component

The Service Ranker Component takes in input a live updated graph and assign to each service a vote based on its regularity. A host is the source of at least one but usually several services. The Services Ranker component provides a ranking among all the services of the hosts. This ranking is used twice: is provided to the operator and is given in input to the Host Ranker Component (see Figure 4.1). The ranking is calculated using the vote assigned (and updated) to each service periodically, basing on a mathematical function of the service outer message rate. Simply spoken, if the average message rate of a service is near its historical rate then the vote is decreased otherwise is augmented. To compute the historical rate we used the exponential moving average, also known as *exponentially weighted moving average* (E.M.A.) defined as follows:

$$E.M.A. := \begin{cases} S_1 = Y_1 \\ S_t = \alpha Y_t + (1-\alpha) S_{t-1} \quad t > 1. \end{cases}$$

where the coefficient α represents the degree of weighted decrease, a constant smoothing factor between 0 and 1. A higher α discounts older observations faster. Y_t is the observation at a time period t. S_t is the value of the E.M.A. at any time period t. Given that any service has its own vote, it is easy to compute a ranking among all the services of one host. This ranking can be used to point out the services with higher vote i.e. a worse behavior. There are two adjustable parameters in this ranking feature: (i) the deviation between the average (how much the Y_t is far from the average) and (ii) α. Both the parameters can be tuned in real time using the CASPER GUI [4].

Hosts Ranker Component

The output of the previous Services Ranker component is used to compute a ranking among the hosts. This ranking is provided to the operator and is combined with the output of the Symptoms Detection module by the Failure Prediction Module. The hosts ranking submodule uses two information to compute its output: (i) the per-host ranking of the services and (ii) a performance metric already used in CASPER, the request over replies (the number of messages whose type is request over the number of messages whose type is reply). Each host will be equipped with a vote, as did with the services. The host ranking is computed by multiplying the sum of all the votes of its services and its ratio request over reply. The ratio will be 1 if the host received the same number of requests and replies. If it receives more requests this means

[4]CASPER GUI can be seen in the video linked in the previous footnote.

that the host is overloaded, the ratio will be major than 1 and their vote will be higher (worse). If otherwise the host sends more replies w.r.t. the requests, the ratio will be less than 1 and the vote will be lower (better).

4.2.5 Failure Prediction module

It is mainly responsible for correlating the information about the state received from the System State Inference component and from Host Ranker component. It takes in input the inferred state of the system at each CASPER clock-cycle and the host ranking. The inferred state can be a safe state or one of the possible unsafe states $unsafe_1 \dots unsafe_k$. Using the CEP engine, this module counts the number of consecutive $unsafe_i$ 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 3.2. The alert will also contain the ranking of the hosts so the operator can have an overview of the single hosts, in the moment of the prediction.

4.3 Training and Tuning of CASPER

4.3.1 Training of the model

The knowledge base (see Figure 4.1) concerning the possible safe and unsafe system states of the monitored system is composed by the matrices A and B (defined before, sec. 3.3) of the system state inference module. This knowledge is built during an initial training phase. If the A matrix represents how the system behaves, the B matrix represents what we can see about the system behavior. To adjust the entries of these matrices is the solution to the *training problem* defined before. It is the most difficult of the three problems: there is no known way to solve for a maximum likelihood model analytically; moreover solve this problem without knowledge about the path of system states is a NP Complete problem and can only be approximate using heuristics and complex algorithms (e.g. Baum-Welch algorithm, see 3.3). If the path of hidden states that generates the observations is known, the parameters of the matrices can be calculated using the maximum likelihood re-estimation technique [89]. This architecture contains the latter technique: during the training, CASPER is fed concurrently by both recorded network traces and a sequence of pairs `<system-state,time>` that represents the state of the monitored system (i.e.,

$safe, unsafe_1, \ldots, unsafe_k$) at a specific time[5]. No training is required for the other parts of the architecture.

4.3.2 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:

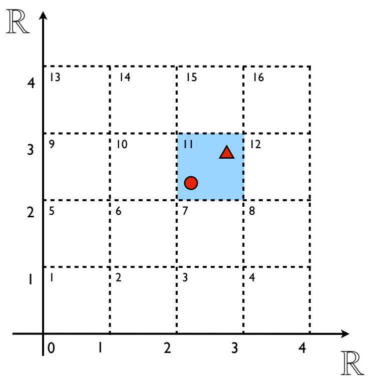
- length of the CASPER *clock period*;
- *number of symbols* output by the performance metrics computation module, i.e. the parameter M of the aggregator;
- length of the failure prediction module *window size.*

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 M influences the system state inference: if a high number of symbols are chosen, a higher precision for each performance metrics can be obtained but also a lower repeatability of the symbols emitted. The repeatability is fundamental since the knowledge base is built according to the frequency of symbols being the system in a given hidden state (see The Baum-Welch Algorithm, sec. 3.3) and likely two "near values" represent the same system state. Too many symbols imply less repeatability as clarified in Figure 4.6 example.

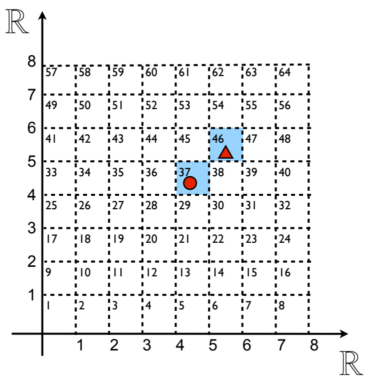
The failure prediction window size corresponds to the minimum number of CASPER clock cycles required for raising a prediction alert. The greater the window size, the more the accuracy of the prediction, i.e., the probability that the prediction actually is followed by a failure (i.e. a true positive prediction). The tradeoff is that the time-to-prediction increases linearly with the windows size causing shorter time-to-failure (see Figure 3.2);

During the training phase, CASPER automatically 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.

[5]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.



(a) $M = 16$



(b) $M = 64$

Figure 4.6: An example of decreased repeatability: two near values, a circle and a triangle, in the (a) case will corresponds to the same symbol 11 while in the (b) case will correspond to two different symbols, 37 and 46.

Chapter 5

Evaluations and results on a real ATC system

We deployed and tested CASPER failure prediction capabilities in several test environments: a CORBA-based distributed system, a CARDAMOM based virtual machines cluster and finally a real Air Traffic Control system. In this chapter we present the results obtained in the real Air Traffic Control testing environment owned by Selex Sistemi Integrati, a Finmeccanica company that develops and maintains ATC systems of the European and Italian market.

The first part of the work on the field has been to collect a number 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 failures conditions, and we collected the relative traces. In our test field, we consider one of the nodes of the ATC system be affected by either Memory or I/O stress. After the collection of all these traces, we trained CASPER and once the training phase was over we deployed CASPER again on the testing environment of the ATC system in order to conduct experiments in operative conditions. The results in a real environment like ATC confirmed the results obtained in the other studied environments, mainly showing (i) the CASPER accuracy in detection of the state of the monitored system and (ii) the CASPER capability to predict a failure caused by these conditions.

5.1 Evaluation Metrics

Our evaluation assesses the system state inference component accuracy and the failure prediction module accuracy (see Figure 4.1). 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 Table 5.1 metrics that define the accuracy of CASPER.

Table 5.1: Accuracy metrics.

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

We evaluate the failure prediction 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.

5.2 Monitored System

“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 [26, 27, 28, 29] 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 [5, 81].

5.2.1 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. 5.1) 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 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.

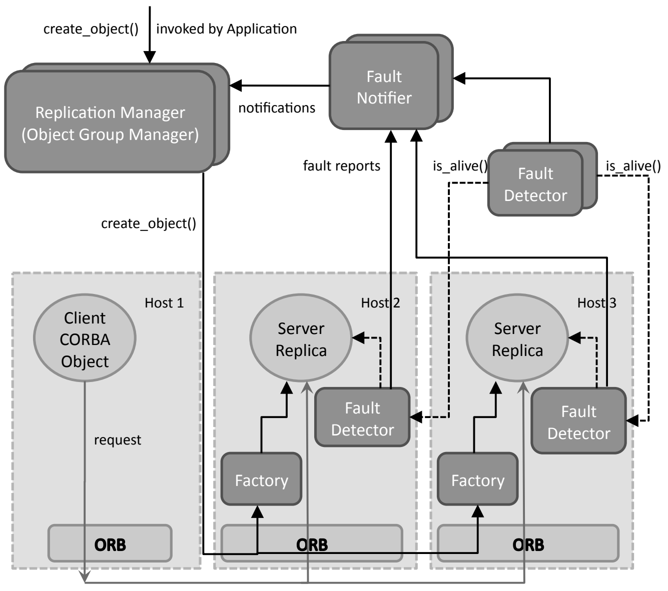


Figure 5.1: FT CORBA framework

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 [35], 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. 5.2) 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

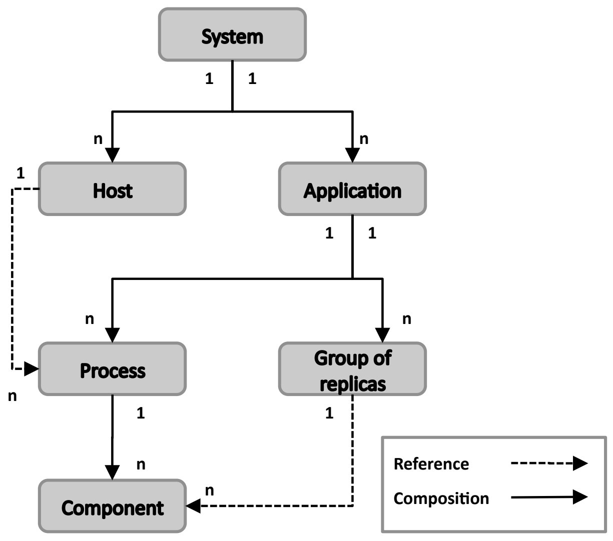


Figure 5.2: System decomposition in application, process, component, group and host.

ports [80]. In this context, the exposed ports (facets) of the server components are defined as objects of FT CORBA groups. This approach suits 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” [83]. 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 [82]. 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. 5.3): in fact it has been chosen as the foundation for a safety critical subsystem, the core part of a next generation ATC system. In order to sep-

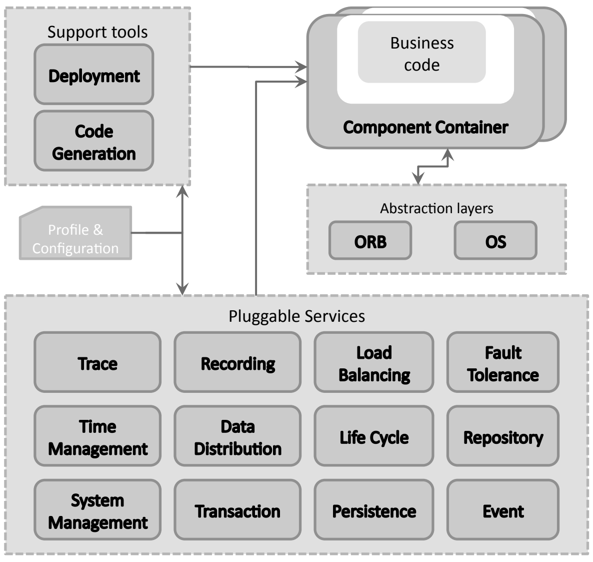


Figure 5.3: CCM and CORBA based middleware services.

arate 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.

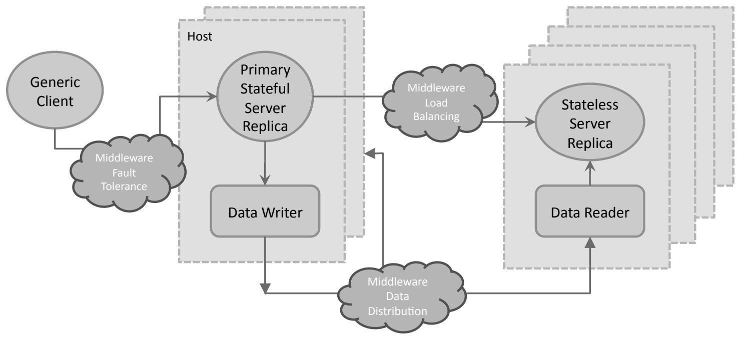

Host
Generic Client
Primary Stateful Server Replica
Middleware Fault Tolerance
Middleware Load Balancing
Stateless Server Replica
Data Writer
Data Reader
Middleware Data Distribution

Figure 5.4: 3-tier architecture.

This architecture (see Fig. 5.4) 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 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.

5.2.2 Testbed

We deployed CASPER in a dedicated host located in the same LAN as the ATC system to be monitored. 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 environments are composed by 8 or 4 machines, 16 cores 2.5 GHz CPU, 16 GB of RAM each one. It is important to remark that CASPER does not know anything about the environment in which has been placed. Neither the application nor the service logic nor the testbed details are available to CASPER.

5.2.3 Faults Injection

Policy restrictions and privacy issues forbid the injection of faults modifying the source code or the machine code, restrictions on installing third-part monitoring software and on hosts log-in forbid techniques like G-SWIFT or fault injection and emulation in general (see [23, 48, 79] for an overview on these techniques). According to these restrictions, we had to apply stress conditions rather then fault injection, in order to emulate the frequent conditions that yield the ATC system to 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. For this reason we decided not to show the results of these tests.

5.2.4 Training Data

We trained CASPER (see Section 4.3.1) using the following recorded traces:

1. between 10 and 13 minute long traces in which the ATC system is behaving in a steady-state.

2. between 10 and 11 minute long traces (at least 3 per each kind of injected stress, i.e., memory and I/O stress) in which al least one of the services of the ATC system fails.

These traces are taken from the testing environment of the ATC system.

5.3 Performance Metrics

To monitor a real ATC system, some performance metrics have to be identified and coded in the EPL language used by ESPER, i.e. the CEP engine used. In order to enhance the failure prediction accuracy we considered the subset of the performance metrics that are more influenced by the stress condition we injected. This is due to the fact that some of these are not influenced in the specific case study (e.g. the *percentage of exceptions* since we don't experienced exceptions, *message size* since remained constant, *percentage of replies* because this metric can be deducted from the *number of requests without reply*, if the system does not use one way interactions).

EPL queries to identify performance metrics

Each of the presented performance metrics (Sec. 4.2.2) needs an EPL query to be computed. Below are reported some of them, in particular the three considered to monitor the real ATC system, i.e. *Round Trip Time, Number of request without reply, message rate.*

Round Trip Time performance metric. The RTT performance metrics is among the more important metric in every request-reply interaction. It measure the time from the moment in which the request has been issued till the relative reply has been received by the client. Variations of the mean RTT can represent overloading situation of a servant, network link problem, several situations on the server implying a grater processing time. In the worst case can represent a workload change, tricking the failure prediction mechanism. The EPL code is reported below.

```
select (rep.numericTimestamp - req.numericTimestamp) as RTT
    from  pattern[every
            req=eventStream(messageType = 'REQUEST') ->
            rep=eventStream(messageType = 'REPLY'
and (requestID = req.requestID) //semantic context identifier
and (src_ip = req.dst_ip)
and (dst_ip = req.src_ip))]//network level context identifier
```

There is an evident happened-before relationship between a request and a reply. The EPL query recognizes this causality by selecting the correct reply among the cloud of the replies in the event stream (using the request id). The "pattern" operator expresses the fact that a reply is generated if and only if the relative previous request has been issued. The main context identifier is an identifier that is present in the event, called "requestID". The difference between the timestamps is clearly the wanted metric.

Rate of the messages. The messages rate is the number of messages are traveling the network per second. It is important to monitor the behavior of this performance metric: deviations may show a particularly overloaded situation or situations in which the nodes are not producing their usual load. The EPL query in order to do this is the following:

```
select rate(1 sec) as messageRate

         from eventStream
```

The query is particularly simple since the predefined function of EPL *rate* compute it. Only the amount of time to consider has to be specified.

Rate of the messages carrying an exception. Complex distributed systems have to deal with exceptions every time. The events we treated embody information about exceptions in the client-server request/reply interaction. We considered the percentage of exception in the set of the replies as a performance metric: a growing number of exception can represent problems on some servant. The EPL query in order to do this is the following:

```
select count(*) as percentage

         from eventStream.win:length(100)

         where replyStatus = 'EXCEPTION'
```

The query is simply a filter on the field "replyStatus".

Number of Requests without Reply. The number of requests that, in a given temporal window, do not receive a reply is a metric that recognize if any server is for example, overloaded and it does not answer anymore. Considering the mean of this value an overview on the load of the system can be obtained. This number should always be near zero. A similar performance metric is the rate requests over replies that represent the same information and should always be around 1. Both of these metrics assumes that in the system there are only request and replies, note that this is not always true (e.g. one-way interactions). A simple way to implement this is reported in the EPL query below. It simply counts the number of the reply events every 100 events. The value returned is the percentage of the replies.

```
select count(*) as count
              from eventStream.win:length(100)
              where messageType='REPLY'
```

5.4 Results

The results are divided in three subsections: the training of CASPER, the tuning of the parameters, and the failure prediction evaluation (using network traces and deploying the system).

Training of CASPER.

During the training phase, the performance metrics computation component produces a symbol at each CASPER clock cycle. Thanks to the set of pairs `<system-state,time>` we are able to represent the emitted symbols in case of safe and unsafe system states. Figure 5.5 illustrates these symbols. Each symbol is calculated starting from a combination of three values, using the *Aggregator* presented in sec 3.4. In this case, we have $D = 6$ possible values per each performance metric; the number of different symbols is therefore $M = 6^3 = 216$, being $N = 3$ the number of performance metrics. Observing Figure 5.5 we can notice that the majority of the emissions belong to the interval $[0, 2]$ for the *Round Trip Time*, and $[0, 1]$ for *Number of Request Without Reply* and *Message Rate*. Note that only the index of the 3-dimensional interval will reach the System State inference component, according to the aggregator logic. Starting from the symbols represented in Figure 5.5, the HMM-based component builds the matrices A and B. After that CASPER can be considered trained.

Tuning of CASPER parameters: clock period and number of symbols.

After the training of HMM, CASPER requires a tuning phase to set the clock period and number of symbols in order to maximize the accuracy (F-measure, precision, recall and false positive rate) of the *symptoms detection module* output. This tuning phase is done by feeding the system with a new recorded network trace (different from the one used during the training), with the relative set of pairs `<system-state,time>`. CASPER will try several values of clock period and number of symbols (the M parameter of the aggregator) and

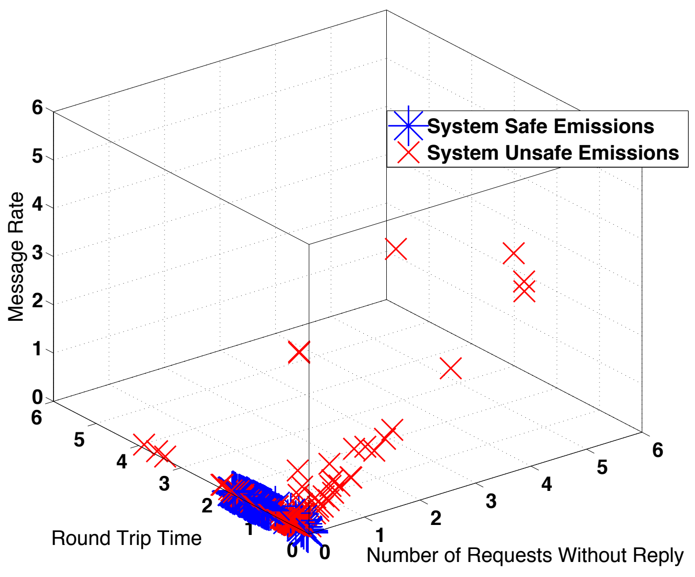


Figure 5.5: Symbols emitted by the performance metrics computation component in case of a recorded trace that exhibits stress of the memory.

will compute the F-Measure. Figure 5.6 plots the obtained F-Measure values when varying clock period and number of symbols.

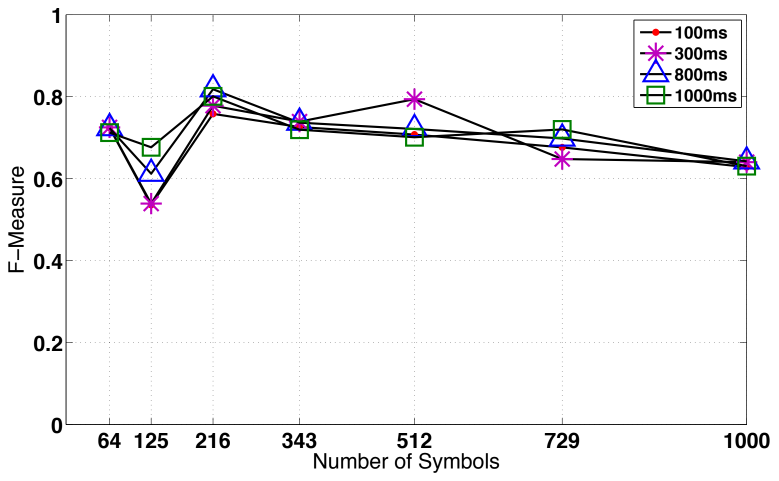


Figure 5.6: Symptoms detection module: F-Measure varying the number of symbols and clock period in case of a recorded trace subject to memory stress

We can see that the best choice of the clock period is 800 milliseconds. This period yields a higher F-Measure value than the other clock values in most of the number of symbols considered in the plot (note that this value of the clock also resulted the best choice in case of I/O stress). Figure 5.6 also shows that the accuracy of the symptoms detection module is highly influenced by the number of symbols and less influenced by the clock period (e.g., if we consider a number of symbols greater than 216, the F-measure values vary of less than 10% when augmenting the clock period from 100ms to 1000ms). Thus, CASPER set the clock period to 800 milliseconds, choosing the max F-measure value. Once fixed this clock period, the second parameter to define is the number of symbols. Figure 5.7 shows the precision, recall, F-measure and false positive rate of the symptoms detection module varying the number of symbols.

CASPER considers the maximum difference between the F-measure and the false positive rate in order to choose the ideal number of symbols (ideally, F-measure is equal to 1 and f.p.r. to 0). As shown in Figure 5.7, considering 216 symbols (6 values per performance metric) we obtain $F = 0.82$ and $f.p.r. = 0.12$ which is actually the best situation in case of memory stress.

Figure 5.8 shows the accuracy of the symptoms detection module in case

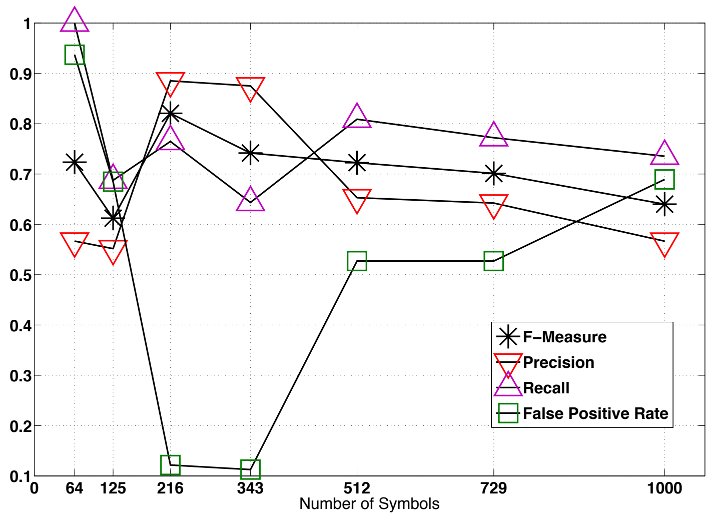


Figure 5.7: Performance of the symptoms detection module varying the number of possible symbols in case of a recorded trace subject to memory stress. CASPER clock period 800 ms

of I/O stress. Best values of F-Measure and false positive rate ($F = 0.86$ and $fpr = 0$) are obtained for 512 symbols (meaning 8 values per performance metrics).

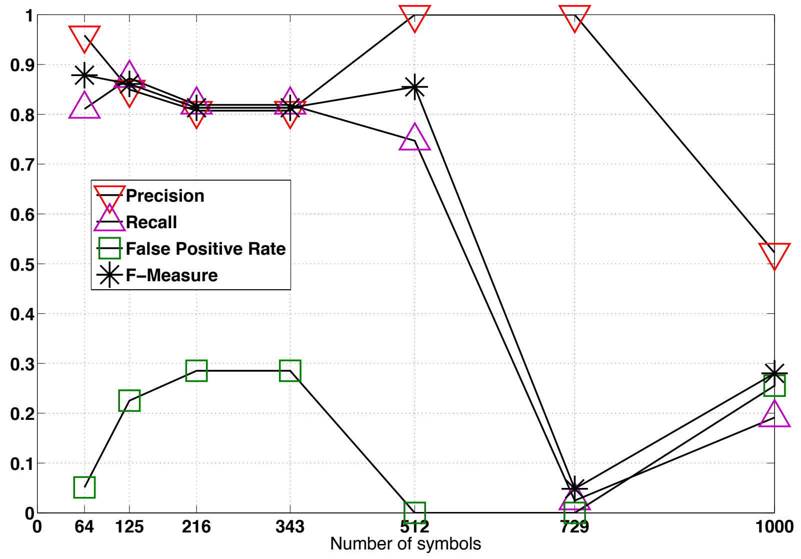


Figure 5.8: Performance of the symptoms detection module varying the number of possible symbols in case of a recorded trace subject to I/O stress. CASPER clock period 800 ms

Tuning of CASPER parameters: window size.

The window size is the only parameter that has to be tuned by the operator according to the tradeoff discussed in Section 4.3.2. We experimentally noticed that during fault-free executions the system state inference still produced some false positives. However, the probability that there exists a long sequence of false positives in steady-state is very low. Thus, we designed the failure prediction module to recognize sequences of consecutive clock cycles whose inferred state is not safe. Only if the sequence is longer more than a certain threshold CASPER triggers a prediction. The length of these sequences multiplied by the clock period (set to 800ms) is the window size. The problem is then to set up a reasonable threshold in order to avoid false positive predictions during steady-state, Figure 5.9 illustrates the number of the false positive varying the window size.

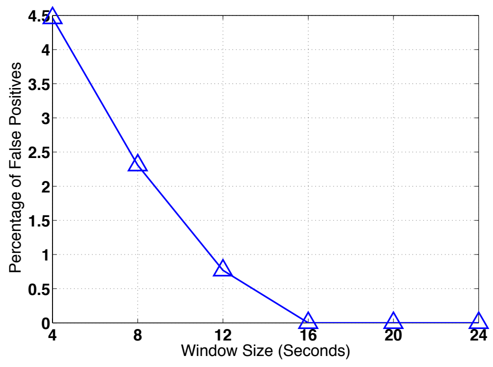


Figure 5.9: False positives varying the window size feeding. CASPER is fed with a recorded trace behaving in steady-state. Its clock period is 800 ms

From this Figure it can be noted that the window size has to be set to at least 16 seconds in order not to incur in any false positives. Let us remark that the window size also corresponds to the minimum time-to-prediction. All the results presented below are thus obtained using a window size of 16 seconds.

Results of CASPER failure prediction.

We run two types of experiments once CASPER was trained and tuned. In the first type, we injected the faults described in section 5.2.3 in the ATC testing environment and we carried out 10 tests for each type of fault[1]. In the second type, we observed the accuracy of CASPER when monitoring for 24h the ATC system in operation. These types of experiments and their related results are discussed in order as follows.
As first test, we injected a memory stress in one of the node of the ATC system till a service failure. Figure 5.10 shows the anatomy of this failure in one of the tests. The ATC system runs with some false positive till the time the memory stress starts at second 105. The sequence of false positives starting at second 37 is not sufficiently long to create a false prediction. After the memory

[1]The number of tests has been limited by the physical access to the ATC testing environment. In fact, every experiment of 10 minutes takes actually 2 hours to be completed due to the storage of the data, the stabilizing and rebooting of the ATC system after the failure.

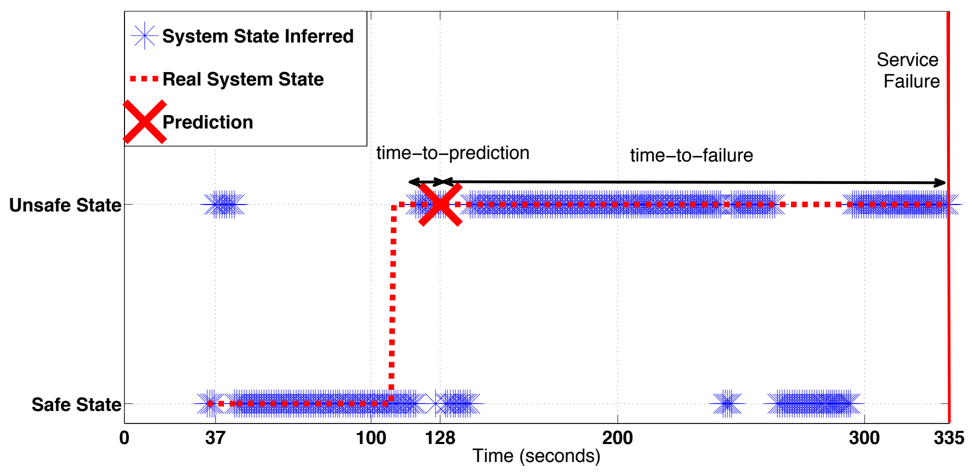


Figure 5.10: Failure prediction in case of memory stress starting at second 105. Window size 16s, clock period 800ms, time-to-prediction 23s, time-to-failure 207s

stress starts, the failure prediction module outputs a prediction at second 128; thus, the time-to-prediction is 23s. The failure occurs at second 335, then the time-to-failure is 207s, which is satisfactory with respect to ATC system recovery requirements. We can also see a little burst of system state inference component's false negatives starting at second 128, successfully ignored by the failure prediction module. Figure 5.11 shows the anatomy of the failure in case of I/O stress in one test. A failure caused by I/O stress happens after 408 seconds from the start of the stress (at second 190) and has been predicted at time 222 after 32 seconds of stress, with a time-to-prediction equal to 376 seconds before the failure. There is a delay due to the false negatives (from second 190 to 205) that the system state inference component produced at the start of the stress period. The time-to-prediction is 21s. 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]. Figure 5.12 summarizes these results.

Finally, we performed a 24h test deploying CASPER on the network of the ATC system in operation. In these 24 hours the system exhibited steady-state

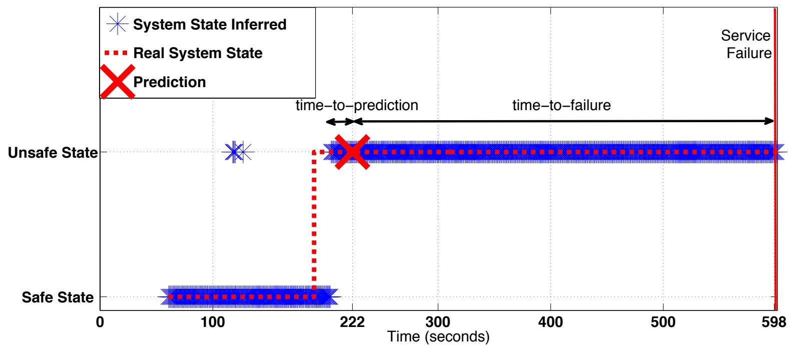


Figure 5.11: Failure prediction in case of I/O stress starting at second 408. Window size 16s, clock period 800ms, time-to-prediction 21s, time-to-failure 376s.

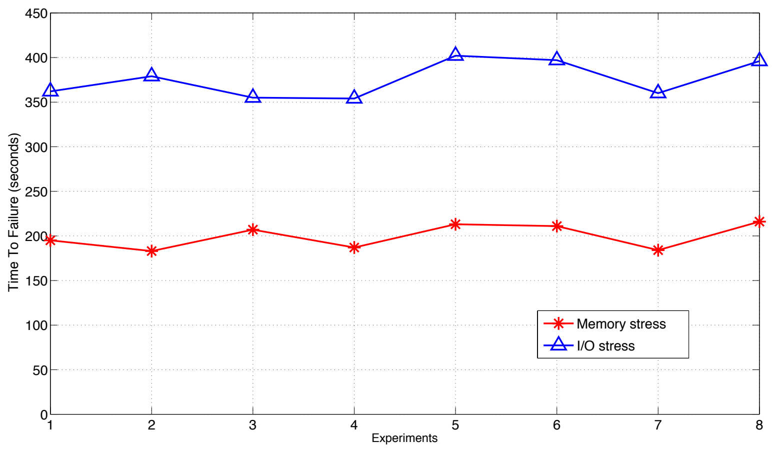


Figure 5.12: Performance of CASPER in terms of time-to-failure.

performance behavior. CASPER did not produce any false positive along the day. Figure 5.13 depicts a portion of 400 seconds of this run in which there are several false positives in the system state inferred but no failure predictions thanks to the choice of the window size.

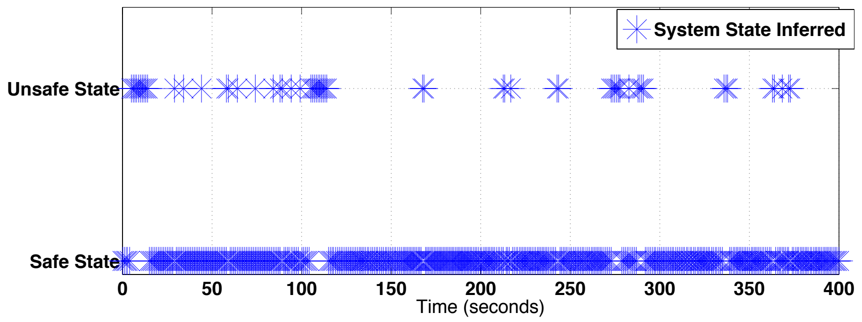


Figure 5.13: 400 seconds of a steady-state run of the ATC system in operation.

Results of CASPER hosts ranking.

The host ranking is a real time ranking among the hosts. Figure 5.14 and Figure 5.15 represent the behavior of the hosts votes during two different runs. Figure 5.14 represents a situation in which, at second 191, a memory stress application has been run in the host named 102. The regularity of this host is highly affected by the stress. The host ranking module recognizes this fact and assigns, to host 102, an higher and higher vote. Also the other hosts are affected by the misbehavior of the host 102 implying the related votes growing. At second 270 the host 102 halted and its vote starts to grown linearly.

Figure 5.15 shows the results of the Host Activity Detection module obtained observing an 8 hosts system. In this case the unsafe host, suffering a memory stress, has not been halted by the stress. A higher and higher votes have been assigned nonetheless.

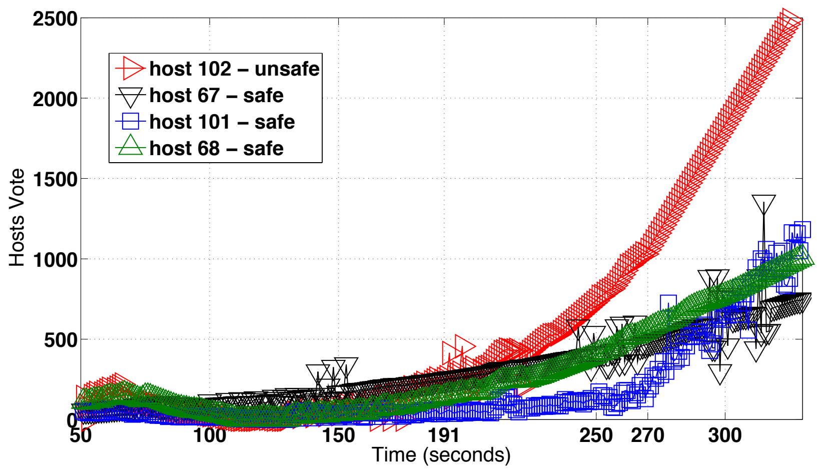


Figure 5.14: Hosts ranking behavior. At second 191 a memory stress starts on host number 102. Starting from second 191 will have always the higher mark.

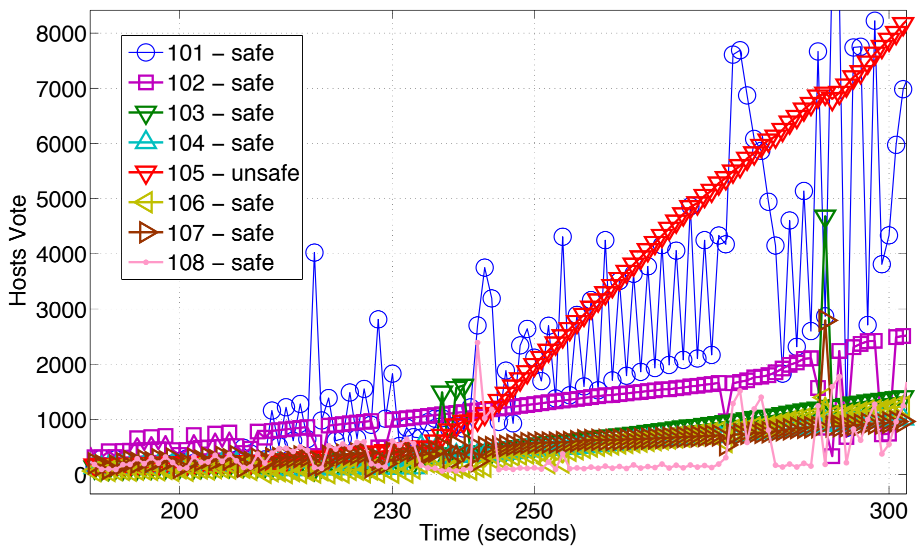


Figure 5.15: Hosts ranking behavior. At second 230 a memory stress starts on host number 105. After some seconds the vote of the host 105 grows rapidly, despite the host remained alive.

Chapter 6

Conclusion

In the last few years, the *online failure prediction* paradigm gained attention as a good approach to enhance the dependability of complex distributed systems. Applications can be found in several fields, but rarely this kind of techniques are applied to mission-critical distributed systems, such as air traffic control, battlefield applications, naval command and control systems. In such complex real-time systems, the failures may happen with potentially catastrophic consequences, hence, fault management techniques are usually at the current state of the art. The problem is that mission-critical distributed systems are designed and deployed in a given time and after that is not easy to equip such systems with novel fault management subsystems. The idea to have a "plug-and-play" mechanism that, in a completely non-intrusive manner, monitors an already deployed mission-critical system is very attractive in the industrial community. This interest of the community allowed us to study, design, develop, apply and test state of the art and novel fault management techniques, on the real field.

In this thesis, we proposed to apply online failure prediction techniques in mission critical distributed systems. We devised a novel combination of two state of the art paradigms, complex event processing and hidden Markov models. In particular, in Chapter 3 we described how we represent the state of complex distributed middleware-based systems by means of a set of performance metrics, computed at runtime using complex event processing. These metrics are influenced by the change of the conditions of the system, or by the change of the system monitored workload. We called the critical degrading performance of the system *symptoms* of faults, we train the HMM-based model to recognize these situations, that usually lead the system services to failures. The model, properly trained, timely recognizes deviations of the correct system behavior and thus can trigger alerts, warning the upcoming failure. All of this is done only using application-indipendent information, captured observing network interactions. The source of the input event-stream of the complex event processing is composed infact by only network-captured data. The approach is both non-intrusive and black-box as there is no need to install any software on the observed system and the model does not infers relations among the system components but only on the whole system behavior.

The main contribution of the thesis was to devise a framework to predict online failures of mission critical distributed systems. The failure prediction architecture, namely CASPER, described in Chapter 4, provides accurate predictions of failures by exploiting only the network traffic of the monitored system. In this way, it results non-intrusive with respect to the nodes hosting the mission critical system and it executes a black-block failure prediction as no knowledge concerning the layout and the logic of the mission critical distributed system is used. To the best of our knowledge, this is the first failure prediction system exhibiting all these features together. Let us remark that the black-box characteristic has a strategic value for a company developing such systems. Indeed from a company perspective the approach is succeeding as long as the failure prediction architecture is loosely bound to the application logic. Additionally there should be no direct or indirect influence between the monitored and the monitoring system, in order not to falsify the measures and create hidden workload transfer. The advantage that no additional load to the monitored system is introduced is not the only one. The non-intrusiveness also implies that the approach can be applied in all the existing middleware based systems without modifications of the architecture. Results showed that CASPER, after a careful training phase, achieved good accuracy in terms of false positives when both failure conditions and steady-state behaviors were considered. Additionally, the exhibited time-to-failure when injecting stress conditions inside the monitored system resulted quite reasonable.

As future work, we are investigating versions of CASPER that are able to improve automatically the knowledge base, starting training session during the functioning of the framework. This will improve the ability of CASPER to predict failure conditions and enhance its “plug-and-play” characteristic. Another problem in investigation is to provide to the model the capability to recognize workload changes never seen before, that usually yield to false positive predictions: a feature that online informs the framework that the situation coming is a workload change and not a deviation from the correct known behavior. This will allow CASPER to automatically train the HMM parameters to recognize the new behavior rather then be tricked by it. Finally, we are working to embody the designed framework in the supervision mechanism of a real Air Traffic Control System. We envision that in this integration several lessons will be learnt with consequent improvements to the model.

Concluding, let us remark that the proposed solution presents some open problems that we plan to address in the close future. Example of problems is the workload change already stated. But also the training of model can be a problematic task: even if it can be easily performed in an ATC system (the ATC systems have a testing environment that is a copy of the deployed system, so the training can be performed on the testing environment) it is not easy recreate the symptoms of all possible faults that can afflict a distributed system. The ideal solution would be the use of state of the art fault injection techniques, but as we seen, is not always possible. We devised a partial solution by adding an "unknown state" in the model: if no one of the known states is recognized, safe or unsafe, an unknown state can be identified as unsafe. Each time an unknown state is recognized can be triggered a relative alert. This solution has partially solved the problem but suffers of the workload changes nonetheless. It is also true that if the training session is done accurately, a workoad change in a mission critical systems is a task extremely rare and if unawares even rarer. Another problem is tuning of the aggregator component. It needs as a matter of fact (i)a good choice of the D parameter (i.e. the number of values that each performance metric can assume), and (ii)a correct estimation of the performance metrics values ranges. Both this parameters influence the failure prediction accuracy.

Bibliography

[1] Rtm analyzer web site. http://www.realtime-monitoring.de/index.php/en/productsaservices/rtm-analyzer. 42

[2] Tibco business event web site. http://www.tibco.com/products/business-optimization/complex-event-processing/businessevents/default.jsp. 42

[3] Esper project web page, 2011. http://esper.codehaus.org/. 20, 36, 46

[4] IBM's System S Web Site, 2011. http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html. 36, 38, 40

[5] Object management group webpage, 2011. http://www.omg.org/. 73

[6] JUNG - Java Universal Network / Graph Framework, 2012. http://jung.sourceforge.net/. 65

[7] C. A. Bayes predictive analysis of a fundamental software reliability model. *In IEEE Transactions on Reliability*, 39(3):177–183, 1990. 23

[8] P. A., R. F., and A. R. Bayesian analysis and prediction of failures in underground trains. *Quality and Reliability Engineering International*, 19(4):327–366, 2003. 23

[9] A. Adi and O. Etzion. Amit - the situation manager. *VLDB J.*, 13(2):177–203, 2004. 38

[10] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. *In SIGOPS Oper. Syst. Rev.*, 37:74–89, October 2003. 18, 30

[11] J. Aitchison and I. Dunsmore. *Statistical Prediction Analysis*. Cambridge University Press, 1980. 23

[12] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, Aug. 2008. 38

[13] J. Alon, S. Sclaroff, G. Kollios, and V. Pavlovic. Discovering clusters in motion time-series data. In *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 375–381, 2003. 33

[14] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *In ICDCS 2006*, pages 71–79, 2006. 38

[15] R. V. a. Andreão and J. Boudy. Combining wavelet transform and hidden markov models for ecg segmentation. *EURASIP J. Appl. Signal Process.*, 2007(1):95–95, Jan. 2007. 33

[16] E. Angori, R. Baldoni, V. Bortnikov, G. Chockler, E. Dekel, G. Laventman, and G. Lodi. A Collaborative Environment for Customizable Complex Event Processing in Financial Information Systems. Technical report, MIDLAB 5/10, 2010. 38

[17] L. Aniello, R. Baldoni, G. D. Luna, and G. Lodi. A Collaborative Event Processing System for Protection of Critical Infrastructures From Cyber Attacks. In *The 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2011)*, 9 2011. 38

[18] R. Anreão, B. Dorizzi, and J. Boudy. Ecg signal analysis through hidden markov models. *In IEEE Transactions on Biomed. Eng.*, 53(8):1541–9, August 2006. 33

[19] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, 1985. 11

[20] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin. The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Trans. Comput.*, 20(11):1312–1321, 1971. 11

[21] A. Avizienis and J. Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. *IEEE Computer*, 17(8):67–80, August 1984. 7

[22] A. Avizienis, J. Laprie, B.Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004. 44, 62

[23] D. Avresky, J. Arlat, J. C. Laprie, and Y. Crouzet. Fault injection for formal testing of fault tolerance. *Reliability, IEEE Transactions on*, 45(3):443–455, 1996. 7, 78

[24] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30:109–120, September 2001. 46

[25] R. Baldoni, S. Bonomi, G. Lodi, and L. Querzoni. Data Dissemination supporting collaborative complex event processing: characteristics and open issues. In A. press, editor, *Workshop on Data Distribution for Large-Scale Complex Critical Infrastructures*, 4 2010. 38

[26] R. Baldoni and C. Marchetti. Three-tier replication for FT-CORBA infrastructures. *Software Practice & Experience, 2003*, 6 2003. 73

[27] R. Baldoni, C. Marchetti, and A. Termini. Active Software Replication through a Three-tier Approach. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02), October 13-16, 2002 Osaka, Japan.*, pages 109–118, 10 2002. 73

[28] R. Baldoni, C. Marchetti, and A. Virgillito. Design of an Interoperable FT-CORBA Compliant Infrastructure. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems Systems (ERSADS'01)*, 5 2001. 73

[29] R. Baldoni, C. Marchetti, A. Virgillito, and F. Zito. Failure Management for FT-CORBA Applications. In *Proceedings of the 6th IEEE International Workshop on Object Oriented Real-time Dependable Systems (WORDS'01)*, 1 2001. 73

[30] H. Berenji, J. Ametha, and D. Vengerov. Inductive learning for fault diagnosis. In *Proceeding of IEEE International Conference on Fuzzy Systems*, 2003. 24

[31] W. Blischke and D. Murthy. *Reliability: Modeling, Prediction, and Optimization.* Wiley Series in Probability and Statistics. John Wiley & Sons, 2000. 10

[32] J. Bowles. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Reliability.*, 41(1):2–12, 1992. 10

[33] A. B. Brown and D. A. Patterson. Embracing failure: A case for recovery-oriented computing (roc), 2001. 12

[34] D. C and S. J. Efficient failure handling in grid computing using failure prediction algorithm. In *proceeding of ICCCI 2012*, 2012. 29

[35] CARDAMOM. Cardamom middleware website, website. `http://www.cardamom.eu/`. 20, 74

[36] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging, 2001. 10

[37] K. M. Chandy. Event-driven applications: Costs, benefits and design approaches. In *Gartner Application Integration and Web Services Summit*, 2006. 34

[38] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Proceedings o IEEE International Conference on Dependable Systems and Networks (DSN 2008)*, pages 452–461, 2008. 29, 30

[39] D. Coleman and C. Thompson. Model based automation and management for the adaptive enterprise. In *12th Annual Workshop of HP OpenView University Association,*, pages 171–184, Porto, Portugal, July 2005. 12

[40] A. Daidone, F. Di Giandomenico, A. Bondavalli, and S. Chiaradonna. Hidden Markov models as a support for diagnosis: Formalization of the problem and synthesis of the solution. In *Proceedings of 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*, pages 245–256, Leeds, UK, October 2006. iii, 19, 33, 34

[41] P. D.B. Training of hmm recognizers by simulated annealing. *Proceedings of IEEE international conference on Acoustic, Speech and Signal Processing*, pages 13–16, 1985. 54

[42] M. C. de Vries P. Haynes P. Corwine M. Trustworthy computing. Technical report, Microsoft Corp., 2002. 12

[43] W. Denson. The history of reliability prediction. *IEEE Trans. Reliability.*, 47(3):321–328, 1998. 10

[44] S. L. Dockstader, N. S. Imennov, and A. M. Tekalp. Markov-based failure prediction for human motion analysis. In *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ICCV '03, pages 1283–, Washington, DC, USA, 2003. IEEE Computer Society. 33

[45] B. G. H. A. K.-L. W. P. S. Y. M. Doo. Spade: The system s declarative stream processing engine. In *Proceedings of ACM SIGMOD inter-*

national conference on Management of data, Vancouver, BC, Canada, June 9–12 2008. 36, 41

[46] P. H. dos Santos Teixeira, R. Clemente, R. A. Kaiser, and D. A. V. Jr. Holmes: an event-driven solution to monitor data centers through continuous queries and machine learning. In *DEBS*, pages 216–221, 2010. 46

[47] T. Dumitraş and P. Narasimhan. P.: Fault-tolerant middleware and the magical 1. In *In: ACM/IEEE/IFIP Middleware Conference*, pages 431–441, 2005. 18

[48] J. A. Duraes and H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Softw. Eng.*, 32(11):849–867, Nov. 2006. 78

[49] EC482. Commission regulation (ec) no 482/2008. *Official Journal of the European Union*, pages 5–9, 2008. 15

[50] S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998. 33

[51] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002. 10, 12

[52] ESARR6. *ESARR 6. EUROCONTROL Safety Regulatory Requirement. Software in ATM Systems.* European Organisation for the Safety of Air Navigation, 2.0 edition, 2010. 15

[53] O. Etzion and P. Niblett. *Event Processing in Action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. 34, 35, 36, 38, 39

[54] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. 13

[55] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997. 10.1023/A:1007465528199. 31

[56] S. Fu and C. zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SCÕ07)*, 2007. 23

[57] M. A. Garcia, A. P. C. da Silva, and M. Meo. Using hidden markov chains for modeling p2p-tv traffic. In *GLOBECOM*, pages 1–6, 2010. 33

[58] X. Gu, S. Papadimitrioul, P. S. Yu, and S. P. Chang. Online failure forecast for fault-tolerant data stream processing. In *Proceedings of IEEE 24th International Conference on Data Engineering (ICDE 2008)*, pages 1388 – 1390, 2008. 24

[59] X. Gu and H. Wang. Online anomaly prediction for robust cluster systems. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1000–1011, Washington, DC, USA, 2009. IEEE Computer Society. iii, 24, 26, 63

[60] G. A. Hoffmann, F. Salfner, and M. Malek. Advanced Failure Prediction in Complex Software Systems. Technical Report 172, Berlin, Germany, 2004. 28

[61] C. Hood and C. Ji. Proactive network-fault detection [telecommunications]. *In IEEE Transactions on Reliability*, 46(3):333 –341, september 1997. 45, 58

[62] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology. 2001. 12

[63] Y. Huang, N. Feamster, A. Lakhina, and J. (jim Xu. Diagnosing network disruptions with network-wide analysis. In *In Sigmetrics*, pages 61–72, 2007. 38

[64] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. *Fault-Tolerant Computing, International Symposium on*, 0:0381, 1995. 10, 11

[65] A. Immonen and E. Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7:49–65, 2008. 11

[66] Z. Jelinski and P. Moranda. *Statistical computer performance evaluation.* Freiberger, W. (ed.), Academic Press, 1972. 23

[67] R. Kapadia, G. Stanley, and M. Walker. Real world model-based fault management. In *18th International Workshop on the Principles of Diagnosis Nashville TN*, 2007. 8

[68] R. Khanna and H. Liu. Control theoretic approach to intrusion detection using a distributed hidden markov model. *Wireless Commun.*, 15(4):24–33, Aug. 2008. 33

[69] A. Koski. Modelling ecg signals with hidden markov models. *Artif. Intell. Med.*, 8(5):453–471, Oct. 1996. 33

[70] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden markov models in computational biology: applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, 1994. 33

[71] J. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. *Proc. of the 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15)*, June 1985. 4

[72] J.-C. Laprie and B. Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr. iii, 4, 5, 6, 7, 58

[73] Z. L. Li Yu, Ziming Zheng and S. Coghlan. Practical Online Failure Prediction for Blue Gene/P: Period-based vs Event-driven. In *Proceeding of 41st IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2011)*, pages 259 – 264, 2011. 28

[74] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 425–434, Washington, DC, USA, 2006. IEEE Computer Society. 2

[75] Liebert. *Regulatory Compliance and Critical System Protection*. Liebert Corporation, 2005. 15

[76] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 35, 58

[77] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, 2002. 31, 58

[78] J. Musa, A. Iannino, and K. Okumoto. *Software reliability: measurement, prediction, application*. Software engineering series. McGraw-Hill, 1990. 10

[79] R. Natella, D. Cotroneo, J. Durães, and H. Madeira. Representativeness analysis of injected software faults in complex software. In *DSN*, pages 437–446, 2010. 78

[80] OMG. Corba component model (ccm), omg specification, formal/2011-11-03, part 3 - components, CCM. http://www.omg.org/spec/CORBA/3.2/Components/PDF. 75

[81] OMG. Fault tolerant corba (ft), omg specification, formal/2010-05-07 , v1.0, FT-CORBA. http://www.omg.org/spec/FT/1.0/PDF. 20, 73

[82] OMG. Lightweight load balancing service (ltload), omg specification, formal/2010-02-04, v1.0, LTLOAD. http://www.omg.org/spec/LtLOAD/1.0/PDF. 76

[83] OMG. Lightweight fault tolerance for distributed rt systems (lwft), ptc/2011-06-05, beta 2, LWFT. http://www.omg.org/spec/LWFT/1.0/Beta2/PDF. 75

[84] B. Parhami. From defects to failures: a view of dependable computing. *SIGARCH Comput. Archit. News*, 16(4):157–168, 1988. 6

[85] D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 11

[86] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCS Workshops*, pages 611–618, 2002. 38

[87] M. Proctor, M. Neale, B. McWhirter, K. Verlaenen, E. Tirelli, A. Bagerman, M. Frandsen, F. Meyer, G. D. Smet, T. Rikkola, S. Williams, and B. Truit. JBoss Drools Fusion. http://www.jboss.org/drools/drools-fusion.html, 2010. 36, 41

[88] Z. Z. Qiang Guan and S. Fu. Proactive failure management by integrated unsupervised and semi-supervised learning for dependable cloud systems. *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 83 – 90, 2011. 27

[89] L. Rabiner and B. Juang. An introduction to hidden markov models. *ASSP Magazine, IEEE*, 3(1):4 – 16, jan 1986. 33, 48, 50, 52, 58, 68

[90] L. Rabiner and B. Juang. Hidden markov models for speech recognition. *Technometrics*, 33(3):251–272, Aug. 1991. 54

[91] B. Randell. System structure for software fault tolerance. *IEEE Trans. Software Eng.*, 1(2):221–232, 1975. 11

[92] M. K. R.K. Iyer, Z. Kalbarczyk. Measurement-Based Analysis of Networked System Availability. *Performance Evaluation Origins and Directions*, 2000. 8

[93] G. C. E. D. G. L. G. L. L. M. Roberto Baldoni, Vita Bortnikov. *Collaborative Financial Infrastructure Protection: Tools, Abstractions and Middleware, chapter 4: CoMiFin Architecture and Semantic Rooms.* Springer-Verlag, Berlin, Germany, 2011. 38

[94] B. Rood, J. P. Walters, V. Chaudhary, and M. J. Lewis. Failure prediction and scalable checkpointing for reliable large-scale grid computing. In *The 16th IEEE International Symposium on High Performance Distributed Computing, Monterey, CA, June 2007*, 2007. 12, 28

[95] D. S. R.S. Swarz. *Reliable Computer Systems (3rd ed.): Design and Evaluation.* A.K. Peters, 1998. 6

[96] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(3):660–674, 1991. 32

[97] F. Salfner. *Event-based Failure Prediction: An Extended Hidden Markov Model Approach.* PhD thesis, Department of Computer Science, Humboldt-Universität zu Berlin, Germany, 2008. iii, 10, 11, 21, 22, 28, 29, 52

[98] R. S. Siewiorek, D. P. Swarz. *Reliable Computer Systems*, volume 2nd edition. Digital Press, Bedford, MA, 1992. 11

[99] O. Sigaud and S. W. Wilson. Learning classifier systems: a survey. *Soft Comput.*, 11(11):1065–1078, 2007. 31

[100] W. Simpson and J. Sheppard. *System test and diagnosis.* Kluwer Academic, 1994. 9

[101] G. M. Stanley and R. Vaidhyanathan. A generic fault propagation modeling approach to on-line diagnosis and event correlation. In *3rd IFAC Workshop on On-line Fault Detection and Supervision in the Chemical Process Industries,*, 1998. 9

[102] Streambase Systems. *StreamBase product documentation*, 2010. Release 6.3. 46

[103] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems, 1991. 11

[104] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*,

PODC '10, pages 173–182, New York, NY, USA, 2010. ACM. iii, 26, 27, 63

[105] M. Thottan and C. Ji. Properties of network faults. In *Proceeding of IEEE/IFIP Network Operation and Management Symposium (NOMS 2000)*, pages 941–942, 2000. 45, 58

[106] K. S. Trivedi and K. Vaidyanathan. Software aging and rejuvenation. In *Wiley Encyclopedia of Computer Science and Engineering*. 2008. 10, 11

[107] Y. Wang, Y. Huang, K.-P. Vo, P. Chung, and C. Kintala. Checkpointing and its Applications. *Proc. of the 25th IEEE Fault-Tolerant Computing Symposium (FTCS-25)*, June 1995. 12

[108] S. White, A. Alves, and D. Rorke. Weblogic event server: a lightweight, modular application server for event processing. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 193–200, New York, NY, USA, 2008. ACM. 46

[109] A. W. Williams, S. M. Pertet, and P. Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Los Alamitos, CA, USA, 2007. iii, 24, 25, 45, 58

[110] J. Yamato, J. Ohya, and K. Ishii. Recognizing human action in time-sequential images using hidden Markov model. *Proceedings 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 379–385, 1992. 33

[111] S.-Z. Yu and H. Kobayashi. An efficient forward-backward algorithm for an explicit-duration hidden markov model. *IEEE Signal Processing Letters*, 10(1):11–14, 2003. 52

[112] S.-Z. Yu and H. Kobayashi. Practical implementation of an efficient forward-backward algorithm for an explicit-duration hidden markov model. *Trans. Sig. Proc.*, 54(5):1947–1951, Oct. 2006. 52

[113] G. P. Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, 30(4):451–462, Nov. 2000. 31, 32

[114] X. J. Zhang, H. Andrade, B. Gedik, R. King, J. Morar, S. Nathan, Y. Park, R. Pavuluri, E. Pring, R. Schnier, P. Selo, M. Spicer, V. Uhlig, and C. Venkatramani. Implementing a high-volume, low-latency market

data processing system on commodity hardware using ibm middleware. In *Proceedings of the 2nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 7:1–7:8, New York, NY, USA, 2009. ACM. 38

[115] Z. Zhang and S. Fu. Failure prediction for autonomic management of networked computer systems with availability assurance. In *Symposium on Parallel and Distributed Processing*, pages 1–8, 2010. 23