DOTS-LCCI

Dependable Off-The-Shelf based Middleware Systems for Large-scale Complex Critical Infrastructures

Program of Relevant National Interest (PRIN)
Research Project Nr. 2008LWRBHF

Project funded by the Italian Ministry for University and Research (MIUR)

Deliverable no.:	2.1
Deliverable Title:	Diagnosis and Reconfiguration: State-of-the-art
Organisation name of lead Contractor for this Deliverable:	UNIPARTHENOPE
Author(s):	A. Bovenzi, R. Pietrantuono, S. Russo, D. Cotroneo, C. Esposito, A Pecchia, R. Natella, S. D'Antonio, F. Oliviero, M. Ficco, I. Elia, V. Formicola, L. Coppolino, L. Romano, M. Colajanni, R. Lancellotti, R. Baldoni, L. Querzoni, R. Beraldi, M. Platania, A. Bondavalli, F. Brancati,
Participant(s):	All
Work package contributing to the deliverable:	2
Task contributing to the deliverable:	2.1
Version:	1

Total Number of Pages	82

Project DOTS-LCCI

Dependable Off-The-Shelf based Middleware Systems for Large-scale Complex Critical Infrastructures

DELIVERABLE D2.1

DIAGNOSIS AND RECONFIGURATION: STATE-OF-THE-ART

Table of Versions

Version	Date	Contributors	Version Description	Reviewers	Date of Approval
0.1	17-01-2011	Antonio Bovenzi, Roberto Pietrantuono	Document structure, Content on Diagnosis	Domenico Cotroneo	-
0.2	14-02-2011	UniParthenope and UniNa	Update Doc. Structure		
0.8	08-04-2011	Antonio Bovenzi	Update Fault and Error Model; Update Diagnosis Section		
0.9	18-05-2011	Francesco Oliviero	ToC Redefinition; SCADA System Vulnerability Section; Recovery and Reconfiguration Section; Network-level Approaches.	Salvatore D'Antonio	
0.92	24-05-2011	Antonio Pecchia, Roberto Natella	Insertion contribution of UniNA in section 1.7.2 (OS monitoring and reconfiguration); References Sort	Christian Esposito	-

0.93	27-05-2011	Francesco Oliviero	Integration of section 6 (Recovery and Reconfiguration)	Salvatore D'Antonio	-
0.94	03-06-2011	Francesco Oliviero	ToC upgrade	Salvatore D'Antonio	
0.95	14-06-2011	Christian Esposito, Marco Platania	Contributions to Section 4.2 (Middleware Detection) and Section 6.2 (Middleware Recovery and Reconfiguration)		
0.96	15-06-2011	Antonio Bovenzi	Contributions to Section 3.1 (Unintentional Failures)		
0.97	15-06-2011	Francesco Oliviero	Contributions to Section 3.2 (Intentional Failures) and Section 4.3 (Network Detection)		
0.98	08-08-2011	Andrea Bondavalli	Document Review		
0.99	01-09-2011	Francesco Oliviero	Reference labels Review		
1	06-09-2011	Antonio Bovenzi, Christian Esposito, Stefano Russo	Final editing		

1 EXECUTIVE SUMMARY

The DOTS-LCCI research project aims to define novel middleware technologies, models, and methods to assure and assess the resiliency level of current and future OTS-based LCCI, to diagnose faults in real time, and to tolerate them by means of dynamic reconfiguration. Assuring the resiliency level of LCCI is crucial to reduce, with known probabilities, the occurrence of catastrophic failures, and consequently, to adopt proper diagnosis and reconfiguration strategies.

Project efforts will progress according to three main directions:

- Distributed architectures for LCCIs, their components (OTS and legacy), and their resiliency requirements will be studied, in order to define algorithms and middleware architectures for improving dependability attributes of future LCCIs;
- Strategies for on-line diagnosis and reconfiguration will be studied and defined, specifically tailored for OTS-based LCCIs, according to the resiliency assurance requirements;
- Tools and techniques for modeling and evaluating LCCIs will be devised.

This document is the Deliverable D2.1 of DOTS-LCCI Project.

2 BASIC CONCEPTS AND TERMINOLOGY

Several vocabulary definitions answer the question of what is diagnosis. From the ancient Greek "διαγιγωσκειν", which stands for to discern, to distinguish, they share the general meaning of identifying the nature and cause of some phenomenon.

Essential for a smooth and efficient operation of modern systems is the ability to analyze quantitative aspects related to performance and security, availability and reliability features, which show and convince us of suitability of systems. In the field of dependable systems, diagnosis consists in the interpretation of information collected from the system in order to obtain an indication of the activation of faults, their nature and their location [Avizienis 2004]. For these reasons Diagnosis activities are crucial to build Dependable system.

In the rest of the chapter will carry out a systematic treatment of dependability, which includes a description of basic concepts such as fault, error, failure, and dependability attributes such as reliability, availability, security.

2.1 Dependability basics

A widely accepted notion of dependability was formulated by Aviezenis et al. as follow: "*the ability to deliver service that can be justifiably be trusted*", thus putting focus on the definition of service. To provide also the criterion to decide if a system is dependable, the dependability of a system is also defined as "*the ability to avoid service failures that are more frequent and more severe than acceptable* ", where ***service*** is defined as the behaviour of a system as perceived by its users, and **failure** is defined as the event that occurs when the delivered service deviates from the specified correct service. The term ***system*** refers to an entity that interacts with other entities, i.e., other systems, including hardware, software, users, and the physical world with its natural phenomena. These other systems are the ***environment*** of the given system.

What the system is intended to do is called ***function*** and it is described by the functional specification in terms of functionality and performance. What the system does to implement its function is the ***behaviour*** and it is described by a sequence of states. The total state of a given system is the set of the following states: computation, communication, stored information, interconnection, and physical condition. From a structural viewpoint, a system is composed of a set of ***components*** bound together in order to interact, where each component is another system, etc. The recursion stops to the so-called ***atomic component***: any further internal structure cannot be discerned, or is not of interest and can be ignored. Consequently, the total state of a system is the set of the states of its atomic components

The dependability concepts are used in all stages of the life cycle of a computer system, from the requirement stage to the operational stage. Rather than being a monolithic concept, dependability may be regarded as an umbrella term. In particular, the attributes through which dependability can be expressed and quantified include:

- Availability: readiness for correct service;
- Reliability: continuity of correct service;
- Safety: absence of catastrophic consequences on the user(s) and the environment;
- Integrity: absence of improper system alterations;

- Maintainability: ability to undergo modifications and repairs.

When addressing security, an additional attribute deserves great importance, namely ***confidentiality***, i.e., the absence of unauthorized disclosure of information. Security is a composite of the attributes of confidentiality, integrity and availability, requiring the concurrent existence of a) availability for authorized actions only, b) confidentiality, and c) integrity with "improper" meaning "unauthorized".

Several causes may lead a system to ***failure*** causing the transition from a correct service to an incorrect one. The period of delivery of incorrect service is a service ***outage***. The deviation from correct service may assume different forms that are called service ***failure modes.***

The service failure modes characterize incorrect service according to four viewpoints [Avizienis 2004]:

- domain, which encompass *timing failure* (i.e., timing requirements are violated) *content failure* (i.e., the service content deviates from implementing the system function) and both;
- The detectability, which address the signaling of service failure, i.e., a failure may be *signaled,* when a warning signal is provided, or *unsignaled*;
- The consistency, which let us distinguish *i) consistent failure*, i.e., the incorrect service is perceived identically by all users, and *ii) inconsistent failure*, i.e., users perceive different behaviors (faulty or correct) of the system;
- The consequences on the environment, which address the impact of failures (e.g., *Minor failure* or C*atastrophic failure*).

With respect to the **consequences** point of view, failure modes are ranked according to failure severities, which are determined following specifics criteria (e.g., for availability is the outage duration; for safety is the possibility of the loss of human lives; for integrity, the extent of the corruption of data and the ability to recover from these corruptions).

Hardware faults and design faults are just an example of the possible sources of failures. A service may fail either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. 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***.

An ***error*** can be regarded as the part of a system's state that may lead to a failure. In other words, a failure occurs when the error causes the delivered service to deviate from correct service. A more detailed discussion about errors will be provided in the Error Detection subsection.

The adjudged or hypothesized cause of an error is called a ***fault***. Faults can be characterized from various point of view which regard:

- phase of occurrence: development or use phase faults;
- dimension: hardware or software faults;
- system boundaries: internal or external faults;

- persistence: permanent or transient faults;
- phenomenological cause: natural or human-made faults;
- objective: malicious or non malicious faults;
- intent: deliberate or non deliberate faults;
- capability: accidental or incompetence faults.

However all faults can be grouped in three major (and overlapping) classes:

- Development: it includes all fault classes occurring during development;
- Physical: it includes all fault classes that affect hardware;
- Interaction: it includes all external faults.

The great research effort striven in dependable systems has provided good results with respect to hardware-related faults, while lot of research is still needed with respect to software faults. Software faults are neither easy to diagnose nor to recover, and represent the major matter of concern for the dependability of complex systems [Sullivan 1991].

A crucial concern for software engineers is to understand how a fault, of any type, may be activated and how it can be reproduced. In [Gray 1986] Gray discussed Software fault reproducibility for the first time. In this work, Gray distinguished faults whose activation is easily reproducible (e.g., through a debugger), i.e., ***solid or hard faults***, from faults whose activation is not systematically reproducible, i.e., ***elusive*** or ***soft faults***.

Solid faults manifest consistently under a well-defined set of conditions and that can easily be isolated, since their activations and error propagations are relatively simple. ***Soft faults***, instead, are intricate enough that their activation conditions depend on complex combinations of the internal state and the external environment. The conditions that activate the fault occur very rarely and can be very difficult to reproduce.

In Gray's paper, the first class of faults (i.e., solid) were named "Bohrbugs", recalling the physicist Niels Bohr and his rather simple atomic model: "Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques, and hence boring". Thus a Bohrbug does not disappear or alter its characteristics when it is activated. These include the easiest bugs to fix (where the nature of the problem is obvious), but also bugs that are hard to find and to fix, which remain in the software during the operational phase. A software system with a Bohrbug is analogous to a faulty deterministic finite state machine.

Soft faults were instead defined as those faults for which "if the program state is reinitialized, and the failed operation is retried, the operation will not fail a second time". Such kinds of faults were named "Heisenbugs", referring to the physicist Werner Heisenberg and his Uncertainty Principle. Based on Gray's paper, researchers have often equated Heisenbugs with soft faults. However, when Bruce Lindsay originally coined the term in the 1960s (while working with Jim Gray), he had a more narrow definition in mind. In his definition, Heisenbugs were envisioned as "bugs in which clearly the system behaviour is incorrect, and when you try to look to see why it's incorrect, the problem goes away". In this sense the term recalls the uncertainty principle, in that the measurement process (in this case the fault probing) disturbs the

phenomenon to be measured (the fault). A software system with a Heisenbug is analogous to a faulty non-deterministic finite state machine. One common example is a bug caused by a race condition. However, between Bohrbugs and Heisenbugs behaviour there is another class of bugs that has been identified later, which cannot be classified in neither of the two categories. They are, like Heisenbugs, very hard to reproduce but their activation is just apparently non-deterministic, i.e., they are deterministically activated by a particular exact condition (like Bohrbugs), but detecting this condition is so difficult that the bug can be considered as non-deterministic.

In scientific literature these software defects are named Mandelbugs (which name derives from the name of fractal innovator Benoit Mandelbrot). According to this classification,

- *A Heisenbug is a bug that disappears or alters its characteristics when it is probed.*
- *A Mandelbug is a bug whose causes are so complex that its behaviour appears to be chaotic (but they are deterministic).*

However, currently there is no agreement in the literature and the term "Heisenbug" is used inconsistently: some authors accepts this classification; others use the term Mandel-bugs as a synonym for Heisenbugs, since they claim that there is no way to distinguish a bug whose behavior appears chaotic and a bug whose behaviour is actually chaotic. On the other hand, Trivedi et al. claim in [Trivedi 2007] that Heisenbugs are actually a kind of Mandelbugs. He identified two main sources of complexity characterizing the occurrence of a Mandelbugs. The first one is that there may be a long delay between the fault activation and the final failure occurrence (e.g., because several error state are traversed before the failure or because the fault progressively accrues an abnormal condition until the system fails). This usually happens with complex software systems employing one or more OTS items. The second source of complexity is due to the system-internal environment: fault activation and/or error propagation depend on the combination of conditions occurring inside the application and conditions that accrue within the system-internal environment (e.g., a fault causing failures due to side-effects of other applications in the same system, or a race condition caused by inadequate synchronization in multi- threaded software). According to this view, "since the system-internal environment induces the change in the behavior -regarding to a fault-, Heisenbugs are actually a type of Mandelbug" [Trivedi 2007]. Referring to this classification:

- *Mandelbugs include those faults which activation is chaotic or non-deterministic (including Heisenbugs);*
- *Heisenbugs are a special case of Mandelbugs in which system-internal environment influences fault's behaviour in a specific application (hence causing the bug to "disappear" when probed).*

In the following we will refer to this classification. Due to their different nature, to deal with Bohr or Mandel-bugs implies distinct techniques to be adopted. Indeed Bohrbugs, due to their deterministic activation conditions are more prone to be detected and fixed during the software design and verification phase (however the largest slice of fault activated during the operational life of systems still belongs to this class!). For instance,

structured design, design review, formal analysis, unit testing, inspection, integration, system and acceptance testing are techniques that are employed in the development phase and that "usually" fix the most of Bohrbugs.

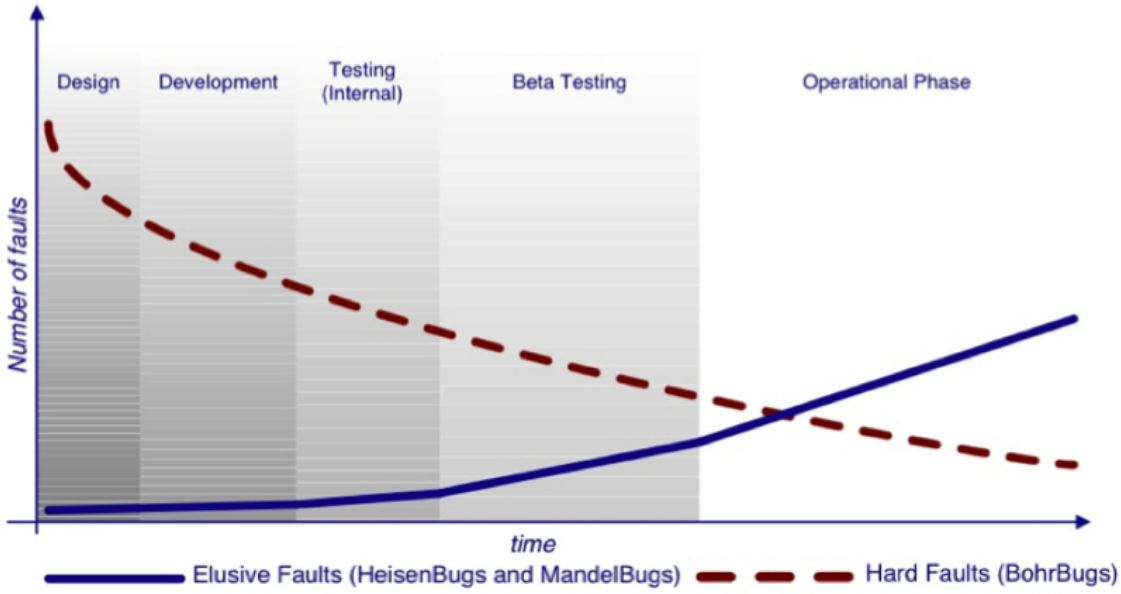

Figure 1 - Evolution of reproducible and non-reproducible faults over software life cycle

For these reasons, the number of detected Bohrbugs decreases over time (as depicted in Figure 1), becoming negligible after a long period of production. On the other hand, the number of Mandelbugs increases over time: during the development phase this number is very low, since the system under test runs always in the same environment and intricate activation conditions rarely occur; during the beta testing, when the system is delivered out of the production environment, a consistent number of Mandelbugs are reported. In that stage, the system runs in several environments under workloads very different from the ones applied in the testing phase. This number further increases once the system is brought to the operational phase.

Another fundamental milestone by Chillarage and Sullivan in [Sullivan 1991] proposes a classification of triggering condition of software faults. A *Defect trigger* is a condition that allows a fault to be activated. Even if extensive testing has been performed, a series of circumstances may allow a fault to surface after that a software system has been deployed. The most used defect trigger categories are:

- Boundary Conditions: software faults were triggered when the systems ran in particularly critical conditions (e.g., low memory);
- Bug Fix: the fault surfaced only after another defect was corrected: this happens either because the first defects was covering the second one, or because the fix was not successful, introducing a new bug;
- Recovery: the fault surfaced after the system recovered from a previous failure.
- Exception Handling: the fault surfaced after an unforeseen exception-handling path was executed;

- Timing: the fault emerged when particular timing conditions occurred;
- Workload: the fault surfaced only when particular workload condition occurred (e.g., only after the number of concurrent requests to serve was higher than a given threshold).

Another category of bug, which recently is gaining attention, is the so-called "Aging-related bugs". Often, software systems running continuously for a long time tend to show a degraded performance and an increased failure occurrence rate. This phenomenon is usually called Software Aging [Kintala 1995]. These bugs are very difficult to detect during the development phase, because a failure only occurs after the accumulation of many_errors. Therefore there is a large interval of time (many hours, often also weeks) between fault activation and failure occurrence. Some examples of these bugs are: memory leaks (memory allocated portions of a process but no longer used or usable), not terminated threads, poor management of shared resources, data corruption, unreleased file-locks, the accumulation of numerical errors (e.g., round- off and truncation) and disk fragmentation.

Also in this case, there is no complete agreement on the classification of these bugs: some authors relegate Aging-related bugs only in the Mandelbugs category, while others view these bugs as an intersection between Mandelbugs and Bohrbugs classes. Indeed, Aging-related bugs can be either deterministic (e.g., a missing delete statement) or non-deterministic (e.g., a fault which is dependent on the message arrival order). However both kinds of bug require a long time to manifest. Hence, also deterministic Aging-related bugs will very likely manifest at the operational time (like Mandelbugs), being very difficult to note their effect at testing time. For this reason, in accordance with the adopted classification for Bohrbugs-Mandelbugs, we adopt this second solution (Aging bugs as subset of Mandelbugs), suggested by [Trivedi 2007], for placing aging-related bugs in the classification scheme.

The most adopted technique to face this problem is known as Software Rejuvenation. The purpose of these techniques is to restore a "clean state" of the system by surgically releasing OS resources and removing error accumulation. Some common examples are garbage collectors and process recycling (e.g., in Microsoft IIS 5.0). In extreme cases these techniques result in partial or total restarting of the system: restarting the application (also known as *micro-reboot*), restarting the node or activating a standby spare.

There are several examples in the literature reporting real system failures due to software aging faults: web-servers [Trivedi 2006], middleware [Cotroneo 2010], spacecraft systems [Marshall 1992], and even military systems [Tai 1997] with severe consequences such as loss of money and human lives. For these reasons, it is a crucial task to detect and diagnose them.

2.2 Fault Tolerance basics

Over the course of the past 50 years many means have been developed to attain the various attributes of dependability. The development of dependable systems requires the combined use of four types of techniques:

- prevention of failure, to prevent the occurrence or introduction of faults in the system;

- fault tolerance, to deliver a correct service even in the presence failure;
- fault removal, to reduce the number or severity 'fault;
- fault forecasting, to estimate the number of faults in the system, their impact in the future, or their likely consequences.

Among these "fault tolerance is determined by the system requirements and the system safety assessment process" and it consists of using proper techniques, during the operational life of the system, to allow the continued delivery of services at an acceptable dependability level, after a fault becomes active [Wilfredo 2000].

The main phases that are typically executed to achieve fault tolerance can be schematized as follows:

- ***Error Detection****:* its goal is to generate an error warning in order that an error does not remain latent. It aims to detect errors prior them to degenerate into failures for other system components, or for the overall system. It can be carried out online (i.e., *concurrent detection*), i.e., while the system delivers the service, or offline (i.e., *pre-emptive detection*), where the system is scanned while the service provisioning is suspended.

- ***System Recovery****:* its goal is to lead the system in an error free and fault free state. Recovery consists of:

    - *Error Handling,* which aims at removing the error condition, by bringing the system in an error-free state (i.e., through Rollback, Rollforward and Compensation)

    - *Fault Handling,* which aim is to prevent fault re-activation, through various stages, i.e., **Diagnosis**, Isolation, **Reconfiguration** and Re-initialization.

Typically, these phases may be followed by a corrective maintenance step, in order to remove the diagnosed faults. Note also that compensation does not necessarily require error detection; however, this can lead to a progressive loss of redundancy if not.

A widely used strategy for the implementation of fault tolerance is redundancy, either temporal or spatial, which aim is to mask faults activation. On one hand, temporal redundancy attempts to re-establish proper operation by bringing the system in an error-free state and by repeating the operation which caused the failure. On the other hand 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 fail independently. This can be achieved through design diversity [Avizienis 1984].

Both temporal and spatial redundancy require 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. Such a recovery can assume the form of rollback (i.e., 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), rollforward (i.e., 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). Fault masking, or simply masking, results from the systematic usage of compensation. Such masking will prevent completely failures from occurring.

The measure of effectiveness of any given fault tolerance technique is called ***coverage***, i.e., the percentage of the total number of failures that are successfully recovered by fault tolerance means.

The key for achieving (software) fault tolerance is the ability to accurately detect, diagnose, and recover from faults during system operation. Many works in the literature focused on these three topics; the most relevant ones are surveyed in the rest of the document.

3 FAULT AND ERROR MODEL

An understanding if faults that may affect a system or a class of systems (depending on the technology and on the environment) is highly needed. The definition of a fault taxonomy, from which fault models for individual systems or for infrastructures can be derived, must be kept in line with evolution of technology. The two main approaches (not mutually exclusive) to understand faults and to build a taxonomy or a model are:

- Field Failure Data Analysis (FFDA): collecting and analyzing data from logs related to errors and failures in systems already in operation. This information is used to determine the causes of errors and failures that actually happened and their occurrence rates. However, the FFDA requires a very long time, because of the low frequency of occurrence of such events in real systems.
- Failure Modes and Effects Analysis (FMEA): A systematic analysis of the system, decomposing into elementary parts and identifying for each (i) the possible failure modes and (ii) the possible causes, effects and countermeasures each failure mode. The failure mode of an elementary component becomes a fault in the next abstraction layer composing the system or the infrastructure. The identification does not take place on an experimental basis, but assuming for each item, the possible events and scenarios that can cause a failure. This approach is not suitable to accurately estimate the probability of occurrence and the level of severity of a fault.

Among the existing fault types, the most frequently considered are hardware faults and software ones.

In particular Hardware faults types are better characterized, having been the electronic circuits the main cause of failures until the 80 [Arlat 1990]. This type of component is subject to failure due to deterioration, manufacturing defects and external electromagnetic interferences. Several possible failure modes have been identified during the last 50 years. For instance, CMOS transistors are subject to the occurrence of short-circuits and open, or the alpha rays can alter the state of capacitors used in memory cells. These faults can cause both permanent failures, such as a fixed output on a single logical value (stuck-at), and transient failures, such as the reversal of a memory cell (bit-flip). However, due to the technological evolution the types and frequency (both relative and absolute) of hardware faults is continuously changing whereby transients are getting more and more frequent.

Software related faults represent a second class of interest. Unlike hardware faults, they are not related to deterioration or natural phenomena, but rather to human mistakes in the phases of analysis, design and implementation of software. They are considered as persistent faults because they are inherent in the code. However, as mentioned in the previous section, the conditions for the activation of software faults can be very difficult to reproduce (elusive faults) and can be profitably modeled as transient.

Others fault classes commonly used are related to incompetence faults (e.g., errors of human operators in systems administration), or the faults that lead to security vulnerabilities (e.g., unauthorized access). Software faults that result in accidental failures are discussed in Section 3.1 while faults exploited for malicious attack, also known as intentional failures, are discussed in section 3.2.

3.1 Unintentional Faults

The fault models used in designing and analyzing dependable distributed systems typically make simplifying assumptions about the nature of faults in the system. Often, the fault tolerance algorithms used treat all faults in the same way, ignoring their nature (intentional or non intentional). Some overly optimistic single fault-type models assume a fixed number of benign permanent faults (and perfect fault coverage). On the other side, other models assume all faults to be malicious, even though only a small portion of the faults is malicious in reality. These more pessimistic models, called Byzantine, assume all faults to be arbitrary.

By distinguishing between different fault types and by considering differing probabilities of occurrence for each type, one can develop different system models to design algorithms capable of handling various fault types [Walter 2003]. In the following we discuss about fault models and the implications of their assumptions.

To simplify our discussion we adopt a generalized system model: A system consists of a set of nodes, which communicate by exchanging information (messages) across links, with a bounded delay assumed for message generation, delivery and processing. Terms "system", "nodes", "links", and "messages" are used in an abstract sense, because some type of information exchange often exists between a node and its components, or between a process and its sub-processes. We point out that the examined fault models are applicable to a several synchronous system models.

Much effort has been expended and lots of taxonomies have been proposed to capture the important attributes of faults. In the previous section we introduce the classification made by Avizienis [Avizienis 2004]. Avizienis et al. classify faults according to the attributes of capability (accidental or incompetence); phenomenological cause (natural or human-made); system boundaries (internal or external); phase of occurrence (design or operational); persistence (permanent or transient); objective (malicious or not); intent (deliberate or not); dimension (hardware or software). The Authors also give a convenient taxonomy of errors, describing them in terms of the elementary service failures that they can cause. Using the terminology of the previous section we can distinguish the following: content vs timing error, detected vs. undetected errors, consistent vs. inconsistent errors, minor vs. catastrophic errors. In the field of error control codes, content errors are further classified according to the damage pattern: single, double, triple, byte, burst, erasure, arithmetic, track, etc., errors.

The taxonomy provided by Avizienis et al. is important and commonly used in the literature. When dealing with a specific system or infrastructure the taxonomy is normally used a the starting point to understand which are the faults which can really occur in the system at hand, which ones have to be removed during the design and testing process, which have to be avoided and how, and finally those that have to be detected and tolerated.

Being a taxonomy, it classifies faults in disjoint classes or groups; moreover so a fault in a specific system can belong to exactly one class. However, the same type of fault can be classified differently in two different systems. Such fault classification ambiguity can arise because, for example, a fault lasting a few seconds can be classified as permanent in a system and as temporary in another. Differences can also occur regarding the

location of the fault in the systems, how long the fault is active relative to the time scale (mission time) for each system, the workload, the technology used to construct each system, the assumed system environment, and other related factors.

Different approaches based on the use of failure semantics are the fault models proposed by [Bondavalli 1990], [Powell 1992][Cristian 1991].

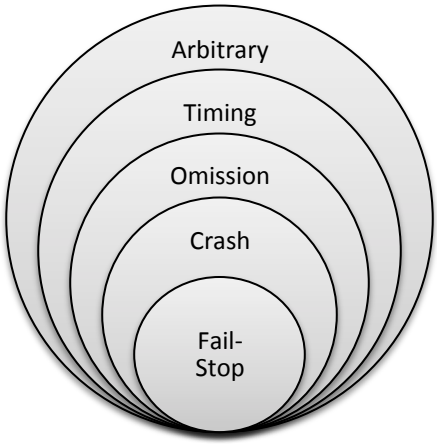

Figure 2 - Onion Failure modes.

Bondavalli and Simoncini [Bondavalli 1990], developed a failure classification having in mind issues related to error detection so their work aims at distinguishing faults based on the capability or impossibility for an external observer to perceive and detect the failures. David Powell [Powell 1992] concentrated on a hierarchy of assumptions that can be made when considering a fault model for a specific system and on the coverage of such assumptions and its evaluation. As an example the advantage highlighted by Powell of the byzantine fault model does not lie on its capability to represent reality but more on the easiness of evaluating its coverage (1 by definition). The three works mentioned are essentially equivalent in the classes of faults considered and in the fact that the different classes are hierarchically organized either in a total order or with some oriented graph (Bondavalli and Powell consider also value failures, neglected by Cristian). **Error! Reference source not found.**, taken from [Cristian 1991], shows the relationships among the five potential failure modes, in a sort of onion model, given in increasing order of severity: fail-stop, crash, omission, timing and arbitrary. Under the fail-stop assumption, a component fails by ceasing execution; no incorrect state transition occurs, and all other good components can detect this failure. A crash or fail-silent mode is identical to fail-stop, except that detection by all good components is not guaranteed. An omission mode occurs when a component fails to respond to an input, perhaps undetectably to some components. A functionally correct, but untimely response corresponds to a timing fault. Other behaviors are classified as *arbitrary*. The more restricted the assumed fault mode, the stronger the assumed failure semantics. Thus, weak failure semantics correspond to little restriction of behavior of a faulty component. Note that in **Error! Reference source not found.**, the containment relationship among different classes is such that limiting failure semantics of a host to, say, timing, means that the host can fail in any of the modes subsumed: fail-stop, crash,

or omission.

The onion model fails to capture the notion of a system consisting of nodes, and does not aid in ensuring that correct system operations are sustained in the presence of a node failure. The focus of onion model on abstract and high level faults limits its coverage of data, or content based, faults.

Another proposal, which considers with value faults is the Customers Fault and Error Model (CFEM) approach proposed by Walter and Suri [Walter 2003]. The CFEM model focuses on runtime fault-effects and the methods associated with tolerating them. The CFEM fault classes are thus determined by the fault-tolerance techniques (detection and masking) implemented in the system and the system topology. Under the CFEM, node faults are classified according to the ability of the system to tolerate their effects. At the highest level, the set of all faults is partitioned into tolerable and intolerable faults according to their effects. Intolerable fault-effects are faults whose effects cannot be detected, masked, or otherwise tolerated by the system. Design faults, generic faults, common mode faults, physical damage faults and other catastrophic faults that immediately render much of a system useless would be categorized as intolerable.

In some applications, It is necessary to point out that the same fault can be classified differently in different systems 8as in the Avizienis taxonomy: a system may be able to tolerate a fault while another system may fail when the same fault occurs. Tolerable fault are classified according to two perspectives: detectability (as in [Bondavalli 1990)], and consistency.

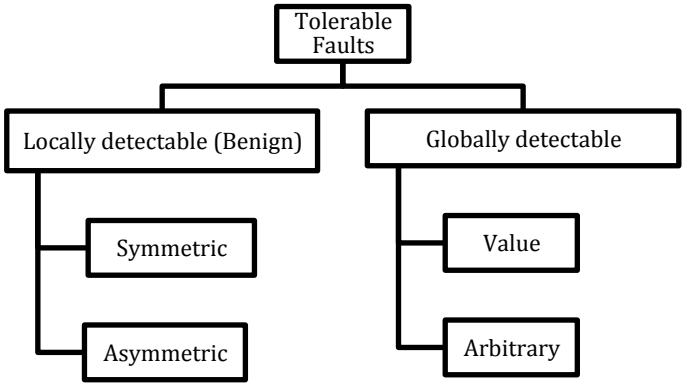

Figure 3 - CFEM tolerable faults classification.

As shown in **Error! Reference source not found.** a faulty node leading to an error detected by some non-faulty, i.e., good node is labeled benign fault. All good nodes must agree on this local judgment. Benign nodes have been further classified in *Symmetric*, i.e., all good nodes receive the same messages (consistent behavior); *Asymmetric*, i.e., all good nodes receive different messages (inconsistent behavior).

A second class of tolerable faults is the Globally Detectable faults: only by sharing or by disseminate information in order to develop a global opinion on the fault-effect. When the faulty node send the same messages to each good node they call the *value-fault* while if the faulty node has a different behavior, i.e., it sends different messages to good nodes, the fault is called Arbitrary.

Besides providing fault taxonomies and models, also being able to deal with errors, classifying them and providing error models is fundamental for achieving fault tolerance and ultimately dependability. Chillarage et al.[Chillarage 1996] provided a milestone work in defining an **error model**. They investigated failure reports pulled from RETAIN, a database system maintained by IBM, which contains world-wide information on hardware problems, software problems, bug fixes, release information, in order to generate errors emulating software faults. In particular they used reports to classify the erroneous system state of IBM OS, which are caused by software faults. Their error classification was similar to the one proposed in [Iyer 1991] thus allowing to compare error distribution of IBM operating systems and of the Tandem Guardian90. The distribution of errors was surprisingly similar adding flavor to the generality of analyzed data, thus the proposed error model can be considered appropriate to mimic representative software faults for OS. However error classification is so general that can be used also for different software systems (e.g., middleware, web server, database). Error types were not classified for all faults but only for: (i) faults related to the development, namely related to coding; (ii) faults causing an erroneous system state; and (iii) faults having impact on the user, i.e., the provided service is degraded or totally unavailable. Erroneous states are grouped into the following error categories:

- **Single address error**. An error falls into this category if the software fault caused an incorrect address word. Errors in this category were further divided into the following types of address word: control block address, storage pointer, module address, linking of data structures and register.
- **Single non-address error**. An incorrect non-address word (e.g. data word) is caused by the software fault. This error was divided into the following types: value, parameter, flag, length, lock, index and name.
- **Multiple errors**. A software fault can generate multiple errors at the same time. The errors generated can be any combination of single errors or be related to a data structure. These errors were divided into the following types: values, parameters, address and something else, flag and something else, data structure and random. It was noteworthy that they did not classify address and flag because it was not revealed.
- **Control errors**. Software faults affecting the memory content in a very subtle and non-deterministic way. These errors impact on the memory in a non-deterministic way, that is, it was not possible to identify and classify the error in any of the first three categories. Furthermore, some faults did not affect the memory at all, e.g. interaction with a user or terminal communication. These errors were divided into the following types: program management, storage management, serialization, device management, user I/O and complex. For instance, the error type storage management tells us that one module deallocates a region of memory before it has completely finished using the region.

3.2 Intentional Faults

The intentional faults are due to malicious human behaviors, committed in order to

cause damage to the system or to obtain secret information. Intentional faults affecting software usually fall in two categories: i) the *malicious logics* (or *malicious software* or *malware*) [Bishop 1991], which are set of instructions that cause violation of system security policies, ii) and the *intrusions* [Verissimo 2003], which are violation of the system security requirements by exploiting some well-known *vulnerabilities*.

Malicious logic concerns an internal malicious fault, which can be introduced in the system either during the system development or while the system is in operation. In spite of a bug, which accidentally occurs, the malicious logic are deliberately introduced in the system in order to generate a failure. These faults include:

- *virus*, which replicates itself and joins other programs when they are executed in order to compromise the system functionalities;
- *worm*, which replicates itself and propagates without the users are aware of that;
- *logic bomb*, which "sleeps" until an event occurs or a timer expires, and it can seriously damage the system;
- *zombie*, which can be activated by an attacker for remotely launching malicious activities against a specific target;
- *Trojan horse*, which is capable to perform illegal activities while it gives the impression to be legitimate.

The malicious logics can be also classified into intentionally installed vulnerabilities and attack agents. The attack agents, in particular, can be further classified based on four characteristics:

- *propagation*, it is possible to distinguish between self-propagating malwares and no-propagation malwares;
- *trigger conditions*, they represent the condition that can activate the malware;
- *target of the attack*, it represents the potential victim of the attack and it can be local or remote;
- *aim of the attack*, it concerns the objective of the attack against a particular security requirement of a system.

All the activities aiming at carrying out an unauthorized operation on a specific resource that can compromise security requirements is classified as intrusion, both external and internal to the system boundaries. Another word commonly used to define an intrusion is *attack*. Actually the two terms differ in the mean. An intrusion is just the final result of a successful attack, which represents an intentional attempt to cause a fault in a computing or communication system.

In order to perform an intrusion, an attack exploits a vulnerability, which is a weakness into the system. The vulnerability may be introduced during the development phase, or during the system operations. Furthermore, the vulnerabilities can be introduced accidentally or deliberately, with or without malicious intent. The AVI (Attack-Vulnerability-Intrusion) chain model [Verissimo 2003] is reported in Figure 4.

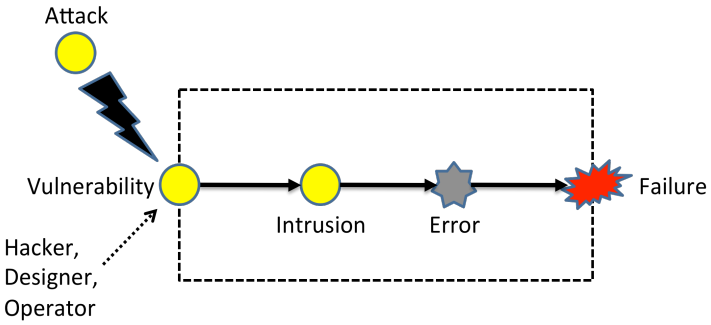

Figure 4: The AVI chain model

Based on this model the intrusions can be described in terms of both attack and vulnerability. In the following few example are reported.

1. An hacker can remotely introduce itself into a system by guessing user password. This attack exploits the vulnerability in the password selection process (common words, too short password, etc.).
2. An insider can abuse of its privileges to compromise the system. The vulnerability consists in a weakness in the configuration or the design of the system.
3. An hacker can make a network resource unavailable by a request overload (Denial of Service). Vulnerabilities are in the network protocol design and in implementation or misconfiguration of the hosts in the network.

In the following of this section some typical taxonomies of intrusions will be detailed. In particular, the main intrusions, which can affect a complex distributed system are reported. According to the classification proposed in [Mazzariello 2007], there are just three consequences that can result from a successful attack. First, otherwise defect-free information can become corrupt. Second, services that should be available can be denied, and last but not least, information can get to places it should not go. According to Cohen [Cohen 1995], each of the named events, can be considered as a disruption of information. He explicitly calls those events corruption, denial, and leakage of information.

Many taxonomies have been proposed, focusing on different aspects of the problem of attack categorization. Some are aimed at giving lists, which define all the possible attacks [Cohen 1995] or a list of possible categories. In [Cheswick 2003] attacks are classified in the following seven categories:

- **Stealing passwords** methods used to obtain other users' passwords;
- **Social engineering** talking your way into information that you should not have;
- **Bugs and backdoors** taking advantage of systems that do not meet their specifications, or replacing software with compromised versions;
- **Authentication failures** defeating of mechanisms used for authentication;
- **Protocol failures** protocols themselves are improperly designed or implemented;
- **Information leakage** using systems such as finger or the DNS to obtain information that is necessary to administrators and the proper operation of the network, but could also be used by attackers;

- **Denial-of-Service** efforts to prevent users from being able to use their systems.

By observing this taxonomy, it is clear that it can possibly include almost every type of known attack. Indeed, it is often confusing, since categories represent either the target of the attack, its final purpose, or the strategy used to obtain the desired effect. Also, large lists of items are rarely fully comprehensive, since it is difficult to keep the pace at which novel attacks are introduced.

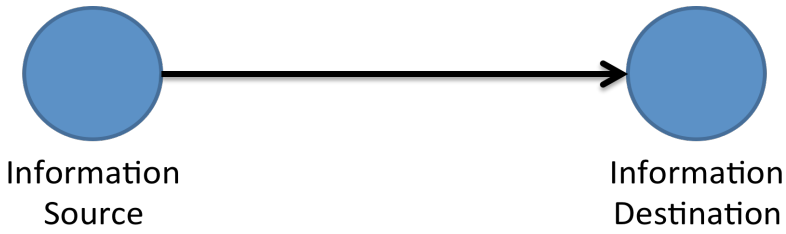

Figure 5: Normal Information Flow

In an effort to overcome taxonomies based on mere listings, Stallings proposed an attack categorization based on modification to the information flow [Stallings 1995] resulting from the attacker's actions. Given the model of normal information exchange between two entities, as represented in Figure 5, he individuates four possible modification of such flow. In Stalling's taxonomy, a distinction is made between active and passive attacks. Passive attacks are those which don't interfere with the normal information flow. What they do is just collect the information, without any effect on its normal transmission.

Interception (Figure 6(b)) is viewed as a passive attack, since it only intercepts the information flow, directing it towards a collecting node. The natural information flow is preserved unmodified. On the other hand, interruption (Figure 6(a)), modification (Figure 6(c)) and fabrication (Figure 6(d)) are viewed as active attacks. This taxonomy is particularly interesting since it expresses the properties of attacks as processes, individuating their algorithmic nature, focusing on the general problem of information exchange, and trying to find common properties in their life cycle.

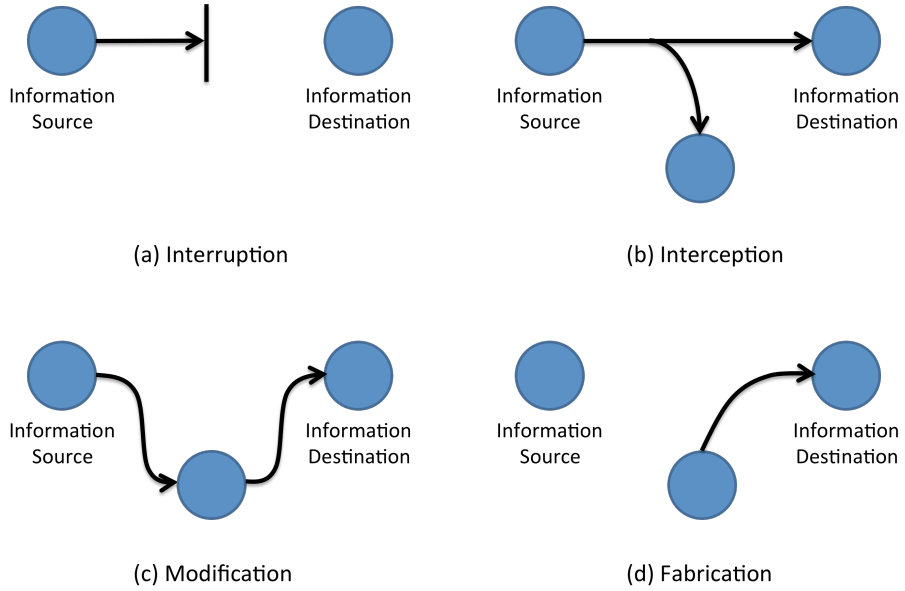

Figure 6: Attacks as Flow Information Modification

Howard [Howard 1995], instead gives a more operational view of attacks. He groups attacks according to the type of attacker, the implemented access strategy, the tools used and the expected results.

In [Hansman 2005] a taxonomy of threats to computer and network security is presented. According to previous well known taxonomies, Hansman and Hunt introduce a number of properties which a good taxonomy should have, such as:

- **Accepted** The taxonomy should be structured so that it can be become generally approved.
- **Comprehensible.** A comprehensible taxonomy will be able to be understood by those who are in the security field, as well as those who only have an interest in it.
- **Completeness/Exhaustive.** For a taxonomy to be complete/exhaustive, it should account for all possible attacks and provide categories for them. While it is hard to prove a taxonomy is complete or exhaustive, they can be justified through the successful categorization of actual attacks.
- **Determinism** The procedure of classifying must be clearly defined.
- **Mutually exclusive.** A mutually exclusive taxonomy will categorize each attack into, at most, one category.
- **Repeatable.** Classifications should be repeatable.
- **Terminology complying with established security terminology.** Existing terminology should be used in the taxonomy so as to avoid confusion and to build on previous knowledge.
- **Terms well defined.** There should be no confusion as to what a term means.
- **Unambiguous.** Each category of the taxonomy must be clearly defined so that there is no ambiguity as to where an attack should be classified.
- **Useful.** A useful taxonomy will be able to be used in the security industry. For example, the taxonomy should be able to be used by incident response teams.

These properties are the result of an accurate study of previous models, and try to respect all the good accomplishments of earlier research in the field of attack categorization. They propose to use a somewhat geometrical framework in order to identify attack types and a structure using four dimensions. The first dimension, also referred to as base dimension, categorizes attacks according to the attack vector. The second dimension covers the attack target. Classification can either be very fine-grained, down to the program version number affected by a specific vulnerability, or cover a class of targets, such as entire operating systems' processes. The third dimension covers the vulnerabilities and exploits, if they exist, that the attack uses. There is no structured classification for vulnerabilities and exploits due to their possible infinite number. The fourth dimension takes into account the possibility for a side effect for a specific attack. Some attacks, like trojan horses for example, carry the burden of what's hidden in the horse, which is usually the most dangerous part of it.

In the remainder of this section we will report some example of potential intentional failures affecting complex critical system focusing on the attacks to control systems based on SCADA (Supervisory Control And Data Acquisition) technology.

3.2.1 Example of intentional failures in Complex Distributed System based on SCADA

A SCADA system is in charge of controlling and monitoring infrastructures, facilities, and industrial processes through a distributed computing system. During the years the SCADA systems have evolved from the pioneering "centralized" system to "distributed system" with proprietary communication protocols, to "networked" architectures consisting of COTS equipment and adopting open communication protocols. While this evolution has allowed the operators to reduce the Operational Expenditure (OPEX) and to optimize the business, on the other hand it has also revealed the weaknesses of these systems and what is even worse, it has incorporated lots of new and old well-known vulnerabilities coming from the ICT world [Byres 2004]. Initially, SCADA systems, indeed, had little resemblance to ICT systems in that they were isolated systems running proprietary control protocols using specialized hardware and software. Widely available, low-cost Internet Protocol (IP) devices are now replacing proprietary solutions, which increases the possibility of cyber security vulnerabilities and incidents. As SCADA systems are adopting ICT solutions to promote corporate connectivity and remote access capabilities and are being designed and implemented using industry standard computers, operating systems (OS) and network protocols, they are starting to resemble ICT systems.

To understand the vulnerabilities associated with control systems, it is of paramount importance to first know all the possible communications paths into and out of the SCADA. There are many ways to communicate with an SCADA network and components using a variety of computing and communications equipment. A person who is knowledgeable in process equipment, networks, operating systems and software applications can use these and other electronic means to gain access to the SCADA system. Basically, an attacker who wishes to assume control of a control system is faced with three challenges:

1. Gain access to the control system LAN
2. Gain understanding of the process through discovery

3. Gain control of the process.

Therefore, when describing all the possible communications paths into and out of the SCADA system it is logical to do so from a control network perspective. Before introducing these concepts it is quite convenient to refresh what the basic components of the control system LAN are. Basically, in most installations there are six different components: the Data Acquisition Server, sometimes also called Master Terminal Unit or SCADA server; the Advanced Applications Server; the Master Database Server, sometimes called Historian server; the Human-Machine Interface (HMI) console; and the engineering workstation. The SCADA network is often connected to the business office network to provide real-time transfer of data from the control network to various elements of the corporate office.

The following subsections describe how the underlying technology may be exposed or introduce cyber vulnerabilities to the SCADA environments.

3.2.1.1 Vulnerabilities affecting the Control System LAN

The first thing an attacker needs to accomplish is to bypass the perimeter defences and gain access to the control system LAN. Most control system networks are no longer directly accessible remotely from the Internet. Common practice in most industries is to have a firewall separating the business LAN from the control system LAN. Not only does this help to keep hackers out, but it also isolates the control system network from outages, worms, and other afflictions that occur on the business LAN. Most of the attacker's off-the-shelf hacking tools can be directly applied to the problem. There are a number of common ways an attacker can gain access, but the miscellaneous pathways outnumber the common pathways.

Vulnerabilities affecting Common Network Architectures

There are three common architectures found in most control systems. Every business has its own minor variations dictated by their environment. All three are securable if the proper firewalls, intrusion detection systems, and application level privileges are in place.

By far, the most common architecture is the two-firewall architecture. The business LAN is protected from the Internet by a firewall and the control system LAN is protected from the business LAN by a separate firewall. The business firewall is administered by the corporate IT staff and the control system firewall is administered by the control system staff.

Large DCS often need to use portions of the business network as a route between multiple control system LANs (see Figure 7). Each control system LAN typically has its own firewall protecting it from the business network and encryption protects the process communication as it travels across the business LAN. Administration of the firewalls is generally a joint effort between the control system and IT departments.

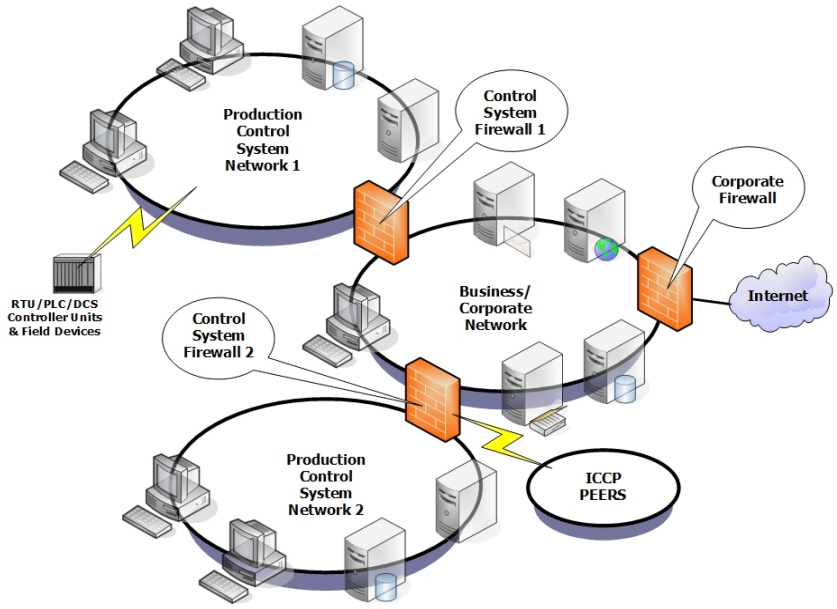

Figure 7 – Use of firewalls to protect control system networks

The second most common architecture is the control system network as a Demilitarized Zone (DMZ) off the business LAN. A single firewall is administered by the corporate IT staff that protects the control system LAN from both the corporate LAN and the Internet.

Finally, a new trend is to install a data DMZ between the corporate LAN and the control system LAN (see Figure 8). This provides an added layer of protection because no communications take place directly from the control system LAN to the business LAN. The added strength of a data DMZ is dependent on the specifics of how it is implemented.

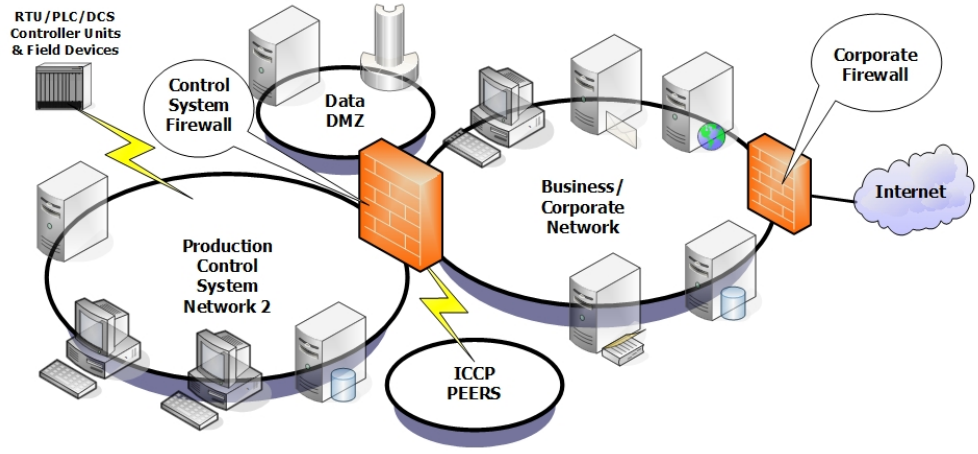

Figure 8 – DMZ between the corporate LAN and the control system LAN

Vulnerabilities introduced by Dial-up Access to the RTUs

One of the most common routes of entry is directly dealing modems attached to the field equipment. Modems are used as backup communications pathways if the primary high-speed lines fail. The attacker dials every phone number in a city looking for modems. Additionally, an attacker will dial every extension in the company looking for modems hung off the corporate phone system. Most RTUs identify themselves and the vendor who made them. Most RTUs require no authentication or a password for authentication. It is common to find RTUs with the default passwords still enabled in the field.

The attacker must know how to speak the RTU protocol to control the RTU. Control is generally, but not always, limited to a single substation.

Vulnerabilities introduced by Vendor Support

Most control systems come with a vendor support agreement. There is a need for support during upgrades or when a system is malfunctioning. The most common means of vendor support used to be through a dial-up modem (see Figure 9). In recent years, that has transitioned to Virtual Private network (VPN) access to the control system LAN. An attacker will attempt to gain access to internal vendor resources or field laptops and piggyback on the connection into the control system LAN.

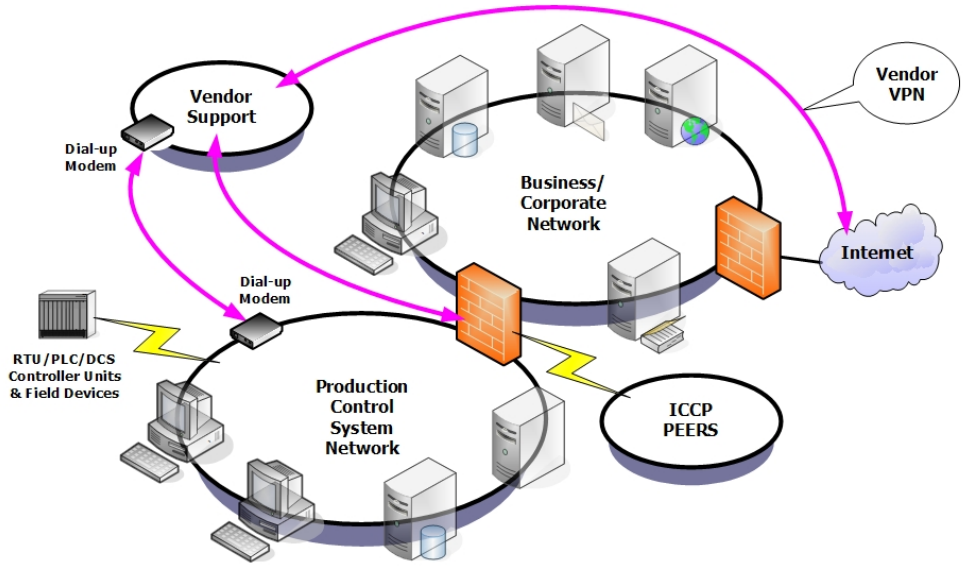

Figure 9 – Vendor support access to the SCADA system

Vulnerabilities affecting IT Controlled Communication Gear

Often, it is the responsibility of the corporate IT department to negotiate and maintain long-distance communication lines. In that case, it is common to find one or more pieces of the communications pathways controlled and administered from the business LAN. Multiplexers for microwave links and fibre links are the most common items. A skilled attacker can reconfigure or compromise those pieces of communications gear to control field communications.

Vulnerabilities affecting Corporate VPNs

Most control systems have some mechanism for engineers on the business LAN to access the control system LAN. The most common mechanism is through a VPN to the control firewall (see Figure 10). An attacker will attempt to take over a machine and wait for the legitimate user to VPN into the control system LAN and piggyback on the connection.

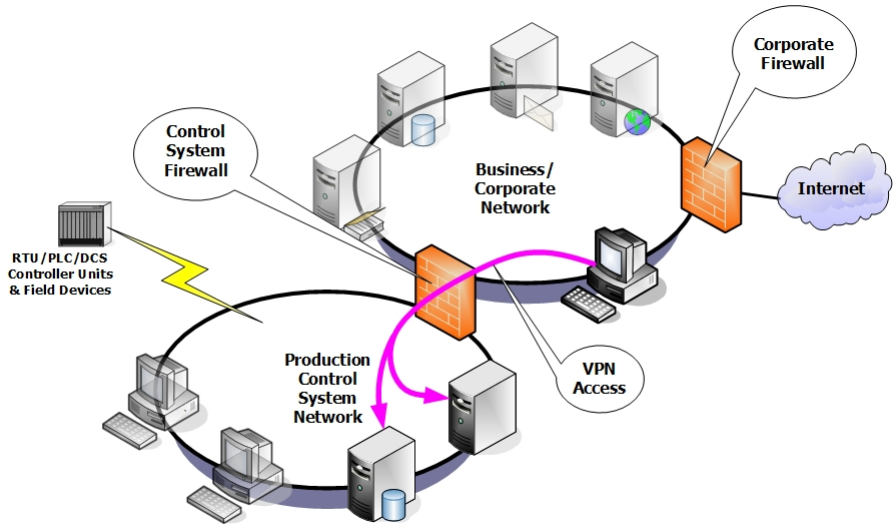

Figure 10 – Corporate VPNs

Vulnerabilities affecting Database Links

Nearly every production control system logs to a database on the control system LAN that is then mirrored into the business LAN. Often administrators go to great lengths to configure firewall rules, but spend no time securing the database environment. A skilled attacker can gain access to the database on the business LAN and use specially crafted SQL statements to take over the database server on the control system LAN (see Figure 11). Nearly all modern databases allow this type of attack if not configured properly to block it.

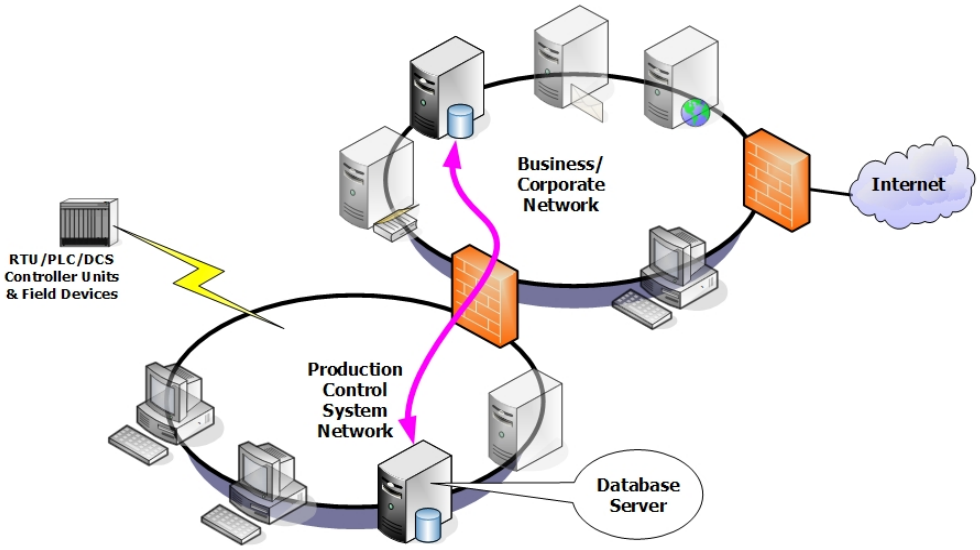

Figure 11 – Database mirroring

Vulnerabilities affecting Poorly Configured Firewalls

Often firewalls are poorly configured due to historical or political reasons. Common firewall flaws include passing Microsoft Windows networking packets, passing remote services, and having trusted hosts on the business LAN. The most common configuration problem is not providing outbound data rules. This may allow an attacker who could sneak a payload onto any control system machine to call back out of the control system LAN to the business LAN or the Internet

Vulnerabilities in Peer Utility Links

Often, the easiest way onto a control system LAN is to take over neighbouring utilities or manufacturing partners. Historically, links from partners or peers have been trusted. In that case, the security of the system is the security of the weakest member (see Figure 12). Recently, peer links have been restricted behind firewalls to specific hosts and ports.

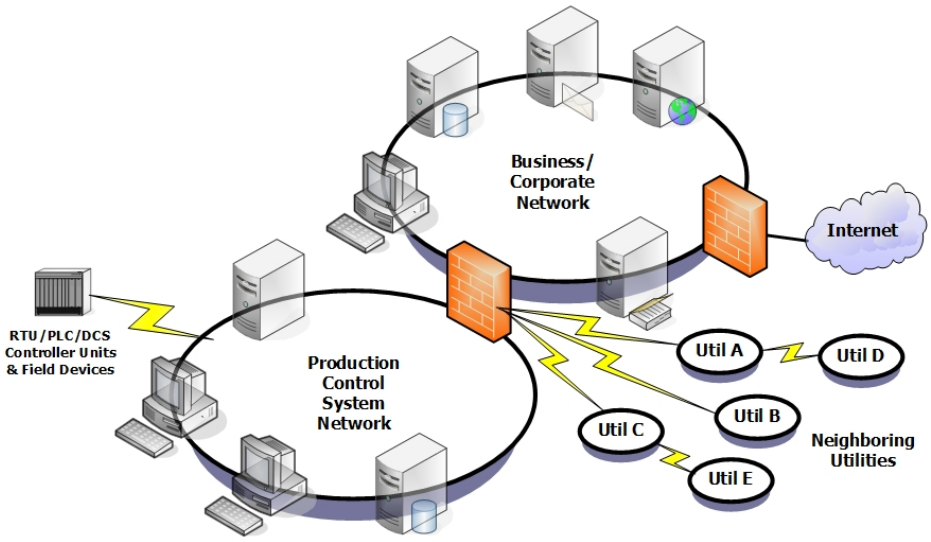

Figure 12 – Peer utility links

3.2.1.2 Vulnerabilities in control system communication protocols

The core precepts of IT security are confidentiality, integrity and authentication, precepts not present in the design of most control systems. Currently, in the majority of control systems, if an attacker can penetrate into the control system network segment, the lack of the core security principle allows the attacker to "own the system". These failures are most notably:

- **The use of poor (or lack of) authentication schemas** (e.g. no passwords, default passwords, and clear text password exchanges). Authentication is critical in any system as it ensures that a user is who they "appear" to be. Authentication also helps in the authorization process, limiting the privileges that a user has to modify data and interact with system processes. Moreover, authentication goes beyond the user-centric point of view. On network communications, authentication can mean ensuring that both communication ends are legitimate. Control systems and their communication protocols do present a lack of or poor authentication schemas. This inherent weakness in control systems allows attackers a variety of attack pathways including firmware attacks and simple vectors into "rooting" control system services on both field devices and pc based servers and workstations.
- **No use of data integrity schemas.** The lack of integrity controls allows attackers to manipulate the data communicated on the system through man in the middle, IP spoofing and various other attacks. This is critical for ICS systems since most of the automated decisions are taken based on this information. A modification of these data can cause the system to take the wrong decision.
- **No encryption.** Though confidentiality of the control system communication may not be of the upmost importance, the encryption of password and key exchanges would serve to greatly improve the security postures of control systems.

The lack of encryption and of authentication or integrity validation mechanisms is not exclusive of control systems' protocols. In fact, the Ethernet or the TCP/IP protocol suite, including ICMP, IGMP and the initial versions of the routing protocols (OSPF, RIP, etc.) are affected by exactly the same design vulnerabilities. Nevertheless, networks and systems based on the TCP/IP protocol suite have evolved during the last years and new versions or even new protocols have appeared to solve this problem. For instance, at the data plane of the network level of the OSI model we have the IPSec protocol. At the control plane, routing protocols now include in their lasts versions brand-new security mechanisms for message authentication and data integrity validation.

Nowadays the new trend in control systems' communication protocols is to migrate from specific protocol stacks (e.g. IEC-101, DNP3, Modbus, etc.) to IP-based protocols (e.g. IEC-104, DNP3 over TCP/IP, Modbus/TCP, etc.). However, while some of the security solutions design for typical IT systems could now be useful for the control systems' world, special precautions must be taken when introducing them to ICS environments. ICS have many characteristics that differ from traditional IT systems. Therefore, in some cases new security solutions are needed that are tailored to the ICS environment. Some of these differences that can condition protocol security measures but also the whole control system design include:

- **Performance Requirements.** ICS are generally time-critical, with the criterion for acceptable levels of delay and jitter dictated by the individual installation. Some systems require deterministic responses. High throughput is typically not essential to ICS. In contrast, IT systems typically require high throughput, and they can typically withstand some level of delay and jitter.
- **Availability Requirements.** Many ICS processes are continuous in nature. Unexpected outages of systems that control industrial processes are not acceptable. Outages often must be planned and scheduled days/weeks in advance. Exhaustive pre-deployment testing is essential to ensure high availability for the ICS. In addition to unexpected outages, many control systems cannot be easily stopped and started without affecting production. In some cases, the products being produced or equipment being used is more important than the information being relayed. Therefore, the use of typical IT strategies such as rebooting a component, are usually not acceptable solutions due to the adverse impact on the requirements for high availability, reliability and maintainability of the ICS. Some ICS employ redundant components, often running in parallel, to provide continuity when primary components are unavailable.
- **Time-Critical Responses.** In a typical IT system, access control can be implemented without significant regard for data flow. For some ICS, automated response time or system response to human interaction is very critical. For example, requiring password authentication and authorization on an HMI must not hamper or interfere with emergency actions for ICS. Information flow must not be interrupted or compromised. Access to these systems should be restricted by rigorous physical security controls.
- **System Operation.** ICS Operating Systems (OS) and applications may not tolerate typical IT security practices. Legacy systems are especially vulnerable to

resource unavailability and timing disruptions. Control networks are often more complex and require a different level of expertise (e.g., control networks are typically managed by control engineers, not IT personnel). Software and hardware are more difficult to upgrade in an operational control system network. Many systems may not have desired features including encryption capabilities, error logging, and password protection.

3.2.1.3 Network Vulnerabilities

In this section we will briefly report some network level vulnerabilities for different technologies and equipment in used at many real SCADA systems. The source of the research has been the National Vulnerability Database. The section also includes flaws identified as network level cyber-weaknesses. We have mainly based our research on the Common Weaknesses and Exposures dictionary.

ID	Title	Description
IV-00036	Stop alarm events using DNP3 protocol	An attacker stops unsolicited responses from field devices to prevent alarms and other critical events.
IV-00037	Unsolicited response Store	An established connection between a HMI or control server and a PLC is hijacked or spoofed to send other attacks to either device.
IV-00038	Unauthorized client may restart devices using DNP3 protocol	An attacker can force a PLC or other DNP3 server to power cycle by issuing a response packet with function code 0D.
IV-00039	Unauthorized access to DNP3 devices	An unauthorized DNP3 client attempts to read or write information from a PLC or other field device.
IV-00040	DNP3 device listing	An attacker may list what DNP3 data points are available in due to insufficient authentication protocol
IV-00041	Unauthorized access to Modbus server information	An attacker learns the vendor, product, version number and other information about a PLC or other MODBUS server.

IV-00042	Unauthorized access to Modbus Devices information	An attacker gains information on a PLC or other Modbus server by issuing the function code 17 Report Slave ID request.
IV-00043	Modbus device listing	An attacker may list what Modbus data points are available in due to insufficient authentication protocol
IV-00044	Takebishi Electric DeviceXPlorer OPC Server fails to properly validate OPC server handles	The Takebishi Electric DeviceXPlorer OPC Server fails to properly validate server handles. This vulnerability may be triggered by an attacker with access to the server's OPC interface. The following versions of DeviceXPlorer OPC Server are affected by this vulnerability: * DeviceXPlorer MELSEC OPC Server * DeviceXPlorer SYSMAC OPC Server * DeviceXPlorer FA-M3 OPC Server * DeviceXPlorer TOYOPUC OPC Server * DeviceXPlorer HIDIC OPC Server * DeviceXPlorer MODBUS OPC Server
IV-00045	NETxAutomation Vulnerabilities	NETxAutomation NETxEIB OPC Server before 3.0.1300 does not properly validate OLE for Process Control (OPC) server handles, which allows attackers to cause a denial of service or possibly execute arbitrary code via unspecified vectors involving the (1) IOPCSyncIO::Read, (2) IOPCSyncIO::Write, (3) IOPCServer::AddGroup, (4) IOPCServer::RemoveGroup, (5) IOPCCommon::SetClientName, and (6) IOPCGroupStateMgt::CloneGroup functions, which allow access to arbitrary memory. NOTE: the vectors might be limited to attackers with physical access.

Table 1 – Cyber-vulnerabilities description

ID	Title	Summary
IW-0043	Predictable from Observable State	A number or object is predictable based on observations that the attacker can make about the state of the system or network, such as time, process ID, etc.
IW-0044	Interpretation Conflict	Product A handles inputs or steps differently than Product B, which causes A to perform incorrect actions based on its perception of B's state. This is generally found in proxies, firewalls, anti-virus software, and other intermediary devices that allow, deny, or modify traffic based on how the client or server is expected to behave.
IW-0045	Insufficient Control of Network Message Volume	The object does not sufficiently monitor or control transmitted network traffic volume, so that an actor can cause the software to transmit more traffic than should be allowed for that actor.
IW-0046	Insufficient Resource Pool	The software's resource pool is not large enough to handle peak demand, which allows an attacker to prevent others from accessing the resource by using a (relatively) large number of requests for resources
IW-0047	Insufficient Verification of Data Authenticity	The target does not sufficiently verify the origin or authenticity of data, in a way that causes it to accept invalid data.
IW-0048	Use of a Broken or Risky Cryptographic Algorithm	The use of a broken or risky cryptographic algorithm is an unnecessary risk that may result in the disclosure of sensitive information. The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Well-known techniques may exist to break the algorithm.

IW-0049	Failure to Enforce that Messages or Data are Well-Formed	When communicating with other components, the software does not properly enforce that structured messages or data are well-formed before being read from or sent to that component. This causes the message to be incorrectly interpreted.
IW-0050	Improper Handling of Syntactically Invalid Structure	The product does not handle or incorrectly handles input that is not syntactically well-formed with respect to the associated specification.
IW-0051	Improper Authentication	The protocol does not properly ensure that the user has proven their identity.
IW-0052	Error Message Information Leak	The protocol generates an error message that includes sensitive information about its environment, users, or associated data.
IW-0053	Covert Timing Channel	Covert timing channels convey information by modulating some aspect of system behavior over time, so that the program receiving the information can observe system behavior and infer protected information.
IW-0054	Interpretation Conflict	Product A handles inputs or steps differently than Product B, which causes A to perform incorrect actions based on its perception of B's state.
IW-0055	Misinterpretation of Input	The software misinterprets an input, whether from an attacker or another product, in a security-relevant fashion.
IW-0056	Insufficient Type Distinction	he software does not properly distinguish between different types of elements in a way that leads to insecure behaviour.
IW-0057	Incomplete Model of Endpoint Features	A product acts as an intermediary or monitor between two or more endpoints, but it does not have a complete model of an endpoint's features, behaviours, or state, potentially causing the product to perform incorrect actions based on this incomplete model.

IW-0058	Behavioural Change in New Version or Environment	A's behaviour or functionality changes with a new version of A, or a new environment, which is not known (or manageable) by B.

Table 2 – Cyber-weaknesses description

4 DETECTION

Error detection stage is crucial to recognize that something unexpected has occurred in the system and to trigger error and fault handling procedures. It can be addressed at different levels, from the lowermost one, the hardware (e.g., machine check, division by zero, etc.), to the uppermost, i.e., the applications. In any case, its aim is to detect errors before they can propagate to errors or failures for other different component(s) or to failure for the overall system.

Anderson [Anderson 1981] has proposed a check-based classification for error detectors, which include:

- *Replication checks* make use of matching components with error detection based on comparison of their outputs. This is applicable only when sufficient redundancy is provided with the assumption of independent faults.
- *Timing checks* are applicable to systems and components whose specifications include timing constraints, including deadlines. Based on these constraints, checks can be developed to look for deviations from the acceptable component behaviour. Watchdog timers are a type of timing detector with general applicability that can be used to detect anomalies in component behavior.
- *Reversal checks* use the output of a component to compute the corresponding inputs based on the function of the component. An error is detected if the computed inputs do not match the actual inputs. Reversal checks are applicable to modules whose inverse computation is relatively straightforward.
- *Coding checks* use redundancy in the representation of information with fixed relationships between the actual and the redundant information. Error detection is based on checking those relationships before and after operations. Checksums are a type of coding check. Similarly, many techniques developed for hardware (e.g., Hamming, M-out-of-N, cyclic codes) can be used in software, especially in cases where the information is supposed to be merely referenced or transported by a component from one point to another without changing its contents. Many arithmetic operations preserve some particular properties between the actual and redundant information, and can thus enable the use of this type of check to detect errors in their execution.
- *Behavioral checks* use known semantic properties of data (e.g., range, rate of change, and sequence) to detect errors. These properties can be based on the requirements or the particular design of a component.
- *Structural checks* use known properties of components structure. For example, lists, queues, and trees can be inspected for number of elements in the structure, their links and pointers, and any other particular information that could be articulated. Structural checks could be made more effective by augmenting data structures with redundant structural data like extra pointers, embedded counts of the number of items on a particular structure, and individual identifiers for all the items ([Taylor 1980] [Taylor 1980b]).

Starting from the discussed error model (see Section 3), error detectors can be implemented, as local or remote modules, within software components or at their outputs. Based on the type of interaction these strategies can be grouped into: *push-mode*, *pull-mode* or *log-based* techniques.

Push-mode techniques trust on the component, which after the execution of operation provide some information that specify if the requested services could not be satisfied [Randel 1995]. For instance, exceptions can be signaled by a component when its error detection mechanisms detect an invalid service request, i.e., an interface error, or when errors in its own internal operations (i.e., local error) are discovered.

Pull-mode techniques rely on the presence of an external module, which is in charge of interrogating (i.e. by means of pre-planned queries) the monitored component, to discover latent errors. The query can be explicitly performed at predetermined time, by sending a "control" request and waiting for the reply, [Tanebaum 2006], or, it can be implicitly send each time another component tries to invoke the monitored component [Aguilera 2002].

Log-based techniques, instead, consist in examining the log files produced by the components, in order to understand the system behavior by correlating different events [Simache 2002] [Iyer 2007]. Logs are the only direct source of information available in the case of OTS items where no other internal monitor can be added.

These techniques belong to *direct detection strategies*: they try to infer the system health status by directly querying it, or by analyzing events it is able to produce (e.g., logs, exceptions). However, some problems arise with these approaches because of the nature of software faults. According to Abbott [Abbott 1990], it is impossible to anticipate all the faults and their generated errors in a system:

"If one had a list of anticipated faults, it makes much more sense to eliminate those faults during design reviews than to add features to the system to tolerate those faults after deployment. The problem, of course, is that it is unanticipated faults that one would really like to tolerate."

For instance, we observe that when a fault is activated it leads to an incorrect state, i.e., an error; however, this does not necessarily mean that the system knows about it. Therefore we have to deal not only with detected errors (i.e., errors which presence is indicated by an error message or error signal) but also with latent errors (i.e., errors which are present but not yet detected). However besides causing a failure, both latent and detected errors may cause the system/component to have an out-of-norm behavior as a side effect. This out-of-norm behavior is called anomaly. For these reasons, *anomaly-based detection* approaches have also been considered to detect errors.

Anomaly-based detectors consist in comparing normal and anomalous runs of the system. These approaches are quite different with the respect to the previous one, because they try to infer the health of the observed components by monitoring system parameters and their interactions with the environment. These approaches can be also labeled as *indirect detection strategies*. As an example, the work in [Iyer 2007] exploits hardware performance counters and OS signals to monitor the system behavior and to signal possible anomalous conditions. A similar approach is followed in [Khanna 2006], which provides detection facilities for large-scale distributed systems running legacy code. The detection system proposed here is an autonomous self-checking monitor, which is architecture-independent and has a hierarchical structure. The monitors are designed to observe the external messages that are exchanged between the protocol entities of the distributed system under study. They use the observed messages to deduce a runtime state transition diagram (STD) that has been executed by the all the

entities in the system. Indirect techniques have been especially adopted for intrusion detection systems. In [Forrest 1996] the execution of a program is characterized by tracing patterns of invoked system calls. System call traces produced during the operational phase are then compared with nominal traces to reveal intrusion conditions. Other solutions, such as [Stolfo 1998] [Srinivasan 2005], are based on statistical learning approaches. They extract recurrent execution patterns (using system calls or network connections) to model the application under nominal conditions, and to classify run-time behaviors as normal or anomalous.

When detectors trigger an alarm to signal a deviation from the norm, one has to evaluate if this alarm is due to the activation of a dormant fault. Therefore to evaluate detectors performance we can use quality metrics in order to evaluate the effectiveness of detection mechanisms, as well as to make comparisons among different approaches.

In order to provide a fair and sound comparison between several approaches we analyze criteria that should be used to characterize on-line detectors performance for LCCI. We first summarize some of the most used metrics in literature then we must specify, which metrics are the most representative in our scenario and why.

Roughly speaking the goal of an on-line failure detector is to reveal all the occurring failures, to reveal them timely and not to commit any false alarm.

To ease the description of metrics and to better understand their meaning we introduce some definitions:

- *True Positive* (TP)*:* if a failure occurs and the detector triggers an alarm;
- *False Positive* (FP): if no failure occurs and an alarm is given;
- *True Negative* (TN): if no real failure occurs and no alarm is raised;
- *False Negative* (FN): if the algorithm fails to detect an occurring failure.

Clearly high number of TP and TN are good, while the vice versa for FP and FN.

Metrics coming from diagnosis literature are usually used to compare the performance of detectors [Daidone 2009]. For instance *coverage* measures the detector ability to reveal a failure, given that a failure really occurs; *accuracy* is related to mistakes that a failure detector can make. Coverage can be measured as the number of detected failures divided by the overall number of failures, while for accuracy there are different metrics.

Basseville et al. [Basseville 1993] consider the *mean delay for detection* (MDD) and the *mean time between false alarms* (MTBFA) as the two key criteria for on-line detection algorithms. Analysis are based on finding algorithms that minimize the mean delay for a given mean time between false alarms and on other indexes derived from these criteria.

In [Malek 2010] metrics borrowed from information retrieval research are used, namely *precision* and *recall*. In their context recall measures the ratio of the failure that is correctly detected, i.e., *TP/(TP+FN)*, while precision measures the portion of the predicted events, which are real failure, i.e., *TP/(TP+FP)*. Thus perfect recall (recall=1) means that all failures are detected and perfect precision (precision=1) means that there are no false positives. A convenient way of taking into account precision and recall at the same time is by using *F-measure*, which is the harmonic mean of the two quantities. Since in diagnosis the ratio of failures correctly detected (recall) is also called coverage, in the following we refer to it as coverage

However using solely precision and coverage is not a good choice because they do not account for true negatives, and since failures are rare events we need to evaluate the detector mistake rate when no failure occurs. Hence, in combination with precision and coverage, one can use *False Positive Rate* (FPR), which is defined as the ratio of incorrectly detected failures to the number of all non-failures, thus *FP/(FP+TN)*. Fixing Precision and Coverage, the smaller the false positive rate, the better. Another metric is *Accuracy* [Malek 2010], which is defined as: the ratio of all correct decision to the total number of decisions that have been performed, i.e., (*TP+TN*)/(*TP+TN+FP+FN*).

Chen, Toueg and Aguilera [Aguilera 2002] propose three primary metrics to evaluate detectors quality, in particular their accuracy. The first one is *Detection Time* (DT), which informally accounts for the promptness of the detector. The second one is the *Mistake Recurrence Time* (T_{MR}), which accounts for time elapsed between two consecutive erroneous transitions from Normal to Failure. Finally they define *Mistake Duration* (T_M), which is related to the time that detector takes to correct the mistake. Other metrics can be simply derived from the previous one. For instance, *Average Mistake Rate* (λ_M), represents the number of erroneous decisions in the time unit; *Good period duration* (T_G) measures the length of period during which the detector does not trigger a false alarm; *Query accuracy probability* (P_A) is the probability that the failure detector's output is correct at a random time.

Bearing in mind classes of our target applications we believe that, when dealing with long running and safety critical systems, mistake duration (and thus T_G) is less appropriate than Coverage, accuracy and λ_M, since just an alarm may be sufficient to trigger the needed actions (e.g., put the system in a safety state). Coverage is essential because if the detector does not reveal a failure, then more severe (and potentially catastrophic) consequences may happen. Accuracy and λ_M are useful to take into account false positives because each failure detector mistake may result in costly actions (such as shut down, reboot, etc.).

The query accuracy probability is not sufficient to fully describe the accuracy of a failure detector, in fact, as discussed in [Aguilera 2002], for applications in which every mistake causes a costly interrupt the mistake rate is an important accuracy metric.

We point out the differences between Accuracy, defined in [Malek 2010], and P_A defined in [Aguilera 2002]. Since we consider a fail stop model, *TP<<TN*, so if *FP<<TN*, then Accuracy≈1. For these reasons we consider P_A as more representative than Accuracy to compare SPS algorithm with Static Thresholds Algorithm.

Table 3. Metrics for performance evaluation of failure detectors.

Metric	*Formula*	*Metric*	*Formula*
Coverage (C)	$TP/(TP+FN)$	**Accuracy-Coverage TradeOff**	$C \cdot A$
Precision (P)	$TP/(TP+FP)$	**MTBFA** (Mean Time Between Mistakes)	$E(T_{MR})$
F-Measure	$(2 \cdot P \cdot C)/(P+C)$	**MDD** (Mean Delay for Detection)	$E(T_D)$
FPR (False Positive Rate)	$FP/(FP+TN)$	**λ M** (Average Mistake Rate)	$1/E(T_{MR})$
Accuracy (A)	$\frac{TP+TN}{TP+FP+TN+FN}$	**P_A** (Query accuracy probability)	$\frac{E(T_{MR}-T_M)}{E(T_{MR})}$

4.1 Middleware Detection

To monitor or monitoring generally means to be aware of the state of a system, especially if the system is in a safe state or it entered into an unexpected, or unwanted, state that implies outages. At the middleware level, the state is represented by two different aspects: (i) received messages and (ii) connectivity among the different components of the middleware.

In the first case, monitoring the received messages implies that the source constantly checks if the destinations have received the same messages or that some message losses occurred. This can be done by sequencing the produced messages and periodically sending a command, named Hearbeat, that force the receiver to reply about the identifications of the latest received messages or even of the unreceived ones. A concrete example of a protocol that adopts such monitoring scheme is the Automatic Repeat request (ARQ) [Lin 1984], adopted in several kind of middleware such as the recent OMG Data Distribution Service (DDS) [OMG 2007]. An other loss-detection mechanisms can be receiver-based, rather than sender-based as the previously introduced one. Specifically, the destination can detect a message loss if the expected message does not arrive within a certain time frame, or even if the received message does not contain the expected identification, or even other destinations notify the messages they have received so far, named notification history, and the given one infers a loss from such received histories. Receiver-based techniques allow providing higher scalability guarantees and are preferred in large-scale systems, and adopted in advanced protocols such as Gossiping Protocols [Costa 2000].

In the second case, let us consider a particular kind of middleware such as a Distributed Event-Based Systems (DEBS), but the following considerations can be easily generalize to any other kind. The Notification Service that glues together publishers and subscribers by enabling the communication among each other, is typically the target of any monitoring activities. The architecture of Notification Service is based on the Broker design pattern, and consists into a network of brokers interconnected by an overlay network. In this case, the goal of any monitoring activities is to check the liveliness of brokers and the correctness of the overlay links. In the first case, entities connected to a certain broker (i.e., other brokers or publishers and subscribers) send a particular command named Keep_Alive. At the reception of such message, the broker has to immediately respond. If a response to such a command is not received within a certain time period, then the broker is considered down. Such simple monitoring mechanism is commonly adopted in all DEBS products, such as the ones compliant to OMG DDS specification. On the other hand, in case of link monitoring, each end-point of a given link holds statistics of the perceived quality of the notification delivery along the given link (such as latency or success rate). In this case, the end-point is aware of any possible degradation due to networking failure of even link crash, and able to react with proper countermeasures. Such mechanism is adopted in several solutions to achieve reliable event notification, such as [Kazemzadeh 2009].

4.2 Network Detection

Due to the complexity of a network infrastructure, the failure detection at this level presents a variety of solution depending on the specific fault to identify. We can distinguish between solutions for unintentional and intentional failures. However, similar approaches can be exploited for both the fault typology.

As concerns the former issue, several mechanisms have been implemented for detecting either error of communication process and failures of network components. In communication, an error is a situation in which the received data does not match whit the sent data. Errors can occur for many reasons, including the signal problems, such as noise, interference, or distortion, and protocol problems. Usually, the communication errors are at the level of individual bits, so the task becomes to ensure that the bit sequence received matches the one sent. Several solutions have been proposed to identify communication errors. Such solutions vary in how effective they are, and all of them impose a transmission penalty in the form of extra bits that must be sent. Detecting errors involves the identification of an incorrect or invalid transmission element, such as an impossible character or a garbled (but not encrypted) message. In general, error-detection strategies rely on a numerical value (based on the bytes transmitted in a packet) that is computed and included in the packet. The receiver computes the same type of value and compares the computed result with the transmitted value. Error-detection strategies differ in the complexity of the computed value and in their success rate. Error-detection methods include cyclic or longitudinal redundancy checks and the use of parity bits.

As concerns the unintentional failures in network components, detection mechanisms usually retrieve and analyze a large amount of diagnostic information about the infrastructure status coming from the network. This information can be either acquired using proper monitoring tools [Brodie 2001][Natu 2006][Natu 2007], or received from network entities in the form of network alarms [Yemini 1996][Gardner 1997][Bouloutas 1993][Wang 1993]. Indeed, two main approaches for fault detection systems can be identified: (i) fault detection systems that actively sample performance data from network components, commonly referred to as *probe-based systems* or *active systems*, and (ii) fault detection systems that utilize network alarms, commonly referred to as *alarm correlation-based systems* or *passive systems*. Both paradigms address certain challenges and offer alternative solutions to the fault network detection problem.

In a probe-based fault detection dedicated software tools (e.g., pinging, trace routing, system log analyzer, etc.) are used for monitoring network infrastructure. Probing stations are first determined and located in different parts of the network. The results of probing activities are periodically sent from the probing station to a network management system. These probes are collectively analyzed to determine whether a failure has been detected. To effectively isolate the malfunctioning network component, the set of probes must cover all the nodes in the managed network. Depending on the number and locations of available probing stations this set has to include a large number of probes to be used. Employing a large number of these probes for fault identification tasks certainly increases the accuracy of locating network malfunctioning components; nevertheless, the fault identification task will become more time-consuming. Furthermore, a large set of probes generates excessive management traffic

injected into the managed network. Hence it is desirable to minimize the negative impact of the extra management traffic induced by these probes.

Alternative solutions for faults detection are based on network alarms generated by network entities as a response to network failures. Usually the alarms are based on the information provided by the SNMP traps and the management information base (MIB) variables. This variable describes the behavior of the network. Fault detection is performed by tracking a set of inherently invariant properties of the system: a fault is detected whenever invariant properties are violated whereas faults are identified by correlating these violations, or alarms, with a unique fault in the system. Common approaches for alarm correlation artificial intelligence [Hood 1996][Ho 2000], Petri net models [Silva 1985], finite state machine [Bouloutas 1992].

With respect to system architecture, traditionally network fault detection frameworks consist of two main components: the agents and the managers. Agents are basically monitoring software components that are installed in every monitored entity in the computer network. The fault detection activities, instead, are performed by the managers, upon retrieving some of these variables from their subordinate agents. This centralized framework can be tolerated in small size networks. As networks grow larger and become more heterogeneous, the centralized model creates a bottleneck. Recently, more advanced fault detection techniques have adopted a distributed approach by which the managed network is partitioned into distinct management domains, each managed by an independent management center. In this way, faults may be handled locally; thus reducing the amount of traffic that should be transmitted across the network.

As concerns the intentional faults, the intrusion detection system (IDS) represents a common approach for identifying malicious failure in network infrastructure. The network monitoring and traffic inspection are the critical tasks for a intrusion detection system. Generally an IDS provides network traffic monitoring in order to identify inappropriate, incorrect or anomalous activity within a system, be it a single host or a whole network. As like a "bouncer", IDS observes the traffic passing through the network, discriminates among different users by analyzing properly the different packets flows, and finally classifies them in normal or malicious based on well-known classification criteria. All this must be done in a transparent fashion, without interfering with legitimate user activities.

Several IDS classification criteria have been proposed. Usually an IDS can be grouped into three main categories:

- Network-based Intrusion Detection Systems (N-IDS),
- Host-based Intrusion Detection Systems (H-IDS),
- Stack-based Intrusion Detection Systems (S-IDS).

This classification depends on the information sources analyzed to detect an intrusive activity. N-IDS [Paxson 1999][Vigna 1999] analyzes packets captured directly from the network. By setting network cards in promiscuous mode, an IDS can monitor traffic in order to protect all of the hosts connected to a specified network segment. On the other hand, H-IDS [Andersson 1995][Tyson 2000] focuses on a single host's activity: the system protects such a host by directly analyzing the audit trails or system logs produced by the host's operating system. Finally, S-IDS [Laing 2000] are hybrid

systems, which operate similarly to a N-IDS, but only analyze packets concerning a single host of the network. They monitor both inbound and outbound traffic, following each packet all the way up the TCP/IP protocol stack, thus allowing the IDS to pull the packet out of the stack even before any application or the operating systems processes it. The load each IDS must afford is lower than the total traffic on the network, thus keeping the analysis overhead into reasonable bounds; hypothetically, each host on the network might run a S-IDS.

Intrusion Detection Systems can be roughly classified as belonging to two main groups as well, depending on the detection technique employed:

- anomaly detection,
- misuse detection,
- signature-based detection.

The first two techniques rely on the existence of a reliable characterization of what is normal and what is not, in a particular networking scenario. More precisely, anomaly detection techniques base their evaluations on a model of what is normal, and classify as anomalous all the events that fall outside such a model. Indeed, if an anomalous behavior is recognized, this does not necessarily imply that an attack activity has occurred: only few anomalies can be actually classified as attempts to compromise the security of the system. Thus, a relatively serious problem exists with anomaly detection techniques, which generate a great amount of false alarms. On the other side, the primary advantage of anomaly detection is its intrinsic capability to discover novel attack types. Numerous approaches exist which determine the variation of an observed behavior from a normal one. A first approach is based on statistical techniques. The detector observes the activity of a subject (e.g. number of open ¯les or TCP state transitions), and creates a profile representing its behavior. Every such profile is a set of "anomaly measures".

Statistical techniques can then be used to extract a scalar measure representing the overall anomaly level of the current behavior. The profile measure is thus compared with a threshold value to determine whether the examined behavior is anomalous or not. A second approach, named predictive pattern generation, is based on the assumption that an attack is characterized by a specific sequence, i.e. a pattern, of events. Hence, if a set of time-based rules describing the temporal evolution of the user's normal activity exists, an anomalous behavior is detected in case the observed sequence of events significantly differs from a normal pattern.

Misuse detection, also known as signature detection, is performed by classifying as attacks all the events conforming to a model of anomalous behavior. This technique is based on the assumption that an intrusive activity is characterized by a signature, i.e. a well-known pattern. Similarly to anomaly detection, misuse detection can use either statistical techniques or even a neural network approach to predict intrusions.

Signature-based detection is the most used to detect an attack. SNORT [Baker 2004] and Bro [Paxson 1999] are well-known systems based on this approach. Intrusions are coded by means of a set of rules: as soon as the examined event matches one of the rules, an attack is detected. A drawback of this approach is that only well-known intrusive activities can be detected, so that the system is vulnerable to novel aggressions; sometimes, few variations in an attack pattern may generate an intrusion

that the IDS is not able to detect.

The main problem related to both anomaly and misuse detection techniques reside in the encoded models, which define normal or malicious behaviors. Although some recent open source IDS, such as SNORT or Bro, provide mechanisms to extend the detection ability of the system in order to include either anomaly or misuse approaches [Esposito 2005a][Esposito 2005b], behavior models are usually hand-coded by a security administrator, representing a weakness in the definition of new normal or malicious behaviors. Recently, many research groups have focused their attention on the definition of systems able to automatically build a set of models. Data mining techniques are frequently applied to audit data in order to compute specific behavior models (MADAM ID [Lee 2000], ADAM [Barbara 2001]). Data mining algorithms are used for extracting specific models from large stored data. In particular machine learning or pattern recognition processes are usually exploited in order to realize this extraction. These processes may be considered off-line processes. In fact, all the techniques used to build intrusion detection models need a proper set of audit data. The information must be labeled as either "normal" or "attack" in order to define the suitable behavioral models that represent these two different categories.

5 DIAGNOSIS

Fault Diagnosis is one of the most important phases of Fault Handling because its aim is to identify the nature, the type location of the cause of errors. The issue of diagnosis has been faced since a long time, maybe since computers came. The first attempt to formalize the problem is due to Preparata et al., which introduced system level diagnosis in [Preparata 1967]. Since then, diagnosis has been faced by following several approaches and in many research fields. Although significant progresses have been done, the problem is still far away from the solution (this is especially for diagnosis of software faults).

Existing diagnosis approaches mainly cope with hardware-induced errors and their symptoms within a software system, thus they have to be revised in order to deal with software faults. Several challenges have to be faced when designing and implementing strategies for Software Fault Diagnosis. First of all, the presence of software faults hampers the definition of a synthetic and accurate mathematical model able to describe system failure behavior. Since software is intangible (i.e., it does not exist in a physical sense), it cannot degrade in the same manner as physical hardware components do. For this reason is not realistic to adopt the failure distributions (e.g., weibull or a log-normal) which are used to model hardware components. Second, software systems are often the result of the integration of many OTS (Off-The-Shelf) components, and a priori fault model may be not available for such components. Third, software fault activation can be very difficult to reproduce since may depend on the complex combination of user input, system state and environmental state.

In the last decade or so, there has being an increasing work focusing on the diagnosis. This problem is getting faced by several perspectives and by means of quite different techniques from a variety of research fields. On the one hand, such a generous literature is a benefit since approaches and techniques exist and can be leveraged and improved with respect to the particular domain. On the other hand, this can be disorienting and drawing a systematic picture of the existing literature becomes a hard task. The bibliographic analysis conducted in this section is aimed at giving a clearer overview of the available research results, by identifying common aspects and assumptions of the existing works.

In the next section before discussing the most common approaches to diagnosis of faults, it is presented an overview of the techniques that each diagnosis approach usually adopts.

5.1 Techniques adopted in fault diagnosis

In the following some techniques commonly exploited when dealing with fault diagnosis are briefly described. Diagnosis approaches illustrated in the next sections typically adopt one, or a combination, of these techniques, by customizing them according to their specific needs.

5.1.1 Bayesian Inference

Bayesian inference is a method of statistical inference in which some kind of evidence or observations are used to calculate the probability that a hypothesis may be true, or

else to update its previously calculated probability. The Bayesian inference is exploited for diagnostic reasoning as described hereafter.

Recalling the Bayesian formula:

$$P(F^* \mid e) = \frac{P(e \mid F^*)P(F^*)}{\sum_{F \in \Omega} P(e \mid F)P(F)} \tag{3.1}$$

Where **Ω** is the universe of all the possible hypothesis (e.g. all the faults listed in the fault model); *F** is the adjudged hypothesis; *P(F|e)* is the degree of belief in hypothesis *F* after observing the evidence *e* (i.e., the posterior probability of hypothesis *F*). *P(F)* is the degree of belief in hypothesis *F* before observing the new evidence *e* (i.e., prior probability of hypothesis *F*). *P(e|F)* is the probability of observing some evidence *e*, given that the hypothesis *F* is true. According to the diagnosis problem:

- *P(F|e)* is the probability that the component is affected by a certain fault (information related with hypothesis *F*) given that a certain exceptional event (an anomaly or an error) was collected (evidence *e*);
- *P(F)* is the probability that the component was affected by a certain fault (hypothesis *F*) before collecting the last evidence *e*;
- *P(e|F)* is the probability that the detection mechanism trigger the evidence *e*, given that the monitored component is affected by a certain fault (hypothesis *F*).

This last term, *P(e|F),* is in some sense related to the imperfection of the detection mechanism.

Assuming (i) all fault–related events occur at discrete points in time and that two successive points in time differ by a constant time unit (or step) and (ii) the detection mechanism evaluates at each step *i* if the component is behaving correctly or not, triggering a boolean event towards the diagnostic mechanism. Therefore the prior probability of a hypothesis *F** at step *i* can be obtained as the posterior probability of the same hypothesis *F** at the previous step *i − 1*, adjusted based on the evidence observed in the meanwhile. The probability of the monitored component being affected by a certain fault at step *i* is hence obtained based both on the probability values evaluated at round *i − 1* and the evidences collected in the meanwhile:

$$P_i(F^* \mid e) = P(F^* / e) = \frac{P(e \mid F^*)P_{i-1}(F^*)}{\sum_{F \in \Omega} P(e \mid F)P_{i-1}(F)} \tag{3.2}$$

5.1.2 Data Mining

The process of applying algorithms to data with the intention of uncovering hidden patterns is called Data mining. The extraction of patterns from data consists in data collection, storage and manipulations. Many techniques can be exploited to accomplish this task. The most relevant are: neural networks, clustering, genetic algorithms, decision trees and support vector machines (SVM). Data mining techniques have the potential to tackle voluminous data and assist in fault management. Hereafter some common methods used for data mining are introduced.

5.1.3 Neural Network

Neural networks (NN) were first created in an attempt to reproduce the learning and generalization processes of the human brain. The main strengths of NN are: the ability to easily deal with complex problems; the ability to generalize the results obtained from known situations to unforeseen situations; the ability to carry out "classifications" of the elements of a given set; low operational response times due to a high degree of structural parallelism. These characteristics allow NN's to be used in many applications. For these, NN's can represent a means to resolve not only pattern recognition problem, but also problems involving complex systems and/or require an immediate response (for example, simulation, control problems, fault detection, online fault diagnosis, and reconfiguration in fault tolerant systems). The basic element of an NN is the "artificial neuron" also known as a "Processing Element" (PE). The PEs may have a local memory and are able to process information received as input. The memory is usually "built" at an early stage, called learning phase, and no longer changes (except in the case of NN models to adaptive learning).

One of the most used schematizations is the one shown in Figure 13.

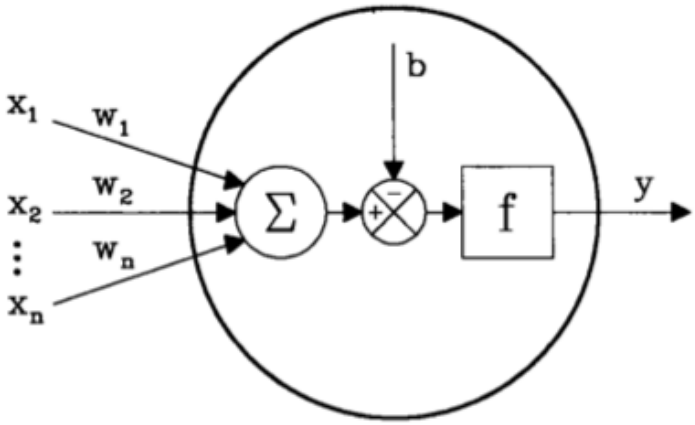

Figure 13 - General schema of NN Processing Element.

The PE *y* output is given by

$$y = f(\sum_i x_i w_i - b) \tag{3.2}$$

where x_i is the PE input supplied by weight links w_i, b is the characteristic PE offset (bias), and f is the transfer function. The transfer function in addition to changing the output of the neuron can change its local memory (typically in the learning phase). There are various types of transfer functions (e.g., step function, sigmoidal, gaussian, radial basis) which use can depend from the applications. For instance, applications that require an NN with continuous outputs, the transfer functions are sigmoidal. Lots of interconnection scheme are been proposed for NNs. Neurons are interconnected via unidirectional channels, which transmit the signals. Each neuron has one output that can send out to many connections (carrying the same signal). Typically, models for interconnection can fall into two main categories: feed-forward networks and recurrent/feedback network. The feed-forward networks are characterized by a single one-way flow (i.e., no cycles), while the recurrent networks provide feedback channels.

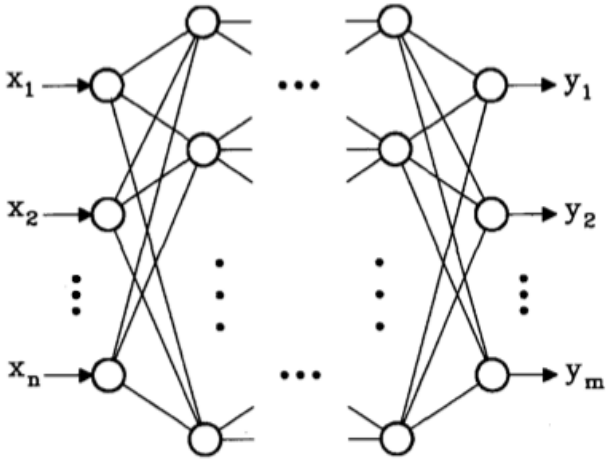

Figure 14 - Feed-forward NN.

For instance in Figure 14 is depicted the "feed forward" neural network. It is made up of a given number of layers, each of which has a specified number of nodes; the interconnections are only between nodes of adjacent layers, and each node belonging to a layer is connected to all the nodes of adjacent layers. The nodes connected directly to the network inputs all belong to the same layer, known as the "input layer," these nodes do not have offset. The nodes that furnish the network outputs also belong to a single layer, called the "output layer." The other nodes are organized in one or more layers, called "hidden layers," because they cannot directly be reached from outside the network. More details can be found in [Russell 2003].

5.1.4 SVM

Support vector machines (SVMs) are a set of learning methods that analyze data and recognize patterns. SVMs are used with significant success for classification (e.g. document classification [Larry 2002], and speech recognition [Smith 2002]) and regression analysis. Vladimir Vapnik invented the original SVM algorithm while Corinna Cortes and Vladimir Vapnik has proposed the current standard incarnation (soft margin).

To explain how SVMs work we start to discuss the case of linearly separable patterns, i.e., the pattern classification problems. In this context the problem of finding the class to which belongs the pattern is turned in solving a quadratic optimization problem: we have to find the hyperplan (see Figure 15), i.e., a decision surface, which maximize a given objective function. This objective function is related to the concept of margin of separation between patterns of the two classes. However the optimal hyperplane is limited to be used only in linear separable patterns identification.

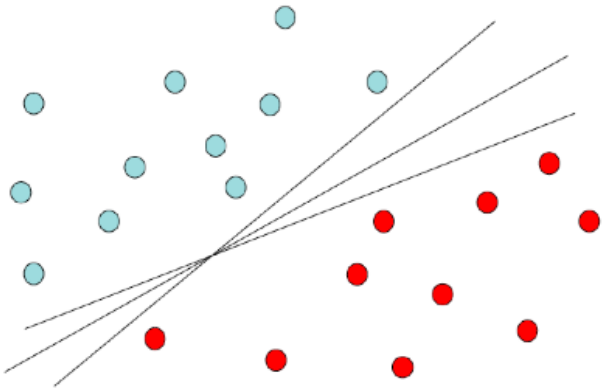

Figure 15 - Decision surface hyperplane (2 dimension).

To overcome this limitation two mathematical operations are exploited: (i) non linear mapping of the input vectors into a high-dimentional feature space (hidden from both input and output); (ii) build an optimal hyperplane (as in the linear separable problem) with the features discovered at the previous step.

The rational for these operations is based on a strong mathematical basis (see [Vapnik 1995]). Intuitively the first step tries to discover a high-dimensional space (theoretically infinite), i.e., the feature space where the pattern can be separate by a hyperplane. The second step follows the procedure of the linear separable case with the difference that the hyperplane belongs to the new feature space. To identify new patterns, they are first mapped into the feature space and then predicted to belong to a category based on which side of the hyperplane they fall on. More details are given in the [Haykin 1998].

5.1.5 Decision Tree

Decision tree induction is one of the simplest, and yet most successful forms of learning algorithm. In data mining a decision tree is used to classify instances of large amounts of data (this is also called classification tree). In this context, a decision tree describes a tree where leaf nodes represent classifications and branches all the properties that lead to those classifications. It takes as input an object or situation described by a set of attributes and returns a "decision": the predicted output value for the input. The input attributes can be discrete or continuous. The output value can also be discrete or continuous; learning a discrete valued function is called classification learning; learning a continuous function is called regression. A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with the possible values of the test. Consequently, each internal node appears to be a macro-class made by the union of the classes associated with its child nodes. The predicate, which is associated with each internal node (which is based on the distribution of data), is called the split condition. When the leaf node in the tree is reached it specifies the value to be returned.

In many situations it is useful to define a criterion for stopping (halting), or criterion of pruning (pruning) in order to determine the maximum depth. This is because the increase of the depth of a tree (or even its size) does not directly affect the goodness of

the model. In fact, an overgrowth of tree size alone could lead to disproportionate increase in computational complexity compared to the benefits regarding the accuracy of the predictions / classifications. Although decision trees are simple to manage and to understand, the performances are not good if compared to other classification methods (e.g., NN or SVM) because they cannot learn complex patterns without lose in generalization.

5.1.6 Model-based

Modeling techniques are often used to define a synthetic representation of a system, which can be useful to (i) abstract its principal proprieties, by characterizing its behavior, (ii) to correlate events and (iii) to control the system by choosing the most proper countermeasures. The models are usually developed on the basis of knowledge on the principles of operation of the system to be observed. Knowledge is expressed in mathematical terms, i.e., relations between the inputs and outputs, (quantitative models) or by qualitative features (qualitative models). Some modeling technique and formalisms commonly used when dealing with diagnosis of faults are hereafter discussed.

HMM

A hidden Markov model is a formalism able to represent probability distributions over sequences of observations; an HMM is basically a Markov chain whose state is "not observable" (the state is indeed "hidden") but emits "observable" symbols depending on a probabilistic function of the state. A good tutorial about HMM and its use can be found in [Rabiner 1990]. Let Ω be the finite discrete set of states and Σ be the finite discrete set of observable symbols; an HMM M can be expressed by the following quintuple:

$$\{\Omega, A, \pi(1), \Sigma\}$$

Ω, A and π(1) are the basic elements describing a first–order time–homogeneous DTMC17 [Bolch 2005], that is:

- the discrete set of (hidden) states $\Omega = \{\omega_1, \ldots, \omega_n\}$;
- the $n \times n$ state transition probability matrix A, where $A(i, j) = a_{ij}$ is the conditional probability that the model moves to state ω_j given that it is in state ω_i at time t;
- the initial state probability vector $\pi(1)$.

$\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ is the set of distinct (observable) symbols emitted by the HMM (the so–called alphabet of the model).

B is the $n \times m$ observable symbol probability matrix, where the element $B(i, k) = b_i(\sigma_k)$ is the conditional probability that the model emits the symbol σ_k at time t given that the model is in the state ω_i.

The use of the formalism of HMM allows both the definition of a framework through which the diagnosis problem can be approached, and the definition of a diagnostic mechanism; its application is discussed in the Section 5.2.4.

Fault Tree

Fault trees were developed in the 1960s and have become a standard tool reliability modeling. Vesely et al. [Vesely 1981] have given a comprehensive treatment of fault trees. The purpose of fault trees is to model conditions under which failures can occur using logical expressions. Expressions are arranged in the form of a tree, and

probabilities are assigned to the leaf nodes, facilitating the computation of the overall failure probability. However fault tree analysis is a statically that does not take the current system status into account. However, if the leaf nodes are combined with online error detectors, and logical expressions are transformed into a set of rules, they can be used as a rule-based online diagnosis system. Although such an approach has been applied to chemical process fault diagnosis [Ulerich 1988] and power systems [Rovnyak 1994], it has not yet applied to IT systems.

MBR

Systems that reason with causal rules are called model-based reasoning systems, because the causal rules form a model of how the observed system operates. The model-based reasoning (MBR) [Weissman 1993] [Lewis 1999] [Wietgrefe 1997] represents a system by modeling each component. A model can represent an entity or a natural or logical entity. The models for the physical entities are called functional models, and models for all the logical entities are called logical models. The description of each model contains three categories of information: attributes, relations with other models and behavior. The event correlation is the result of collaboration between the models. Once all the components of the system represented by their behavior, you can run simulations to predict the behavior of the system. However, for large and complex system, it can be hard to model properly and completely the behavior of all components and their interactions. In fact, these models do not capture all aspects of the system but only identify those aspects that help to identify system failures and shape them properly [Pieschl 2003]. The system NetExpert [NETeXPERT] is an example of MBR.

5.1.7 Automatic Debugging

Automated debugging techniques try to simply the activities commonly performed by developers during debugging providing them a set of differences between normal and faulty executions that permit to localize the fault within a software component. These techniques, commonly used during off-line diagnosis, focus on the identification of the faulty instructions but do not provide any description of the misbehavior. However such techniques when combined with others methods (e.g., log analysis, symbolic debagging) can still be useful. The most commonly used by developer are introduced as follow.

In [Zeller 1999] Zeller introduced the Delta Debugging algorithm to identify the smallest set of changes that caused a regression in a program. The technique requires the following information: a test that fails because of a regression, a previous correct version of the program and the set of changes performed by programmers. The Delta Debugging algorithm works in this way: it iteratively applies different subsets of the changes to the original program to create different versions of the program and identify the failing ones. The minimal set of changes that permit to reproduce the failure is reported. Delta Debugging has been applied in different contexts (e.g. in [Tucek 2006] discussed in the Section 5.2.7).

Another effective technique is presented in [Zeller 2002]. Given two executions of a program, one failing one not, the algorithm identifies, with the help of a debugger, a set of differences between the memory graph of the program in different execution states (for example after the start, in the middle of the execution and before the failure). The algorithm repeats a legal execution and alters the state of the program by changing the value of program variables with values in the difference set till the minimum set of

variables that lead to the failure is found. Although this approach completely automates the debugging process usually performed by developers manually, it still leaves to developers the task of understanding what is causing these differences. Other automated debugging approaches present adaptations of the Delta Debugging algorithm to identify program inputs leading to failures [Choi 2002] [Misherghi 2006] [Zeller 2002b] and to identify the fault(s) not only in case of regression faults [Sterling 2007].

Program Slicing was first introduced by Weiser in 1979 [Weiser 1979] as a decomposition technique that automatically extracts the program statements relevant to a particular computation. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a slicing criterion. Typically, a slicing criterion consists of a pair <p, V>, where p is a program point and V is a subset of program variables. The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C are called the program slice with respect to criterion C. Program slicing approaches can be grouped according to the nature of data considered to derive the slice: static, dynamic, or both. A broad description of the different slicing approaches is provided in [Baowen 2005]. Static slices are built by analyzing variables reachability through the inspection of abstract representation of the program such as Control Flow Graphs [Weiser 1984], Program Dependence Graphs [Ottestein 1984], Value Dependence Graphs [Ernst 1994], System Dependence Graphs [Binkley 2003] or Data Dependencies [Orso 2001]. Effectiveness of static slicing approaches is often limited by programming language features like indirect access, pointers, function, polymorphism and concurrency. These features make complex pointer analysis or conservative approximations necessary thus increasing both slices size and computation time. Dynamic slicing approaches have been introduced to reduce the size of the computed slice. Dynamic slicing approaches monitor the system during an execution and trace the program elements covered. Only the statements covered at runtime are considered to build the slices [Korel 1988] [Kamar 1993]: this lead to smaller slices with respect to those generated by static approaches. Despite the reduced size of generated slices, dynamic slicing is seldom applied because of the impact on system performance and memory consumption caused by the runtime monitoring. Combined dynamic-static approaches have been proposed in order to overcome the limitations of static and dynamic approaches. Hybrid slicing reduces monitoring costs by focusing only on the statements covered between the breakpoints set in a debugging session [Gupta 1995]. Other approaches reduce the slicing overhead by identifying data dependencies relationships at runtime [Umemori 2003], tracing executed method calls [Nishimatsu 1999] or reduce the amount of monitored data by executing only the statements in the static slice [Rilling 2001].

Spectra based techniques overcome the limitations of slicing techniques by identifying and comparing the set of instructions executed during correct and failing executions without considering data dependence relations [Harrold 1998]. For example developers collect the set of code blocks executed during correct and failing executions (these sets are known as Block Hit Spectra) and manually compare the blocks contained in the two sets. Code blocks present in the failing sets only are considered suspicious and are manually inspected by developers to look for erroneous instructions. Researchers adopted resemblance coefficients [Romesburg 1984] to automatically identify the suspicious instructions. Three most effective resemblance coefficients presented in

software engineering literature: Jaccard [Chen 2002], Tarantula [Harrold 2002] and Ochiai [Abreu 2007]. Resemblance coefficients are used in combination with block hit spectra to identify the program statements mostly related with the fault. Each coefficient calculates a score for each program statement that indicates how much related to the fault is. Statements are presented to developers sorted in descending order: statements with the highest score present the greater probability to contain the fault. Spectra coefficients differ for the way the score is calculated, but in general they relates the number of faulty executions in which each statement is executed or not with the number of correct executions in which the statement is executed or not. In [Abreu 2008] Abreu presents an approach that combines resemblance coefficients with model based debugging in order to filter the results provided by spectra and augment the efficacy of the analysis.

Program spectra can be quite effective in localizing faults within a component, however they are limited in case only small program portions are analyzed (e.g. unit tests during development). Furthermore these techniques only highlight suspicious faulty locations but do not provide any additional information to help in the final diagnosis of the problem.

5.1.8 Probing

A probe is a test transaction whose outcome depends on some of the system's components; diagnosis can be performed by appropriately selecting the probes and analyzing the results. For instance, in the context of distributed systems, a probe is a program that executes on a particular machine (called a probe station) by sending a command or transaction to a server or network element and measuring the response. The ping and traceroute commands are probably the most popular probing tools that can be used to detect network availability. Other probing tools, such as IBM's EPP technology ([Frenkiel 1999]), provide more sophisticated, application-level probes. For instance, probes can be sent in the form of test e-mail messages, web-access requests, a database query, and so on.

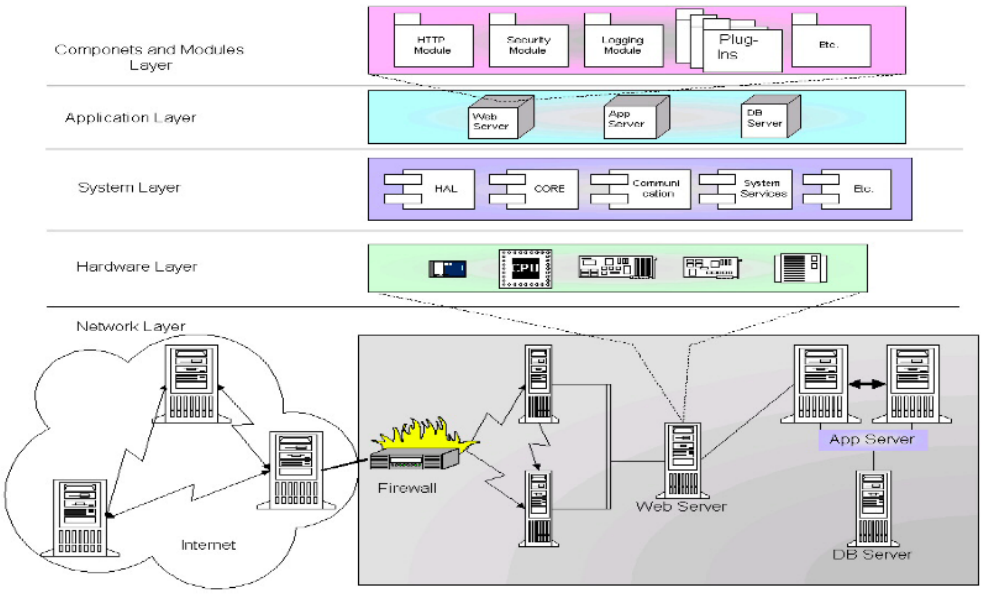

Figure 16 - Examples of probing at different layer.

Figure 16 taken from [Rish 2005] illustrates the core ideas of probing technology. The bottom left of the picture represents an external cloud (e.g. the Internet), while the greyed box in the bottom middle and right represents an example intranet e.g. a web site hosting system containing a firewall, routers, web server, application server running on a couple of load balanced boxes, and database server. Each of these contains further substructure the figure illustrates the various layers underlying the components. Probing can take place at multiple levels of granularity; the appropriate choice of granularity depends on the task probing is used for. For example, to test a Service Level Agreement (SLA) stating response time we need only probe one point of Figure 16, the point of contact of the external cloud and the intranet. In order to find more detailed information about the system one could probe all network segments as well the web server, application server, database server all the elements of the intranet in the network layer. If we need to do problem determination or tune up the system for better performance, we may also need to consider more detailed information; e.g. from the system layer (some systems allow instrumentation to get precise information about system components) or component and modules layer, and so on. For each task appropriate probes must be selected and sent and the results analyzed.

In practice, probe planning (i.e., choice of probe stations, targets and particular transactions) is often done in an ad-hoc manner, largely based on previous experience and rules-of-thumb. Thus, it is not necessarily optimized with respect to probing costs (related to the number of probes and probe stations) and diagnostic capability of a probe set. More recent work [Brodie 2001], [Rish 2002] on probing-based diagnosis focused on optimizing the probe set selection and provided simple heuristic search techniques that yield close-to-optimal solutions. However, the existing probing technology still suffers from various limitations:

1. Probes are selected off-line (pre-planned probing), and run periodically using a fixed schedule (typically, every 5 to 15 min). For diagnostic purposes, this approach can be quite inefficient: it needs to construct and run repeatedly an unnecessarily large set of probes capable of diagnosing all possible problems, many of which might in fact never occur. Further, when a problem occurs, there may be a considerable delay in obtaining all information necessary for diagnosis of this particular problem. Thus, a more adaptive probe selection is necessary.
2. Another limitation of existing techniques, including both event correlation [Kliger 1997] and probing [Brodie 2001], [Rish 2002], is their non-incremental ("batch") processing of observed symptoms (alarms, events, or probes), which is not quite suitable for continuous monitoring and real-time diagnosis. An incremental approach is required that continuously updates the current diagnosis as more observations become available.

Finally, existing approaches typically assume a static model of the system (i.e., the system state does not change during diagnostic process). While "hard" failures of components are indeed relatively rare, "soft" failures such as performance degradations (e.g., response time exceeding certain threshold) may happen more frequently; in a highly dynamic system 'failure" and "repair" times may get comparable with the average diagnosis time. This can lead to erroneous interpretation of some contradictory observations as "noise" [Kliger 1997] when in fact they may indicate changes in the

system. A more sophisticated model that accounts for system dynamics can provide a more accurate diagnosis in such cases.

5.2 Approaches to Fault Diagnosis

In this section a broader classification of diagnosis approach is discussed, then some of most common approaches, which exploit one or a combination of the previous discussed techniques, are surveyed.

5.2.1 Offline vs. Online approaches

The diagnosis approaches, which we are going to discuss, can be broadly grouped into two categories: off-line and on-line.

A large slice of the work on software faults diagnosis proposed *off-line diagnosis* approaches. Off-line approaches consist of two distinct operations. On one hand, the analysis of data gathered from (i) monitoring devices and logs during system execution and (ii) reports of off-line diagnostic tests (i.e., core dump, cache, RAM and CPU tests). On the other hand, there is the inference process, which provides the fault hypothesis, i.e., the cause and type of occurred failure. Starting from the output of the diagnosis some maintenance activities must be performed to fix the problem (e.g., software update, component overhaul/replacement, etc.) and preventing it to occur again. Off-line approaches assume that the system, or one of its parts, is no more operative or that the monitored components are temporarily isolated from the rest of the system (thus depriving it of some resources). Moreover expertise involvement could be required to conduce the diagnosis activities. The category of off–line analysis procedures includes (i) all automatic and semi-automatic debugging technique and (ii) more or less sophisticated procedures which analyze system error logs to derive trend analysis and predictions of future permanent faults, as proposed in [Lin 1990], [Iyer 1991]. However some issues arise when performing off-line diagnosis. For instance, it could be difficult to reproduce a failure during off-line diagnosis because of different conditions of in-house environment or because the occurred failure is due to the activation of elusive fault(s).

On–line approaches are performed while the system is still running and rely on the continuous analysis of data coming from on-line monitoring and detectors. They require that part of system resources (e.g. time, CPU cycles, etc.) are dedicated to the diagnosing activity and, most important, require the use of not-destructive detection mechanisms; sometimes these requirements cannot be met. Since on–line mechanisms work while the monitored system is working, the diagnostic judgments can be exploited at run–time to act on the system by triggering the most proper countermeasures aiming to tolerate the diagnosed fault(s). On–line diagnostic mechanisms include a variety of threshold–based over–time diagnostic mechanisms [Spainhower 1992] [Mongardi 1993], [Bondavalli 2000], [Bondavalli 2004], [Daidone 2006], [Serafini 2007], [Nickelsen 2009], using either heuristic or probabilistic approaches.

With the respect to the off-line case, on-line diagnosis has much more rigid timeliness requirements to give the fault hypothesis. Moreover, since the system is still running, only non-invasive diagnostic tests (i.e., test performed to check the presence of errors) can be performed because the state of the system has to be preserved at the completion of the tests (e.g., in power grid context, we cannot use on-line the Degree of

Polymerization test [US BR 2000] for assessing the paper degradation in a transformer). On-line diagnosis may also need to dedicate a part of system resources (e.g., a fraction of operational cycle [Coccoli 2003]) to the diagnostic activities, thus penalizing system performance. In [Coccoli 2003] the problem of choosing between an on–line or off–line approach for implementing Diagnosis in a control systems for critical applications has been analyzed.

5.2.2 One-shot approach

The model proposed by Preparata et al. (also known as the PMC model) assumed the system to be made up of several components, which test one another. The fault model assumed is permanent and independent faults. Test results are leveraged to diagnose faulty component(s). This model is still the subject of a lot of research work (e.g. [Manik 2009]).

[Barsi 1976] is another seminal work, which is based on many premises of the PMC model; the "BGM model" differs from the PMC model because of the interpretation of test results, simplifying the diagnostic procedure. Also the BGM model is still the subject of a lot of research work (e.g. [Vedeshenkov 2002]). In the above mentioned works diagnosis was based on the so called "one–shot diagnosis of a collected syndrome", so that the component was diagnosed as "failed" as soon as an error was observed, immediately triggering reconfiguration operations devoted to isolate, repair or replace the failed component.

5.2.3 Heuristic approach

Heuristic approaches are simple mechanisms suggested by intuitive reasoning and then validated by experiments or modeling; for example, they count errors, and when the count crosses a pre–set threshold a permanent fault is diagnosed. An example of heuristic mechanism for the discrimination between transient and permanent faults is the α–count mechanism [Bondavalli 2000].

These approaches emerged in those application fields where components oscillate between faulty and correct behavior because a large fraction of faults are transient in nature; in those application fields one–shot diagnosis is not effective, because the cost for treating a transient fault as a permanent one is very disadvantageous. The idea of counting the number of errors accumulated in a time window has long been exploited in IBM mainframes: in the earlier examples, as in IBM 3081 [Tendolkar 1982], the automatic diagnostics mechanism uses the above criteria to trigger the intervention of a maintenance crew; in later examples, as in ES/9000 Model 900 [Spainhower 1992], a retry–and–threshold scheme is adopted to deal with transient errors. In the TMR11 MODIAC railway interlocking system by Ansaldo, the architecture proposed in [Mongardi 1993], two failures experienced in two consecutive operating cycles by the same hardware component, which is part of a redundant structure, make the other redundant components consider it as definitively faulty. In [Sosnowski 1994] big attention is given to the error detection and error–handling strategies to tolerate transient faults: a fault is labeled as transient if it occurs less than k times in a given time window.

The most sophisticate representative of the heuristic mechanisms is α–count

[Bondavalli 2000]. It is a general approach that encompasses all the heuristic presented above. The details about the α–count mechanisms are given hereafter.

The α–count heuristic implements the "count–and–threshold" criteria. It has been proposed to discriminate between transient, intermittent and permanent faults. The fallowing assumption are made: many repeated errors within a little time window are easily evidence for a permanent or an intermittent fault, while some isolated errors collected over–time could be evidence for transient faults. α–count hence counts error messages and signals collected over–time, raising an alarm when the counter passes a predefined threshold, and decreasing the counter when not-error events are collected.

Assuming that (i) all fault–related events occur at discrete points in time and that two successive points in time differ by a constant time unit (or step) and (ii) the detection mechanism evaluates at each step *i* if the component is behaving correctly or not, triggering a boolean event, $J^{(i)}$, towards the diagnostic mechanism. $J^{(i)}$ is defined as follows:

$$J^{(i)} = \begin{cases} 0 \text{ if correct behavior is detected at step i} \\ 1 \text{ if incorrect behavior detected at step i} \end{cases}$$

α–count collects over–time the signals coming from the deviation detection mechanism in a score variable α associated with the monitored component; at each step the score variable α is updated and compared with a threshold value α: if the current score value exceeds the given threshold α ($\alpha \geq \alpha_T$), the component is diagnosed as affected by a permanent or an intermittent fault, otherwise it is considered healthy or hit by a transient fault.

The score variable α is evaluated at each step *i* by using the following formula:

$$\alpha^{(0)} = 0$$

$$\alpha^{(i)} = \begin{cases} \alpha^{(i-1)} K \ \ if\ J^{(i)} = 0 \qquad (0 \leq K \leq 1) \\ \alpha^{(i-1)} + 1 \ \ if\ J^{(i)} = 1 \end{cases}$$

K is the rate at which memory of past errors disappears. Extended studies about the tuning of the internal parameters (K and α_T) are described in [Bondavalli 2000], where the trade–off between promptness and accuracy is evaluated. A discussion about the dynamic selection of thresholds for QoS adaptation could be find in [Casimiro 2008]. Applications and variants (e.g. the double–threshold variant12) are described in [Bondavalli 2004][Serafini 2007]. Practical applications of the α–count heuristic can be found also in the following works: [Powell 1998], where α–count has been implemented in the GUARDS13 architecture for assessing the extend of the damage in the individual channels of a redundant architecture; [Romano 2002], where α–count is used in a COTS based replicated system to diagnose replica failures based on the result of some voting on the replica output results.

5.2.4 Probabilistic approach

Probabilistic approaches are tailored to evaluate the probabilities of the monitored component being faulty, based on information collected. Diagnostic mechanisms following a probabilistic approach are tailored to evaluate over–time the probability of the monitored component being faulty. Moreover they reach a good compromise

between diagnosis accuracy and responsiveness because they use the available knowledge about the system to improve diagnosis accuracy in the best way [Pizza 1998].

Different fault models are used for diagnosis. For instance, [Pizza 1998] takes into account permanent, transient and propagation fault (i.e., an error occurred into a component activate a fault into another component). The authors use a diagnostic mechanism which exploits the Bayesian inference to compute the probability *P(f)* of the monitored component being affected by a certain fault (hypothesis *f*) is computed. This probability is updated when new detection event (evidence *e*) comes from the detection mechanism. Then the evaluated probabilities are exploited to trigger the proper countermeasures. In [Sanders 2005] an automatic model driven recovery in distributed systems is proposed. Authors exploit a set of a limited coverage monitors whose outputs are correlated to diagnose the faulty component. The reasoner exploits the Bayesian inference to update the probabilities of having a faulty component.

[Nickelsen 2009] applies Bayesian Networks (BN) for probabilistic fault diagnosis for TCP end–to–end communication both in the wireless and wired domain. Bayesian networks are used to infer system insights and a priori system knowledge, by using probabilistic reasoning systems. The structures and initial parameters of the Bayesian networks depend on the prior knowledge of the domain expert, however if the system is properly modeled, these techniques lead to high diagnosis quality, in terms of accuracy and latency.

Other examples of diagnosis, which follows the probabilistic approach, is the mechanism based on Hidden Markov Model (HMM) [Daidone 2006] presented as follow. Daidone et al. proposed using a hidden Markov model approach to infer whether the true state of a monitored component is healthy or not. The diagnostic mechanism thus follows the probabilistic approach presented in the previous section, taking advantage of the peculiarities of the hidden Markov models in order to give an intuitive but formal way of solving the diagnosis problem. Observation symbols are associated to outcomes of component probing, and hidden states (i.e., not observable directly) are related to the component's true state. The true state (in a probabilistic sense) was inferred from a sequence of probing results by the so-called forward algorithm of hidden Markov models. The diagnostic mechanism is flexible enough to take into account the uncertainty of both observed events and observers, so that it can be also used to diagnose malicious attacks: e.g., in [Jecheva 2006] HMMs are used to implement an anomaly-based intrusion detection system. In order to better comprehend the rationale behind the diagnostic framework based on HMM, more details can be found in [Daidone 2009].

To avoid problems related to event correlation, alternative probabilistic-based diagnostic approaches rely on end-to-end probing technology [Frenkiel 1999], [Brodie 2001], [Rish 2002].

However probe planning (i.e., choice of probe stations, targets and particular transactions) is often done in an ad-hoc manner, largely based on previous experience and rules-of-thumb. Thus, it is not necessarily optimized with respect to probing costs (related to the number of probes and probe stations) and diagnostic capability of a

probe set. More recent work [Brodie 2001], [Rish 2002] on probing-based diagnosis focused on optimizing the probe set selection and provided simple heuristic search techniques that yield close-to-optimal solutions. However, the existing probing technology still suffers from various limitations:

1. Probes are selected off-line (pre-planned probing), and run periodically using a fixed schedule (typically, every 5 to 15 min). For diagnostic purposes, this approach can be quite inefficient: it needs to construct and run repeatedly an unnecessarily large set of probes capable of diagnosing all possible problems, many of which might in fact never occur. Further, when a problem occurs, there may be a considerable delay in obtaining all information necessary for diagnosis of this particular problem. Thus, a more adaptive probe selection is necessary.
2. Another limitation of existing techniques, including both event correlation [Kliger 1997] and probing [Brodie 2001], [Rish 2002], is their nonincremental ("batch") processing of observed symptoms (alarms, events, or probes), which is not quite suitable for continuous monitoring and real-time diagnosis. An incremental approach is required that continuously updates the current diagnosis as more observations become available.
3. Finally, existing approaches typically assume a static model of the system (i.e., the system state does not change during diagnostic process). While "hard" failures of components are indeed relatively rare, "soft" failures such as performance degradations (e.g., response time exceeding certain threshold) may happen more frequently; in a highly dynamic system 'failure" and "repair" times may get comparable with the average diagnosis time. This can lead to erroneous interpretation of some contradictory observations as "noise" [Kliger 1997] when in fact they may indicate changes in the system. A more sophisticated model that accounts for system dynamics can provide a more accurate diagnosis in such cases.

In [Rish 2005], the focus is on a more adaptive and cost-efficient technique, called active probing, that is based on information theory. It avoids the waste inherent in the pre-planned approach since always selects and sends probes as needed in response to problems that actually occur. Also, active probes are only sent few times in order to diagnose the current problem and can be stopped once the diagnosis is complete. Only a relatively small number of probes needed for fault detection should circulate regularly in the network. Combining probabilistic inference with active probing yields an adaptive diagnostic engine that "asks the right questions at the right time", i.e. dynamically selects probes that provide maximum information gain about the current system state. They approach diagnosis problem as the task of reducing the uncertainty about the current system state X (i.e., reducing the entropy $H(X)$) by acquiring more information from the probes, or tests T. Active probing repeatedly selects the next most-informative probe T_k that maximizes the information gain $I(X; T_k | T_1, ..., T_{k-1})$ given the previous probe observations $T_1, ..., T_{k-1}$. Probabilistic inference in Bayesian networks is used to update the current belief about the state of the system $P(X)$.

5.2.5 Rule-based approach

The diagnosis based on rule-based approaches uses a set of rules (also called *knowledge*) for event correlation. These methods were developed in the mid-1970s and formed the basis for a large number of expert systems in medicine and other areas [Russel 2003]. The *knowledge* can be represented by rules of the form "IF condition THEN action" and formalized by the use of the first order logic or more powerful expressive languages (e.g., OWL). The *condition* uses the events received and the information system functions, while *action* contains proper countermeasures that can lead to system changes or the use of parameters for choosing the next rule.

Li and Malony applied Rule-based reasoning, by using rules in the CLIPS expert system, based on architectural patterns [Li 2006] for performance bottleneck diagnosis. Benoit et al. applied the same approach for database system diagnosis [Benoit 2003].

Another rule-based approach is proposed in Hatonen et al. [1996]. It is based on reported error events. The authors described a system that identifies episode rules from error log. They correlate errors in log by counting the number of similar sequences. For instance an episode rule expresses temporal ordering of events in the form "if errors A and B occur within five seconds, then error C occurs within 30 s with probability 0.8." Several parameters such as the maximum length of the data window, types of error messages, and ordering requirements have to be specified a-priori. However, the algorithm returned too many rules such that they had to be presented to human operators with system knowledge in order to filter out informative ones.

In the field of power system process fault diagnosis a rule-based on-line diagnose has been proposed in [Rovnyak 1994]. Here the authors exploited fault tree analysis: the leaf nodes are combined with online error detectors, and logical expressions are transformed into a set of rules.

An advantage of rule-bases approaches is that rules are easily readable by the user, and then their effect is intuitive. The rule-based correlation is suitable for cases in which relations between events are well known and can be formulated clearly. Despite the advantages of rule-based systems, there are some limitations on the frequent changes in the environment that could be modeled. This forces the user to make updates to the rules. In other words, if the system changes rapidly, it is hard to maintain an accurate set of rules. Moreover, these approaches do not deal with incomplete and incoherent error messages or signals of a complex software system, and thus it is difficult to determine the faulty component among the inferred possibilities.

5.2.6 Machine Learning approach

Accurate diagnosis of faults in complex systems requires acquiring data, processing the information using advanced algorithms, extracting required features for efficient identification of faults. Machine learning approach which involve feature extraction, feature selection and classification of faults give a systematic methodology to fault diagnosis and can be used in automated or unmanned environment. Especially classification algorithm can be used to deal with diagnosis problem. The most significant works are surveyed as follow.

The goal of identifying the root cause of a failure automatically is pursued in [Yuan 2006] by means of statistical learning techniques, based on Support Vector Machines

(SVMs). Authors propose to trace low-level system behaviors to deduce problems of computer systems. Events occurring in the system (e.g., system calls, I/O requests, call stacks, context switches) are collected in order to (i) identify the correlations between system behaviors and known problems and (ii), use the learned knowledge to solve new coming problems. [Murray 2003] applied SVMs and naive Bayes approach in order to diagnoses failures of hard-disk drives. In the case of hard-disk failure prediction, five successive samples of each selected SMART attribute set up the input data vector. The training procedure of SVMs adapts the classification boundary such that the margin between the failure-prone and non-failure-prone data points becomes maximal. In [Carrozza 2008] a diagnosis framework exploiting an SVM supervised classifier to determine the source of failure, (i.e., the faulty component) is proposed. In the training phase a large number of features are extracted from anomaly-based detector outputs and application and system logs. The anomaly-based detectors are tailored to analyze data gathered from monitors, which observe low-level OS and components variables such as the scheduling time between processes, the number of syscall errors, the throughput, the waiting time on mutex and semaphores, etc. During the operational phase, when a detector triggers a *detection event,* features are extracted from the system. This detector exploits the Bayesian Estimation to compute the conditional probability that a fault has been activated, given the received alarms. An alarm, A_i, here, represent the event: an anomaly (e.g., number of syscall errors > T_i) has been detected. This approach seems promising because the detection and diagnosis infrastructure are not application specific, however it needs further improvements for the Feature Selection issue (they could jeopardize the diagnosis) and the threshold mechanism (the static threshold approach is not well suited for dynamic systems).

Bayesian classification algorithms can be used too. However they requires that input variables take on discrete values. Therefore, monitoring values are frequently assigned to finite number of bins [Hamerly 2001]. However, this can lead to bad assignments if monitoring values are close to a bin's border. Fuzzy classification addresses this problem by using probabilistic class membership. [Turnbull 2003] used radial basis functions networks (RBFNs) to classify monitoring values of hardware sensors such as temperatures and voltages on motherboards. More specifically, all N monitoring values occurring within a data window were represented as a feature vector which was then classified as belonging to a failure-prone or non-failure-prone sequence using RBFNs. Experiments were conducted on a server with 18 hot-swappable system boards with four processors, each. The authors achieved good results, but failures and non-failures were equally likely in the data set. In [Berenji 2003] Authors used an RBF rule base to classify whether a component is faulty or not: using Gaussian rules, a so-called diagnostic model computes a diagnostic signal based on input and output values of components ranging from zero (fault-free) to 1 (faulty). The rule base is algorithmically derived by means of clustering of training data, which consists of input/output value pairs both for the faulty and the fault-free case. The training data is generated from so-called component simulation models that try to mimic the input/output behaviour of system components (fault-free and faulty). The same approach is then applied on the next hierarchical level to obtain system-wide diagnostic models. The approach has been applied to model a hybrid combustion facility developed at the NASA Ames Research Centre. The diagnostic signal can be used to predict slowly evolving failures.

A decision tree based technique is presented in [Zheng 2004] to diagnose problems in Large Internet Services. Runtime properties of the system (e.g., clients requests) are monitored; automated machine learning and data mining techniques are used to identify the causes of failures. To validate the proposed approach they measure only precision and recall, without consider false alarm rate. In [Kiciman 2005] Authors proposed constructing a decision tree from runtime paths in order to diagnose faulty components. The term runtime path denotes the sequence of components and other features for identification of the node (e.g., IP addresses of server replicas in the cluster). Runtime paths are obtained using Pinpoint. Having recorded a sufficiently large number of runtime paths including failed and successful requests, a decision tree for classifying requests as failed or successful.

In [Chen 2002] Pinpoint diagnosis framework is proposed with the goal of recognizing the most likely faulty component of a distributed system. It employs statistical learning techniques to "learn from system behavior" and to diagnose failures in a Web farm environment. This approach relies on collecting traces of system run-time behaviour, with respect to different client requests. Then data mining algorithms are used to identify the components most relevant to a failure. The major limitations of this approach are that (i) it is suitable only for small-scale software programs, and (ii) it exhibits a significant logging. Thus only off-line diagnosis could be performed.

5.2.7 Other approaches

Triage

Authors in [Tucek 2006] propose a protocol for automatic on-line diagnosis of software failures (TDP), which occur during production runs. The idea is to exploit a lightweight checkpoint to reproduce the failure condition and execute a first step diagnosis to help subsequent effort to fix the bug. The key aspects are summarized as follow:

- Capture variable related to the moment of failure: the failure state and the failure-triggering environment. The aim is to perform diagnosis immediately after the failure.
- Exploit sandboxed re-execution to reduce normal run overhead (rather than instrumenting continuously system). Support for rollback and re-execution from a recent checkpoint allows analysis only after a failure has occurred and greatly reduces the overhead. Also, failure symptoms and the results of previous analysis can allow a dynamic determination of what needs to be done.
- Automatically perform human-like top-down failure diagnosis. Based on the failure symptom and previous diagnosis results, TDP automatically decides the next step of instrumentation or analysis.
- Combine TDP with re-execution, to allow many existing (or future) bug detection and fault isolation techniques to be applied.

Delta generation and delta analysis are exploited to speculatively modify the inputs and execution environment to create many similar but successful and failing replays to identify failure-triggering conditions. Although it seems a promising approach, some assumptions limit its applicability. For instance, when a fault is activated the rollback to the previous checkpoint presumes that a fault has not yet been activated. Although the checkpoint-approach permits to reduce the overhead of other methods that monitor

heavily the environment, it may be unfeasible for applications, which require many operations that cannot be repeated and thus it is needed to record them. Moreover, such an approach cannot deal with *elusive fault* because their conditions of activation are hard to identify, thus to reproduce. Furthermore, long time is required to complete the process.

6 RECOVERY AND RECONFIGURATION

In the system dependability the recovery concerns the capability of the system to evolve from a state containing errors and faults in a state without them. The system recovery can be aimed by using techniques for both errors and faults handling. The former usually eliminates the system errors through either a *rollback* approach, which regresses the system to a "safety" state before the error detection called *checkpoint*, or a *rollforward* approach, which evolves the system in a new state without errors. The fault handling, instead, prevents that a localized fault can occur again in the system. The detection and the isolation are the first two steps in a fault management process. The system reconfiguration mainly concerns the capability of the fault-tolerant system either to switch from failed components to spare components, or to reassign system tasks among the non-failed components. Usually a corrective maintenance is required after the faults are handled in order to definitively remove them from the system.

Usually the least costly and most effective recovery action is triggered based on an estimate of the nature of the fault affecting the system [Bondavalli 2004]. Such approach is adopted in order to limit error accumulation and ultimately contrast the propagation of failures among the system components. Note that, in many cases, given two recovery actions R_i and R_j (where R_j has a greater cost than R_i), the execution of R_j also fixes inconsistent states which could be cured by R_i.

Underlying any reconfiguration techniques is the basic concept of the redundancy. It provides "alternative" solutions, which allow the system to operate even if a failure occurs. The redundancy can be implemented through the use of parallel components. Such technique, called hot redundancy, requires that two or more elements with the same functionalities simultaneously work at the same job. An alternative redundancy approach consists to have an element on-line and another one powered down which can be switched in use, either automatically or manually, whenever the on-line component fails. Such approach is called standby redundancy or cold redundancy.

The growing complexity of the systems requires more and more to define innovative strategies for their recovery and reconfiguration. Unfortunately, mechanisms based on single strategy usually result inappropriate for correctly restoring the functionalities of a systems as whole. In particular, in the area of Critical Infrastructure, for example, the systems are complex distributed architectures, consisting of heterogeneous components and technologies interacting among them, which can be affected by specific faults and errors. Clearly, such peculiarity calls for recovery mechanisms, which can be capable to operate on single component of the architecture, by reducing the risks that a single fault affecting a part can be propagated to the overall infrastructure.

In the recent years multi-level reconfiguration approaches has been more and more adopted in order to improve the capabilities of a system to properly recover from a fault. At each level a different strategy is implemented in order to adequately response to the faults. In order to improve the reliability of the recovery mechanisms, any strategy should be completely independent from the other strategies at different levels.

It is possible to identify four main levels where recovery mechanisms can be implemented: Network-level, OS-level, Middleware-level, Application-level. In the following section we will analyse the strategies proposed for each level.

6.1 OS Recovery and Reconfiguration

The Operating System (OS) is a fundamental layer of any computer system. OSs provide services, such as process scheduling, memory/network management, device interfacing, and filesystem organization: current software systems strongly rely on these services.

OSs are inherently concurrent since they allow managing several hardware devices and user processes simultaneously. Furthermore, they are made up of millions of lines of code providing support services/libraries to the applications. As a result, OSs are prone to software faults like every complex software system. This issue has been shown by studies on field failure data and static code analysis [Sullivan 1991][Chou 2001].

A first class of systems is represented by OSs enhanced with fault tolerance mechanisms. This class of OSs has not been designed to be fault tolerant, but their native architecture is extended with additional components that increase dependability. Nooks [Swift 2005] provides isolation for device drivers in order to prevent kernel memory corruption and the vast majority of driver-caused crashes. This is achieved with little or no change to existing driver and system code. In particular, Nooks isolates drivers within lightweight protection domains inside the kernel address space, where hardware and software prevent them from corrupting the kernel. Nooks also tracks a driver's use of kernel resources to hasten automatic clean-up during recovery. Nooks has subsequently been extended with shadow drivers. A shadow driver is a kernel agent that (1) conceals a driver failure from its clients, including the operating system and applications, and (2) transparently restores the driver back to a functioning state. In this way, applications and the operating system are unaware that the drivers failed, and hence continue executing correctly themselves.

Other works focus on fast recovery techniques. For example, in [Baker 1992] the "recovery box" technique is proposed to increase availability by storing system data in a stable area of memory, which is exploited to avoid a full reboot. In [Cotroneo 2009], hang detection in the Linux kernel is improved by introducing a monitor implemented as a kernel module. The module tries to detect whether the system is stuck or not, by monitoring the interface between the drivers and the kernel. However, this approach was not aimed at hang detection in individual OS components. In [Wang 2007], the support of special hardware (e.g., performance counters in the CPU) is exploited to detect hangs in the OS and in user programs. In [Pelleg 2008], a machine learning approach is proposed to identify anomalies in virtual machines, by monitoring information collected by the hypervisor such as CPU usage and I/O operations. However, this approach comes with the overhead and complexity of virtualization, and it needs to be preliminarily tuned for a specific system workload. Another class of systems is represented by OSs designed to be reliable. For these OSs, the design is driven by specific reliability requirements. For example, the kernel memory protection feature has been added into the design of the Windows 2000 OS since, in the previous releases of the Windows OS, errant applications could access the kernel space during

the installation procedure. Consequently, in order to protect the kernel space from misbehaving applications Windows 2000 has been designed with a kernel address space protection [Murphy 2000]. This approach is effective to increase the availability of Windows 2K systems: a comparative analysis of Windows NT and 2K machines proposed in [Simache 2002] shows that the number of reboots needed to recover from system failures is reduced by 16%.

Minix 3 is a microkernel OS whose design is specifically devoted to achieve high reliability. Its architecture has been recently revisited and loosely resembles Minix 2. The first improvement was to execute all the servers and device drivers in user space on the top of a microkernel [4, 16]. Minix 3 has the capability to recover faulty device drivers without user intervention [9]. This feature is achieved by means of a dedicated component (namely Reincarnation Server, RS, or Driver Manager) that monitors drivers and recovers them when needed. If a monitored process is stuck or unexpectedly exits, RS restarts it. In [Mancina 2008], a resource reservation framework is introduced in order to add "temporal protection" to Minix 3. A new server (Constant Bandwidth Server, CBS) guarantees that a process executes within a given interval, and no one can monopolize the CPU. CBS implements a CPU reservation algorithm that replaces the native Minix scheduler. The reliability-driven design also involved the Minix Inter Process Communication system (IPC) [Herder 2008]. The native IPC is enhanced with new primitives and rules that restrict the communication among trusted processes (e.g., file system server) and untrusted ones (e.g., drivers). In [Herder 2009], the device driver isolation mechanism has been assessed using a practical approach based on extensive SoftWare-Implemented Fault-Injection (SWIFI) testing. SWIFI testing was also helpful to fix rare bugs that occurred even in the microkernel, and the results showed the high dependability level achieved by Minix 3. In [Herder 2009.B] a filter driver, settled between the file system and disk driver, controls the integrity and the sequence of exchanged messages. The filter driver aims at protecting the file system from data corruption. Moreover, in Minix 3 RS constantly monitors device drivers by sending heartbeat messages. The RS heartbeat mechanism has been improved in [Cotroneo 2010]; it evolved in an adaptive heartbeat mechanism, in which the timeout is estimated from past response times.

6.2 Middleware Recovery and Reconfiguration

Reconfiguration and Recovery aim at returning the middleware to a safe state by counter measuring the occurred failure. As mentioned previous in this document, there can be two different kinds of malfunctioning that needs to be taken care. First, we have message losses that compromise the quality of the event notifications; second, there are disconnections due to link and/or broker failures.

In the first case, the correct history of received messages is recovered by commencing retransmission actions, as in ARQ schemes [Lin 1984] or Gossiping Protocols [Costa 2000]. Specifically, if a sender is aware that one of its destinations has lost a certain message, it promptly resends it. Otherwise, if a destination detects a loss, it can ask for retransmission to the sender, as in the ARQ schemes, or it can ask to other destinations that have successfully received the dropped message, as in the Gossiping Protocols.

In the second case, broker and link can be reconfigured by replacing the failed element with another one. Reconfigurable Dispatching System (REDS) [Jafarpour 2008] performs link substitution for reconfiguring the broker network. Specifically, It uses subscription and unsubscription commands to reconfigure the topology, when one of the previous events are detected by the listeners. In particular, link removal is addressed using unsubscriptions, while link insertion is carried out in a dual way by using subscriptions. Another reconfiguration approach is adopted by Reliable Routing of Event Notifications over Peer-to-Peer [Mahambre 2008]. Its key idea is to have several paths from a source to a destination, which differ each other with respect to route reliability. When the monitoring of a link returns notification of a strong degradation and may compromise the satisfaction of the application requirements, then the middleware switch from the current link to a new one (among the several known ones), with a suitable quality level. The work presented by M.A. Jaeger et al. in [Jaeger 2008] proposes a novel approach, called Self-Stabilizing Publish/Subscribe (referred as SelfStab in this section), to perform topology reconfigurations of notification architecture due to link and node crashes (the fault assumptions of this work do not include notification losses) by using self-stabilization theory. Such theory has been introduced by E. W. Dijkstra in his work [Dijkstra 1974] dated back to 1974, and later on S. Dolev has drawn on this concept in [Dolev 2000] to design self-stabilizing distributed systems able to recover from the faults that brought them to an arbitrary and incorrect state. SelfStab provides tolerance to node and link crashes by assuming the routing information within each broker as leased, i.e., it is valid only for a time period, whose expiration causes its removal. If clients want to keep such information within the brokers, they have to renew the lease by resubmitting their subscriptions to the Notification Service. Switching from a failed broker to a new one implies a high failover time, due to the time needed to select a new broker and to re-establish all the connections and the internal state of the broker (made of routing and subscribing information and past received messages). To speed-up this process, a solution is to have a passive replication of brokers, so that when the master broker fails, the reconfiguration implies only to elects one of its replicas as master without needing to re-establish the internal state of the broker. Such solutions in adopted in several approaches, such as XNET [Chand 2004].

6.3 Network Recovery and Reconfiguration

Communication network represents a fundamental component of any complex infrastructure. A distributed system, indeed, requires a reliable communication network to correctly operate. For this reason network-level recovery strategies become mandatory.

A communication infrastructure can be affected by a variety of failures including unintentional failures, due to natural disasters, software bugs, human errors, and intentional failures, usually caused by attacks or sabotages. Generally the faults can affect either the transmission or switching elements. Unfortunately not all the possible faults affecting a communication infrastructure can be prevented. According with this assumption, alternative strategies are needed to increment the network reliability.

Although duplication of vulnerable network elements, most helpful in hardware

failures, and dual homing principle, which duplicates the network access link, are common solutions widely adopted in the modern communication networks, in many cases these are insufficient for guaranteeing an acceptable network reliability level. Furthermore, such approaches are extremely expensive, and they are unable to differentiate between critical traffic and less important traffic.

Innovative recovery mechanisms, or *resiliency schemes*, have been designed in the last years for increasing the network reliability. When a fault is detected these mechanisms automatically divert the traffic allocated on the path where the failure occurs on an alternative fault-free path. In this way the traffic can reach the destination also when a fault happens, and this assures the availability of the services, especially the critical ones, transported by the network infrastructure. In contrast with the common duplication approaches, the resiliency schemes adopt network level solutions instead of element level mechanisms.

Under normal conditions the data are transported along the *working* or *primary paths*. When a fault is detected at a network element belonging to the primary path, a recovery mechanism is activated to remedy such problem. Through a resiliency scheme the working path subject to the fault, the so-called *recovered network segment*, is entirely or partially replaced by a *recovery* or *alternative path*. All the traffic flowing into the damaged element is diverted along the backup path at the *recovery head-end* (RHE), the last node in the working path shared with the recovery path. This operation is called *switch-over operation*. When the traffic passes the *recovery tail-end* (RTE), the first node beyond the fault point shared between the working and the recovery paths, it is again forwarded along the primary path. In order to assure that a single fault does not affect both the recovery and primary path, *diverse routing* solutions are adopted for selecting a backup path completely disjointed from the working path.

Any network recovery mechanism needs some requirements on the network infrastructure in order to be implemented. For example topology requirements are considered since for any failure to recover, an alternative route is need for serving as backup path. Such requirement is fundamental to avoid single point of failure, which can disconnect part of the network from the rest when a fault occurs. Also QoS parameters must be preserved by any resiliency scheme. An equivalent QoS should be assured when the traffic is rerouted on alternative path. This imposes further requirements on the network resources. Indeed, a recovery scheme needs additional capacity in term of available bandwidth, the *backup* or *spare capacity*, to satisfy QoS requirements. Furthermore, recovery mechanism must minimize the increase of propagation delay from source to destination.

A wide variety of recovery and reconfiguration mechanisms exist. Every solution has its strengths and weaknesses. In order to evaluate these mechanisms several design and evaluation criteria have been defined. Firstly, a recovery mechanism should be designed for covering a well-known failure scenario. Typically, the more are the concurrent faults covered by a resiliency scheme, the higher is the network availability that can be guaranteed.

Another important parameter for the evaluation of a network recovery mechanism is the percentage of coverage. In some scenario, indeed, a fault in a specific network area can be entirely or partially solved by the recovery mechanism. For example a fault occurring at edge node can compromise the network availability more than a fault

involving a core node. An appropriate strategy should provide high percentage of coverage.

The recovery time also represents a fundamental issue for any resiliency mechanism. Usually, the smaller is the recovery time, the less is unavailability of the network services.

Traditionally every network recovery strategy presents a different backup capacity needed for recovering from a specific network failure. The backup recovery requirement differs based on the mechanism implemented, and it depends on the algorithm adopted for recovery path selection, the profiles of the traffic flowing into the network, and the layer where the mechanism is implemented.

Recovery mechanisms can be also distinguished according their capacity to assure full bandwidth availability when the traffic is rerouted on alternative paths. This issue is strictly related to the backup capacity.

When a fault occurs in the network a resiliency scheme can severally impact on the data duplication and reordering. The ability to reduce the packets duplication as well as the possibility to reorder the data represent important issues of a network recovery strategy, which increase the complexity of the designed solution.

A recovery mechanism can also introduce latency and jitter on data traffic. Based on the services supported by the network an appropriate solution should be adopted in order to guarantee the latency and jitter requirements.

The more are the backup resources used by a recovery strategy, the more are the information to be stored in the individual network element for implementing the reconfiguration mechanism. A network recovery strategy should also consider the overhead introduced by this information.

Another important parameter is the scalability. It represents a fundamental characteristic for any resiliency scheme. Specifically, a recovery mechanism is considered scalable if the performance does not depend on the size of the network and the traffic transported.

Usually, each network technology imposes particular constraints on the implementation of a recovery mechanism. With respect to the criteria described above, several design choices can be adopted in order to satisfy specific requirements.

With respect to the backup capacity a dedicated or shared solution can be designed. In the first case a one-to-one relationship exists between a specific backup resource and a specific working path. In the second approach, instead, a backup capacity is shared among different working paths. In this case a one-to-many relationship is realized. Clearly the shared solution is more complex than the dedicated one: when a failure occurs the mechanism should guarantee that enough resources are still available to allow the recovery in case of further failures. On the other hand, a shared resources mechanism is much more efficient and flexible than a dedicated resources approach.

Another design option is related to when the recovery path is selected. In the preplanned solution for each failure scenario the recovery path is calculated in advance. With dynamic approach, instead, the path is calculated when the fault occurs by the RHE and RTE nodes. The recovery mechanism starts to search in the network an alternative path when a failure on the working path is detected. Clearly a preplanned mechanism

allows fast recovery with respect to dynamic one, which requires time to identify the alternative path. On the other hand, a dynamic solution is capable to handle also unexpected faults (i.e. the uncovered failure scenarios). The dynamic approach is usually adopted for recovery mechanisms that implement shared backup capacity.

The recovery mechanisms can be classified based on the main goals of their strategies. In particular we can distinguish between protection approaches and restoration approaches. Both the options require signaling, but the main difference is related to the timing when the messages are exchanged. In case of protection schemes the recovery paths are preplanned and signaled before the failure occurs. This reduces the time of reconfiguration since no additional signaling is required in case of fault. With restoration approach, instead, the backup path can be both preplanned and dynamic, but additional signaling is required to establish the path when the failure occurs. Such solution needs more recovery time, but at the same time it is more flexible and it requires less backup capacity.

In order to bypass a failed network element, the recovery mechanisms change the path affected by the failure. The *recovery extent* represents the portion of the working path affecting by rerouting. In a *local recovery* scheme only the failed elements are bypassed. In this case the RHE and the RTE are chosen as close to the failed point. In other words, a local strategy establishes a recovery path between the neighbor nodes of the failed network element, ether a node or a link. With a *global recovery*, instead, the complete path between the source and the destination is replaced by an alternative path. In other words, the source and the destination represent respectively the RHE and the RTE. With a preplanned mechanism, a global recovery strategy requires alternative paths to be completely disjointed from the working paths, which imposes more complexity in the scheme. Local strategies are usually much faster than global ones. On the other hand, global recovery mechanism optimizes alternative path in terms of hops and bandwidth with respect to a local mechanism.

Another important issue that classifies the recovery mechanisms is related to which entity implements the reconfiguration strategy. In a *centralized recovery scheme* a central controller determines which action to perform in case of fault. It has a global view of the network by periodically gathering information about the status of the infrastructure. The controller performs commands for reconfiguration of all the elements involved in the recovery process. In a distributed recovery scheme, instead, no central controller is implemented. The control is distributed over the network and each node is capable to autonomously initiate the recovery actions. This requires just a local view of the network, and a messages exchange provides each other with sufficient information about the network status and for coordinating the recovery strategy. A centralized approach is simpler to implement and it has a better global view of the network. On the other hand, a distributed solution is more scalable and robust with respect to centralized one.

7 REFERENCES

[Abbott 1990] Russell J. Abbott, "Resourceful Systems for Fault Tolerance, Reliability, and Safety", in ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35 – 68.

[Abreu 2007] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. "On the accuracy of spectrum-based fault localization", in Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. (TAICPART-MUTATION 2007), pages 89–98, 2007.

[Abreu 2008] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund, "An observation-based model for fault localization", in Proceedings of International Workshop on Dynamic Analysis, pages 64–70. ACM, 2008.

[Aguilera 2002] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera, "On the quality of service of failure detectors", in IEEE Trans. Comput., 51(1):13–32, 2002.

[Anderson 1981] T. Anderson and P.A. Lee, "Fault Tolerance: Principles and Practice", Prentice/Hall, 1981.

[Andersson 1995] D. Andersson, "Detecting Usual Program Behavior using the Statistical Component of the Next-Generation Intrusion Detection Expert System (NIDES)", Technical Report, 1995.

[Arlat 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications", in IEEE Transactions on Software Engineering, 16(2):166–182, 1990.

[Avizienis 1984] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments", in Computer, 17(8):67–80, 1984.

[Avizienis 2004] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing", in IEEE Trans. Dependable Secure Computing, 2004.

[Baker 1992] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," in Summer USENIX Conf., 1992.

[Baker 2004] Andrew R. Baker, Brian Caswell, and Mike Poor. "Snort 2.1 Intrusion Detection", Syngress Ed., 2004.

[Baowen 2005] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. "A brief survey of program slicing". in SIGSOFT Software Engineering Notes, 30(2):1–36, 2005.

[Barbara 2001] D. Barbara, J. Couto, S. Jajodia, L. Popyack, and N. Wu. "ADAM: Detecting Intrusion by Data Mining", in IEEE Workshop on Information Assurance and Security, 2001.

[Barsi 1976] Ferruccio Barsi, Fabrizio, Grandoni and Piero Maestrini. "A Theory of Diagnosability of Digital Systems", in IEEE Transactions on Computers, vol. C-25, no. 6, pages 585–593, June 1976.

[Basseville 1993] M .Basseville, I.V. Nikiforov. "Detection of abrupt changes: theory and application", Prentice-Hall, Inc. 1993.

[Benoit 2003] Benoit, D. G., "Automatic Diagnosis of Performance Problems in Database Management Systems", PhD thesis, Queen's University, June 2003.

[Berenji 2003] Berenji, H., Ametha, J., And Vengerov, D. "Inductive learning for fault diagnosis", in Proceedings of the IEEE 12th International Conference on Fuzzy Systems (FUZZ). Vol. 1.

[Binkley 2003] David Binkley and Mark Harman, "A large-scale empirical study of forward and backward static slice size and context sensitivity", in Proceedings of the International Conference on Software Maintenance, page 44. IEEE Computer

Society, 2003.

[Bishop 1991] Matt Bishop, "An Overview of Computer Viruses in a Research Environment", Technical Report PCS-TR91-156, Dartmouth College, Hanover.

[Blake 1989] Blake, J.T. and Trivedi, K.S., "Reliability analysis of interconnection networks using hierarchical composition", in IEEE Transactions on Reliability, 32, 111–120.

[Bolch 2005] Gunter Bolch, Stefan Greiner, Hermann de Meer & Kishor S. Trivedi, "Queueing networks and Markov chains", Wiley-Interscience, 2005.

[Bondavalli 1990] A. Bondavalli and L. Simoncini. "Failures classification with respect to detection" in Proceedings of 2nd. IEEE Workshop on Future Trends in Distributed Computing Systems, pages 47-53, Cairo, Egypt, September-October 1990.

[Bondavalli 2000] Andrea Bondavalli, Silvano Chiaradonna, Felicita Di Giandomenico and Fabrizio Grandoni, "Threshold-Based Mechanisms to Discriminate Transient from Intermittent Faults", in IEEE Transactions on Computers, vol. 49, no. 3, pages 230–245, 2000.

[Bondavalli 2004] Andrea Bondavalli, Silvano Chiaradonna, Domenico Cotroneo and Luigi Romano, "Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications", in IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 4, pages 223–237, 2004.

[Bouloutas 1992] A. Bouloutas, G. W. Hart, and M. Schwartz, Simple Finite State Fault Detectors for Communication Networks, IEEE Transactions on Communications, vol. 40, pp. 477–479, March 1992.

[Bouloutas 1993] A. T. Bouloutas, G. W. Hart, M. Shwartz, "Fault Identification using a FSM Model with Unreliable Partially Observed Data Sequences", in IEEE Transactions on Communications, 41(7), 1993.

[Brodie 2001] M. Brodie, I. Rish, and S. Ma, "Optimizing probe selection for fault localization," in Distributed Systems Operation and Management, 2001.

[Byres 2004] Eric Byres, J. Lowe, "The Myths and Facts behind Cyber Security Risks for Industrial Control Systems", in Proceedings of VDE Congress, Berlin, October 2004.

[Carrozza 2008] Carrozza, G. and Cotroneo, D. and Russo, S., "Software Faults Diagnosis in Complex OTS Based Safety Critical Systems", in Proceedings of the 2008 Seventh European Dependable Computing Conference, 2008.

[Carrozza 2008] G. Carrozza. "Software Faults Diagnosis In Complex, Ots-Based, Critical Systems", PhD Thesis 2008. Università degli studi di Napoli, Federico II.

[Casimiro 2008] A. Casimiro, P. Lollini, M. Dixit, A. Bondavalli, P. Verissimo. "A framework for dependable QoS adaptation in probabilistic environments" in Proceedings of the 2008 ACM symposium on Applied computing, 2008.

[Chand 2004] R. Chand and P. Felber, "XNET: a Reliable Content-based Publish/Subscribe System", in Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS 04), pp. 264– 273, October 2004.

[Chen 2001] Chen, D.Y. and Trivedi, K.S., "Analysis of Periodic Preventive Maintenance with General System Failure Distribution", in Proceedings of Pacific Rim Intl Symposium on Dependable Computing (PRDC).

[Chen 2002] M. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services", in Proceedings of the International Conference on Dependable Systems and Networks, 2002.

[Cheswick 2003] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, "Firewalls and Internet Security: Repelling the Wily Hacker", Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Chillarage 1992] Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J, Moebus, D., Ray, B., Wong, M., "Orthogonal Defect Classification – A Concept for In-Process Measurement", in IEEE Trans. on Software Engineering, vol. 18, no. 11, pp. 943-956, November 1992.

[Chillarege 1996] Christmansson, J., Chillarege, R., "Generation of an Error Set that Emulates Software Faults", in 26th IEEE Fault Tolerant Computing Symp., FTCS-26, Sendai, Japan, 1996.

[Cohen 1995] F. B. Cohen, "Protection and Security on the Information Superhighway", John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Choi 2002] J.D. Choi and A. Zeller, "Isolating failure-inducing thread schedules", in Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pages 210–220. ACM, 2002.

[Chou 2001] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating System Errors," in Proceedings of ACM Symp. on Operating Systems Principles, 2001.

[Christmansson 1996] Christmansson, J., Santhanam, P., "Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms – Criteria for Error Selection using Field Data on Software Faults", in Proceedings of 7th IEEE Intl Symp. on Software Reliability Engineering, ISSRE'96, NY, USA, 1996.

[Coccoli 2003] Andrea Coccoli and Andrea Bondavalli. "Analysis of Safety Related Architectures", in Proceedings of IEEE International Workshop on Object-oriented Real-time Dependable Systems, Capri, Italy, 2003. IEEE Computer Society Press.

[Costa 2000] P. Costa, M. Migliavacca, G. Picco, and G. Cugola, "Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation", in Proceeding of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS 04), pp. 552–561, March 2000.

[Cotroneo 2009] D. Cotroneo, R. Natella, and S. Russo, "Assessment and Improvement of Hang Detection in the Linux Operating System," in Proceedings of Symp. on Reliable Distributed Systems, 2009.

[Cotroneo 2010] Carrozza, G. and Cotroneo, D. and Natella, R. and Pecchia, A. and Russo, S., "Memory leak analysis of mission-critical middleware", in The Journal of Systems and Software, 2010.

[Cotroneo 2010] D. Cotroneo, D. Di Leo, R. Natella, "Adaptive Monitoring in Microkernel OS," in Proceedings of DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance, 2010.

[Cotroneo 2010b] R. Natella and D. Cotroneo, "Emulation of Transient Software Faults for Dependability Assessment: A Case Study". in Proceedings of Dependable Computing Conference (EDCC), 2010 European, pages 23–32, 2010.

[Cristian 1991] F. Cristian, "Understanding fault-tolerant distributed systems", in Comm. ACM 34 (1991) 57-78.

[Daidone 2006] Alessandro Daidone, Felicita Di Giandomenico, Andrea Bondavalli and Silvano 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.

[Daidone 2009] A. Daidone, "Critical infrastructures: a conceptual framework for diagnosis, some application and their quantitative analysis", PhD Thesis. Università degli studi di Firenze, December 2009.

[Dijkstra 1974] E.W. Dijkstra, "Self-stabilizing Systems in spite of Distributed Control," in Communications of the ACM, vol. 17, no. 11, pp. 643–644, November 1974.

[Dolev 2000] S. Dolev, "Self-Stabilization", MIT Press, 2000.

[Ernst 1994] Michael D. Ernst, "Practical fine-grained static slicing of optimized code", Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994.

[Esposito 2005a] Marcello Esposito, Claudio Mazzariello, Francesco Oliviero, Simon Pietro Romano, and Carlo Sansone," Real Time Detection of Novel Attacks by means of Data Mining Techniques", in Proceedings of 7th International Conference of Enterprise Information Systems, ICEIS, 2005.

[Esposito 2005b] Marcello Esposito, Claudio Mazzariello, Francesco Oliviero, Simon Pietro Romano, and Carlo Sansone, "Evaluating Pattern Recognition Techniques in Intrusion Detection Systems", in Proceedings of the 5th International Workshop on Pattern Recognition in Information Systems, PRIS, 2005.

[Forrest 1996] S. Forrest, S. A. Hofmeyr, A. Somaya Ji, T. A. Longstaff, "A sense of self for unix processes". in Proceedings of IEEE Symposium on Security and Privacy, page 120, Washington, DC, USA, 1996. IEEE Computer Society. 1996.

[Frenkiel 1999] A. Frenkiel and H. Lee, "EPP: A Framework for Measuring the Endto-End Performance of Distributed Applications", in Proceedings of Performance Engineering 'Best Practices' Conference, IBM Academy of Technology, 1999.

[Gardner 1997] R. Gardner, D. Harle, "Alarm Correlation and Network Fault Resolution using Kohonen Self-Organizing Map", in Proceedings of Globecom'97, 1997.

[Garg 1995] Garg, S., Puliafito, A., Telek, M. and Trivedi, K.S., "Analysis of Software Rejuvenation using Markov Regenerative Stochastic Petri Net", in Proceedings of Intl. Symposium on Software Reliability Engineering (ISSRE), 1995.

[Gray 1986] Jim Gray. "Why Do Computers Stop and What Can Be Done About It?", in Proceedings of the Fifth Symposium On Reliability in Distributed Software and Database Systems, January 13-15, 1986, pp. 3 - 12.

[Gupta 1995] Rajiv Gupta and Mary Lou Soffa, "Hybrid slicing: an approach for refining static slices using dynamic information", in Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, pages 29–40. ACM, 1995.

[Hamerly 2001] Hamerly, G. And Elkan, C.,. "Bayesian approaches to failure prediction for disk drives", In Proceedings of the Eighteenth International Conference on Machine Learning. Morgan Kaufmann Publishers, San Francisco, CA, 202–209, 2001.

[Hansman 2005] S. Hansman and R. Hunt. "A Taxonomy of Network and Computer Attacks", in Computers & Security, 24(1):31–43, 2005.

[Harrold 1998] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi, "An empirical investigation of program spectra", in Proceedings of ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 83–90. ACM, 1998.

[Harrold 2002] James A. Jones, Mary Jean Harrold, and John Stasko. "Visualization of test information to assist fault localization". in Proceedings of the 24th International Conference on Software Engineering, pages 467–477. ACM Press, 2002.

[Hätönen 1996] K. Hätönen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen, "Tasa: Telecommunication alarm sequence analyzer, or: How to enjoy faults in your network", in Proceedings of IEEE Network Operations and Management Symposium, volume 2, pages 520 – 529, Kyoto, Japan, Apr. 1996.

[Haykin 1998] Haykin, Simon, "Neural Networks: A Comprehensive Foundation", second edition, 1998.

[Herder 2006.B] J. N. Herder et al., "MINIX 3: A Highly Reliable, Self- Repairing Operating

System," in Proceedings of ACM SIGOPS, vol. 40, no. 3, 2006.

[Herder 2006] J.Herder,H.Bos,B.Gras,P.Homburg,andA.Tanenbaum, "Construction of a Highly Dependable Operating System," in Proceedings of European Dependable Computing Conference, 2006.

[Herder 2008] J. N. Herder et al., "Countering IPC Threats in Multiserver Operating Systems", in Proceedings of Pacific Rim Symp. on Dependable Computing, 2008.

[Herder 2009.B] J. N. Herder et al., "Dealing with Driver Failures in the Storage Stack," in Proceedings of Latin-American Symp. on Dependable Computing, 2009.

[Herder 2009] J. N. Herder et al., "Fault Isolation for Device Drivers," in Proceedings of Conf. on Dependable Systems and Networks, 2009.

[Ho 2000] L. L. Ho, D. J. Cavuto, S. Papavassiliou, and A. G. Zawadzki, "Adaptive and Automated Detection of Service Anomalies in Transaction-oriented WAN's: Network Analysis, Algorithms, Implementation, and Deployment", in IEEE J. Sel. Areas Commun., vol. 18, no. 5, pp. 744–757, May 2000.

[Hood 1992] C. S. Hood and C. Ji, "Probabilistic Network Fault Detection", in Proceedings of Global Telecommunications Conf., vol. 3, Nov. 1996, pp. 1872–1876.

[Howard 1995] J. D. Howard, "An Analysis of Security Incidents on the Internet 1989-1995", PhD Thesis, 1998, Pittsburgh, PA, USA.

[Iyer 1991] I. Lee, R.K. Iyer, and D. Tang, "Error/Failure Analysis Using Event Logs from Fault Tolerant Systems", in Proceedings of the 21st IEEE Symposium on Fault Tolerant Computing (FTCS-21), June 1991

[Iyer 1997] M. Kalyanakrishnan, R. K. Iyer, and J. Patel, "Reliability of Internet Hosts - A Case Study from the End User's Perspective", in Proceedings of the 6th International Conference on Computer Communications and Networks, September 1997.

[Iyer 2007] Long Wang, Z. Kalbarczyk, Weining Gu, and R.K. Iyer, "Reliability microkernel: Providing application-aware reliability in the os", in IEEE Transactions on Reliability. 56(4):597–614, Dec. 2007.

[Jaeger 2008] M. Jaeger, G.Muhl, M.Werner, H. Parzyjegla, and H.-U. Heiss, "Algorithms for Reconfiguring Self-Stabilizing Publish/Subscribe Systems," in Autonomous Systems - Self-Organization, Management, and Control, Springer, pp. 135–147, September 2008.

[Jafarpour 2008] H. Jafarpour, S. Mehrotra, and N. Venkatasubramanian, "A Fast and Robust Content-based Publish/Subscribe Architecture", in Proceedings of the 7th IEEE International Symposium on Network Computing and Applications (NCA 08), pp. 52–59, July 2008.

[Jecheva 2006] Veselina Jecheva, "About Some Applications of Hidden Markov Model in Intrusion Detection Systems", in Proceedings of International Conference on Computer Systems and Technologies, 2006.

[Kamkar 1993] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri, "Three approaches to interprocedural dynamic slicing", in Microprocessing and Microprogramming, 38 (1-5):625–636, 1993.

[Kazemzadeh 2009] R.S. Kazemzadeh and H.-A. Jacobsen, "Reliable and Highly Available Distributed Publish/Subscribe Service", in Proceedings of the IEEE International Symposium on Reliable Distributed Systems, 2009.

[Khanna 2006] Gunjan Khanna, Padma Varadhara jan, and Saurabh Bagchi, "Automated online monitoring of distributed applications through external monitors", in IEEE Trans. Dependable Secur. Comput., 3(2):115–129, 2006

[Kiciman 2005] E. Kiciman, A. Fox, "Detecting Application-Level Failures in Component-Based Internet Services", in IEEE transactions on neural networks, 2005.

[Kintala 1995] Yennun Huang and Chandra Kintala and Nick Kolettis and N. Dudley Fulton,

"Software Rejuvenation: Analysis, Module and Applications", in Proceedings of International Symposium on Fault-Tolerant Computing, 1995.

[Kliger 1997] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, "A coding approach to event correlation," in Proceedings of International Symposium on Integrated Network Management (IFIP/IEEE), Santa Barbara, CA, USA, 1997, pp. 266–277.

[Korel 1988] B. Korel and J. Laski. "Dynamic program slicing", in Information Processing Letters, 29(3):155–163, 1988.

[Laing 2000] B. Laing and J. Alderson. "How to guide - implementing a network based intrusion detection system", Technical Report, 2000.

[Larry 2002] Larry M. Manevitz and Malik Yousef, "One-class svms for document classification". in J. Mach. Learn. Res., 2:139–154, 2002.

[Lee 2000] W. Lee and S. J. Stolfo. "A Framework for Constructing Features and Models for Intrusion Detection Systems", in ACM Transactions on Information and System Security (TISSEC), 3(4), 2000.

[Lewis 1999] L. Lewis, "Service level management for enterprise networks", in Arch House, 1999.

[Li 2006] Li, L., Malony, A.D., "Model-Based Performance Diagnosis of Master-Worker Parallel Computations", in Proceedings of Euro-Par 2006 Parallel Processing, LNCS 4128, pp 35-46, 2006.

[Lin 1984] S. Lin, D. Costello, and M. Miller, "Automatic-Repeat-Request Error-Control Schemes", in IEEE Communications Magazine, vol. 22, no. 12, pp. 5–17, December 1984.

[Lin 1990] Ting-Ting Y. Lin & Daniel P. Siewiorek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis", in IEEE Transactions on Reliability, vol. 39, pages 419–432, 1990.

[Lynch 1975] W.C. lynch, W. Wagner, and M.S. Schwartz, "Reliability Experience with Chi/OS", in IEEE Transactions on Software Engineering, 1(2):253–257, June 1975.

[Madeira 2006] J. A. Duraes and H. Madeira, "Emulation of software faults: A field data study and a practical approach", in IEEE Transactions on Software Engineering, 2006.

[Madeira 2010] Roberto Natella, Domenico Cotroneo, Joao Duraes, and Henrique Madeira, "Representativeness Analysis of Injected Software Faults in Complex Software", in Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2010, pages 437–446, 2010.

[Mahambre 2008] S. Mahambre and U. Bellur, "An Adaptive Approach for Ensuring Reliability in Event based Middleware", in Proceedings of the Second International Conference on Distributed Event-based Systems (DEBS 07), pp. 157–168, June 2008.

[Malek 1990] M. Malek, "A consensus-based framework for responsive computer system design", in Proceedings of NATO Advanced Study Institute on Real-Time Systems, Springer, Berlin, 1992.

[Malek 1993] M. Barborak, M. Malek, A. Dahburra, "The consensus problem in fault tolerant computing", in ACM Comput. Surveys 25 (1993) 171–220.

[Malek 2010] F. Salfner, M. Lenk, M. Malek, "A survey of online failure prediction methods", in ACM Computuping Survey, 2010.

[Malhotra 1995] Malhotra, M. and Trivedi, K.S., "Dependability Modeling Using Petri Net Based Models", in IEEE Transactions on Reliability, 44 (3), 428–440, 1995

[Mancina 2008] A. Mancina et al., "Enhancing a Dependable Multiserver Operating System with Temporal Protection via Resource Reservations," in Proceedings of Conf.

on Real-Time and Network Systems, 2008.

[Manik 2009] Miroslav Manik and Elena Gramatova, "Diagnosis of faulty units in regular graphs under the PMC model", in Proceedings of 12th International Symposium on Design and Diagnostics of Electronic Circuits&Systems, pages 202–205, Washington, DC, USA, 2009. IEEE Computer Society.

[Marshall 1992] E. Marshall, "Fatal Error: How Patriot Overlooked a Scud", in Science, 255:1347, 1992.

[Mazzariello 2007] C. Mazzariello, "Multiple Classifier Systems for Network Security: from data collection to attack detection", PhD Thesis. 2007, Università degli studi di Napoli, Federico II.

[Misherghi 2006] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging", in Proceedings of the 28th international conference on Software engineering, pages 142–151. ACM, 2006.

[Mishra 2006] Kesari Mishra, K.S. Trivedi, "Model Based Approach for Autonomic Availability Management", in Proceedings of the Intl. Service Availability Sympo- sium, Helsinki , Finlande, 2006 , vol. 4328, 1-16

[Mongardi 1993] Giorgio Mongardi, "Dependable computing for railway control systems". in Proceedings of DCCA-3, pages 255–277, Mondello, Italy, 1993.

[Murphy 2000] B. Murphy and L. Bjorn, "Windows 2000 dependability," in Proceedings of Conf. Dependable Systems and Networks, 2000.

[Murray 2003] Murray, J., Hughes, G., And Kreutz-Delgado, K., "Hard drive failure prediction using non-parametric statistical methods", in Proceedings of ICANN/ICONIP, 2003

[Natu 2006] M. Natu, A. S. Sethi, "Active Probing Approach for Fault Localization in Computer Network", in Proceedings of E2EMON'06, Vancouver, Canada, 2006.

[Natu 2007] M. Natu, A. S. Sethi, "Efficient Probing Techniques for Fault Diagnosis", in Proceedings of Second IEEE International Conference on Internet Monitoring and Protection, 2007.

[NETeXPERT] NETeXPERT. http://www.agilent.com/comms/OSS. Agilent Technologies.

[Nickelsen 2009] Anders Nickelsen, Jesper Grønbæk, Thibault Renier and Hans-Peter Schwefel, "Probabilistic Network Fault-Diagnosis Using Cross-Layer Observations", in Proceedings of International Conference on Advanced Information Networking and Applications, volume 0, pages 225–232, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[Nishimatsu 1999] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue, "Callmark slicing: an efficient and economical way of reducing slice", in Proceedings of the 21st international conference on Software engineering, pages 422–431. ACM, 1999.

[OMG 2007] OMG. (2007, January) Data Distribution Service (DDS) for Real-Time Systems, v1.2. [Online]. Available: www.omg.org.

[Orso 2001] A.Orso, S. Sinha, and M. J. Harrold, "Incremental slicing based on data-dependences types" in Proceedings of the IEEE International Conference on Software Maintenance, page 158. IEEE Computer Society, 2001.

[Ottenstein 1984] Karl J. Ottenstein and Linda M. Ottenstein, "The program dependence graph in a software development environment", in SIGPLAN Notes, 19(5):177–184, 1984.

[Paxson 1999] Vern Paxson, "Bro: A System for Detecting Network Intruders in Real-time", in Computer Networks, 31(23-24), December 1999.

[Pelleg 2008] D. Pelleg et al., "Vigilant: Out-of-Band Detection of Failures in Virtual Machines", in Operating Systems Review, vol. 42, no. 1, 2008.

[Pieschl 2003] B. Peischl and F. Wotawa. "Model-Based Diagnosis or Reasoning from First Principles", in IEEE Intel ligent Systems, 18(3):32–37, 2003.

[Pietrantuono 2010] R. Pietrantuono, S. Russo, K. S. Trivedi, "Software Reliability and Testing Time Allocation: An Architecture-Based Approach", in IEEE Trans. Software Eng., 2010.

[Pizza 1998] Michele Pizza, Lorenzo Strigini, Andrea Bondavalli and Felicita Di Giandomenico, "Optimal Discrimination between Transient and Permanent Faults", in Proceedings of 3rd IEEE High Assurance System Engineering Symposium, pages 214–223, Bethesda, MD, USA, 1998.

[Powell 1992] David Powell, "Failure Mode Assumptions and Assumption Coverage", in FTCS 1992: 386-395

[Powell 1998] David Powell, Christophe Rabéjac and Andrea Bondavalli, "Alpha-count mechanism and inter-channel diagnosis", Technical report, ESPRIT Project 20716 GUARDS Report, N°I1SA1.TN.5009.E, 1998.

[Preparata 1967] F. Preparata, G. Metze, R. Chien, "On the Connection Assignment Problem of Diagnosable Systems", in IEEE Transactions On Electronic Computers, 1967.

[Rabiner 1990] Lawrence R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition", in Alex Waibel & Kai-Fu Lee, editors, Readings in speech recognition, pages 267–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[Randel 1995] Brian Randell and Jie Xu, "The Evolution of the Recovery Block Concept", in Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 1 – 21.

[Rilling 2001] J. Rilling and B. Karanth, "A hybrid program slicing framework" in Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation, pages 12–23, 2001.

[Rish 2002] I. Rish, M. Brodie, and S. Ma, "Intelligent probing: a Cost-Efficient Approach to Fault Diagnosis in Computer Networks," in IBM Systems Journal, vol. 41, no. 3, pp. 372–385, 2002.

[Rish 2005] I. Rish, M. Brodie, S. Ma, "Adaptive Diagnosis in Distributed Systems", in IEEE Transaction on Neural Networks, Sep. 2005.

[Romano 2002] Luigi Romano, Andrea Bondavalli, Silvano Chiaradonna and Domenico Cotroneo, "Implementation of Threshold-based Diagnostic Mechanisms for COTS-based Applications", in Proceedings of 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02), pages 296–303, Osaka University, Suita, Japan, October 13-16 2002.

[Romesburg 1984] Charles Romesburg, "Cluster Analysis for Researchers", Lulu.com, 1984.

[Rovnyak 1994] Rovnyak, S., Kretsinger, S., Thorp, J., And Brown, D., "Decision trees for real-time transient stability prediction", in IEEE Trans. Power Syst. 9, 3, 1417–1426, 1994.

[Russell 2003] Russell, Stuart J. and Norvig, Peter, "Artificial Intelligence: A Modern Approach", 2003.

[Sanders 2005] Kaustubh R. Joshi, William H. Sanders, Matti A. Hiltunen, and Richard D. Schlichting, "Automatic model-driven recovery in distributed systems". in Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems, pages 25–38. IEEE Computer Society, 2005.

[Serafini 2007] Marco Serafini, Andrea Bondavalli and Neeraj Suri, "Online Diagnosis and Recovery: On the Choice and Impact of Tuning Parameters", in IEEE Transactions on Dependable and Secure Computing, vol. 4, no. 4, pages 295–312, 2007.

[Silva 1985] M. Silva and S. Velilla, "Error Detection and Correction in Petri Net Models of Discrete Events Control Systems", in Proceedings of ISCAS 1985, the IEEE Int. Symp. on Circuits and Systems, pp. 921–924, 1985.

[Simache 2001] C. Simache and M. Kaaniche, "Measurement-based Availability Analysis of Unix Systems in a Distributed Environment", in Proceedings of the 12th IEEE International Symposium on Software Reliability Engineering, November 2001.

[Simache 2002] C. Simache, M. Kaaniche, and A. Saidane, "Event Log based Dependability Analysis of Windows NT and 2K Systems", in Proceedings of the 8th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'02), December 2002.

[Simache 2002] C. Simache, M. Kaaniche, A. Saydane, "Event Log Based Dependability Analysis of Windows NT and 2k Systems", in Proceedings of 9th Pacific Rim Int'l Symposium on Dependable Computing (PRDC 02), 2002

[Smith 2002] N. Smith and M. Gales, "Speech recognition using svms", in Proceedings of Advances in Neural Information Processing Systems 14, pages 1197–1204. MIT Press, 2002.

[Sosnowski 1994] Janusz Sosnowski, "Transient Fault Tolerance in Digital Systems", in IEEE Micro, vol. 14, no. 1, pages 24–35, 1994.

[Spainhower 1992] Lisa Spainhower, Jack Isenberg, Ram Chillarege and Joseph Berding, "Design for Fault-Tolerance in System ES/9000 Model 900", in Proceedings of Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS-22) Digest of Papers, pages 38-47, Jul 1992.

[Srinivasan 2005] S. H. Srinivasan R. P. Jagadeesh Chandra Bose, "Data Mining Approaches to Software Fault Diagnosis", in Proceedings of the 15th Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications, pages 45–52. IEEE Computer Society, 2005.

[Stallings 1995] W. Stallings, "Network and Internetwork Security: Principles and Practice", Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[Sterling 2007] C. D. Sterling and R. A. Olsson, "Automated bug isolation via program chipping. Software", in Practice and Experience, 37(10):1061–1086, 2007.

[Stolfo 1998] Wenke Lee and Salvatore J. Stolfo, "Data mining approaches for intrusion detection", in Proceedings of the 7th conference on USENIX Security Symposium, pages 6–6. USENIX Association, 1998.

[Sullivan 1991] Sullivan, M. and Chillarege, R., "Software defects and their impact on system availability-a study of field failures in operating systems", in Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium, 1991.

[Sullivan 1991] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability—A Study of Field Failures in Operating Systems," in Proceedings of Symp. on Fault-Tolerant Computing, 1991.

[Swift 2005] M.Swift, B.Bershad, and H.Levy, "ImprovingtheReliability of Commodity Operating Systems," in ACM Transactions on Computer Systems, vol. 23, no. 1, pp. 77–110, 2005.

[Tai 1997] Tai, Ann T. and Hecht, Herbert and Chau, Savio N. and Alkalaj, Leon, "On-Board Preventive Maintenance: Analysis of Effectiveness and Optimal Duty Period", in Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems - (WORDS '97), 1997.

[Tanenbaum 2006] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum, "Construction of a highly dependable operating system", in Proceedings of the Sixth European Dependable Computing Conference, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[Taylor 1980] David J. Taylor, et al, "Redundancy in Data Structures: Improving Software Fault Tolerance", in IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 585 - 594.

[Taylor 1980b] David J. Taylor, et al, "Redundancy in Data Structures: Some Theoretical Results", in IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 595 - 602.

[Tendolkar 1982] Nandakurnar N. Tendolkar and Robert L. Swann, "Automated Diagnostic Methodology for the IBM 3081 Processor Complex", in IBM J. Research and Development, vol. 26, pages 78–88, 1982.

[Trivedi 2006] Avritzer, Alberto and Bondi, Andre and Grottke, Michael and Trivedi, Kishor S. and Weyuker, Elaine J., "Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms", in Proceedings of the International Conference on Dependable Systems and Networks, 2006.

[Trivedi 2007] Michael Grottke and Kishor S. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate", in Computer, Vol. 40 n. 2, 2007.

[Trivedi 2007] Haberkorn, M. Trivedi, K., "Availability Monitor for a Software Based System", in Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium, 2007. HASE '07, 21-328.

[Tucek 2006] J. Tucek, S. Lu, C. Huang, S. Xanthos, Y. Zhou, "Automatic on-line failure diagnosis at the end-user site", in Proceedings of the 2nd conference on Hot Topics in System Dependability, 2006.

[Turnbull 2003] Turnbull, D. And Alldrin, N., "Failure prediction in hardware systems", Technical report University of California, San Diego, CA, 2003 http://www.cs.ucsd.edu/~dturnbul/Papers/ServerPrediction.pdf.

[Tyson 2000] M. Tyson, "Derbi: Diagnosys Explanation and Recovery from computer Break-Ins", Technical report, 2000.

[Ulerich 1988] Ulerich, N. And Powers, G., "On-line hazard aversion and fault diagnosis in chemical processes: The digraph plus fault-tree method", in IEEE Trans. Reliab. 37, 2 (Jun.), 171–177, 1988.

[Umemori 2003] Fumiaki Umemori, Kenji Konda, Reishi Yokomori, and Katsuro Inoue, "Design and implementation of bytecode-based java slicing system", in Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation, page 108. IEEE Computer Society, 2003.

[US BR 2000] US Bureau of Reclamation FIST manuals, 3-30, Transformer Maintenance, October 2000.

[Vapnik 1995] Floyd, Sally and Warmuth, Manfred, "Sample Compression, Learnability, and the Vapnik-Chervonenkis Dimension", in Mach. Learn. 1995.

[Vedeshenkov 2002] V. A. Vedeshenkov, "On the BGM Model-Based Diagnosis of Failed Modules and Communication Lines in Digital Systems", in Automation and Remote Control, vol. 63, no. 2, pages 316-327, February 2002.

[Velardi 1984] P. Velardi and R.K. Iyer, "A Study of Software Failures and Recovery in the MVS Operating System", in IEEE Transactions on Computers, C-33(6):564–568, June 1984.

[Verissimo 2003] P. Verissimo, N. Neves, M. Correira, "Intrusion-tolerant architectures: concepts and design", in Architecting Dependable Systems, pp. 3–36, 2003.

[Vesely 1981] Vesely, W., Goldberg, F. F., Roberts, N. H., And Haasl, D. F., "Fault tree handbook", Tech. rep. NUREG-0492. U.S. Nuclear Regulatory Commission, Washington, DC, 1981.

[Vieira 2010] I. Irrera, J. Duraes, M. Vieira, H. Madeira, "Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults", in Proceedings of Pacific Rim International Symposium on Dependable Computing, IEEE, 2010.

[Vigna 1999] G. Vigna and R. Kemmerer, "Netstat: a Network based Intrusion Detection System", in Journal of Computer Security, 7(1), 1999.

[Walter 2003] C. J. Walter, N. Suri, "The customizable fault/error model for dependable distributed systems", in Theor. Computer Science, 2003.

[Wang 1993] C. Wang, M. Schwartz, "Identification of Faulty Links in Dynamic-routed Networks", in IEEE Journal on Selected Areas in Communications,11 (3), 1993.

[Wang 2007] L. Wang, Z. Kalbarczyk, W. Gu, and R. Iyer, "Reliability MicroKernel: Providing Application-Aware Reliability in the OS", in IEEE Trans. on Reliability, vol. 56, no. 4, 2007.

[Wein 1990] A.S. Wein and A. Sathaye, "Validating Complex Computer System Availability Models", in IEEE Transactions on Reliability, 39(4):468–479, October 1990.

[Weiser 1979] M. Weiser, "Program Slicing: Formal , Psychological and Practical Investigations of an Automatic Program Abstraction Method", PhD Thesis, The University of Michigan, 1979.

[Weiser 1984] M. Weiser, "Program slicing", in IEEE Transactions on Software Engineering, 10(4):352–357, 1984.

[Weissman 1993] G. Jakobson and M. Weissman, "Alarm Correlation", in IEEE Network, pages 52–59, 1993.

[Wietgrefe 1997] H. Wietgrefe, K. Tuchs, and G. Carls, "Using neural networks for alarm correlation in cellular phone networks", in Proceedings of. International Workshop on Applications of Neural Networks in Telecommunications, 1997.

[Wilfredo 2000] Wilfredo, Torres, "Software Fault Tolerance: A Tutorial", NASA Langley Technical Report Server, 2000.

[Yemini 1996] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, D. Ohsie, "High Speed and Robust Event Correlation", in IEEE Communications Magazine 34 (5), 1996.

[Yuan 2006] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma, "Automated known problem diagnosis with event traces", in Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, pages 375–388, New York, NY, USA, 2006. ACM.

[Zeller 1999] A. Zeller, "Yesterday, my program worked. today, it does not. why?", in Proceedings 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 253–267. ACM, 1999.

[Zeller 2002] Andreas Zeller, "Isolating cause-effect chains from computer programs", in SIGSOFT Softw. Eng. Notes, 27(6):1–10, 2002.

[Zeller 2002b] A. Zeller and Ralf Hildebrandt, "Simplifying and isolating failure-inducing input", in IEEE Transactions on Software Engineering, 28(2):183–200, 2002.

[Zheng 2004] Alice X. Zheng, Jim Lloyd, and Eric Brewer, "Failure diagnosis using decision trees", in Proceedings of the First International Conference on Autonomic Computing, pages 36–43. IEEE Computer Society, 2004.