INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

**Temporal Alarm Correlation
in Communication Networks**

by

Richard Brunet, Ph.D. Mathematics

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Science (Information and Systems Science)

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
May 1998

@Copyright, 1998
Richard Brunet

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32426-5

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis

Temporal Alarm Correlation in Communication Networks

submitted by Richard Brunet, Ph.D. Mathematics
in partial fulfillment of the requirements for the degree of
Master of Science (Information and Systems Science)

Thesis Supervisor

Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Carleton University
May 1998

Abstract

Given a set of fault messages (alarms) in a computer network, Alarm Correlation is a process by which information is extracted from the alarms in order to determine the origin of the faults. This thesis investigates and develops a Temporal Alarm Correlation algorithm that uses time information from a set of alarms received.

Acknowledgments

I would like to thank Professor Bernard Pagurek and Professor Suhayya Abu-Hakima for their guidance and support during the course of my research. I also wish to thank Mr. Tony White and Professor Andrzej Bieszczad from Carleton University for their suggestions and comments. My gratitude is also given to the members of the Seamless Personal Information Networking (SPIN) group of the National Research Council for having hosted me and helped me in many ways. I also would like to thank the Telecommunications Research Institute of Ontario (TRIO) for their financial support during this research.

I also wish to express my gratitude to my wife Yayoi and my sons, Yugo and Ken, for their support, patience and love.

Table of Contents

Acceptance Sheet	ii
Abstract.....	iii
Acknowledgements.....	iv
Table of Contents.....	v
List of Figures.....	vii
Introduction	1
Chapter1 Alarm Correlation and Fault Diagnosis: Related Work	3
1.1 Introduction	3
1.2 Fault Management in Networks	4
1.3 An overview of Alarm Correlation approaches and models	6
1.3.1 Case-Based Diagnosis	6
1.3.2 Fault-Based (Expert System) Diagnosis	8
1.3.3 Model-Based Reasoning	9
1.3.4 Probabilistic Methods	14
1.3.4.1 Grammar Methods	14
1.3.4.2 Probabilistic Finite State Machine	16
1.3.4.3 Bayesian Networks	20
Chapter 2 Problem Description	22
2.1 Introduction	22
2.2 Alarm Correlation and Time	22
2.2 Peer-to-peer interaction	27
2.3 Existing work in Temporal Alarm Correlation.	34
Chapter 3 Katzela-Schwartz and Yemini's algorithms	37
3.1 Introduction.....	37
3.2 Katzela-Schwartz approach	37
3.2.1 Definition of the Problem and Model	37

3.2.2 Fault Identification	41
3.2.3 Exhaustive search	45
3.2.4 Greedy Search	46
3.2.5 Divide and Conquer Maximum Mutual Dependency Algorithm ...	47
3.3 Yemini's algorithm	56
3.3.1 Introduction	56
3.3.2 Theoretical Basis of the framework	56
3.3.3 Coding, Alarm Correlation and description of the Algorithm	61
3.4 Application to a specific example	69
Chapter 4 Algorithms	71
4.1 Introduction	71
4.2 Centralized Algorithm	71
4.2.1 Formal Expression of the Problem	71
4.2.2 Algorithm	84
Chapter 5 Tests	101
5.1 Introduction	101
5.2 Description of the testbench	102
5.2.1 Server system	102
5.2.2 Alarm Generating System	104
5.2.3 Alarm Correlation Engine	104
5.3 Results	106
5.4 Discussion of the results	107
Chapter 6 Future Work and Conclusion	113
6.1 Introduction	113
6.2 Some further research paths	113
6.2.1 Multiple primary faults detection	113
6.2.2 Incorporating probabilities of failure of nodes	114
6.2.3 Learning about the thresholds	114
6.2.4 Distributed algorithm	115
6.3 Conclusion	121
References	123

List of Figures

Figure 1.1 Model-based diagnosis	10
Figure 1.2 IMPACT architecture	12
Figure 2.1 Dependency graph of example	26
Figure 2.2 Dependency graph of example	26
Figure 2.3 Dependency graph of the domino example	28
Figure 2.4 Dependency graph of the domino example	29
Figure 3.1 Network and its causality graph	39
Figure 3.2 Clusters of alarms	42
Figure 3.3 Causality graph	51
Figure 3.4 Cluster of alarms	51
Figure 3.5 Phase I - Partitioning	53
Figure 3.6 Phase II - Selection	54
Figure 3.7 Example of problem in Katzela-Schwartz method	55
Figure 3.8 Network and its dependency graph	57
Figure 3.9 Correlation graph	60
Figure 3.10 Correlation graphs	62
Figure 3.11 Examples of Codebooks	64
Figure 3.12 Examples of causality path	67
Figure 3.13 Causality graph of example	69
Figure 4.1 Directed tree and reversed tree	75
Figure 4.2 Propagation of faults	79
Figure 4.3 Lemma 1	82
Figure 4.4 Time inconsistency in causality graphs	85
Figure 4.5 Cause-Alarm graph	90
Figure 5.1 Dependency Graph of the testbench	108
Figure 5.2 Table of results	109
Figure 5.3 Graph of the ratio of correct analysis as a function of the percentage of missing alarms received	110
Figure 6.1 Global Causality Graph	116
Figure 6.2 Local Causality Graph	117

Introduction

In those years of exponential growth of internetworking, communication networks became especially vulnerable to faults in equipment and software. Because of the recognized importance of malfunctions in networking in general, the Open Systems Interconnection standards (OSI) [Stallings1993] includes, among its five areas related to network management, fault management as being concerned with the detection, identification and correction of malfunctions in the network.

In the next chapters, we will address the problem of identifying primary faults (that is, the first malfunctions in a network at a given time that explain other generated faults) in a communication network, and more specifically, we will discuss how temporal information can be used for that purpose. Our objective is to devise a heuristic algorithm that allows the determination of primary faults using time information. Notice that the algorithm must handle missing faults, signals, or *alarms*.

In Chapter 1, we will first introduce the reader to the area of fault identification, and we will review some interesting approaches to this problem from the literature. We will define with more precision, in Chapter 2, the problem that this thesis addresses. In Chapter 3, we will look more closely at two interesting approaches to the general alarm correlation problem: those are the methods from Katzela and Schwartz, and the algorithm from

Yemini. We focus on those two methods since some concepts from them will be used for the temporal alarm correlation problem. Our solution to the temporal alarm correlation problem will be described in Chapter 4. In Chapter 5, we describe some tests performed, using a model of a network, to verify the performance of the centralized algorithm of Chapter 4. Finally, some further research perspectives will be discussed in Chapter 6, where the thesis concludes.

Contributions of this thesis

This thesis provides the following contributions:

1. An investigation of alarm correlation methods in computer networks, with a special focus on time-based methods; and
2. A novel alarm correlation framework based on temporal information. The framework uses graph-theoretical methods. A centralized algorithm based on this framework is constructed. Testing shows that it correctly diagnoses faults 80% of the time even when 35% of the alarms are missing. Furthermore, the implementation of the algorithm correlates the alarms from a 50 node network in less than 2 seconds.

Chapter 1

Alarm Correlation and Fault Diagnosis:

Related Work

1.1 Introduction

Interoperating and managing heterogeneous systems demands complex network management tools that can deal with multi-platform differences. Furthermore, the increasing bandwidth of networks (ATM, Fast Ethernet, for example) needs those network management systems to have an efficiency and speed that can cope with them. This is true in particular for the management of faults in the network. The quick detection of failures, and the fast analysis of the cause of those failures, have become essential characteristics of a good network management system.

In this chapter, we will first introduce the basic concepts of *fault identification* and *alarm correlation*. We will then explore some of the related work done on alarm correlation in the literature, some through examples.

1.2 Fault Management in Networks

Fault management has traditionally been performed by human operators whose experience and understanding of a particular network were the only tools relied on. Network managers were helped in their task by consoles with simple network status monitoring software. This is much less true now with the complexity of the new interconnected networks. Operators must be assisted in their task by more than a simple monitoring system. The complexity of alarms generated in a system must be *pre-processed*, so that the operators can relate those to a particular set of faults. By an *alarm*, we mean a signal indicating a failure or malfunction of some element (software or hardware) in the network.

In the Introduction, we have seen that in the OSI Management Architecture, the fault management area includes *fault identification* among its functions. The fault identification activity can be decomposed in turn into three processes: *alarm reception*, *alarm correlation* and *testing*. The first process consists of collecting the alarms that are sent by the faulty elements or by an intermediate process monitoring the faulty elements. In general, if an element fails or malfunctions, it will produce a cascade of many alarms. The number of alarms can get so high that it becomes impossible for an operator to analyse them in a reasonable time frame and to isolate the primary fault(s). This is where the second process, *alarm correlation*, comes into action. The alarm correlation engine processes the alarms and correlates them, i.e. relates them to one another so that information can be extracted from their association. The alarm correlation especially suggests hypotheses as *primary faults*, which are the causal origin of the set of all other faults (those other faults

are then referred to as *secondary faults*).

Once we have obtained the hypotheses resulting from the application of the alarm correlation engine, we have to test those hypotheses. Usually, those tests will be performed by the network administrator, either from a console or directly on the suspected device or software. Once identified as the source of the primary fault, the device or software is then isolated from the rest of the network and repaired.

The time taken to identify a fault is the sum of the time to find candidates for the primary causes of the alarms plus the time to test those hypotheses. Therefore, the two main aspects of a fault identification process we wish to optimize are:

1. The accuracy in the hypotheses given by the alarm correlation engine and;
2. The reduction of the number of tests for those hypotheses.

Clearly, optimizing 1 helps reduce the number of tests in 2. Hence, *the speed and accuracy of the whole fault identification process depends heavily on the output of the alarm correlation process*. In this thesis, we will focus on the alarm correlation aspect of fault identification.

As in network computing in general, we observe two trends in alarm correlation: centralized alarm correlation and distributed alarm correlation. In a centralized paradigm, the alarms are sent to a main fault identification engine located in one workstation, while in a distributed framework, many alarm correlation processes collaborate to help identify the primary fault. Each alarm correlation process is responsible for a part of the network

and they collect the alarms in the corresponding sector.

Constructing a good alarm correlation engine involves many tasks and many factors must be taken into consideration. We will illustrate, in the next section, the main issues in alarm correlation by discussing different approaches to the problem.

1.3 An overview of Alarm Correlation approaches and models

Alarm Correlation (known also under the name of *Fault Diagnosis*, especially in the Artificial Intelligence field) is an activity involving many disciplines. This is reflected by the different ways the problem has been addressed thus far. Historically, automated fault diagnosis was applied to fields like medicine [Shortliffe1976], integrated circuits, distributed processors [Preparata1967], aircraft engines (or systems) [Abu-Hakima1996], chemical and power plants [Washio1996]. Fault diagnosis has eventually moved to the area of computer networks, which is the objective of this thesis.

In the next subsections, we will present what we believe are the main approaches or trends in alarm correlation. We will illustrate some of them by describing briefly an example of interesting work in this area.

1.3.1 Case-Based Diagnosis

Perhaps the most natural way to generate hypotheses from alarms is to work in an intuitively reversed direction. Indeed, we can generate a fault dictionary by simulating or analysing the behaviour (symptoms) of a given system over a selected list of anticipated faults. A list of coupled symptoms relevant to each fault is then compiled. It provides a list of possible causes of a problem given symptoms. However, such a dictionary becomes

rapidly useless as the amount of data and their complexity in modern systems are quite imposing. A more sophisticated way to generate those dictionaries and to analyse their content is known under the name of *case-based reasoning*. This process is very similar to the different cases of law jurisprudence that a judge must consult in order to give a good judgement in a trial. His (her) own judgement, together with all the conditions of the trial, are as well recorded later in a book of cases. In case-based reasoning, a database of previous problems in the network, including the different alarms generated and the causes found to explain those, is constructed in order to produce recommendations to the network administrator. It must be encoded in such a way that references to it through a search engine are simplified. This phase, where cases are encoded and stored, is called the *indexing of cases*. Various indexing schemes have been developed. Among them we find Discrimination Nets [Schank1982] and Associative Retrieval [Gotlieb1993]. When a problem occurs in the network, the information about the new problem is encoded, and a best match between the problem and the case library is performed. This matching can be performed by assigning numerical values to the attributes, and the matching is performed using a specific metric on the space of those numbers. An adaptation process is also performed between the best match and the problem. Upon acceptance of the solution and its use, the new case is stored in the case library.

One important advantage of case-based methods is the simplicity of the process and the ability of such systems to learn other cases and augment the database. Its disadvantages relate to the fact that Case-Based Diagnosis is largely system dependent, and the time to acquire sufficiently enough knowledge to perform reasonable diagnostics can be non-trivial. Also, the system depends largely on the indexing of the cases, which can utilize a

structure of the data that is non-causal.

1.3.2 Fault-Based (Expert System) Diagnosis

Another way to perform Fault Diagnosis is to encode, in the form of empirical rules or knowledge-based frames, different associations of symptoms and faults accumulated by human experts. The empirical rules or heuristics represent precise knowledge and experience relating expertise to the managed system. Fault-Based Diagnosis includes work known under the name of *Rule-Based Diagnosis* or *Expert Systems* [Abu-Hakima1994]. Stimulated by the expansion of Artificial Intelligence in the 60's and 70's, Expert Systems have been developed to an extremely sophisticated level. Early examples of Expert Systems include ABEL [Patil1981], MYCIN [Shortliffe1976] and system D [Reggia1983]. Basically, an expert is asked for basic principles for practical diagnosis of a system. Since this knowledge obtained from the expert will be the basis on which the diagnoses are made, this knowledge must be as general, accurate and complete as possible. It is then encoded under the form of *logical predicates*, which contain proper variables and constants representing the different data involved in the expert's analysis. This process is usually a difficult one, since the "logical model" must represent accurately and totally the set of rules the expert gave. One assumption here is that the expert's knowledge can be encoded symbolically, which is not always the case: some subjective knowledge, hardly encodable, might be impossible to put in the database. For example, the "intuitive" skills of the human expert is something that would appear (we hope!) from the Expert System as a whole at the final stage rather than some predicate inserted in the database. The alarms are then encoded as a set of values for some variables in those predicates. An inference engine (using Prolog programming language, for example) analyses what the values of the other variables can be

to satisfy the predicate system given the alarms. Those inferred values then lead to possible causes for the alarm generation.

There are many advantages to the use of Fault-Based Diagnosis. First, systems do not forget while humans do. Therefore, a good Expert System represents some permanency of the expertise of a human expert. Also, Rule-Based Systems work even if the exact mechanisms of the system are not well understood. Another advantage is related to the reproducibility of the system: an Expert System can be easily reproduced or copied, while it takes years to train a good human expert. Also, an Expert System is in general quicker to process a given sequence of rules than a human. The disadvantages to Expert Systems are however non-negligible. As we have already mentioned, one major difference between human and automated experts is intuition. The mechanisms behind human intuition are still largely unknown. They involve a high-degree of pattern-recognition and planning, among other things. Some frameworks for Expert Systems, like Neural Networks, allow some degree of pattern recognition but are still far from what humans can accomplish. One of the main problems is the difficulty and time involved in the creation and maintenance of the rule base. This is a serious problem for applications such as telecommunication networks where the systems are dynamically changing and extremely complex. It is also difficult to find human experts whose knowledge can be as general as possible.

1.3.3 Model-Based Reasoning

Another way to understand the behaviour of a network under some stress is to create a model of it. Hence, by making predictions with the model about the behaviour of the network, we can compare with the actual faulty behaviour of the network (see Fig 1.1). The

difference between the two sets of observations can help us pinpoint the faulty component: actually, we can selectively eliminate the components and compare the effect on the model to the observed data. This method is called *Model-Based Reasoning (MBR) for diagnosis*.

Figure 1.1 Model-based diagnosis

This approach has emerged recently as the computing power to support the complexity of those models grows ([Bibas1996], [Jakobson1993]). In [Bibas1996], Finite-State Machines are used to model the system: each structural and functional component corresponds with a set of Finite-State Machines whose nodes are component states. The transitions between states give the behavioural structure of the network.

Generating models is a main problem in MBR. Some approaches involve causal models, others generate models with both structure and behaviour, while others involves

functionality in the devices. A key element in MBR methods is to make an *accurate* model of the network. It is important to note that model-based systems encompass both novel and known faults. The diagnosis is based on a system-independent program, which usually can be easily upgraded to take into account new configurations of the network. There is however a main problem with the complexity of such a diagnostic system and the important processing resources and time they require. However, some researches in MBR combine FBR (*Fault-Based Reasoning*) with MBR to bound the computational complexity [AbuHakima1994].

Object-Oriented Design (OOD) is an important tool in the construction of the network model. By its nature, OOD decomposes the problem (here the network) into a set of classes of objects that can be combined both hierarchically (that is, an object B *inherits* properties from object A if object B is a specialized version of object A), and by containment relations (an object A can contain an object B). An example of the use of OOD in the construction of a MBR fault diagnoser is GTE's IMPACT [Jakobson1993]. IMPACT's alarm correlation system contains the *structural* and *behavioural components* (Fig 1.2).

The structural component is an object-oriented description of the managed network. It consists of the *Network Element Class Hierarchy* and the *Network Configuration Model*. The Network Element (NE) classes describe network equipment types, such as switches, multiplexers, digital cross-connects etc. Those classes are organized into a hierarchy under the class-subclass relationship. The top class (the most general class) consists of the GENERIC-NE-CLASS: it contains the common information and fields to all NEs. The next

level in this hierarchy consists of the basic NE classes, like switch-class, trunk-class, etc. Notice that each of those classes continues the hierarchy on its own by having as subclasses the classes representing the physical and logical aspects of the object: for example, trunk-class refers to logical-trunk-class and to physical-trunk-class.

Figure 1.2 IMPACT architecture

Those subclasses then continue a subhierarchy of objects. As for general OOD, each subclass inherits parameters, fields, behaviours from its superclass. Notice in this

regards that IMPACT permits multiple inheritance (a class can have more than one immediate superclass). The *Network Configuration Model* describes the NEs and the connectivity between them as well as their containment relations. This process may be performed by the network operating controllers using the IMPACT Network Element Editor. Notice that those relationships must respect the attributes and behaviours of classes already in place.

The behavioural component of the IMPACT system describes the dynamics of alarm correlation. It contains three major parts: the *message class hierarchy*, the *correlation class hierarchy* and the *correlation rules*.

Like NEs, the alarms are organized in a hierarchy of classes. This specification of the nature of the alarms can be extremely useful. For example, suppose that a rule specifies that an action must be performed if Carrier-Group-Alarm-Blue or Carrier-Group-Alarm-Yellow or Carrier-Group-Alarm-Red is received. Then we can introduce a superclass to those three alarms, namely Carrier-Group-Alarm, and then simplify the alarm correlation rule considered. This generalization or specification of alarms plays an important role in the IMPACT system.

A correlation class is a generalized description of the state of the network based on the interpretation of the alarms received so far. Those are instantiated from the correlation rules, which asserts a correlation statement upon some conditions. A correlation class contains components, a message template and some slots for parameters. The components may be NEs, alarm messages or some other correlations involved in the process. The role of the parameters is to pass the information from a correlation to a higher level correlation.

The correlation rules use Boolean logic to assert the correlations from conditions resulting from events. The primary terms of the rules are messages, NEs, tests and correlations (instances of a correlation class. The action part of a correlation rule is the assertion (or instantiation) of correlations and their clearing.

In their paper [Jakobson1993], the authors described an implementation of IMPACT on a SUN Sparc 10 workstation that can correlate 12 to 15 alarms/sec.

1.3.4 Probabilistic Methods

Probabilistic methods are also used in Fault Diagnosis. In many of those, the network is represented both structurally and behaviourally by a dependency graph, where a directed edge indicates the possible immediate cause of a failure of a node. By associating probabilities of failures to each node and a dependency probability to the edges between them, it is possible to evaluate (up to some measure) the most probable candidates as primary failures of the network, given some alarms. One major problem in those approaches is the fact that the system relies entirely on the probability measures that are assigned to each component. It is quite difficult to apply a probability of failure to a component taken as a subpart of a network. Another difficulty lies in the dynamic assignment of probabilities as the network changes and interacts as a subnetwork within a larger network, as is now the case in today's typical heterogeneous networks (e.g. ATM LAN Emulation subnetworks attached to legacy LANs).

1.3.4.1 Grammar Methods

Other tools are also used in alarm correlation in conjunction with probability methods. Bouloutas, Calo and Finkel [Bouloutas1994] proposed a method where they use

two representations for the communication systems: a *Phrase Structure Grammar (PSG)* and a *dependency graph*. Dependency graphs will be explained in more details in chapters 3 and 4. By a PSG, we mean a recursive representation of the elements of a system by "words", each of them containing a sequence of other word representing the components that composed the first word. For example, if a particular network consists of four nodes (node-A, node-B, node-C, node-D) and three links (link-AB, link-BC, link-CD), we can first represent the network as a collection of links:

$$\text{NETWORK} \rightarrow \text{LINK-AB} \cdot \text{LINK-BC} \cdot \text{LINK-CD}$$

Each link can be further expanded into the following forms:

$$\text{LINK-AB} \rightarrow \text{NODE-A} \cdot \text{CHANNEL-AB} \cdot \text{NODE-B}$$

$$\text{LINK-BC} \rightarrow \text{NODE-B} \cdot \text{CHANNEL-BC} \cdot \text{NODE-C}$$

$$\text{LINK-CD} \rightarrow \text{NODE-C} \cdot \text{CHANNEL-CD} \cdot \text{NODE-D}.$$

In a similar fashion, we can expand the channels and the nodes if those can be decomposed as well. This technique provides a representation of any component in the network that may fail. This approach may be used to represent both the structure and the faults as well. PSG can be used to help the conceptual tasks in alarm correlation and can be used as a guide in the design of algorithms for that purpose.

This approach brings, among other methods for fault identification, the *Positive Information Algorithm*. The input for this algorithm is a set F of possible fault locations (that is, the terminal symbols of the grammar representing the faults), an information cost

$I(f)$ for all fault $f \in F$ (the negative logarithm of the probability of the fault); a set of n alarms, and for each alarm, a set A_j of possible causes for the alarm j , where $j = 1, \dots, n$. The output of the algorithm is a set $F' \subseteq F$ such that

$$I(F') = \min_{G \subseteq F} \sum_{g \in G} I(g)$$

where the minimum is taken over all $G \subseteq F$ such that $G \cap A_j \neq \emptyset$ for all $j = 1, \dots, n$.

This method very simply relies heavily on the probabilities assigned to the faults and assumes indirectly the independence of faults. Another version of their algorithm involves the joint probability of the faults and the alarms and considers the cases where the alarms are either accidentally sent or lost, as long as the probability of such events is known a priori.

1.3.4.2 Probabilistic Finite State Machine

Hart and Rouvellou [Hart1995a], [Hart1995b] use a method involving a *Probabilistic Finite State Machine*. A *Finite State Machine (FSM)* is a triplet $(Q(F), L(F), A(F))$ such that

1. $Q(F) = \{Q_1, Q_2, \dots, Q_n\}$ is a finite set of *states* of the system;
2. $L(F) = \{s_1, s_2, \dots, s_m\}$ is a finite set of *labels* (that is, the outputs associated with each transition between states);
3. $A(F)$ is a finite set of *labelled arcs*,: $A(F) \subseteq Q(F) \times Q(F) \times L(F)$. The elements Q_i , Q_j , and s are respectively called the origin, the destination and the label of the transition (Q_i, Q_j, s) .

Basically, a finite state machine is a labelled graph where the nodes represent the states and the edges represent the transitions between the states.

A *Probabilistic Finite State Machine (PFSM)* is a sextuple $(Q(F), L(F), A(F), P^0(F), P^f(F), P(F))$ such that

1. $(Q(F), L(F), A(F))$ is a FSM;

2. $P^0 = (p_{Q_1}^0, p_{Q_2}^0, \dots, p_{Q_n}^0)$ is the *initial state distribution*;

$$\sum_{Q_i \in Q(F)} p_{Q_i}^0 = 1 \quad \text{and} \quad p_{Q_i}^0 \geq 0 \quad \forall j$$

3. $P^f = (p_{Q_1}^f, p_{Q_2}^f, \dots, p_{Q_n}^f)$ is the *final state distribution*;

$$\sum_{Q_i \in Q(F)} p_{Q_i}^f = 1 \quad \text{and} \quad p_{Q_i}^f \geq 0 \quad \forall j$$

4. $P = \{p_{Q_i Q_j s}\}$ is a table where $p_{Q_i Q_j s}$ is the conditional probability associated with the arc (Q_i, Q_j, s) given that the process is in state Q_i . We have that:

$$p_{Q_i Q_j s} > 0 \quad \text{if} \quad (Q_i, Q_j, s) \in A$$

$$p_{Q_i Q_j s} = 0 \quad \text{otherwise}$$

and

$$\sum_{Q_j \in Q(F), s \in L(F)} p_{Q_j, s} = 1 \quad \forall j$$

A sequence $S = s_1, s_2, \dots, s_m$ of labels is *accepted by the PFSM F* if there is a traversal (or route) of the F such that the sequence of the labels of the arcs traversed in the order along the route corresponds to S . If $\mathcal{R} = a_1 a_2 \dots a_n$ is a route in F , the *probability $p_F(R)$ of R* is defined as

$$p_F(R) = p_{Q_1}^0 \times p_{Q_{m+1}}^f \times \prod_{i=1}^m p_{a_i}$$

A fault is modelled as a PFSM, in which the possible outputs of the PFSM corresponds to the possible alarm sequences generated by the fault. As a fault occurs, the result is a propagation of alarms that spread out from the original location of the fault: at first, the alarms are notices in a close neighbourhood of the fault, and later, other symptoms appear in farther locations. An abnormal behaviour of a network is usually detected by some network protocol, and the information is then relayed to another network protocol and so on. Historically, network protocols have usually been modelled using FSM. Thus, for modelling faults and their propagation, the choice of PFSMs to model faults makes sense. In particular, the probabilistic nature of a PFSM captures the likelihood of the presence of some alarms, and it gives a weight to the different sequences. The method assumes that the output could be noisy, that is, we define the noise as a sequence $\eta = N_1, N_2, \dots, N_{n_c}$ of "corruption operations" such as alarm addition, alarm deletion, alarm change and order

change of two alarms. A probability is also assumed for each sets of corruptions. For the alarms, the model assumes a very simple model: each alarm carries the identification of the device, the symptom and the time of occurrence of the symptom.

This algorithm is composed of two phases: *Learning* (Phase I) and *On-Line Correlation* (Phase II). In Phase I, the problem consists of the following: once a sequence of alarms $S(f)$ is received assuming their generation under the presence of a known fault, infer the finite state machine that best explains the sequence of alarms as output. For this purpose, we need to put some cost function on the space of PFSM. The cost function $C(F, S(f))$ defined by Rouvellou and Hart takes into account the complexity of the FSM (in number of arcs, states, and labels), and the probability of a possibly corrupted route in the PFSM consistent with $S(f)$. More precisely,

$$C(F, S(f)) = C_{\text{complex}}(F) + C_{\text{fit}}(F, S(f))$$

where

$$C_{\text{complex}}(F) = A(2\log(Q) + \log L)$$

and

$$C_{\text{fit}}(F, S(f)) = \sum_{j=1}^{n_d} \min_{(R_i, N_i) \in \Lambda(F, S_i(f))} -\log P_F(R_i, N_i)$$

where the minimum is taken over pairs (R_i, N_i) such that the sequence $S_i(f)$ is consistent with the PFSM F under the corruption operations N_i and $P_F(R_i, N_i)$ is the product of the

probability of the route R_i with the probability of the occurrence of the corruptions N_i . The problem in Phase I is therefore to find a PFSM F that minimizes $C(F, S(f))$. This is known as a hard problem, and Rouvellou and Hart propose a heuristic algorithm that first performs a greedy search in the space of FSM, and then minimizes the cost function by considering the space of PFSM with the structure of the chosen FSM.

This phase produces a "vocabulary" of finite state machine as faults. It therefore assumes some "purity" of the fault (of course perturbations are allowed, but the fault should be as simple as possible).

Phase II is based as well on the fitting algorithm of Phase I. Once received a sequence of alarms and given a set of faults (PFSM), Phase II consists in decomposing the sequence received into a set of subsequences, each of which can be recognized as some fault (as a PFSM).

1.3.4.3 Bayesian Networks

Among the frameworks used in the probabilistic approaches, we can mention Bayesian Networks [Deng1993], and Dempster-Shaffer theory of evidence [Dawes1995]. A *Bayesian Network* is a directed, acyclic causal graph where each vertex represents an event and where the directed edges represent a causality between two vertices (namely, from the cause to the effect). A main tool in Bayesian Networks is the use of the Bayes formula:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

The use of Bayesian Networks is based on knowing some of the probabilities and conditional probabilities between neighbouring nodes. We can then use a dependency graph representing the causality between the events to compute the probability of some hypotheses [Charniak1991]. As we can easily see in this formula, we can also calculate the probability of B given that event A occurred.

Deng et al.[Deng1993] have used Bayesian Networks for developing a Fault Diagnostic algorithm that applies to Linear Lightwave Networks. In this method, the diagnosis is made on a set of linear divider/combiner of optical paths, which are switches in a lightwave network. The structure of the network allows the use of an acyclic Bayesian Network, for which the belief propagation updating algorithm is known to be polynomial [Neapolitan1990], [Pearl1987]. However, in general, the resolution of a Bayesian Network is NP-hard, and such a computational complexity is not practical for in a real-time system.

All the approaches we have seen so far applied to Alarm Correlation in general. In the next chapter, we will introduce the problem of Temporal Alarm Correlation.

Chapter 2

Problem Description

2.1 Introduction

In this chapter, we show how temporal information can be essential in the solution of the Alarm Correlation problem. This will allow us to address the Temporal Alarm Correlation problem, which is the objective of this thesis. We will also see some situations where Temporal Alarm Correlation becomes a critical problem, namely in *peer-to-peer interactions*. Some examples of networks, in which those interactions appear and lead to especially critical situations (in the context of Alarm Correlation), will be given as well.

2.2 Alarm Correlation and Time

A computer network is made of components that differ from each other by their nature (workstations, printers, gateways, etc.) and functions (servers, processes, etc.). This heterogeneity implies as well a heterogeneity of faults, and the interdependence of the components of the network is reflected in the relationships between the faults. Therefore, a fault can propagate through the network and generate other faults in cascade. The analysis of the different pieces of information from all those alarms can be extremely difficult. Hence, in such a scenario, too much information is almost as bad as no information at all.

We must find the right elements of information to treat, and the best way to analyse them. We have seen in Chapter 1 various methods to diagnose faults that use different pieces of information about the alarms or the network components. We will develop a method that uses time as information. But to show the relevance of our method, we will first demonstrate how the knowledge of time can be essential in fault diagnosis.

We suppose that whenever a component is faulty or malfunctioning, it emits an alarm that identifies the malfunctioning component and the time (*timestamp*) at which the element started to fail.

Why is time so important? To answer this question, let us use a metaphor example. Suppose that on a table, a set of dominos is organised in circle, each placed at an equal distance from the next one and in a vertical position. Suppose that an observer is in another room. This observer plays the role of our network administrator, and the dominos represent the network elements. Suppose that one domino falls down. Through the cascade effect, all the other dominos will fall down as well. If, when a domino falls, a card with the number of the domino is prepared for the administrator, the administrator receives at the end only the set of the cards for each domino. There is no way he/she can tell which domino fell first and is the primary fault. However, the unique information of the time of the fall of a domino together with the domino number clearly identifies uniquely the exact sequence of the events. The reader may argue that a communication network is something very different from a circle of dominos. However, we will see later some situations in a network that are quite similar to our example.

Another important reason why time is important in Alarm Correlation is that some configurations in a network are set *dynamically*: that is, on the fly, as requested at a specific time for some of the elements. The setting up of an ATM virtual link is an example of such a configuration. The different ATM switches are set according to a request for a link between two hosts for a virtual path. In such conditions, the failure of an element must be traced back to the time of its dynamic configuration for a specific service with a respective timestamp. A set of rules implemented in an Alarm Correlation engine prior to dynamic network configuration actions must be capable of reacting to dynamic behaviour, since the knowledge of new configurations are nonexistent at the time of the knowledge acquisition. Note that some Alarm Correlation approaches indirectly use time. The dynamic aspects and behaviour of Model-Based Diagnosis is an example: since the model is dynamic, the knowledge of the timed interactions of the elements is implied.

Temporal Alarm Correlation involves the correlation of time information from the received alarms. This temporal information can take many forms. We have mentioned timestamps, but we could also consider the time at which the alarms are received. Some Alarm Correlation systems use the interval of times between some events or received alarms. We will see in a later section examples of frameworks taken from the literature that relates to different uses of time information.

It is now clear that in some situations, the knowledge of the time of the occurrence of the fault is essential. Since we assume that our alarms carry the timestamp at which the faults have occurred, we can rephrase the general Alarm Correlation problem into the following form. The following form of the Temporal Alarm Correlation problem is the

main problem that this thesis addresses:

Problem 1. *Assuming that we have received a sequence of alarms identifying the malfunctioning components in the network and the time of the failure, how can we identify the primary fault(s) ?*

We now introduce briefly the notion of a dependency graph. For more details on this concept, we refer the reader to Chapter 4. A dependency graph is a directed graph where the vertices represent some events (here faults) and a directed edge from a vertex u to a vertex v means that event v results from the occurrence of event u .

The reader may find that the problem can be easily solved by taking the alarm with the earliest timestamp. However, losses of alarms can make this assumption erroneous. Let us see an example. Figure 2.1 is a dependency graph of a network. Suppose that fault v_1 occurred and leads to a cascade of faults. We have received some alarms (a black vertex in the dependency graph represents a fault for which an alarm has been received). Suppose we have received the alarms corresponding to faults v_3 , v_4 , v_5 and v_7 . The timestamps of those alarms are respectively 1, 1.6, 1.5 and 2.5 time units. Assuming that the earliest alarm (v_3) is the primary cause of all the alarms would certainly be erroneous, since the faults v_4 and v_7 would not be explained. If we did not know about the actual fault origin, other explanations may have been made: either fault v_2 is a unique primary and its alarm is missing, or we have two independent primary faults, namely v_3 and v_4 .

The firing of spurious alarms has to also be considered. Figure 2.2 shows such an example. With the same dependency graph as in the previous example, assume that the

Figure 2.1 Dependency graph of example

Figure 2.2 Dependency graph of example

original primary fault is v_4 . Suppose we have received alarms v_3 , v_4 and v_7 with their timestamps being the same as in the previous example. Then alarm v_3 may be a spurious alarm, and taking it to be the primary cause because of its smallest timestamp leads to errors.

In the next section, we will come back to the domino example we discussed earlier, since it explains some tricky problems in Temporal Alarm Correlation, and we will generalize the characteristics of those problems. We will then see in concrete examples where those characteristics come into action.

2.2 Peer-to-peer interaction

The domino example we discussed earlier underlines two important aspects of the Temporal Alarm Correlation problem: the possibility of cycles of interactions, and the possibility that an edge in the dependency graph may be bidirectional. The fall of one domino will bring the other domino to fall down in cascade until the "wave of falls" come to the domino next to the domino that fell down first. If we would represent the situation in a dependency graph, this would give us a directed cycle (see Figure 2.3, where d_i represents the event "fall of domino number i ").

Another important characteristic in the domino interaction is the fact that for two adjacent dominos, say d_1 and d_2 , the fall of one domino can bring down the fall of the other domino symmetrically: either d_1 makes d_2 fall or d_2 makes d_1 fall, and with the same type of action. This can be represented on the dependency graph by a bidirectional edge. Therefore, a more accurate representation of the dependency graph for the domino system would be the one given in Figure 2.4.

Figure 2.3 Dependency graph of the domino example

We now need to see in a real network how those characteristics occur. We will introduce the notion of *client-server* systems and *peer-to-peer* interaction. We will then see that those concepts induce in networks the same type of characteristics we saw in the domino example.

A *process* is an executing instance of a program. A *server* is a process that waits for another process, called a *client*, to contact it and to request some type of service. The requested service could be a search for some data or the handling of some task. The client-server framework is widely used today. It is the basis of many existing systems such as the

Figure 2.4 Dependency graph of the domino example

UNIX operating system and most database interactions. Another way to describe a client-server interaction is to see it as a consumer-producer system. The client here is a consumer that requests some resource or service from a producer (here the server).

The client-server model is usually based on a simple request/reply protocol. With this protocol, the client sends a request message to the server for some service, like reading a file. When the server has fulfilled the request, it replies to the client by sending back the result of the operation or sends an error code explaining why the work could not be done. There are two modes by which the client can communicate with the server: *blocking (synchronous)* and *non-blocking (asynchronous)*. With a blocking primitive, the client sends a request and is then blocked until the reply from the server is sent to it. In the case

of a non-blocking primitive, the client can execute another process during the waiting period for the reply from the server. The advantage of non-blocking sends is that the CPU of the client is not idle while the request to the server is being handled.

One way to implement the messaging between the client and the server is to imitate the call for a procedure in a program. Basically, the request from the client to the server is processed by having the client calling a procedure in the server. This method is called a *Remote Procedure Call*.

A very specific situation occurs when a client can be a server and vice versa. This type of interaction between clients and servers is known as *peer-to-peer server systems*. We will see later many examples of such systems. Note that if two servers are of the peer-to-peer type, their requests and replies don't have to be symmetric. A request type from A to B can be different from B to A, and the same is true for the reply. This protocol means that, the roles being dependent on the services, the representation of a host by an object assuming the different roles (client or server) induces a peer-to-peer representation of the network. This is simply a high level Object-Oriented description of the network where the different client-server interactions are represented by the messages that the objects send to each other.

How can we represent interactions and dependencies between clients and servers in a network? One way is to represent each server or client by a vertex in a dependency graph, and each interaction by an edge that is directed from the server to the client of a service. When we can measure the strength of the interaction, we can assign the number representing this strength to the corresponding edge. This measure could be, for example,

the conditional probability that the event represented by the vertex at the termination occurs given that the event at the origin of the edge occurred (we call this measure the *dependency weight* of the edge). To represent a peer-to-peer interaction between two client/servers u and v , we must indicate the dependencies of u upon v and of v upon u . Therefore, in the dependency graph, we should have a directed path from u to v , and a directed path from v to u . Now both paths could use the same edges, and in particular, if the interaction is direct, this interaction could take the form of a directed edge from u to v and a directed edge from v to u (if the interaction is symmetric, we can simply put a bidirectional edge between u and v). This kind of representation will create cycles and bidirectional edges in the dependency graph, and so we obtain a dependency graph that has the same characteristics as the dependency graph in the domino example. Therefore, in the peer-to-peer interaction systems, we see a generalization of those characteristics of the domino example we wanted to enhance.

To avoid cycles in such a dependency graph, one can make copies of the vertices, distinguish them, and use single-direction edges. However, it is at the expense of a large size for the representation of the dependency graph that this single-direction of edges can be achieved. It is one important axiom we make in this work of allowing bidirectional edges in the dependency graph in order to simplify the construction of the dependency graph of a network, so that the dependency graph can be efficiently used. It is especially important if the fault management system involves an object-oriented description of the network. We do not wish to multiply the objects in the dependency graph of such systems. Each object can represent many components that can become faulty. So there is a trade-off to the simplicity in the dependency graph of the peer-to-peer servers: the diagnosis of a fault can

become more complex. It is also important to note that in the dependency graph, vertices other than servers can have double-sided edges between them. The fact that the graph represented the causality of peer-to-peer systems only augments the frequency of such edges.

In the next chapter, we will describe some approaches that solve this problem. Under some specific circumstances, those approaches, as we will see, do not perform very well. In later chapters, we will describe our algorithm for solving the problem. Let us now examine some concrete examples where this problem can occur.

In *air traffic control*, each major airport in a country can be seen as a vertex in which the relative position of airplanes flying in their covering area is maintained. In order to keep an accurate view of the parameters of the planes (trajectory, speed), each vertex can request that its neighbours supply information about each or any subset of planes. All the requests can be made in parallel, and so the failure of one airport server can be difficult to pinpoint for the other location servers. This can be detected as a lack of precision in the location of some planes. This imprecision can be cascaded as the information is assembled together by many servers. The fact that the controllers at each airport see an imprecise sphere around the real position of a given airplane can be seen as an alarm (the alarms could be fired by automatic systems as well), since the lack of accurate information about the position of a plane could imply a crash of two planes. Note that if each airport uses a system of radars of the same type under the same circumstances, the probability of a failure is identical for each vertex, and the dependency weight of each edge is also identical.

Another example of peer-to-peer system comes from the field of distributed processing. Suppose that a network is composed of a set (*pool*) of processors [Tanenbaum1995]. This is the case of many distributed operating systems (Amoeba, Plan9 [Tanenbaum1995]). Suppose that each processor can be connected to the other (the connection may be direct or indirect through some intermediate processors). A set of tasks is sent to some processors. In processing their assigned task, it may happen that some of those processors need to transfer a part of their task to another processor. In some Processor Allocation algorithms (see [Tanenbaum1995]), the overloaded machine randomly picks another machine and sends the job to it. If this machine is overloaded as well, it randomly forwards the job to another processor, and so on. An alarm is generated at a given processor by a signal from the kernel of the operating system (or the *microkernel*, as in the case of Amoeba). The failure of a processor (which includes the failure of the microkernel or the failure of one of its functions) can lead to the failure of many other processors. Considering that there could also be many tasks from many users being processed, the search for the primary failure can be challenging. Like the previous example, the servers (here the processors of the pool) are generally identical.

We consider one last example. This last one will be especially important for us since we will test our algorithm on a special version of this example. Here we have a network of computers (LAN or WAN), and each server uses remote procedure calls for their requests. Each of those servers is located in one host, and each host can contain many RPC servers. We assume that the servers query each other for some information. Suppose for example that server A needs some data. A's request is sent to server B. If B can immediately satisfy the request, the data is sent from B to A. Otherwise, a request is sent from server B to

another server and so on. It is possible that the request from a server can be accommodated from a compound of possibly different requests at a later stage in the cascade of requests. Therefore, it is possible that server A be requested at its turn, creating therefore a directed cycle of requests. We allow this possibility. As mentioned earlier, we will test our algorithm on this type of example in the context of RPC servers in a UNIX LAN (see Chapter 5). In such a paradigm, the failure of a server will then bring a timeout (a failure) in the servers that made a request to it. This will result in a cascade of faults possibly cyclic.

2.3 Existing work in Temporal Alarm Correlation.

Up until now, few papers have addressed the problem of time in Alarm Correlation. As we have mentioned earlier, there are different ways of using temporal information: either by directly using the timestamps of the alarms, or processing those timestamps to obtain the time intervals between some events in the network, or by looking at a relative sequence of some events in time.

The issue of time as an attribute in the management information base (MIB) model by the IETF (*Internet Engineering Task Force*) was addressed in [Apostopoulos1997]. The proposed network management information base is known under the name of *Temporal Management Information Basis (TMIB)*.

Nejdl and Gamper [Nejdl1994] have considered a framework for Model-Based Diagnosis of dynamic systems by using qualitative interval relations.

In telecommunication networks, recent work using models of the network based on time intervals characterization of the interactions between the different components

(*scenarios*) have also been published [Dousson1996], [Bibas1996], [Esfandiari1996]. In the work of [Dousson1996] and [Bibas1996], a first stage of the system consists in the off-line generation of *scenarios* or *chronicles* (set of event patterns and temporal constraints between them). Those simulations of alarms scenarios form the model of the network. Notice that they use intervals of time in the generation of each alarm. The model also calculates the probability of masking alarms (that is, an incomplete sequence of alarms). The second phase consists of the on-line recognition of instances of occurring scenarios. This is performed by matching the sequence of real incoming alarms with one of the chronicles. This matching can be done by forecasting from the candidate chronicle the next incoming alarms. It is to be noted that in those works, the telecommunication network studied has the structure of a tree, and the methods used rely on this structure.

The previous method shows that one of the main tasks in Temporal Alarm Correlation is the recognition of a pattern in the sequence of received alarms. While other methods, not involving time, use only the information that constitutes the presence of alarms as a set, one more structural level in the information must be processed here.

The pattern recognition system on Finite-State Machines output of Rouvellou and Hart [Hart1995a], [Hart1995b], in a work previously mentioned in Chapter 1, consider a sequence of alarms in the *received order*, hence taking into account time factors in the Alarm Correlation process. In the Learning Phase of the algorithm, Probabilistic Finite State Machines (PFSM) are actually constructed using sequences of received alarms. Other PFSMs are then constructed as the product of the PFSMs constructed in the Learning Phase. In a second phase, a sequence of alarms is constructed according to their reception

time, and we then try to fit it to outputs of the PFSMs constructed in the Learning Phase. The faults whose output creates the best fit then constitutes the solution.

The use of time intervals in the Alarm Correlation task is also made in a newer version of the IMPACT system discussed in Chapter 1 [Jakobson1994]. In this revised framework, events are characterized by a content (*message*) and a time quantifier that represents either a point in time where the event occurred or an interval of time for this occurrence. The IMPACT system is upgraded from its original form (see Chapter 1) by the introduction of temporal relations, which is basically an algebra of events with respect to their time quantifier, which is then used to build the correlation rules of IMPACT.

In Chapter 3, we will describe two different Alarm Correlation algorithms we found especially interesting in their approaches. One uses probabilities, while the other uses coding theory. They possess some characteristics on which we will build our Temporal Alarm Correlation algorithms described in chapter 4.

Chapter 3

Katzela-Schwartz and Yemini's algorithms

3.1 Introduction.

In this chapter, we analyze more specifically two interesting approaches to the Alarm Correlation problem. Elements from those two systems will be the basis of our temporal Alarm Correlation algorithms that will be explained in chapters 4 and 5. Later in the present chapter, we will see how the two methods perform on a system containing peer-to-peer interactions.

3.2 Katzela-Schwartz approach

3.2.1 Definition of the Problem and Model

The problem considered by the two authors [Katzela1996], [Katzela1995] can be formulated as follows: given a model of the network and a set of observed alarms, how can we design an algorithm that proposes a number of primary fault hypotheses and assigns a measure of confidence in each of them?

In this framework, a centralized paradigm for fault management is assumed. The proposed scheme essentially uses a graph representation of the network and probability on the edges and the vertices of the graph. Based on that model, an algorithm for Alarm Cor-

relation, called the *Maximum Mutual Dependency* algorithm, is designed.

As in many other Network Management paradigms, the typical telecommunication network in this approach consists of a number of managed objects, to which we will refer as objects, or entities. An *object* here is simply a part of the network that has a separate and distinct existence. It can be a subnetwork, a host, a process, a layer in the stack of a protocol, a virtual link, etc. It is possible for an object to be composed of other objects, and so on hierarchically. A *terminal object* is one that is indivisible, at least in the scope of the model. An object is *active* if it is in use during the time of the correlation.

We assume in this framework that an alarm carries only the identity of the object that failed. The timestamp may appear facultatively, but is not forbidden.

By a *causality graph*, we mean a directed graph $G = (V, E)$, where V is a set of active terminal objects and E is a set of directed edges between vertices of V . A directed edge $(v_i, v_j) \in E$ simply indicates that the failure of object v_i depends on the failure of object v_j , or in other words, that the failure of v_j may cause the failure of object v_i . Basically, a causality graph is a dependency graph where we have reversed the direction of each edge.

We associate with each vertex v_i a probability of failure p_i . This is the probability that the object v_i fails, independently of any other *active terminal object* in the network. In practice, those probabilities can be estimated from system specification information, history of previous failures, etc. It is important to mention that this probability in general is hard to evaluate. To obtain the probability of failure of a component, it is often not suf-

ficient to analyse the component while it is off-line. This probability must be evaluated as the component is active in the network. We will come back to this problem later.

Each directed edge (v_i, v_j) is assigned a *weight* p_{ij} representing the strength of dependency of the objects under failure. It is a measure of the “likelihood” of the failure of v_i , given that the active terminal object v_j has failed. Basically, $p_{ij} = P(v_j \text{ fails} \mid v_i \text{ fails})$ (this equation can be read as follow: given the failure of v_i , p_{ij} is the probability that v_j fails). In addition, if $\vec{P} = v_1 v_2 \dots v_n$ is a directed path in the causality graph, the weight of $w(\vec{P})$ is defined as the product of the weights of the directed edges that form the path, basically $w(\vec{P}) = \prod_{i=1}^n p_{i(i+1)}$.

Figure 3.1 Network and its causality graph

Many special cases of this graph can occur, for example, the edges can be bidirectional. If each edge of G is bidirectional, then we say that G is an *undirected causality graph*.

Let us consider an example of a network and its causality graph. Figure 3.1 represents a network that consists of three nodes and four links. For the selected level of abstraction, we assume that most objects are active terminal objects. The reader will notice that during our observation object L_4 was not in use (or active). We therefore obtain the causality graph on the right-hand side of figure 3.1, where each vertex represents either a host or a link from the network, and the directed edges represents the dependency between the terminal objects.

Such a model of the network presents many advantages. It is *simple*, since the model can be easily understood. It is also *general*, since the model can be applied to very different types of networks, and it can even be applied to systems that are not networks per se (for example, the main component of a workstation). It is flexible because it can easily be modified if needed.

It has disadvantages however: for example, it presupposes the knowledge of the probability of failure of a terminal object. To start with, we have to agree on a probability space to describe such an event; for example, should we estimate this probability over a large interval of time? Or do we define that probability with respect to the set of occurrences of all alarms in the system? In any case, we face the problem that the estimated probability of failure of a terminal object depends on the state of the network over the period in which this probability is evaluated. If the network is modified, the value of the

probability might be completely changed. These problems of the determination of the probability of failure of an object are much less acute in the case of the determination of the conditional probability of failure of an object B upon the failure of another object A. In this case, we simply have to count the number of times object B fails over all the times that object A fails. Those questions will be taken into account in the development of our algorithms in the next chapters.

In the next subsection, we will describe the algorithm based on the network model we have designed previously.

3.2.2. Fault Identification

For the sake of simplicity, we will represent the alarm associated with an active terminal object v_i by the symbol a_i for some index i . Given another object v_j , we say that v_j *explains alarm* a_i if there is a directed path in the causality graph from a_i to v_j . As we have defined in general, we say that an object that became faulty independently of other objects is a *primary fault*, while an object that became faulty as a result of the malfunctioning of another object is called a *secondary fault*.

For each alarm a_i generated, we define the domain $D(a_i)$ of the alarm a_i to be the set of all objects in the causality graph that explains a_i . A set of alarms is called a *cluster of alarms* if it can be organized into a sequence such that, for any alarm a_i in the sequence, the domain of a_i shares a nonempty intersection with at least one preceding alarm in the sequence (except the first one). The reader will notice that this is the exact definition of what is intended as a cluster in [Katzela1995], [Katzela1996], but it was not define correctly there. Figure 3.2 shows two clusters of alarms. Those are made up of 6

received alarms altogether: the domain of each alarm is represented by an ellipse in the figure. As we can see in the figure, some objects are in the intersection of the domain of more than one alarm. This representation of alarm domains shows that a cluster is represented by a “connected piece” of the picture. It also shows clearly that an alarm belongs to a cluster if its domain has a non-null intersection with the domain of at least one alarm that is in the cluster.

Figure 3.2 Clusters of alarms

Our problem is to determine the subset of the domain of an alarm that best explains the alarm. Looking for all possible causes that can explain an alarm would be pointless, as

this set could end up containing all the vertices in the dependency graph. However, *we can use the causality graph to find all causes that exceed a certain dependency weight W from each alarm*. Obviously, it is the choice of W that will create a different level of associative dependencies between objects. The right choice of W is a a major decision in the heuristics of the Schwartz-Katzela method. If W is too small, the clusters will contain too many objects and the computational time for the search for the primary faults may be too large. If W is too large, the clusters may contain only a few objects, and it is possible that a disconnectivity between clusters will hide a common cause. As an example, one can choose W to be the smallest of all the edge weights of the graph; the model obtained is called the *single linkage* or *nearest neighbour* model.

The problem of finding the domain $D(a_i)$ is a variation of the *single source problem* in Combinatorial Optimization [Katzela1996]. We remind the reader that the single source problem consists of finding the minimum cost paths from a source vertex in a graph to all other vertices. In the case of the causality graph considered here, the length of a path is measured by taking the product of the weights of its edges. But the domain alarm can easily be transformed into the single source problem by replacing the weight p_{ij} by the number $-\log(p_{ij})$, reversing the direction of the edges and by adding, rather than multiplying, those numbers. Hence, the single source problem equivalent to finding $D(a_i)$ consists of finding all the vertices in the graph whose minimal distance to a_i is less than $-\log(W)$. This problem can be solved by many algorithms (see [Aho1974], [Chartrand1993], [Papadimitriou1982], [Christofides1975]) including Dijkstra's algorithm which has complexity $O(|V|^2)$. One of the ideas in our approach in the next chapter will be to solve *a priori* (before any session of the Alarm Correlation engine) the single

source problem for each vertex by computing the transitive closure of the graph, and by ordering the neighbourhood of each vertex in this new graph by increasing value of the weight of their adjacent edge (see definitions next chapter).

Suppose now that a set of alarms $\{a_i\}$ occurs in the network. Suppose that we have found the domain of each alarm in the sequence and the cluster associated with those domains. Any object that is in the domain of an alarm is a possible explanation for the alarm, and clearly, the alarms in a same cluster should be examined together since there is a high probability that a common fault caused those alarms. We assume here that there should be one primary fault per cluster. Once we have found the primary fault for each cluster, the union of those faults will explain at best the whole set of alarms. Assume for the sake of simplicity that there is only one cluster, say C .

Let $S = \bigcup_{\forall a_i \in C} D(a_i)$, i.e. the set of all objects in the cluster C . Let $N = |S|$. We assume we have computed the causality graph of the network and also, we assume that the p_i 's and the p_{ij} 's are now for all objects in the graph, and in particular for the elements of S .

Let us go back to the cluster. The objective in our fault identification algorithm is to identify the best primary cause to the alarms contained in C . Each of those alarms may have many explanations. For the Katzela-Schwartz method, the best explanations consist of the minimum subset S' of active objects that satisfies the following two properties:

1. each element of S' explains all the alarms in the cluster; and
2. the set S' maximizes the probability that at least one of the objects is at fault.

For a set S' of objects, the probability that at least one object in S' is at fault is given by

$$\begin{aligned}
 P(S') &= P(\text{at least one fault in } S') \\
 &= 1 - P(\text{no fault in } S') \\
 &= 1 - \prod_{\forall i: e_i \in S'} \left[1 - P(e_i \text{ is a primary fault}) - P(e_i \text{ is a secondary fault}) \right] \\
 &= 1 - \prod_{\forall i: e_i \in S'} \left[1 - p_i - \left(\sum_{\forall j: e_j \in S', j \neq i} p_i \cdot p_j + \varepsilon \right) \right]
 \end{aligned}$$

where ε is the probability that a secondary fault occurs by the concurrent primary failure of two or more objects. Since the p_i 's are in general quite small, we can approximate the last formula by:

$$P(S') = \sum_{\forall i: e_i \in S'} \left[p_i + \sum_{\forall j: e_j \in S', j \neq i} p_i \cdot p_j \right]$$

Let us see now some algorithms to find this optimum set S' .

3.2.3 Exhaustive search

The simplest way to evaluate the set S' is to go in a straightforward way by:

1. Getting all the subsets of S ;

2. Selecting the minimum subset of S that explains all the received alarms in S under the following condition: if there are more than one such set with the same cardinality, select the set that maximizes the probability that the set contains a fault.

This algorithm, though simple, is computationally expensive: we have to remember that there are 2^N subsets of S , and so the computational time would be of the order of $O(2^N)$.

3.2.4 Greedy Search

One way to reduce the speed of the exponential in the complexity of the exhaustive search is to apply a binary partition of the set at each step, that is, instead of looking for all subsets, we split the set in two. More precisely:

1. Start with S ;
2. Find all the possible partitions of the set into two equal subsets;
3. Choose, out of those partitions, the subset that explains all the alarms and has the maximum probability of containing a fault;
4. Repeat Steps 2 and 3 with the chosen set until the resulting sets either are singletons or neither of them can explain all the alarms.

At the end, we will obtain the best explanation for the set of alarms. One can show that the complexity here is $O\left(\log N \cdot 2^{\left(\frac{N}{2}\right)\log\left(4\frac{N}{N+2}\right)}\right)$. The complexity is still exponential, but the exponent grows slower than the exponential in the exhaustive search case.

3.2.5 Divide and Conquer Maximum Mutual Dependency Algorithm

As its name implies, the goal of the algorithm is to reduce the problem into smaller parts, and then to solve the problem for each of these parts. It is especially well suited for large inputs of the set S . We need first to define some concepts. We say that the objects of a set S are *maximally mutually dependent* if for any pair of objects v_i and v_j , and for any object v_k not in S , the dependency weight p_{ij} is greater than p_{ik} , p_{ki} , p_{jk} , and p_{kj} . So the basic steps in the algorithm are as follows:

1. Start with the set S ;
2. Partition the set S into two sets, each of which showing the maximum mutual dependency between its objects.
3. Of the two sets obtained, choose the one that explains all the received alarms and has the highest probability of containing a fault. If no set exists that explains all the received alarms, find the subset of alarms that each set explains, and select both.
4. Apply recursively steps 2 and 3 to the set chosen in 3, and continue until we obtain a partition of the set with singletons (sets with only one element).

As for the other algorithms (greedy search and exhaustive search), the objective here is to narrow down the set of candidates to the minimum set that explain all the alarms and has the greatest probability of containing a fault. The reason for partitioning into sets that shows maximal mutual dependency is simple. We expect a very small number of primary causes for all the generated alarms, and if we have a maximal mutual dependency

set, it is much more probable that only a small number of these faults is primary. If we think back to the example of the domino (Chapter 2), each domino is highly dependent on the next one, and we expect in that set of dominos only a few primary causes since the fall of all dominos usually necessitates only the fall of one or few dominos.

This algorithm will always find the best set of causes that can explain all the alarms (if such a cause exists), but may not be the optimal one. The choice of the maximal mutually dependent set might eclipse the best solution.

Let us examine the algorithm in more detail. It is two-phased. Phase I is the *partitioning* of S into the maximal mutually dependent sets, and Phase II is the *selection* of the optimal set that explains all the alarms and exhibit the best probability of containing a fault.

Phase I. Partitioning

We must partition the set S into two maximally mutual dependent sets, and we continue recursively with each of the resulting sets. The top-down approach here can be quite expensive computationally. Instead, we can proceed bottom-up:

1. Partition S into all its singletons, and set $m = N$;
2. Choose out of the $\binom{m}{2}$ pairs of sets the pair that is mutually closest. Merge the sets together, compute the probability that at least one element in the new set is a fault, and calculate the dependency of the new set to all other sets.
3. Repeat 2 with $m = N - 1, N - 2, \dots 2$.

At the end, we obtain the original set S , and the sequence of sets constructed recursively is the sequence of partitions of S into maximal mutual dependent sets that will be used in the algorithm.

Phase II Selection

We now need to find the smallest subset S_{sol} of S that explains all the alarms and has the highest probability that one of its elements is a fault.

Initialization. At first, we set S_{sol} to be empty. As a first input to a typical step of the algorithm, we take (S, A) where A is the set of alarms in the alarm cluster considered.

Typical Step. Let (S', A') be the input for the step. From Phase I, we have computed the children of S' (S'_r and S'_l) in the maximal mutual dependency decomposition. The different cases for the typical step are the following:

- If S' is a singleton, add the element contained in it to S_{sol} and stop.
- If S' is not a singleton:
 - if both S'_r and S'_l explain all the alarms in A' , select the child that exhibits the greatest probability of containing a fault, and repeat the typical step for that child;
 - if only one of the children explains all the alarms and this child exhibits the greatest probability of containing a fault, choose this set and repeat the typical

step of the algorithm with it;

- if only one of the children explains all the alarms but the other child has the greatest probability of containing a fault, or if neither of the children explains all the alarms, then:
 - Find the child that has the greatest probability of containing a fault and find the subset of alarms (say A^i) it explains;
 - Repeat the typical step with this child and the set A^i ;
 - Repeat the typical step with the other child and the set of alarms $A^i - A^i$.

We will understand this algorithm better by applying it to an example. This example is taken from [Katzela1996].

Figure 3.3 shows the causality graph of a network. Notice that all edges are bidirectional, so that we have a causality graph that could model a system of peer-to-peer servers. Let us assume that object v_1 fails and alarms generated from objects v_2 , v_3 and v_4 were received. The resulting alarm cluster is represented in Figure 3.4, where we have taken a dependency weight $W = 0.01$. Notice that under this choice of parameter, we have only one cluster to consider.

Figure 3.3 Causality graph

Figure 3.4 Cluster of alarms

Phase I. The steps of the maximum mutual partitioning of the objects is given schematically in Figure 3.5.

Phase II. The steps of the selection phase of the algorithm are described in Figure 3.6. Initially, the solution set is empty, and we see that the input set is made of the set $S^f = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $A^f = \{a_2, a_3, a_4\}$. Now using the partitioning of the set S^f in Phase I, we have that the children of this set are $S_l^f = \{v_1, v_2, v_3, v_4\}$ and $S_r^f = \{v_5, v_6\}$. Now the probability that one of the objects is a fault is greater in S_r^f than in S_l^f . Also, S_r^f explains all the alarms received. We therefore select S_r^f for the next step, and so the input in step2 is $S^f = \{v_1, v_2, v_3, v_4\}$ and $A^f = \{a_2, a_3, a_4\}$. The right and left children of S^f are respectively $S_r^f = \{v_3, v_4\}$ and $S_l^f = \{v_1, v_2\}$. Both of those children explain all the alarms, but the probability that one of the objects in the set is a fault is greater for the left child. Taking $S^f = \{v_1, v_2\}$ and $A^f = \{a_2, a_3, a_4\}$, the result is that the right and left children of S^f are respectively $S_r^f = \{v_2\}$ and $S_l^f = \{v_1\}$. Again, S_l^f has a higher probability of containing a fault, so we choose it for the next input. Now let $S^f = \{v_1\}$ and $A^f = \{a_2, a_3, a_4\}$. Since S^f is a singleton, we add it to the solution set and stop. We notice then that v_1 is the only element in the solution set, and so it is the object that has the highest probability of being in fault (it is actually the simulated fault).

Katzela and Schwartz have evaluated the complexity of their Divide-and-Conquer algorithm. They showed that the complexity of Phase I is $O(N^3)$ (where N is the number of objects in the causality graph). The complexity of Phase II is $O(N)$. Therefore, the overall complexity of the algorithm is $O(N^3)$. However, we have to keep in mind that this is a heuristic algorithm, and therefore this algorithm might miss the solution completely. Katzela and Schwartz have also evaluated an upper bound on errors to the solution obtained with the exhaustive search algorithm.

Figure 3.5 Phase I - Partitioning

Figure 3.6 Phase II - Selection

Let us talk about the advantages and disadvantages of the Katzela-Schwartz method. This framework provides a simple model of the network, and the correlation methods handle incomplete information (missing alarms) up to some level. They give a quick and reasonably accurate answer. It can easily be adapted to a different environment, as long as the causality graph can be calculated and that the probabilities are also evalu-

ated. This, as was mentioned in Chapter 1, is one of the major problems of probability-based algorithms. It is very hard to evaluate the probability that a node will fail in general.

Another problem is that in the algorithms we just saw, the presence of the alarm from a best candidate cause is not evaluated as a criterion for optimality of the solution. For example, consider the causality graph in Figure 3.7, and assume that alarms a_2 , a_3 and a_4 have been received. Even though v_2 explains all the alarms, it is object v_1 that is considered as the best candidate for primary cause, even though no alarm has been received from it.

Figure 3.7 Example of problem in Katzela-Schwartz method

We believe that since the loss of alarms is a low probability event, cause v_2 would be a better candidate under the circumstances. In the design of our algorithms (see next chapter), we will take into consideration the number of non-alarms in a causal chain, and

favour a causal chain where the number of alarms is close to the number of vertices in the chain.

3.3 Yemini's algorithm

3.3.1 Introduction

This algorithm, invented by Yemini [Yemini1994] et al, is based on Coding Theory. Basically, each problem is a message “encoded” into some of the alarms generated. The problem of Alarm Correlation is described here as finding the message (problem) that is best decoded by a set of received alarms. We therefore have to select an optimal set of alarms in order to “decrypt” the message. In the present algorithm, this choice of set is made for each set of alarms expected to be received.

There are two phases to this algorithm. The first one is the *selection of the codebook*. We select a set of alarms to be monitored in order to optimally pinpoint the problem associated with a group of alarms received. This choice of codebook must reflect the desired level of insensitivity to noise during the decoding of the message. The second phase is the *decoding phase*. During that phase, the message is decoded using the selected alarms from the codebook.

In the next section, we will analyse the theoretical basis of this algorithm. We will then explain the algorithm itself, and then discuss its strong and weak points.

3.3.2 Theoretical Basis of the framework

Similar to Katzela-Schwartz's method, Yemini's framework uses *dependency graphs* (a causality graph where all the edges have been reversed), but in a different way.

The vertices here represent *events*, and we are interested especially in two kinds of events: a *problem*, which is an event that may require handling, and a *symptom* (or *alarm*), which is an event that is observable. Those vertices can be marked respectively by P or S. Some events can be neither of those types, or could be both types. In the dependency graph, a directed edge from an event e to an event f indicates that event e causes event f . Figure 3.8 represents a dependency graph over 11 vertices.

Figure 3.8 Network and its dependency graph

Yemini et al assume that cycles in the dependency graph do not contribute to the information on the Alarm Correlation. The argument the authors bring is that a cycle corresponds to causal equivalence. This depends on the type of dependency the edges represent, but also on the meaning of the vertices. Here the vertices really represent *events*, and not objects like in Katzela's-Schwartz's method. Under those assumptions and considerations, Yemini's method reduces a cycle to a vertex.

So far, the model does not include the notion of strength of causality. Similar to Katzela-Schwartz, we could use probabilities to give a measure of the likelihood of the causality to happen. Yemini et al. introduce an algebraic model that generalizes those measures. A *semi-group* H is a set of elements equipped with a binary law of composition “ $\bullet$ ” (that is, a rule that associates with any two elements a and b of H a third element of H represented by $a \bullet b$) such that is *associative* (i.e., for any $a, b, c \in H$:, we have that $(a \bullet b) \bullet c = a \bullet (b \bullet c)$).

A *semi-ring* is a partially ordered set L with an order $\leq$ and two binary operations $\bullet$ (catenation) and $+$ (combination) such that:

1. $\langle L, \bullet \rangle$ is a semi-group with a unit **1** (a *monoid*);
2. $\langle L, + \rangle$ is a commutative semi-group with a unit **0** ;
3. (i) $\forall a, b \in L$, (i) $a \bullet b \leq a$ and $a \bullet b \leq b$ and (ii) $a \leq a + b$ and $b \leq a + b$;
4. $\forall a \in L, 0 \leq a \leq 1$.

The use of a semi-ring is to provide some arithmetic on a measure of causality. The elements of L represent the actual measure. Through 4, this measure takes values between **0** and **1** . The catenation operation gives the causality strength of a given causal chain (a directed path), while the combination measures the strength of a causal implication between two events from many causal chains between those two events. Notice that those two operations correspond to the dependency strength defined in Schwartz-Katzela’s framework.

Let us see now some specific examples of semi-rings in causality models.

In the *deterministic model*, the semi-ring $L = \{0, 1\}$ (that we represent in this case by D) has for the catenation operation the Boolean *and* ($\wedge$) and for combination the Boolean *or* ($\vee$).

In the *probabilistic model*, the semi-ring $P = [0,1]$ is equipped with the product of probabilities as the catenation operation, and the combination operation is defined as $q_1 + q_2 = (1 - q_1) (1 - q_2)$.

We can also represent a *temporal model* using a semi-ring. Let T be the set of all non-negative real numbers that represent the duration of time of a causal action; for example, if $c \rightarrow e$ is a causality from event c to event e , a label of 7.8 means that e will occur 7.8 seconds (or any time unit that is agreed upon) after event c . Now the catenation operation is simply the addition of real numbers, and the combination operation is the minimum of two causalities. In practice, this model would be very difficult to use since the duration of causalities may vary greatly. At best, it would still require significant network analysis to obtain the values.

We define a *causal likelihood model* to be a triplet $\langle G, L, \phi \rangle$ where G is a dependency graph, L is a semi-ring describing the likelihood model and ϕ is a mapping from the edge set of G to the elements of L .

Given a causal likelihood model, we can construct from its dependency graph a *correlation graph*: this graph is a bipartite graph (essentially, this is a graph where the vertices are partitioned into two sets; for more details, see the definition in Chapter 4) where

one part of the partition is the set of symptoms and the other part is the set of problems. There is an edge from a problem p to a symptom s in the correlation graph if and only if there is a dependency chain (a directed path) from p to s in the dependency graph. The label applied to this edge is obtained from applying the combination operation in L to every dependency chain from p to s . Notice that a measure has already been given to each dependency chain from p to s by applying the catenation operation to each dependency chain from p to s . The correlation graph is denoted by G^* . We can then define the *correlation likelihood model* by the triplet $\langle G^*, L, \phi \rangle$. Figure 3.9 represents the correlation graph of the dependency graph of figure 3.8 after removing inessential symptoms and aggregating cycles.

Figure 3.9 Correlation graph

There is some equivalence of Yemini's algorithm to the correlation likelihood model in Katzela-Schwartz's algorithm. If we make the identification of alarms with

symptoms and causes with problems, the clusters of Katzela-Schwartz are exactly set-theoretic versions of Yemini's correlation graph (assuming that the graph is connected, that is, it consists of one piece). Actually, the probability measures put on the dependency chains (the dependency strengths) and between every pair of active objects were the operations of the semi-ring P (probabilistic model) seen earlier. In our algorithms (next chapter), we will use the concept of correlation graphs but because of the nature of our approach, we will consider a multiple edges version of Yemini's correlation graph. We will come back to this in the next chapter.

3.3.3 Coding, Alarm Correlation and description of the Algorithm

Having discussed the theoretical background of the problem, we can now explain how Coding Theory can solve the Alarm Correlation problem. Let S be the set of symptoms and P be the set of problems associated with a given correlation likelihood model. For each problem p , we can represent it by a vector where each component (or position) correspond to a symptom, and the value of the vector component at a given symptom s is the likelihood measure assigned to the edge between p and s . We denote this vector for p by $\underline{p}$, and we call it the *code of p with respect to S* . Clearly, the code vector is a point in the $|S|$ -dimensional space $L^{|S|}$. Now each alarm is the expression of a symptom, and we can represent the set of alarms received by a code vector where we put 1 in the position of a received alarm and 0 if the alarm is not received. We call this vector the *alarm vector*.

The problem of Alarm Correlation in the present context of Coding Theory is now to find problems that match optimally the received alarm vector. Let us see an example to see the issues involved. Take the example described in figure 3.10. Let us consider these

two correlation graphs with respect to the deterministic model and the probabilistic model.

Let us assign some values to the edges according to each model (Figure 3.10 (a) and (b)).

Figure 3.10 Correlation graphs

In case (a), the code vectors (considering the set of symptoms $\{3, 6, 9\}$ as the symptom space) are respectively $\underline{1} = (1, 0, 1)$, $\underline{2} = (1, 1, 0)$ and $\underline{11} = (1, 0, 1)$. For the probabilistic case (case (b)), we have $\underline{1} = (0.8, 0, 0.3)$, $\underline{2} = (0.4, 0.9, 0)$ and $\underline{11} = (0.5, 0, 0.9)$. In the deterministic case, the reader sees immediately that codes $\underline{1}$ and $\underline{11}$ are identical and therefore indistinguishable. Another symptom would hence be needed to distinguish between those two codes. For case (b) however, the two codes though having the same symptom space are distinguishable.

It is important to keep in mind that some alarms may be added (such as spurious alarms) or others could be missing. Suppose for example that in the deterministic case we receive the alarm code $\underline{a} = (0, 1, 0)$. No cause has for code $(0, 1, 0)$, but only one cause

contains the symptom 6 in its code, namely 2. This means that there is a possibility that symptom 3 was missing and that this cause is the correct answer. But there are also other possibilities if we stay in this space of causes: alarm 6 may be a spurious alarm and symptom 3 and 9 may be missing. Of course, the conjunction of those three events is more remote than the event in which alarm 3 is missing. This last example shows that it is possible to refine the symptom space so that instead of processing many alarms to decode an alarm vector code, we only need to analyse the alarm vector for the presence of some symptoms that may distinguish the code right away from some other possibilities. On the other hand, we must be careful that the code of the causes under this new set of symptoms can be distinguishable enough, as the previous case of codes 1 and 11 taught us. A carefully selected subset of symptoms will be named a *codebook*. Figure 3.11 (a) shows a correlation matrix of problems with their codes in the space of all symptoms, and Figure 3.11(b) shows the correlation matrix with a codebook of three symptoms, namely $\{1, 2, 4\}$

As we can see, the choice of this codebook allows the distinction between all 6 problems. But a question comes immediately to mind: what happens if an alarm is missing or if a spurious alarm occurs? Such events, if not carefully controlled at the moment of the selection of the codebook, can lead to an erroneous diagnosis.

(a)

		problems					
		p_1	p_2	p_3	p_4	p_5	p_6
symptoms	1	1	0	0	1	0	1
	2	1	1	1	1	0	0
	3	1	1	0	1	0	0
	4	1	0	1	0	1	0
	5	1	0	1	1	1	0
	6	1	1	1	0	0	1
	7	1	0	1	0	0	0
	8	1	0	0	1	1	1
	9	0	1	0	0	1	1
	10	0	1	1	1	0	0
	11	0	0	0	1	1	0
	12	0	1	0	1	0	0
	13	0	1	0	1	1	1
	14	0	0	0	0	0	1
	15	0	0	1	0	1	1
	16	0	1	1	0	0	1
	17	0	1	0	1	1	0
	18	0	1	1	1	0	0
	19	0	1	1	0	1	0
	20	0	0	0	0	1	1

correlation matrix

(b)

		problems					
		p_1	p_2	p_3	p_4	p_5	p_6
symptoms	1	1	0	0	1	0	1
	2	1	1	1	1	0	0
	4	1	0	1	0	1	0

codebook of radius 0.5

(c)

		problems					
		p_1	p_2	p_3	p_4	p_5	p_6
symptoms	1	1	0	0	1	0	1
	3	1	1	0	1	0	0
	4	1	0	1	0	1	0
	6	1	1	1	0	0	1
	9	0	1	0	0	1	1
	18	0	1	1	1	0	0

codebook of radius 1.5

Figure 3.11 Examples of Codebooks

In order to have some noise resilience, we need to introduce a measure of distinction between the codewords. Such a measure is given by the *Hamming distance* between the codes. The Hamming distance between two codes is defined as the sum of the componentwise difference of the two vectors of code. For example, The Hamming distance between codes $\underline{p}_1$ and $\underline{p}_2$ is given by $d_H(\underline{p}_1, \underline{p}_2) = (1 - 0) + (1 - 1) + (1 - 0) = 2$. A codebook is characterized by half the minimal distance between the codes in its symptom space. This number is called the *radius of a codebook*. In the case of the codebook in Figure 3.11(b), the radius is 0.5. The bigger the radius of a codebook is, the better the resilience to noise the decoding will be. For example, a codebook with radius more than one can correctly decode any alarm vector that experienced the loss or spurious generation of one symptom, and detect any two errors in an alarm code. These observations can be generalized to the following theorem: a codebook C will correctly decode a code vector with k errors and detect $2k$ errors as long as $k < radius(C)$.

An important point needs to be made here: the appearance of an error in the alarm vectors is equivalent to an error in the correlation model. The idea of using a correlation model and a codebook of restricted symptoms is to reduce the processing time in the identification of problems, and at the same time, make the model resilient to possible errors in symptoms. We will see in the next paragraph some ways to construct an optimal codebook. In order to determine an optimal codebook, we need to generalize the Hamming distance. We have seen so far the example of the deterministic model using the original Hamming distance.

For the probabilistic model, we define a distance measure by $d(\underline{a}, \underline{b}) = |\log(\underline{a}) - \log(\underline{b})|$, where $\log(\underline{a}) = \sum_{i=1}^n \log(a_i)$ for $\underline{a} = (a_1, a_2, \dots, a_n)$. Note that we define here $\log(0/0) = 0$ and $\log(0/a) = 0$ for $a \neq 0$. For example, for the codebook described in Figure 3.11(c), the distance between 1 and 11 is given by $d(\underline{1}, \underline{11}) = |\log(8/5) + \log(3/9)| = |\log(8/15)| \cong 0.6286$. Hence, probabilistically, the two codes are distinct.

If S is the global set of symptoms and C is a codebook, that is, a proper subset of S , we clearly have that $r_C(P) \leq r_S(P)$, where P is the set of all problems. Suppose that in constructing a codebook C , we want a level of distinction d . The problem of designing the optimal codebook C is then the problem of finding a subset C of symptoms from S such that $r_C(P) \geq d$. How can we get such an optimal subset of symptoms? One way is to simply use a pruning algorithm. For example, one could remove symptoms from the correlation matrix model until a suitable codebook is obtained. The chosen symptoms should be in order according to the minimum effect on $r_S(P)$.

Having discussed how to find a codebook, we need to find a way to decode the received alarm codes. The maximum likelihood of an alarm to a code vector varies according to the distance measure. This is the main tool for decoding. But in some situations, in order to properly decode, we need to determine if either loss of symptoms in the alarms occurred or spurious symptoms in the alarms were generated. In the case of a symmetric measure, distinctions between a spurious and a lost alarm is in general difficult. This decoding process can be greatly improved by extending the distance measure to an asymmetric correlation measure made of two non-negative functions, each measuring the

distance from 1 (measuring a lost symptom) and the distance from 0 (measuring a spurious alarm).

Let us talk now about the advantages and drawbacks of this algorithm. The advantage is that this algorithm provides a simple model on which Alarm Correlation can be achieved. It is resilient to noise (either loss of symptoms or generation of spurious alarms). The use of a codebook (that is, a reduced set of encoding symptoms) reduces the time in decoding significantly in comparison to the full use of all symptoms in the model.

There are also disadvantages to the coding method. There is a hidden assumption that we can always greatly reduce the number of symptoms in order to construct a high radius codebook. This depends totally on the dependency graph. If the graph has long dependency chains, the model will be harder to simplify in its number of diagnostic symptoms. In order to understand this point, consider the extreme example where a dependency graph is actually a path as in Figure 3.12. Then, to correctly diagnose any problem, we

Figure 3.12 Examples of causality path

need all the symptoms in the graph. Another problem is that there is no mention of the time period over which the alarms have to be received to start the diagnosis. This problem is reduced somewhat if we make sure the codebook has a radius large enough so that it can handle missing symptoms needed for the diagnosis, but the time of transmission of some alarms is required for the choice of the codebook, so that a safe time window can be used for alarm reception. In the absence of such a time window, “non-codebook” symptoms must be considered in order to improve the correlation: even though the correlation is improved by the restriction of the analysed symptoms to those of the codebook (in the situation where most of the codebook symptoms are received), the other symptoms carry as well non-negligible information that can help in the case of an absence of codebook symptoms.

Yemini [Yemini1994] suggested a model in which the time plays a role; we remind the reader about the temporal model T where the edges are labelled with the expected duration time for a dependency to happen. The problem with such an assumption is the variability under different situations of this time parameter. Under some faults, the alarms will be sent very fast and will be received reasonably fast by the Alarm Correlation engine, but in other situations, the alarm may be retained or dropped or masked in the occurrence of another fault. A better model may be obtained by the use of a time interval $[t_1, t_2]$ on which we expect the value of the duration of the dependency to happen. Such a model would involve *interval arithmetic* methods in the construction of the new semi-ring model T' . Another possibility is to work with the timestamps of the alarm (which reflects the time of occurrence of an event) and their relative order, rather than the duration time for a dependency. This approach is the one we will put forward in the design of our

algorithms in the next chapter.

3.4 Application to a specific example

We now look at a very special situation in a network that contains peer-to-peer servers and where the probability of failure and the dependency weight of the system of servers is the same for all those servers. Figure 3.13 shows the causality graph of our example network.

Figure 3.13 Causality graph of example

The network is made of four peer-to-peer servers and of two more nodes. The different probabilities are given in the figure. Notice that v_1 , v_2 , v_3 and v_4 all have the same probability of failure, and the edges between those servers all have the same dependency weight. Suppose that alarms a_1 , a_2 , a_3 and a_4 exist. Suppose that the timestamps of

those alarms can actually be ordered along that sequence, that is $t_1 = 1$, $t_2 = 2$, $t_3 = 3$ and $t_4 = 4$ (those timestamps are chosen only to simplify the discussion).

Before applying Yemini's algorithm, we must notice that the peer-to-peer servers form a cycle. Since the preliminary pre-processing phase of the algorithm reduce any cycle to a node, we see readily that Yemini's algorithm cannot determine the primary cause to the problem.

Now let's see what Katzela-Schwartz's method can do. Even if we apply the exhaustive search algorithm, there will be four minimal sets that explain all the alarms (we use here a parameter $W = 0.001$) and whose probability of containing a fault is maximal (and actually equal): $\{v_1\}$, $\{v_2\}$, $\{v_3\}$, $\{v_4\}$. Hence, the algorithm cannot find the solution. Furthermore, if we apply the Maximum Mutual dependency algorithm, we could select in the partitioning algorithm the sets $\{v_1, v_2\}$ and $\{v_3, v_4\}$ as children of the set $\{v_1, v_2, v_3, v_4\}$. Since we must select one child at each step of the selection phase, we might choose the $\{v_3, v_4\}$ and end up with the wrong solution (either $\{v_3\}$ or $\{v_4\}$ as final solution). The reader will of course notice that a simple look at the timestamps permits one to select the correct solution, that is $\{v_1\}$.

Such examples as the preceding one have inspired the algorithms we will describe in the next chapter.

Chapter 4

Algorithms

4.1 Introduction

In this chapter, we describe our solution to the temporal Alarm Correlation problem involving peer-to-peer interactions. We describe here a *centralized algorithm*, where we assume that the Alarm Correlation engine is in one (physical or logical) location. The reader who wishes to refresh his/her memory on the subject can consult a general Graph Theory book (like [Bondy1976] or [Harary1972] for example).

4.2 Centralized Algorithm

4.2.1 Formal Expression of the Problem

We consider here a formal expression of the problem in the context of combinatorial optimization. We also assume that the timestamps applied by the different hosts are consistent, i.e., the clocks of all hosts are synchronized. This assumption, as long as we assume also persistence of the clocks, brings no loss of generality to our approach, since if the clocks vary, we can bound the variability of the timestamps (with respect to a given reference clock for the whole system) within a certain interval of time, and work with those intervals rather than actual values of time. Our algorithm involves the comparison of

timestamps of events, and working with time intervals would involve very minor changes in the steps. However, for the clarity of exposition, we suppose consistent, single-valued timestamps. The reader may argue (very rightly) that clocks could be non-persistent in network nodes, that is to say, they reset as the host on which they reside restart after a major fault. Such a phenomenon implies that bounding the maximum variability of timestamps is very hard. However, we still believe that our assumption on persistence of clocks is reasonable, even if it is only for the sake of fixing the conditions of the problem.

We base our algorithms on a type of causality graph similar to those developed by Katzela and Schwartz in the preceding chapter. We assume that the set of vertices in our causality graph represents a set of *objects* in the network, that is, an entity that has a separate and distinct existence up to a desired level of abstraction. Those entities could be hardware entities, software entities, or states for example. We suppose that each object is *active*: it performs a function in the network. A *fault* or *malfunction* occurs when an abnormal condition happens in the network, and such an abnormality or fault is represented by the corresponding object in the graph.

As already mentioned in Chapter 2, we assume that each object can generate an *alarm*, and that this alarm is simply a message that contains two fields: the *identity* of the object generating the alarm, and the *timestamp* of the alarm. The identity of the object generating the alarm is simply the name of the object in the causality graph from which the alarm originates. We therefore assume that the identity maps one-to-one the set of alarms in the network to the set of objects or vertices of the causality graph that can be in fault. The timestamp is the time at which the alarm was generated. We assume that the

timestamp is the exact time at which the fault occurred. We do not address the problem of intermittent faults in the network. We also do not address the problems associated with sensor reliability. A vertex or object is either functioning or malfunctioning, but not both at the same time.

We also make the assumption here that an alarm is generated with a *unique* timestamp. We do not address the case where an object sends the same alarm more than once and with different timestamps (this may happen for example when different objects' malfunctions bring a malfunction in a same object but at different times). However, such a case can be included in our framework by subdividing the object into two distinct objects in the causality graph, or by attaching a sequence number to the alarms (so receiving an alarm with a sequence number greater than 1 indicates that an alarm was previously generated from this object). If many similar alarms are sent from the same object but with the same timestamp, this does not matter to the algorithm since it will keep only one copy of this set of identical alarms.

Each directed edge in the causality graph represents *causality*; a directed edge $v_i v_j$ represents that a failure of object v_i has for possible cause a failure of object v_j . In order to be able to qualify the different possible causes of a failure, we associate with each directed edge $v_i v_j$ a (*dependency*) *weight* $w_{ij} = -\log(p_{ij})$, where p_{ij} simply represents the conditional probability $p_{ij} = P(v_i \text{ fails} \mid v_j \text{ fails})$. For consistency of our weight function, we always assume that $p_{ij} > 0$: instead of allowing the case $p_{ij} = 0$, we simply delete the edge between v_i and v_j . The value of p_{ij} does not necessarily have to be a probability *stricto sensu*: it can be a likelihood measure or fuzzy measure of the implica-

tion v_j fails implies v_i fails. A default value of 1 can be assigned without any problem. Of course, the performance of the fault identification process will be affected. Notice that because of the definition of our weight function, probabilities of 1 gives a weight of 0. Given a path (or *causal chain*) $\vec{P} = v_1 v_2 \dots v_n$ in G , we define the (*dependency*) *weight* of $\vec{P}$ by the expression $w(\vec{P}) = \sum_{i=1}^{n-1} w_{i(i+1)}$. Since we address the problem of temporal Alarm Correlation involving peer-to-peer interactions, the causality graphs that we consider may contain bidirected edges as well as directed cycles. We also suppose for the moment that a unique primary fault occurred. We will see later in our algorithm how to include the possibility of multiple faults. We also assume for the moment that the set of alarms is filtered, and that spurious alarms are eliminated as much as possible. Again, later adjustments in the algorithm will allow us to incorporate erroneous alarms.

We define in this scheme a *dependency chain* to be a causal chain in G where we reverse the orientation of its edges.

A *tree* is an undirected graph that contains no cycle (or “loop”). We define a *directed tree* to be an undirected tree with a special vertex called the *root* where we orient all the edges in the direction outward from the root (and therefore there is a directed path from the root of the tree to each vertex in the directed tree) We define a *reversed tree* to be a directed tree where we have reversed all the edges (and therefore there is a unique directed path from each vertex to the root of the tree). The “root” of the reversed tree is called the *sink* of the reversed tree. Figure 4.1 shows a directed tree and a reversed tree.

With those hypotheses and definitions, we now express the Temporal Alarm Correlation Optimization Problem (TACOP).

Figure 4.1 Directed tree and reversed tree

TACOP: Given a causality graph G , find a vertex v and a reversed tree T in G such that:

1. v is the sink of T ;
2. T contains all the alarms;
3. The timestamps of T are consistent, that is, the timestamps are decreasing in a traversal of any path in T ;
4. the sum of the number of non-alarm vertices in each path in T from the sink (excluding it) to an alarm is minimum;
5. Under the assumption that conditions 1-4 are met, we also impose that the sum of the weights of all the paths from the sink to each alarm in T is minimized.

We now explain each restriction on the feasible space of solutions.

It is important to notice here that we actually have two optimization problems: the first one finds all solutions to conditions 1 to 4, and we then find the solution out of this set that satisfies conditions 5.

The reason for which we are looking for a reversed tree containing all the alarms appears naturally: the reversed tree reflects a causal cascade of events that includes the generated alarms. The sink of the reversed tree represents the primary fault that caused the cascade of faults following its occurrence.

Conditions 4 and 5 reflect that we want to obtain causal chains that are as certain as possible. The different dependency chains that lead from a common candidate cause to each alarm will then contribute to make the sink candidate the best possible choice.

Condition 4 represents optimality for the presence of non-alarms vertices in the causal chains. It is important to note that condition 4 reflects the main objective function in this optimality problem: we first find a solution that satisfies 1-4, and then satisfies 5. Note that even in the case of the exact causal tree representing the real chain of events, it is possible that the tree is not composed of alarms only. We assume here indeed that it is possible that some alarms may be missing. However, we assume that we would receive most of the alarms, as we expect to receive the maximum number of alarms from the cascade of faults. It should be clear that the determination of the possible primary causes depends on the number of missing alarms on the number of alarms generated. The greater this number, the less information we have and the more difficult the Alarm Correlation will be.

Note that according to condition 4, a missing alarm that is common to many paths will be counted many times in the objective function (number of non-alarm vertices), and so the weight of those paths will reduce the optimality of the solution. Because such a missing alarm implies many independent events (the events implied by this vertex according to those paths), it is reasonable to put a higher weight on it. Note also that because we make the sum of the length of the paths according to the missing alarms rather than simply counting the alarms in the tree, this objective function favors paths that have missing alarms around the *origin* rather than around the *termination*. This characteristic of the objective function makes sense, since a dependency chain has more certainty if a missing vertex is at the end than at the beginning. It is important to note that the sink is never counted more than once as a missing alarm. The explanation is very simple: we allow certainly one missing alarm as the sink but two or more at the top would lead to less solid conclusions.

The fact that we take into account the alarms received is one important difference between our approach and the Katzela-Schwartz algorithm. We remind the reader that, in the Katzela-Schwartz approach, the choice of a candidate as a primary cause did not rely much on the presence of its alarm. In our algorithms, we base our conclusions on the available information, and in particular the received alarms. In a modern network, there could be situations where many alarms are missing, for example, in the break of a link. One endpoint may not receive the alarms from the other endpoint. However, the best approach to take in that situation is to rely on the alarms received and to try to find a common cause to those alarms. This is a principle we value in this work. We make however one important assumption; if a vertex has out-degree greater than 1 and in-degree at least 1

in the causality graph, we suppose that the alarm will be sent a number of times proportional to the out-degree. This reduces the probability that the alarm from such a vertex would be lost. As we will see later, those alarms are crucial in the correlation.

Condition 5 reflects the fact that solutions with long causal chains of low dependency weight should be avoided. In the type of situations we address, the propagation of the faults is assumed to behave in a “breadth-first” expansion rather than in a “depth-first” expansion: indeed, the cascade of faults, after a break, propagates rather in an arborescent expansion rather than along a few long paths. This comes from the assumption that if a fault occurs at an object, its effect will be sensed by all other entities depending on this object (depending on their conditional probability of failure of course). We believe it would be unrealistic to model the fault as cascading along a single path for example, when many other objects would depend on the original faulty object. Let us consider a graphical example to illustrate this point.

Suppose we have a vertex u that can induce three faults v , w and r . Suppose that $P(v|u) = 0.7$, $P(w|u) = 0.8$, $P(r|u) = 0.8$, $P(w|v) = 1$, and that $P(r|w) = 1$. Let C_1 be the objective function associated with condition 5. Then the cost of the reversed tree in Figure 4.2 is

$$C_1(T_1) = -\log(0.7) - \log(0.8) - \log(0.8) = 0.8029$$

while the cost of reversed tree T_2 is

$$C_1(T_2) = -3\log(0.7) - 2\log(1) = 1.07$$

Figure 4.2 Propagation of faults

and so the cost of T_1 is smaller than the cost of T_2 .

The TACOP problem is related to the *Steiner Tree Problem* ([Winter1987], [Christofides1972]), which can be stated in the following form:

Steiner Tree Problem: *Given a weighted undirected graph $G = (V, E, w)$ and a subset of vertices U , find a minimum cost tree T in G such that T contains all the vertices of U .*

This problem is well-known to be NP-complete. Our problem is actually “harder” than the STP, since our tree must be directed and consistent with respect to the timestamps. The TACOP is therefore NP-complete.

In order to find a solution to the TACOP, we will rather solve a derived problem. We need to introduce first some concepts related to shortest paths in graphs. Given a directed graph G and two vertices u and v , a path $\vec{P}$ is said to be a *shortest path from u to v* if the distance of $\vec{P}$ is minimum among all paths in G from u to v . The usual distance in a graph (number of edges contained in the path) can be generalized to a *metric* in G : that is, a function $d: G \times G \rightarrow \mathfrak{R}$ (where $\mathfrak{R}$ is the set of real numbers) that satisfies the following three axioms: (1) for all u and v in G , $d(u, v) \geq 0$; (2) $d(u, u) = 0$ and (3) for all $u, v, w \in V(G)$, $d(u, w) \leq d(u, v) + d(v, w)$. The reader will notice that our definition of metric slightly differs from the mathematical definition. Here we do not (and do not want to!) impose that $d(u, v) = 0$ implies that $u = v$. This would lead to unrealistic situations.

There are many examples of metrics in a directed graph G , and especially if G is a causality graph. For example, we can define the *non-alarm metric* of a path to be the

number of vertices in the path that do not have alarms. The *non-alarm metric between two vertices* can be defined in the following way: for any two vertices u and v , it is the minimum number of vertices that are not alarms over all directed paths from u to v . The dependency weight also induces a metric in a causality graph G : the *dependency metric* of u and v is the minimum dependency weight over all directed paths from u to v in G . The preceding two metrics will allow us now to introduce a new optimization problem in Alarm Correlation, and we will show later that a solution to this problem under some conditions gives a solution to the TACOP.

TACOP2: Find a vertex v and a set of directed paths $\Pi = \{ \vec{P}_{a_i} \mid a_i \text{ is an alarm} \}$ in the causality graph G such that :

1. v is the destination of all paths in Π ;
2. Each alarm a_i is the origin of one path $\vec{P}_{a_i}$ of Π ;
3. each $\vec{P}_{a_i}$ is a minimum path with respect to the non-alarm metric (i.e., the number of non-alarms in $\vec{P}_{a_i}$ is minimum over all the directed paths in G from a_i to v);
4. for all paths that satisfy conditions 1-3, we also impose that each $\vec{P}_{a_i}$ is a minimum path with respect to the dependency metric.

We now show that a solution to TACOP2 is a solution to TACOP. But before some observations must be made.

Lemma 1. *Subpaths of shortest paths are shortest paths: that is to say, let $\vec{P}_1$ be a path that is a shortest path between two vertices u and v according to some metric, and let w be an intermediary vertex between u and v in $\vec{P}_1$. Then the portion of $\vec{P}_1$ between u and w is a shortest path between u and w .*

Proof. Suppose on the contrary that there is a path $\vec{P}_2$ from u to w whose length is less than the length of $\vec{P}_1$. Let us assume that $\vec{P}_2$ has for intersection with $\vec{P}_1$ only the vertices u and w (the case where the intersection set is bigger can be proved in a similar manner). The situation is illustrated in Figure 4.3. Then, the path $\vec{P}_2$ concatenated to the subpath of $\vec{P}_1$ from w to v would be a path from u to v of length shorter than the length of $\vec{P}_1$, which is a contradiction. This proves the lemma.

Figure 4.3 Lemma 1

Let v and Π be a solution to TACOP2. Let H be the graph obtained by putting together all the paths contained in Π . It is important to notice here that even though Π is a set of paths, H may not be a tree. There could be vertices that belong to more than one

path in Π . We say that H is *time consistent* if any path in H is consistent with respect to timestamps of the alarms contained in it. In other words, the timestamps of the alarms met along a traversal of any path in H to v are in decreasing order.

Theorem 1. *Let v and Π be a solution to TACOP2, and suppose that its resulting graph H is time consistent. Then H contains a reversed tree that is a solution to TACOP (whose sink is v).*

Proof. We first show that H contains a solution T_0 to conditions 1–4 of TACOP. It is important to notice that, as a consequence of Lemma 1, if a vertex u of H is at the origin of many paths to v , then all these paths have the same length with respect to the non-alarm metric (for otherwise, there would be another vertex w that would have a path from itself to u and then along the short path to v that would be shorter than its path in Π , a contradiction). This implies that as we remove edges (and possibly resulting non-alarm leaves) from H to produce a reversed tree, say T_0 , the length of a path from any alarm a_i to v is equal to the length of the corresponding path $\vec{P}_{a_i}$. The smallest possible sum of the non-alarm length of all paths from the alarms to v we can obtain is the sum of non-alarm length of all paths in Π (those are shortest non-alarm length paths). Therefore, T_0 is a solution to conditions 1–4 of the TACOP. Since the same argument applies to the dependency weight as for the non-alarm metric, the proof shows also that condition 5 of the TACOP is met by T_0 . This proves the theorem.

The reason we put in the previous theorem the hypothesis at the effect that H must be time consistent is illustrated in Figure 4.4. We show a very simple graph H correspond-

ing to a solution to TACOP2, where the alarms are denoted by the vertices in black. We notice that the graph is time inconsistent, and if we remove the edge v_3v_1 to make a reversed tree, we obtain an inconsistent path $v_5v_3v_2v$ in the resulting graph. However, this situation can be avoided if we assume the following hypothesis: if an object fails, then the dependent objects that will be affected will fail together in approximately the same time and the same region of the causality graph (one dependent object will not fail much later than its other siblings) . This assumption is reasonable, and we must also remember that an intersection at an alarm forces the graph to be “locally” time consistent at this intersection (in our example, if the vertex v_3 would be an alarm, there would be time consistency). The problem may occur only at a non-alarm intersection vertex. Since we assume that most of the alarms are present in our paths, this reduces the probability of having a solution that may be time inconsistent. But if we cannot make this assumption in a given network, it is always possible to remove an edge so that the resulting graph is time consistent, and the solution tree will be an approximation of the solution to the TACOP.

In the next section, we will describe the centralized algorithm that solves TACOP2.

4.2.2 Algorithm

We propose a heuristic algorithm that will find a feasible solution to TACOP2 under constraints 1, 2, and 3, and is a solution or an approximation under conditions 4 and 5. The next paragraphs are ordered so as to discuss the successive steps of the algorithm.

Figure 4.4 Time inconsistency in causality graphs

We should first mention that we assume that the causality graph G is implemented as an array of *adjacency lists*. Those are the linked lists of neighbors for each vertex of the graph, and the set of all those linked lists is grouped in an array.

Preprocessing Phase: Computation of the Path Repository and the Transitive Closure

The following paragraphs constitute a brief discussion of the preprocessing phase to the actual diagnostic process. This phase is not a part of the diagnostic process *per se*, but its importance is capital.

The essence of the algorithm resides in a selection of possible causality chains from each of the received alarms to a possible candidate cause. In order to perform a selec-

tion of causality chains, it is necessary to have *a priori* a list of all the possible paths in the causality graph from vertex, their termination, together with their length and their dependency weight. It is easy to show that this pre-processing activity has exponential complexity. Since Alarm Correlation must be performed rapidly, such a process must be done *prior* to any diagnosis activity. We will call the resulting list of paths the *paths repository* (of the *causality graph*). This repository is essentially an array containing all the possible directed paths in the causality graph. Each element of the array contains information about the paths associated with this array element, but contains also the path itself (which is represented by an array of pointers, each pointer containing the address of a vertex of the path). The reason we use an array to represent the paths rather than a linked list with pointers, is that it can be useful for later analysis to access a vertex of the path directly, rather than traversing a linked list up to that vertex.

The paths repository is created by traversing all the paths in the causality graph and by recording each path in an element of the paths repository array. This can be achieved by taking all ordered pairs of vertices in the causality graph, and then by finding all paths from the first vertex of the couple to the second vertex of the couple of vertices chosen. Finding all such paths can be done by an exhaustive traversal of the graph from the first vertex until we reach the second vertex. It is at the moment of this discovery of the paths that we can better get information about that path, like the length of the path, its dependency weight, its origin and destination, etc.

Given any two vertices in the graph, we need a more condensed source of information about the paths between those two vertices. For this purpose, we construct an auxil-

iary graph, the *transitive closure* $\bar{G}$ of G , in the following way: the vertices of $\bar{G}$ are exactly the vertices of G , and there is an edge $v_i v_j$ in $\bar{G}$ if and only if there is a directed path $\vec{P}$ between two vertices v_i and v_j in G . Notice that $\bar{G}$ may contain multiple edges between two given vertices: there are as many directed edges between two vertices $\bar{G}$ as there are directed paths between the two same vertices in G . We assign to each edge in $\bar{G}$ the weights relative to its corresponding path in G : in our case, we assign the graph-distance of the path and the dependency weight. The set of neighbors of any vertex of the transitive closure are ordered in an increasing value of their minimum distance from this vertex. For example, suppose a vertex u has for neighbors in $\bar{G}$ v and w , and suppose that they are at distance of 3 and 7 respectively from u (the minimum length of a directed path from u to v is 3, and the minimum length of a directed path from u to w is 7). Then, in the neighborhood list of u , w will come after v . Also, the edges between any couple of vertices are ordered by increasing value of the length of the path in G they represent, and for a given value of length, they are then ordered by increasing value of the dependency weight of the paths they represent in G . In other words, given two vertices u and v in the transitive closure, their edges are organized by lexicographic order relative to the couple *(length, dependency weight)*. This suggests that the set of edges between two given vertices u and v can be represented by a double linked list attached to the element v in the linked list of the neighbors of u .

For each edge in $\bar{G}$, we may need to access the corresponding directed path in G . Each path in the paths repository being in bijection to its edge in the transitive closure of G , we represent this bijection (one-to-one mapping) by a couple of pointers between the path of the path repository and its edge in $\bar{G}$.

Both the transitive closure and the path repository of a causality graph can be large, but they are fixed data structures in the main database, and they do not need to be updated during the Alarm Correlation process. We can actually store them in a linear file.

We can now describe the main algorithm itself. In the input of the algorithm, besides the causality graph G we just described and its auxiliary graphs, we need two parameters: a *distance bound parameter* δ_d and a *(dependency) weight bound parameter* δ_w . The alarms are taken from the set of alarms received within a time window t .

Algorithm Temporal_Alarm_Correlator

Input: - causality graph G :

- its transitive closure $\bar{G}$;
- its path repository $\text{PathsRep}(G)$;
- the set of alarms received during a window of time t ;
- a distance bound value δ_d ;
- a dependency weight bound value δ_w .

Output: a reversed tree solution T_{sol} to TACOP2.

Phase I: Processing of the alarms

As we receive the alarms, the first step is to order them according to their timestamps. We assume that the alarms are grouped in a linked list. Their ordering can be done very easily and efficiently in polynomial time ($O(n \log n)$), where $n = |G|$, and will help in later analysis. Subsequent alarms are also inserted in this list according to their times-

tamp.

Phase II: Construction of the Cause-Alarm Bipartite Graph

A *bipartite graph* is a graph whose vertex set can be partitioned into two subsets X and Y , and each edge in the graph has one end in X and one end in Y . In our case, the Cause-Alarm bipartite graph G' has its vertex set partitioned into the *set of received alarms* A and the *set of possible causes* C .

The edges are always oriented from a cause towards an alarm. They represent a dependency path that links an alarm to a cause. Therefore, they correspond to an edge of the transitive closure of the causality graph, but with reverse direction. As for $\bar{G}$, multiple edges are possible in G' . Figure 4.5 shows an example of a Cause-Alarm graph

The main point in the construction of G' is to optimally choose the candidate causes (the vertices of C) and the causality paths from the received alarms to these causes. This stage of the algorithm is extremely important, as the number of possible candidates chosen will directly influence the complexity of the algorithm. If we choose all the possible destinations of causality chains from each alarm, we could end up with exponential complexity. We need to choose the candidates extremely carefully. The *distance bound* (respectively *dependency weight bound*) represents the greatest distance or length (respectively dependency weight) within which the paths from each alarm should be taken.

Figure 4.5 Cause-Alarm graph

In other words, we search and analyze only paths that have length and dependency weight less than K_d and K_w respectively. We propose two methods: a general method (method A) where we use as bounds the values of $K_d = \delta_d$ and $K_w = \delta_w$ directly. Another method (method B) scale these two parameters for each alarm according to the position of the alarm in the sorted alarm vector.

Method A. For this purpose, we will bound our search within the values $K_d = \delta_d$ and $K_w = \delta_w$: for each alarm a_i received, we find all the vertices v that have at least one path from a_i to v with length less than δ_d and dependency weight less than δ_w .

Method B. This second method is similar to method A, except that we scale δ_d and δ_w for each alarm according to its position in the alarm vector. More specifically, for each alarm a_i received, we find the vertices v whose distance from a_i is within a certain function value $K_d(\delta_d, a_i)$ and whose dependency weight is within a certain function value $K_w(\delta_w, a_i)$. One option in our algorithm is to make the distance bound an actual function of the timestamp of the alarm: for example, for a_1 , we assume that the cause should be within a distance at most 2 from it. For the latest alarm, we will assume a distance bound that should be bigger. A distance bound $K_d(\delta_d, a_i) = \delta_d(1 + 2\log_2 i)$ is a reasonable guess, since it represents the distance in a breadth-first expansion of the alarms along a tree. For the dependency weight bound, the value depends on the degree of certainty we want, depending on the alarm. For example, if we know that usually a chain that leads to the emission of a certain alarm a_i is near certainty, we will use a bound value $K_w(\delta_w, a_i)$ that is close to 0.

In both methods, the candidate causes are selected from the transitive closure. The reader remembers that we have ordered the neighbors of a vertex according to their distances (shortest distance), and their edges according to their distances but also to their dependency weight (within a common distance value) in a lexicographic manner. From this ordering, it becomes easy to select all vertices that are within the assigned bounds by traversing the neighbors of an alarm in the transitive closure up to the distance bound and the edges up to the two bounds. When we add a candidate cause to the Cause-Alarm graph, we add as well a reversed copy of the transitive closure edges between the alarm and the cause whose value is lower than K_d and K_w . Thus, the Phase II Algorithm is:

Phase II Algorithm

For (each alarm) do

While (there is a neighbor to alarm at distance $< K_d$) do

*While (there is an edge from alarm to cause in G of length $< K_d$ and of
dependency weight $< K_w$) do*

Add neighbor to G' ;

Add the reversed edge to G' ;

End While

End While

End For

The Cause-Alarm graph is identical to Yemini's correlation graph, except for the fact that we add all the edges representing possible causality chains in G between a given cause and an alarm, within the bounds K_d and K_w . This graph represents all the causes that can explain some alarms (or all) within a certain approximation, and the possible explanation chains to the alarms. While the causality graph, its transitive closure and the paths repository are fixed data that do not depend on the alarms, the Cause-Alarm graph is constructed with each alarm window received.

The Cause-Alarm graph will be our main instrument to determine the best candidates as primary causes of the alarms received. The next step in the algorithm is to consider the *connected components* of the Cause-Alarm graph. The alarms of a connected component of the bipartite graph should be investigated together, since this "proximity" within a connected subgraph of any two alarms suggests the likelihood of a unique cause

to those alarms. Such a connected component of the Cause-Alarm graph constitutes exactly what Katzela and Schwartz [Katzela1995] call a *cluster of alarms*. The determination of the clusters, within the construction of the Cause-Alarm graph, is easy. Indeed, we process the alarms in order and check for each alarm if a possible cause to it in the graph is adjacent to some other alarms. We make lists of those adjacent alarms and causes as we progress, and each of those lists becomes the set of vertices of a cluster. This can be done in polynomial time.

Each cluster is processed one at a time, and we consider one possible primary cause per cluster. This does not mean that there has to be one primary cause in a cluster that can explain all the alarms in the cluster. For the rest of the explanation of the algorithm, we will assume that there is only one cluster in the Cause-Alarm graph, since the multiple clusters case is simply a repetition of the single cluster case.

The next step in Phase II of the algorithm consists of partitioning the causes into two sets: the *single cause list* and the *multiple causes list*. The single cause list consists of all causes that are adjacent to each alarm. Those have clear potential to be the primary causes to all the alarms. The multiple causes list is therefore the rest of the causes. It does not mean that those are necessarily multiple faults. It may happen that because the chosen bounds are too small, we could not find a specific single candidate to each alarm. Those will be examined later. The single cause list is constructed as a linked list where the best candidate (because of its timestamp) is always put at the head of the list. Then, we simply scan in turn each other cause of the Cause-Alarm graph: if the cause is adjacent to each alarm and its timestamp (if it has one) is lower or equal to each timestamp of the rest of the

alarms, then we insert it in the single cause list. If its timestamp (if it has one) is lower or equal to each timestamp of the rest of the alarms but does not explain all the alarms, we put this cause in the multiple cause list. Otherwise, we delete the cause.

It is important to notice that the edges of the Cause-Alarm graph are ordered, for each pair of vertices, in increasing value order.

Phase III: Pruning the Cause-Alarm Graph

The next phase of the algorithm consists in “pruning” the Cause-Alarm graph. As we have accumulated all possible candidate causes and causal chains within the heuristic parameters of distance and dependency weight, we want to eliminate as many of those candidates as we can in order to pinpoint the primary cause of the alarms. For this, we will scan the candidate causal paths in G to see if there is some inconsistency in the timestamps of the alarms and the causes. If, as we traverse the path from the alarm to the cause, the sequence of timestamps is not strictly decreasing, we have to reject this causal path, since it implies that the cause came after the effect. If an edge in the Cause-Alarm graph is the only edge between a candidate for primary cause and an alarm, invalidating this way the quality of this cause to be a global primary cause, the cause is removed from the global cause list and the algorithm moves to the next cause in this list.

In order to get a solution that respects condition 3 of TACOP2, we must keep track of the non-alarm measure of the paths as we traverse them. Later, after having pruned the paths with respect to timestamps, we will re-order the remaining paths according to their non-alarm measure, their weight and their length.

In order to avoid exponential complexity in traversing all these paths, there are two routines we can use: eliminating causal paths hierarchically and in parallel.

A. Hierarchical Pruning of the Paths

In order to improve the computational speed of the pruning process, we make the following observation. Suppose that a path $\vec{P}$ is found to be inconsistent. Clearly, any other path that contains it will be as well inconsistent. This offers a first way to speed up the pruning process.

To perform such a speed-up routine, we need the paths to be organized in a hierarchical tree T called the *Paths Tree* where the root is the empty set (dummy vertex), the next level is made of all the directed edges in the causality graph, and such that each vertex representing a directed path $\vec{P}$ has for children the paths $\vec{P}'$ that are made of the path $\vec{P}$ with one more directed edge, either at the origin or at the terminus of $\vec{P}$. Notice that the paths increase in size as we go deeper in the tree. Since we restrict the size of our paths within the graph distance bound K_d , we can focus our search. It is also important to notice that the different vertices of the Paths Tree can be actually pointers to its corresponding path in the Paths Repository, since the space occupied by a Paths Tree actually composed of paths would be considerably large and not very useful. Let us illustrate the mechanism by an example.

Consider the path $\vec{P}_1 = v_4 v_5 v_7 v_9 v_{10}$. This path contains many subpaths: for example, $v_5 v_7 v_9$, $v_4 v_5 v_7 v_9$, $v_5 v_7 v_9 v_{10}$, etc. If the path $v_5 v_7 v_9$ has been found to be inconsistent, then so are the paths $v_4 v_5 v_7 v_9$, $v_5 v_7 v_9 v_{10}$ and $\vec{P}_1$. We can

use the Paths Tree to mark all the paths that contains $v_5 v_7 v_9$ and marked them as time inconsistent, up to length K_d . In this manner, we avoid marking all paths in the graphs, which could lead to exponential complexity.

B. Use of MultipleThreads

In order to speed up the pruning process, we also use parallel processing. We use one thread of control (basically one process) per possible candidate up to a maximum number of parallel scanning threads. It is clear that we need a mechanism to synchronize the threads, and to lock and unlock the data that are shared between those scanning threads. This is especially necessary if we use the preceding hierarchical pruning routine, where we have to “lock” sections of the Paths Tree.

Phase III Algorithm

For (each cause in G') do

Spawn a thread;

For (each alarm adjacent to cause) do

For (each unmarked causal path P from cause to alarm) do

time = 0;

$P.numberOfNonAlarms = 0$;

While((P is not marked as inconsistent) and (there is an unscanned vertex in P)) do

If (vertex is an alarm) then

If (vertex.timestamp < time) then

```

        Mark P as inconsistent;

        Mark paths reachable from P in T (up to distance  $K_d$ ) as
            inconsistent;

    Else

        time = vertex.timestamp;

    End If

    Else

        P.numberOfNonAlarms = P.numberOfNonAlarms + 1;

    End If

    End While;

    End For

    End For

    End For

```

At the end, we obtain a pruned bipartite graph where the edges and the vertices are consistent according to their timestamps. The paths are also sorted towards each alarm according to their non-alarm length and their dependency weights. To find the best solution, we scan each cause to see which of them is a neighbor to each alarm in the Cause-Alarm graph with the minimum number of non-alarms in the reversed tree it belongs to. Clearly, this gives us a best explanation to the alarms (no alarm is missing).

Suppose now that no candidate explains all the alarms. Then either the bounds used were too small, or there are multiple primary faults. We then augment our bounds

values, and run the algorithm one more time. If no single candidate appears, we can then suspect that multiple, primary causes may have occurred. This is especially the case if no cause explains all alarms after many bounds relaxation, or if there is a cause explaining the alarms but with a high number of non-alarms in its reversed tree (say with $>50\%$ non-alarms).

There is also the possibility where we obtain a time inconsistent graph. Then we find the intersection vertices (those must be non-alarm vertices) where there is local time inconsistency. In order to obtain a consistent reversed tree, we simply keep the outward edge to this vertex as the edge that goes the alarm with the earliest timestamp (we might have to track down the path to find such an alarm), and we remove all the other outward edges to the intersection. Proceeding similarly for all other inconsistent intersection, we obtain a reversed tree that is time consistent and is an approximation to the solution to TACOP2 (it is an approximation since we cannot use Theorem 1 for this graph).

At the time of writing this work, we do not see a way to evaluate how close this approximation is to the real solution. But again, we wish to emphasize, as explained earlier, that under the assumption that most alarms are received and that resulting failures from an object must occur at similar times, such a time inconsistent situation is rare.

Complexity

It is difficult to bound the complexity of our algorithm, as it depends on the number of alarms, the type of graph, the connectivity and the bounds used. However, it should be clear to the reader that the most complex task is the pruning. However, if we let n_A be

the number of alarms and n_R the number of vertices that can be reached from an alarm by a path of length at most K_d , then the complexity of traversing each path bounded in length by K_d from a cause to each alarm is bounded by $O\left(K_d n_A n_R^{K_d-2}\right)$. However, if we assume that the causality graph has a tree structure with each node having degree at most d (so we can choose $K_d = \left(\log_d n_R\right)$), there is a unique path from each alarm to the candidate cause and so the complexity would rather be bounded by $O\left(\left(\log_d n_R\right) n_A\right)$. This compares to the exhaustive search of Katzela-Schwartz that is $O\left(2^{n_R}\right)$. However, their Maximum Mutual Dependency algorithm, which is an approximation of the exhaustive search, has complexity $O\left(n_R^3\right)$. Yemini claims that the decoding part of his algorithm has complexity of the order of $(radius - 1) \log p$, where the radius is the radius of the codebook and p is the number of problems. This supposes of course that a satisfactory codebook has been found. In both cases (Katzela-Schwartz and Yemini), the computation of the transitive closure is a hidden part of their algorithms, and since this has to be exponential, we can claim that our algorithm compares in complexity to theirs.

Our centralized algorithm can hardly scale to many thousands of vertices, since the number of paths in the transitive closure grows exponentially. We could still bound the search in the transitive closure, but as the transitive closure grows exponentially with the size of the causality graph, the bounds would become very restrictive. One way to avoid the complexity associated with a large causality graph is to use a distributed approach. A hierarchical version of the centralized algorithm we have described will be briefly discussed in Chapter 6.

In the next chapter, we will show an implementation of the algorithm and the results obtained.

Chapter 5

Tests

5.1 Introduction

In this chapter we present the testbench on which we have tested the centralized algorithm explained in the previous chapter. The aim of the tests is to verify the performance of the algorithm. As we mentioned in Chapter 2, we have considered a model containing a system of peer-to-peer servers. The source code is available from the supervisors.

The reader will recall that, at the end of Chapter 3, we have seen an example of a causality graph and alarms where Katzela-Schwartz and Yemini's methods failed, while our method succeeded. It would have been interesting to test our algorithm on examples already successfully tested for Katzela's and Yemini's algorithm for a "sanity check". For very small graphs, it appears clearly that the computational complexity of the three approaches is similar. To get a non-trivial comparison, we would need a bigger causality graph. But then, because of the difference in the nature of the inputs for each method (for example, Yemini's method needs a choice of codebook), it would be difficult to set a common ground of comparison.

We wish to emphasize that the tests performed here were aimed at verifying that the algorithms find a correct answer in a reasonable amount of time, and a reasonable fraction of the time. The implementation of the correlation engine, in particular, has been optimized while simplicity of coding and functionality were respected. As we will see in the next chapter, future considerations of this work could involve a high performance coding of the algorithm, involving compactness of the code and memory used as well as speed at run-time.

5.2 Description of the testbench

The testbench we developed is made of three components: a set of node processes representing the servers of the network, an alarm mechanism and finally the alarm correlation engine based on the centralized version of the algorithm of chapter 4.

5.2.1 Server system

The main characteristic of the server system we wish to simulate here is the breaking down of some of the servers individually, and the propagation of the fault through dependency on other servers in a chain reaction fashion. We also want our system to be able to simulate peer-to-peer server failure, so the chain propagation could possibly go along a cycle of servers. For this purpose, we have constructed a model where each server process is equipped with two files:

- a *status file*, which contains the information on the status of the server (OK or FAILED), and
- a *dependency file*, which contains the identity of dependent servers and their

dependency strengths from the actual server.

We will come back to those files later

Each server is made of two processes. A first process represents the simulation of an internal break at the server. For this purpose, a lottery draw is made in a loop by using a random number generator. The probability of failure of the server is given to the process in input under the form of the numerator and denominator of the fraction representing the probability of failure of the given server process (both numbers are integers). The random number generator draws a number from 0 to the value of the denominator, and if the number is between 0 and the numerator, a message "FAILED" is written in the status file of the server. This outcome represents of course the failure of the node.

The second process polls the status file of the server until it reads "FAILED". When it does read "FAILED", it then writes an alarm message to the alarm file: it writes the identity of the server that fails (the actual server) and the time at which it fails. For this timestamp, it simply evaluates the time from the operating system. After sending an alarm, the second process enters a phase aimed at propagating the fault to some other servers. It reads the dependency file it owns: this file consists of the name of the dependent servers, the numerator and the denominator of their dependency probability of failure. The process reads the data in turn for each dependent server, draws a number in the same fashion as the first process accordingly to the probability data given in the file, and writes a "FAILED" message to the status file of the dependent servers for those servers for which the draw was successful.

The reason we want to use two processes is that we want independence between the fault messaging coming from the server itself and the fault messaging of this same server coming from other servers by dependency. Indeed, it is most important in this simulation to not restrict one source of failure over another. The UNIX programming environment provides record locking functions (read and write locks for the files), which were used for this purpose.

The communication between the different servers is made through messaging in files to avoid a simulation that would be uselessly complicated. Remembering that UNIX's Network File System (NFS) is based on Remote Procedure Call, we can claim that our simulation somehow reflects an RPC interaction between servers.

The servers processes were hosted by a network of SPARC Ultra1 workstations using the Solaris 2.5.1 operating system.

5.2.2 Alarm Generating System

Having established how to simulate faults, we must have a system to generate alarms. As we have mentioned already, there is a process in each server that generates an alarm upon reading a "FAILED" message to the status file of the server. Here even more than for the reading and writing of the status of each server, read and write locks are important since all the failed servers will write to the same alarm file.

5.2.3 Alarm Correlation Engine

We now come to the heart of the simulation. We have implemented the centralized version of the algorithm.

For many reasons, we have decided to implement our algorithm using the Java programming language. The first reason is that it is an Object-Oriented language, which is well suited for the different parts of our algorithms. Another reason is that Java is platform independent, and this is important since the alarm correlation system should be able to work on different platforms at the same time. A third reason is that Java is best suited for network programming, especially for the use of mobile code. The reader will remember that our distributed version of the algorithm is simplified by the use of mobile agents investigating the validity of a causal chain.

We have chosen to make a very simple implementation of the ideas of the algorithm. The main objective here was simply to test the feasibility of the implementation of the algorithm, but also basic performance of the algorithm (speed, scalability). Normally, here would be an off-line pre-processing to the diagnoser, which consists of the computation of the causality graph, the computation of the transitive closure of the causality graph and the paths repository, i.e. the array containing all the possible paths in the causality graph. However, since we wanted to make a simple implementation, those objects were instantiated right before the diagnosis of any test, giving therefore an idea of the performance of the system under a "cold start". For the propagation of faults, in order to test how well the algorithm behaves under heavy propagation, we have assumed that each dependency probability is 1. As much as possible, we have optimize the memory used in our implementation by using multiple linked lists and avoiding doubling a common object in the heap of the computer. We have also tried, in order to maintain simplicity of the code, to make heavy use of class inheritance. However, we believe that more thorough memory optimization through clever choice of classes is possible.

5.3 Results

We have proceeded with a simulated network of 50 nodes (this number is similar to the size of the network on which Katzela-Schwartz's method was tested [Katzela1996]). The dependency graph of this network contains many cycles (in order to simulate peer-to-peer interactions) and single paths (the reader will remember that our objective function models expansion of faults in a "bouquet": Having single paths brings us therefore to the limits of the objective function model).

We proceeded to 10 simulations of faults where many single fault simulations, and from each of the 10 scenarios obtained, we have randomly removed a fixed percentage of alarms. Those percentages go from 0% to 35% of the alarms removed, which means that we correlate 65% to 100% of the alarms received. Therefore, the correlation engine had seventy scenarios of alarms (10 originals simulations times 7 choices of alarms). For each case, the choice of the alarms to remove were made with a random number generator following a uniform distribution.

We then input our Java implementation of the centralized algorithm and compared the results to the actual sequence of events (the actual simulation). The convergence time of the algorithm was on average 35 seconds using the JDK1.1.3 compiler on a SPARC Ultra1 workstation. Using the native Unix time function, we discovered that the actual computation time for the correlation itself (which means once the dependency graph and the transitive closure are computed) takes on average 2 seconds only for converging. This compare easily to other alarm correlation methods ([Katzela1996], [Yemini1994]). It is important to mention that when using a JIT (just in time) Java compiler on a specific test

run, the total processing time reduced by half. The JIT compiler we used was the one provided by Sun's JavaWorkShop 2.0.

We have considered the results from two point of views:. The first one consists of only considering the optimal solution as the actual primary cause of alarms generated in the simulation. Another way of looking at the results is to accept a second best solution (that is, considering as possible candidates the first best optimal solution and the next one following in value of the objective function). The reason for this second rule is the following. Suppose that the alarm of a primary cause is lost, but a fault depending on it produces an alarm that is received and explains all the other alarms optimally. It would be difficult to expect the algorithm to find the actual primary cause. After all, the optimal solution explains everything best and we know nothing of any previous fault from the information we have. Therefore, it is reasonable to accept as an equal candidate the second best solution. The results appear in the tables of Figure 5.2 and the cartesian graph of Figure 5.4.

5.4 Discussion of the results

It is important to notice that if we allow as solutions the two best optimal solutions from the algorithm, the algorithm correctly finds the cause in at least 90% of the cases when the number of alarms missing is less than 35%, according to our simulations. As we mentioned earlier, we have allowed many single paths in the dependency graph. The discrepancy between the two curves can be explained in part by the presence of single paths in the dependency graph. The reader will remember that we have allowed a great number of those paths on purpose to measure the performance under the worst conditions. We

would expect the difference between the two curves to be less important if we would have a graph where the dependency at most nodes is more than one, therefore eliminating single paths.

Figure 5.1 Dependency Graph of the testbench

Test run #	100% alarms	95% alarms	90% alarms	85% alarms	80% alarms	75% alarms	70% alarms	65% alarms
1	Y	Y	Y	Y	Y	Y	Y	Y
2	Y	Y	N	N	N	N	N	N
3	Y	Y	N	Y	N	Y	N	N
4	Y	Y	Y	Y	Y	N	Y	Y
5	Y	Y	N	N	N	N	N	N
6	Y	N	Y	Y	N	N	N	N
7	Y	Y	Y	Y	Y	N	Y	Y
8	Y	Y	Y	Y	Y	Y	N	Y
9	Y	N	Y	Y	N	N	N	N
10	Y	Y	Y	Y	Y	Y	N	Y
# of suc-cesses	10	8	7	8	5	4	3	5

Solution as best optimal solution

Test run #	100% alarms	95% alarms	90% alarms	85% alarms	80% alarms	75% alarms	70% alarms	65% alarms
1	Y	Y	Y	Y	Y	Y	Y	Y
2	Y	Y	Y	Y	Y	N	Y	N
3	Y	Y	Y	Y	Y	Y	Y	Y
4	Y	Y	Y	Y	Y	Y	Y	Y
5	Y	Y	Y	Y	Y	Y	N	N
6	Y	Y	Y	Y	Y	Y	Y	Y
7	Y	Y	Y	Y	Y	Y	Y	Y
8	Y	Y	Y	Y	Y	Y	Y	Y
9	Y	Y	Y	Y	Y	Y	Y	Y
10	Y	Y	Y	Y	Y	Y	Y	Y
# of suc-cesses	10	10	10	10	10	9	9	8

Solution as best or second best optimal solution

Figure 5.2 Table of results

We have also proceeded to some test runs on a Pentium 100, and found that the total computation time was of 1 m 10s in average against 35 seconds on the Sparc Ultra1 host.

Figure 5.3 Graph of the ratio of correct analysis as a function of the percentage of missing alarms received

Overall, we can claim that the performance of the algorithm is quite satisfactory. The performance could even be improved if we would have taken dependency weights in

the causality graph different from 0, therefore helping the search for candidates by another level of optimization. The overall running time can be improved as well by even more careful code optimization. Also, since the centralized version of our algorithm has greater overall computational complexity than its distributed version, it would be interesting to see in the latter case the performance of an implementation of it.

We can compare our results to the results obtained from Yemini's and Katzela-Schwartz's methods. In the case of the Katzela-Schwartz method, the Maximum Mutual Dependency algorithm finds the correct answer in only 25% of the time. Notice however that the Maximum Mutual Dependency algorithm is an $O\left(n^3\right)$ approximation of the Exhaustive Search algorithm, which is exponential in the number of vertices n . In the case of Yemini's algorithm, if we suppose that we can find a codebook of radius 0.5, a correct diagnosis of faults is found in 96% of the cases even with a 20% alarm loss. As was already discussed in Chapter 4, the complexity of this algorithm (for the decoding part) is logarithmic in the number of vertices. This method can process thousands of alarms per second. As already mentioned, our experimentation (for our centralized algorithm) showed a processing rate of about 10 alarms per second, which compares to other methods such as IMPACT [Jakobson1993], which processes 12-15 alarms/sec.

We tested the algorithm with a simulation of a small scale network of 50 nodes. On a real network of 50 nodes, one would expect the complexity of faults to be much larger, the causality graph would as a result be therefore much larger, and hence the performance of the algorithm would decrease exponentially. But we believe that a simple fault model in a real network is still a useful framework, and so the performance would be, in that case,

comparable to our results. In the case of a more complex fault system, or in the case of a large network, a distributed version of our algorithm (discussed in Chapter 6) may help overcome scalability issues.

Chapter 6

Future Work and Conclusion

6.1 Introduction

In this chapter, we briefly discuss which aspects of our work it would be interesting to see being developed further. We have chosen only the areas where we believe further research would be interesting extensions of the present thesis. We will give a brief overview of a distributed version of the centralized algorithm described in Chapter 4. We will then conclude the present work by giving an overview of what we accomplished.

6.2 Some further research paths

Let us see now some aspects of the work that could benefit from being further analyzed.

6.2.1 Multiple primary faults detection

One of the first improvements our methods would benefit from would be to allow the correct detection and identification of multiple primary causes. Indeed, it can be quite difficult to decide when a primary cause explains reasonably all the alarms and should be accepted as the unique primary cause, and when multiple causes should be considered. This aspect needs to be researched further

6.2.2 Incorporating probabilities of failure of nodes

As was mentioned in earlier chapters, one of the strengths of the algorithms developed in Chapter 4 is the non-necessity of knowing the probability of failure of the objects being represented as nodes in the causality graph. This aspect was considered important in the development of the algorithm as we consider that obtaining the knowledge of such probabilities require extensive statistical analysis of the network under different situations. The use of those probabilities need to be done under careful analysis of the alarm correlation method. However, when a clear knowledge of those probabilities exist, it would be advantageous to take advantage of it in our method. This could lead, for example, to a "hybrid" method involving our temporal alarm correlation engine with Katzela-Schwartz's method. This would result in bringing together the strengths of both methods. The similarity of some routines and data structures in both algorithms would make such a combination an interesting perspective.

The knowledge of the probability of failures of the objects in our algorithm could also be handled by using more probabilistic methods like Bayesian Networks.

6.2.3 Learning about the thresholds

Another aspect of the algorithms that would require further research would be to develop a mechanism for learning the length of the causality paths to search. The reader remembers that a value must be assigned for the length bound, and that there is always the possibility that this length bound would be too small to effectively find solutions. Therefore, it would be interesting to see if a mechanism of assignment of the correct value for this bound upon the alarms received can be constructed.

6.2.4 Distributed algorithm

As already mentioned in Chapter 4, the distributed (more specifically *hierarchical*) version of the algorithm would allow major improvements for the computational complexity through the use of distributed computing and mobile code. Here we give a brief overview of this version, and we hope that future work can be based on this idea.

Clearly, the causality graph of a whole network is made of the causality graphs of its elements. Those causality subgraphs are actually connected subgraphs of the whole graph. We assume that the network management system is distributed among some of its elements, so that each element is responsible for the fault management in its region of influence. This partitioning can be done *geographically* (each region is made of adjacent elements) or *functionally* (a region constituting a set of hosts, possibly distant from each other, dedicated to one functional purpose), or both. In any case, we make the assumption that the causality subgraph of an element is *connected*: that is, for any pair of vertices in the subgraph, there is an underlying undirected path that links them. A less formal characteristic of connectivity is that the subgraph is “made of one piece”.

We also suppose that the set of causality subgraphs of the elements constitute a real partition of the graph: the intersection of any two such subgraphs is made only of the edges that connect causality graphs between adjacent regions, and the union of all those subgraphs is the graph itself. As an example, Figure 6.1 represents the partitioning of a global causality graph of a network.

Figure 6.1 Global Causality Graph

The combinatorial optimization problem is identical to the centralized version (see 4.2.1). However, as we mentioned, we assume that the causality graph of the network can be represented as the union of the causality graphs at each region. Also, in order to solve the problem, the different network management entities that are each responsible of a part of the global causality graph will have to contribute to the solution by analyzing their sub-graph. This is the main improvement in this version of the problem over the centralized version. The use of distributed computing on different parts of the causality graph to improve the speed of the fault identification process.

Figure 6.2 Local Causality Graph

A general view of the network is however necessary to each *network management entity* (*NM entity*). We do not want this view to be detailed, since then there would be no need for a non-centralized paradigm.

This general view can be obtained, for example for a region A, by shrinking the causality graph at each region to a vertex, except for the subgraph at A. Figure 6.2 represents the view of the network from region A of Figure 6.1. We need also to permit multiple edges in this form of a causality graph. This poses no problem in the analysis. The shrinking of a subgraph to a vertex is an operation known in Graph Theory under the name of *taking the minor of the graph*.

The alarms, in this distributed version, are slightly different from the centralized version. Here we assume that the alarms contain three fields: the *identity of the fault* (as for the centralized algorithm), the *timestamp*, and the *identity of the NM entity* that supervises the region from which the alarm originates. This last field identifies the minor vertex from which the alarm originates. The information we get from the NM entity's identity helps us to adjust the timestamp's value. Indeed, we assume that we have *a priori* a table of the maximum difference in time between the different clocks of the NM entities. This table allows us to estimate the consistency of a causal path involving timestamps from different NM entities: if the actual difference of timestamps between two causally related events is outside the difference from the table, we will reject the causal chain.

For each host, the algorithm works as for the centralized version. Upon receiving alarms, a NM entity tries to find a causal tree that explains best the sequence of alarms received from its local causality graph. It will consider the identification field for the alarms inside its region, but it will consider the NM identity field for all the alarms outside its region. In other words, the NM entity identifies all the alarms coming from a common foreign NM entity. This former information matches perfectly the reduction of the information that the local causality graph of the NM entity reflects: the NM entity does not know how to use the identity of the fault at a foreign region, but it will ask this other NM entity to help in discovering the actual global causal path. Among the possible candidates for all the alarms received, there will be alarms from shrunk vertices. However, it is important to note here that if no alarm comes from a particular region, the associated NM entity will not take part in the diagnosis, unless another entity asks for information related to this region.

The NM entity needs to get the exact paths inside the shrunk vertices in order to determine the exact chain of causality. We could use messaging between the different NM entities to exchange information. However, the amount of communication necessary to achieve the right interoperation of the NM entities could easily clutter the network. Besides, it might be necessary, depending on the “conclusions” of one NM entity about the primary cause, to start some process at this NM entity remotely from another NM entity. For those reasons, we use *mobile agents* instead of messaging. A *mobile agent* is simply a process that can move from one NM entity to another NM entity. In order to migrate, the agent is serialized. In fact, its code and state are transformed into a bit stream that is sent to another NM entity. The host where the agent migrates must be equipped with a mobile code daemon, which simply de-serializes the stream and therefore re-activates the agent. Many mobile agents systems were developed recently, using the Java programming language facilities ([Susilo1997], [Kimiry1997]).

Hence, each NM entity uses its reduced causality graph to find candidates for primary faults. For each possible cause, it then sends a mobile diagnoser (a mobile agent) with a candidate primary cause and the set of paths (*path map*) that emanate from this candidate. This mobile diagnoser first migrates to the candidate’s NM entity. It then asks the NM entity upon the possible subpaths through its local causality graph that can replace the shrunk vertex in its view (this shrunk vertex coming from the mobile diagnoser’s originating NM entity). Therefore, the search is made in a hierarchical manner, since the NM entity of the shrunk vertex will help the other NM entities in diagnosing the fault. This implies that the algorithm is more specifically a *hierarchical algorithm*. It is important to mention that each NM entity prunes its own local causality graph as well. Therefore,

assuming that the local NM entity pruned enough of it, it can give the migrating diagnosing agent a “pruned” set of subpaths to insert in the shrunk vertex representing the NM entity’s region.

If there are such paths, the diagnosing agent then spawns mobile diagnosers to continue the pruning to the other NM entities that represent the next vertices in the path map. It is important to note here that the edges in the Cause-Alarm graph attached to the shrunk vertex of the originating NM entity are pruned together by the mobile diagnoser. Therefore, there must be one agent per such edge to continue the pruning. Each of those children diagnosers then move to the next vertex in their respective path map and continue the diagnosis. If there are no paths in the shrunk vertex, then the diagnoser sends the information to an *agent broker*, which is an “intelligent” Path Tree. Its role is to keep a repository of all detailed causality paths in the causality graph of the whole network and to mark as inconsistent any path containing a path found inconsistent by an entity. When a mobile diagnoser initiates a search, it transmits its paths map to the broker. The broker informs agents on the marked paths, therefore saving the mobile diagnoser from uselessly migrating if all the paths have been marked as inconsistent. At the end, after having pruned all the path maps, the mobile diagnosers return to their home NM entity. They then compute the total length and dependency weight of the reversed tree resulting from each primary cause that explains all the alarms. Then all the NM entities “vote” for the most probable primary cause.

There are many advantages to this approach. As we have mentioned already, it utilizes distributed computing to reduce the computational complexity at one host. Also, the

method produces an NM system that is more robust, in the sense that if one entity fails, the other NM entities can still perform diagnosis. Note that in the case of the centralized version, a failure at the central Alarm Correlation system catastrophically stops all Alarm Correlation processes in the network.

6.3 Conclusion

We conclude this thesis by briefly reviewing what we have done so far, and by stating the contributions this work has brought.

In Chapter 1, we briefly reviewed the literature in the area of alarm correlation. We also gave some examples to illustrate some approaches.

In Chapter 2, we introduced and defined the form of the temporal alarm correlation problem we solved, namely: *assuming that we have received a sequence of alarms identifying the malfunctioning components in the network and the time of the failure, how can we identify the primary fault(s)?* We have also seen how solving this problem can help resolving situations where cyclic faults occur.

In Chapter 3, we looked more closely at two interesting approaches to the general alarm correlation problem: Katzela's and Schwartz's probabilistic method and Yemini's coding method. Those methods are crucial in our approach since some of our routines were derived from those methods.

In Chapter 4, we described our solution to the temporal alarm correlation problem expressed in Chapter 2 in the form a centralized algorithm

In Chapter 5, we demonstrated with an implementation that our centralized algorithm solves the temporal alarm correlation problem in a very satisfactory manner: the algorithm finds the correct primary cause at least 80% of the time even though as much as 35% of all alarms have been lost.

Finally, we explored some interesting perspectives that our discussions opened. In particular, it would be interesting to develop an implementation of the distributed version of our algorithm briefly discussed here. Such a development would be especially interesting in these years of expansion of internetworking.

This thesis provides the following contributions:

1. An investigation of alarm correlation methods in computer networks, with a special focus on time-based methods; and
2. A novel alarm correlation framework based on temporal information. The framework uses graph-theoretical methods. A centralized algorithm based on this framework is constructed. Testing shows that it correctly diagnoses faults 80% of the time even when 35% of the alarms are missing. Furthermore, the implementation of the algorithm correlates the alarms from a 50 node network in less than 2 seconds on a SparcUltra1 workstation.

References

- [Abu-Hakima1994] S. Abu-Hakima, "DR: the Diagnostic Remodeler Algorithm for Automated Model Acquisition through Fault Knowledge Re-Use". PhD thesis, Carleton University, Ottawa, Canada, 1994.
- [Aho1974] A.V Aho, J. E. Hopcroft, J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.
- [Apostopoulos1997] T.K. Apostolopoulos, V.C. Daskalou, "Network Management Services Using a Temporal Information Model", in *Integrated Network Management V*, Elsevier Science Publishers, 1997, pp. 638-649.
- [Bibas1996] S. Bibas et al., "Alarm driven supervision for telecommunication network: I-Off-line scenarios generation", *Ann. Telecommun.*, vol. 51, no 9-10, pp 493-500, 1996.
- [Bondy1976] J. A. Bondy and U. S. R. Murty, "Graph Theory with Applications". North-Holland, 1976.
- [Bouloutas1993] A. Bouloutas et al., "Fault Identification Using a Finite State Machine Model with Unreliable Partially Observed Data Sequences", *IEEE Transactions On Communications*, vol. 41, no 7, 1993.
- [Bouloutas1994] A. Bouloutas et al., "Alarm Correlation and Fault Identification in Communication Networks", *IEEE Transactions On Communications*, vol. 42, no 2/3/4, pp 523-533, 1994.
- [Charniak1991] E. Charniak, "Bayesian Networks without Tears", *AAAI AI Magazine*, pp 50-63, Winter 1991.
- [Chartrand1993] G. Chartrand, O. R. Oellermann, "Applied and Algorithmic Graph Theory", McGraw-Hill, 1993.
- [Christofides1975] N. Christofides, "Graph Theory: An Algorithmic Approach", Academic Press, 1975.
- [Dawes1995] N. Dawes, J. Altoff and B. Pagurek "Network Diagnosis by Reasoning in Uncertain Nested Evidence Spaces", *IEEE Transactions*

- on Communications*, vol. 43, no. 2/3/4, pp. 466-476, 1995.
- [Deng1993] R. H. Deng, A. A. Lazar and W. Wang, "A Probabilistic Approach to Fault Diagnosis in Linear Lightwave Networks", in *Integrated Network Management III*, H.-G. Hegering and Y. Yemin editors, Elsevier Science Publishers, 1993, pp. 697-708.
- [Dousson1996] C. Dousson, "Alarm driven supervision for telecommunication network: II-On-line chronicle recognition", *Ann. Telecommun.*, vol. 51, no 9-10, pp 501-508, 1996.
- [Esfandiari1996] B. Esfandiari et al., "Agent-oriented techniques for network supervision", *Ann. Telecommun.*, vol. 51, no 9-10, pp 521-529, 1996.
- [Gotlieb1993] C.C. Gotlieb, "Data Types and Structures", Prentice-Hall, Englewood Cliffs N.J., 1978.
- [Harary1972] F. Harary, "Graph Theory". Addison-Wesley, 1972.
- [Hart1995a] G. W. Hart and I. Rouvellou, "Automatic Alarm Correlation for Fault Identification", *IEEE Infocom '95*, pp 5a.3.1-5a.3.9, 1995.
- [Hart1995b] G. W. Hart and I. Rouvellou, "Inference of a Probabilistic Finite State Machine from its Output", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 25, no. 3, pp 424-437, 1995.
- [Jakobson1993] G. Jakobson, M. Weissman, "Alarm Correlation", *IEEE Network*, vol 7, no 6, pp 52-59, November 1993.
- [Jakobson1994] G. Jakobson, M. Weissman, "Real-time telecommunication network management: extending event correlation with temporal constraints", in *Integrated Network Management IV*, Elsevier Science Publishers, 1994, pp. 290-301.
- [Katzela1996] I. Katzela, "Fault Diagnosis in Telecommunication Networks", Ph.D. Thesis. Columbia University, 1996.
- [Katzela1995] I. Katzela, M. Schwartz, "Schemes for Fault Identification in Communication Networks", *IEEE/ACM Transactions on Networking*, vol. 3, no. 6, 1995.
- [Kimiry1997] J. Kimiry and D. Zimmerman, "A Hands-On look at Java Mobile Agents", *IEEE Internet Computing*, vol. 1, no. 4, July-August 1997.

- [Neapolitan1990] R.E. Neapolitan, "Probabilistic Reasoning in Expert Systems: Theory and Algorithms", John Wiley and Sons, Inc, NY, 1990.
- [Nejdl1994] W. Nejdl and J. Gamper, "Model-Based Diagnosis with Qualitative Temporal Uncertainty", World Wide Web, 1994.
- [Patil1981] R. Patil, P. Szolovits, and W. Schwartz, "Causal Understanding of Patient Illness in Medical Diagnosis", in *Proceedings, 7th IJCAI*, (Morgan Kaufmann Publishers, Inc.), pp. 893-899, 1981.
- [Pearl1987] J. Pearl, "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference", Morgan Kauffman Publishers, 1987.
- [Preparata1967] Franco P. Preparata, G. Metze and R.T. Chien, "On the Connection Assignment Problem of Diagnosable Systems", *IEEE Transactions on Electronic Computers*, vol. EC-16, no 6, pp 848-854, 1967.
- [Reggia1983] J. Reggia, D.S. Nau, and P.Y. Wang, "Diagnostic Expert Systems Based on a Set Covering Model", *International Journal of Man-Machine Studies*, vol. 19, pp. 437-460, 1983.
- [Schank1982] Schank, "Dynamic Memory: A theory of Learning in Computers and People", Cambridge University Press, 1982.
- [Shortliffe1976] E. Shortliffe, "MYCIN: Computer-Based Consultation in Medical Therapeutics", American Elsevier, New York, 1976.
- [Stallings1993] W. Stallings, "SNMP, SNMPv2. and CMIP: The Practical Guide to Network Management Standards", Addison-Wesley, 1993.
- [Susilo1997] G. Susilo, "Infrastructure for Advanced Network Management based on Mobile Code", Technical Report SCE-97-10, Systems and Computer Engineering, Carleton University, June 1997.
- [Tanenbaum1995] A. S. Tanenbaum, "Distributed Operating Systems", Prentice-Hall, Englewood Cliffs, 1995.
- [Washio1996] T. Washio, K. Okusa, A. Endou, "Intelligent Virtual Measurement in a Process Plant Based on Model Based Diagnosis", *Seventh International Workshop on Principles of Diagnosis*", pp. 249-257, 1996.
- [Winter1987] P. Winter, "Steiner Problem in Networks: A Survey", *Networks*", vol. 17, pp. 129-167, 1987.

- [Yemini1994] Y. Yemini et al., "A Coding Approach to Event Correlation", in *Integrated Network Management IV*, Elsevier Science Publishers, 1994, pp. 266-277.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved