Fault-Tolerant Network Monitoring

A Dissertation Presented as Partial Requirement for The Degree of Doctor of Philosophy to The Faculty of

Tokyo Institute of Techonology
Graduate School of Information Sciences
Department of Computer Science

By

Elias Procópio Duarte Jr.

1997

Abstract

As networks expand, applications become critical for organizations and individuals. Consequently, there is a pressing need for dependable network management systems. However, considering current approaches, a network fault may cause a partial collapse of the management entity. As fault management is a key functional area of network management systems, this situation constitutes a paradox: the system is meant to solve faults, but those same faults impair the system. To tackle this problem, we have worked on algorithms and tools for fault-tolerant network monitoring. All theoretical work developed has been also implemented using SNMP (Simple Network Management Protocol), however, the proposed solutions are in no way limited to this framework.

In the first part of this work, we present an strategy to improve the dependability of current centralized network management systems based on SNMP. We propose the use of SNMP proxy agents to bridge communications between manager and agent whenever the corresponding network route is not working. This allows network management, which is an application layer entity, to have a simple and powerful routing engine. Algorithms are introduced to locate proxies for each agent. The impact of this solution on the steady-state availability of the system is shown. An SNMP MIB implementation of the proxy is proposed that allows any agent to become a proxy with virtually no cost.

Next, we propose the application of distributed system-level diagnosis to develop network monitoring applications that are resilient to network faults, i.e., no matter which portion of the network is faulty, monitoring continues on the fault-free portion.

The second part of this work introduces a new Hierarchical Adaptive Distributed System-level Diagnosis algorithm (Hi-ADSD) applied for SNMP-based local area network fault management. The algorithm has latency of $log^2 N$ testing rounds, for a network of N nodes. Nodes are mapped into progressively larger logical clusters, so that each node executes tests in a hierarchical fashion. The algorithm assumes no link faults and a fully-connected network. There are no bounds on the number of faults. Both the worst-case diagnosis latency and correctness of the algorithm were formally proved. Experimental results are given through simulation of the algorithm for large networks. Practical results were given from an implementation of the algorithm on a 37-node Ethernet LAN using SNMP.

The third part of this work is a new algorithm for diagnosis of general topology networks, applied for fault management of wide area networks. Nodes test links periodically, and disseminate link time-out information to all its fault-free neighbors in parallel. Upon receiving link time-out information, a node computes which portion of the network has become unreachable. This approach is closer to reality than previous algorithms, for it is impossible to distinguish a faulty node from a node to which all routes are faulty. The diagnosis latency of the algorithm is optimal, as nodes report events in parallel, and latency is proportional to the diameter of the network. The dissemination step includes mechanisms to reduce the number of redundant messages introduced by the parallel strategy. We present a MIB which can be used to implement the algorithm using SNMP. The evaluation of the algorithm's impact on network performance shows that the amount of bandwidth required is less than 0.1% for popular link capacities.

Acknowledgements

It's so nice that I have this little space to be less formal, and tell a bit of the story behind this thesis. I've been so lucky to meet and work with so many wonderful persons during my PhD course, I feel that in these acknowledgements the writing style will be definitely compromised by the use of the most extravagant superlatives!

First of all, it was a dream come true to be a student of the world famous, top computer science professor and scientist Prof. Takashi Nanya. He is *great*. Furthermore, my very busy supervisor is also a wonderful person, with a very captivating personality: it was nothing short of a deep pleasure and honor to work with him.

Next, Glenn Mansfield, of course! Back in the mountains beyond Sendai, in October 1993, I had the honor to be accepted by this most distinguished researcher to work on a project of his, about proxies to save disrupted SNMP management routes - the rest is history, as they say. For his continued supervision and friendship, I owe my deepest gratitude.

Nothing would have been possible if Prof. Masatoshi Miyazaki were not the wonderful person that he is. He gave me the right support when I needed it.

During the summers of both 1996 and 1997 I had the pleasure and the honor to work with Prof. Douglas Blough, in Tokyo and Irvine, California, respectively. It was a great experience to work with him, and I could clearly understand why he is known all over the world in the Fault Tolerance research field.

I'd also like to express my gratitude to Prof. Shoichi Noguchi, of Sendai, and now Aizu, for his constant support.

I want to continue working with you all! Yoroshiku onegaishimasu!

Now, I'd like to acknowledge that my scholarship was a mix of the Brazilian CNPq and the Japanese Monbusho. Furthermore, during all the PhD I had also a license from UFPR, and I thank my colleagues for taking all the burden while I've been out.

I've met some wonderful persons in Japan. I must start with Eizo Hashimoto, his wife Hiromi, his mother, my Obaachan, and the little Kyoko. Their friendship has been more than important for me. Their support fundamental. For them, my eternal gratitude.

A great friendship leads to another, and I must thank Hideko Nishimoto for her friendship and support, in Sendai, Takadanobaba, Meguro, and next Curitiba! Our life ways are really interconnected! What a pleasure for me!

I'd like also to thank the great friends of Sendai: the guys at Miyazaki-ken, Raquel, Cris, Kiyama-San and Chiba-San. I cannot mention 1% of all Sendai friends here, you know you are all in my heart!

Next come the people at the Fault-Tolerance Lab at Titech, starting with Prof. Tomohiro Yoneda. Thanks to Ueno-San for his expert technical support, which I also got occasionally from Rafael Morizawa. A special thank you to Andreas Savva and Mohit Sahni for many nice conversations. And also to all other friends I met here: Tomita-San, Bin Zhou, Patrick Trane, Mart Saarepera, Ravi Kishore, Sung-Bum Park. Also my friends at Titech from other labs: Sima, Pavlin, Chris, Louis. Oh I'll never be able to mention everybody...

Rodinei Gomes is my friend since the initial flight from Brazil to Japan, up to this conclusion. Rodney, let's keep it on!

I thank N. Ohta for important life lessons.

Dr. Jaider, D. Cotinha and all their group, thanks a lot! Your support was very important, my sincere gratitude to you.

M. Yazici: if only life were perfect. I still have hope!

Mirian Halfeld has been so close, although she was miles away in Paris. Mi, thanks for your support, for our friendship.

My hobby has been a major source of strength throughout my life, and I owe it to my father.

To my mother, I owe the world. Her brain is behind it all. She brought me up from 7 alone, she is my hero, I love you!

To my sister Angela, for her support, for her friendship, my thanks, my love, let's keep on moving forward together!

And to complete, to my sister Fatima, there won't be enough words to acknowledge the joy and strength I received from our daily communication during all this period through the Internet. As if all else were not enough, the simulation of Hi-ADSD using SMPL you'll find in chapter 4 is mostly her work. As if all else were not enough we had great vacations during this time... I guess all I can say is thank you.

Contents

Chapter 1

Introduction

The information revolution is changing the world and deeply affecting human lives, relationships, and enterprises. Although innumerable factors have allowed this revolution to take place, it can be seen as a direct result of the development of computer and communication technologies, and the synergy that arises from their interaction.

Computer networks are continually expanding to such an extent that it is possible to say that the enterprise and the network are becoming indistinguishable [1]. In other words, if the network is not working properly, the enterprise is in trouble. Network applications are thus becoming mission-critical: risks and costs associated with network faults and performance problems are significant. At the same time, networks are becoming larger and more complex, being made up of a variety of heterogeneous devices, based on different technologies, produced by different organizations. Management of such a system is not a trivial task. There is thus a pressing need for automated systems to allow effective network management.

As network management is needed to ensure that the network is operating efficiently at all times, it should be clear that the network management system must be itself operating efficiently at all times, including periods in which there are faults in the network. This work is about some of the algorithms and tools that can be used for building dependable, fault-tolerant network management systems.

This chapter is organized as follows. After defining network management systems and their functionality, we discuss current standards and why they are needed. Network monitoring is then introduced as a key part of any network management system, and popular approaches are reviewed. This is followed by the definition of faults, as well as of system-level diagnosis. The contributions of this work are then introduced. This is followed by an overview of the remaining chapters.

1.1 Network Management

A broad definition of network management is given by Terplan [2]: "network management means deploying and coordinating resources in order to plan, operate, administer, analyze, evaluate, design, and expand communication networks to meet service-level objectives at all times, at a reasonable cost, and with optimum capacity".

ISO (International Organization for Standardization) has proposed a classification of network management functionality into five areas: fault, performance, configuration, security, and accounting management. This functionality was proposed as part of the OSI (Open Systems Interconnection) systems management specification, but has been widely accepted to describe of the requirements for any network management system [3]. The core of each function is described below:

- *Fault management:* allows the detection, isolation, and correction of abnormal operation of the network.
- *Performance management:* allows performance monitoring and evaluation of the network.
- *Configuration management:* allows the human manager to reconfigure the network from a management station. Its goal is to guarantee continuous operation and quality of service.

Security management: includes procedures to protect the system from unauthorized access.

Accounting management: enables charges and cost to be defined for network usage.

1.2 Standard Frameworks

As networks are made up of heterogeneous elements, i.e. computers and communication technologies produced by different organizations and vendors, it is important that network management systems be based on international open standards, shared by all technologies. Both the Internet TCP/IP (Transmission Control Protocol/Internet Protocol) and the OSI communities have produced open standards for network management. The OSI set of standards includes the Common Management Information Protocol (CMIP) standard. The TCP/IP set of standards includes the Simple Network Management Protocol (SNMP) standard, also known as the Internet-standard Network Management Framework.

SNMP has had a huge success, it is the *de facto* standard for network management today. A large number of organizations, both academic and business-related, have adopted SNMP. A vast number of network devices, routers, bridges, hubs, and operating systems offer support for SNMP. Although the OSI framework has also received attention, its complexity and the slow pace in which documents were produced have influenced in its low acceptance and scarce deployment compared to SNMP. There is little hope that CMIP will ever become more popular than SNMP.

In August 1988, the specification for SNMP was issued and soon afterwards it became widely adopted. A significant revision of SNMP, known as secure SNMP, was issued in 1992. Then, in 1993, a second generation protocol was issued: SNMP version 2 (SNMPv2), which improves both SNMP and secure SNMP with new functionality. Work on new versions continues under the auspices of the IETF (Internet Engineering Task Force).

1.3 Network Monitoring

Network management can be divided into two parts: *monitoring* and *control*. Monitoring is the process of observing the behavior of the network and its components, both to detect faults and monitor performance. Control is the process of changing the network's behavior in real time by adjusting parameters when the network is up and running, in order to improve performance or repair faults.

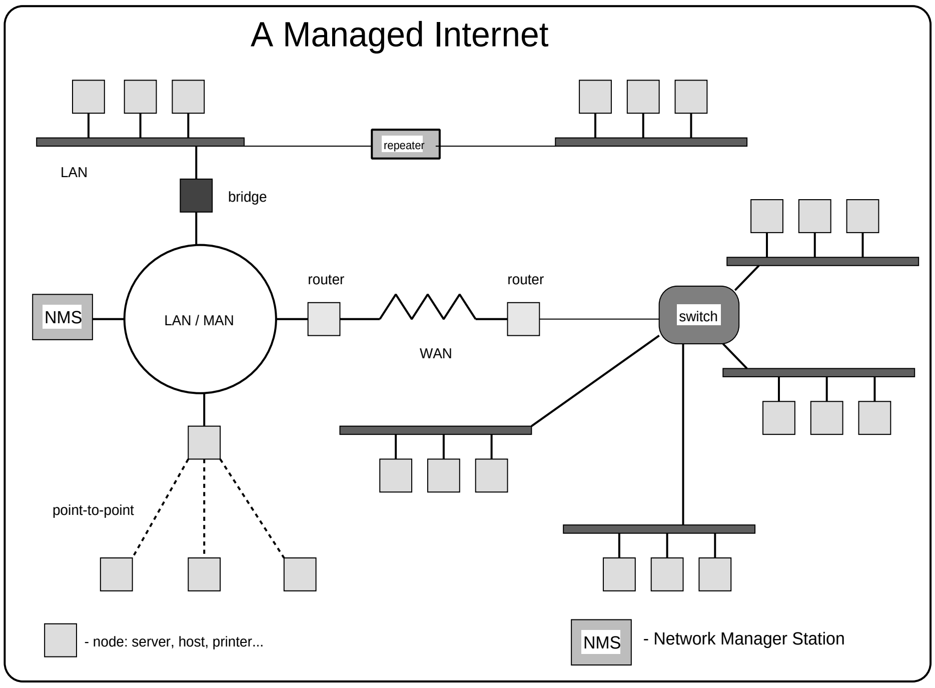

Figure 1.1: An example of a managed network, nodes and devices are agents.

SNMP is based on the *manager/agent* paradigm. Each agent maintains a database of management objects, called MIB (Management Information Base). The manager, also called Network Management Station (NMS), monitors and controls the network by reading and writing (or getting and setting) the management objects in the agent using a standard management protocol. As Partridge and McCloghrie mention in [3]: "the object database provides an abstract representation of the managed system to the management application, and it is the agent's responsibility to translate operations on this abstract database into real operations upon the managed system". Figure 1.1 shows a heterogeneous network with one NMS and a collection of agents on the various nodes and devices.

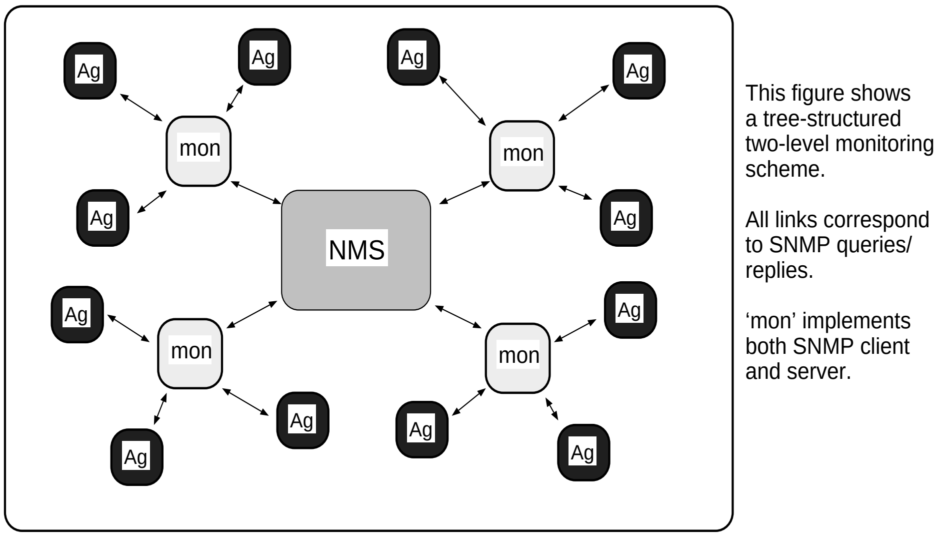
Ag
mon
NMS
This figure shows a tree-structured two-level monitoring scheme.
All links correspond to SNMP queries/ replies.
'mon' implements both SNMP client and server.

Figure 1.2: A common approach to network management monitoring.

The traditional approach to monitoring is to have a number of managers, usually only one, organized in a tree, each of them responsible for querying a set of agents, and reporting to monitors in higher levels of the tree, as shown in figure 1.2. Agents (Ag) are the leaves of the tree, and intermediate nodes are monitors that implement both an agent process (i.e. SNMP server) and a manager process (i.e. SNMP client). This approach presents two drawbacks: (1) if monitors become faulty or unreachable, diagnosis stops on an entire portion of the network; (2) all monitors are required to test a large number of network nodes.

Network monitoring is accomplished not only through polling, but agents are also able of asynchronously notify managers of potential problems by using alarms, called *trap* messages. In general, network management systems employ a combination of polling and alarm management. If the manager monitors a large number of agents and each agent maintains a large number of objects, then it may become impractical to regularly poll all agents for all of their data. However, trap processing also has problems of its own: as agents must generate events when thresholds are achieved, they must continuously monitor the value of the management objects. This process may have a bad impact on the performance of the agent. Furthermore, when there is a fault in the network, the manager is usually flooded with alarms sent from agents that have different perspectives of the problem, and diagnosis can be difficult in those circumstances. If the network problem is congestion, the traps will make it worse. The SNMP community favors polling over traps [4].

Later in this work, we present new distributed algorithms for fault-tolerant network monitoring, and strategies to improve the dependability of current centralized systems.

1.4 Fault Diagnosis

Initially, we define a *fault*. There are in fact three concepts that must be clearly understood [5]: failure, error, and fault. A failure of a system occurs when the behavior of the system first deviates from the system's specification. In other words, a system fails when it cannot provide the desired service. An error is a property of the system state that may lead to a failure. The cause of an error is a fault. The concept of a fault is associated with a notion of defect. A faulty system is one with defects. The definition of a fault is: a defect that has the potential of generating errors.

A classical example is that of a memory cell that always returns 0, independent of what is stored in it. This memory cell is faulty, i.e. it contains a fault. However, an error only occurs when the value stored in the cell was 1, and the contents of the cell are read for some computation. That is an error, an observable event. This error may somehow make the system behave in a way different from its specifications, if it does, there is a failure. A failure is not a property of the system state, and cannot be observed easily.

A large portion of the research on fault management has concentrated on alarm correlation. The ability to filter and correlate alarms is an important feature of network management systems. Whenever there is a problem in the network, the manager is likely to receive a large number of alarms from different parts of the networks, each of them with a different point of view of the problem, and mechanisms should be employed to determine the real cause of the problem. An example of recent work in this area is [6].

Another approach for fault management is to use expert systems. They use knowledge based on past experience to diagnose network faults. The main problem with this approach is that it is difficult to extract and maintain this knowledge. Furthermore, new types of faults are not properly diagnosed. An example of recent work in this area is [7].

1.5 System Level Diagnosis

The field of system-level diagnosis has flourished for years. Its objective is determine the state of every unit of a system. In the PMC model, [8] a system is composed of units that are capable of testing each other. It is assumed that the status of each unit is either faulty or fault-free and the status does not change during diagnosis. A *test* involves controlled application of some stimuli and observation of the corresponding responses; it is assumed that a fault-free unit always reports the status of the units it tests correctly, while the faulty units can return incorrect results of the tests conducted by them. The PMC model assumes the existence of a *central observer* that, based on the syndrome, can diagnose the state of all the units. For a given testing assignment, the diagnosability of a system may be limited by the number of faulty units.

Early system-level diagnosis algorithms assumed that all the tests had to be decided in advance. The tests were then executed, and from the test results the central observer determined which units were faulty. An alternative approach, which requires fewer tests, is to assume that each unit is capable of testing any other, and to issue the tests adaptively, i.e. the choice of the next tests depends on the results of previous tests, and not on a fixed pattern. This approach was called *adaptive* [9]. Early adaptive system-level diagnosis results assumed the existence of the previously mentioned central observer. Furthermore, a bound on the number of faulty nodes was imposed for the system to achieve correct diagnosis.

Kuhl and Reddy [10], introduced distributed system-level diagnosis, in which fault-free nodes reliably receive test results through their neighbors, and each node independently performs consistent diagnosis.

The Adaptive Distributed System-Level Diagnosis algorithm, *Adaptive DSD*, was introduced by Bianchini and Buskens [11, 12]. Adaptive DSD is at the same time distributed and adaptive, each fault-free node uses the minimal number of messages per testing round, i.e., one message, to achieve consistent diagnosis in at most N testing rounds. One testing round is the period of time in which each unit has executed at least one test successfully.

We have proposed the usage of adaptive distributed system-level diagnosis algorithms for network monitoring and fault management. In [13] system-level diagnosis results which are not based on the PMC model are also applied to fault management. Their results are not based on the PMC model, and are basically centralized and not executed on-line.

1.6 Contributions of the Thesis

This thesis has three main contributions. Initially, we worked on improving the dependability of current centralized network management systems by using proxies to bridge management communications and thus give the manager the ability to reach agents even if there is a fault along the route determined by the network layer. Next, we worked on new distributed system-level diagnosis algorithms applied for fault-tolerant network monitoring. We developed the Hierarchical Adaptive Distributed System-level Diagnosis algorithm which can be applied to a Local Area Network (LAN), and a new algorithm for Wide Area Network (WAN) diagnosis. Each of these contributions was implemented using SNMP, but they are not limited in any way to this framework. The three contributions are briefly introduced below.

1.6.1 Fault-Tolerant SNMP Query Routing

Although fault management is one of the most important functional areas of network management systems, currently these systems themselves often become partially faulty as a consequence of the faults they should instead be solving. For instance, if the communication path from the NMS to an agent is down, there will be a collapse of the system, as management is an application layer entity and depends on the network routing layer for all routing decisions. To solve this problem we proposed the usage of SNMP proxies to bridge communications between the NMS and the agents, whenever the corresponding network layer routes are faulty. An example is shown in figure 1.3. Paths using the proxies are set up in the application layer, and are known as application routes, in contrast to the usual network routes.

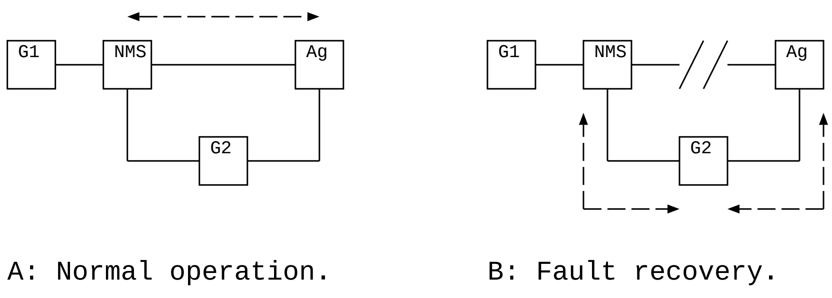

Figure 1.3: A proxy bridges communications between the NMS and an agent.

An algorithm was developed for selecting which nodes may act as proxies for each agent in the network. An evaluation of the impact of the approach on network management dependability was conducted. The evaluation was based on the percentage of network management queries and replies that are correctly delivered using the proxy mechanism, under different network faults. Using routing proxies is a simple approach to prevent large parts of the network from becoming unreachable. The proxy was implemented as a conventional SNMP MIB (Management Information Base), an efficient and flexible approach, that allows the deployment at virtually no cost.

Fault-tolerant SNMP routing is presented in detail in chapter 3.

1.6.2 LAN Fault Diagnosis

We proposed a new Hierarchical Adaptive Distributed System-level Diagnosis (*Hi-ADSD*) algorithm and its implementation integrated to a network management system based on SNMP. Hi-ADSD is a fully distributed algorithm that has diagnosis latency of at most $\log^2 N$ testing rounds for a network of N nodes. The algorithm assumes the PMC fault model. Nodes are grouped in progressively larger logical clusters, so that each node executes tests in a hierarchical fashion, as shown in figure 1.4. The algorithm assumes no link faults, a fully-connected network and imposes no bounds on the number of faults.

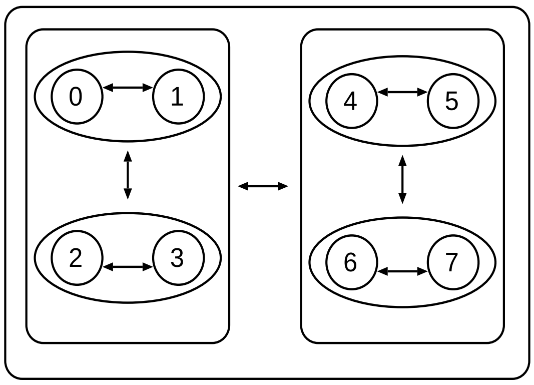

Figure 1.4: Nodes test clusters.

The algorithm was formally proved correct. Each node tests one cluster per testing interval, and tests continue until a fault-free node is found. A testing round is defined as the period after which every fault-free node in the system has tested another fault-free node, or all remaining nodes are tested as faulty. The longest testing path is of length $\log N$, as shown in figure 1.5. As, for each test to occur it may take up to $\log N$ testing rounds, it takes at most $\log^2 N$ testing rounds for all nodes to diagnose the state of the system.

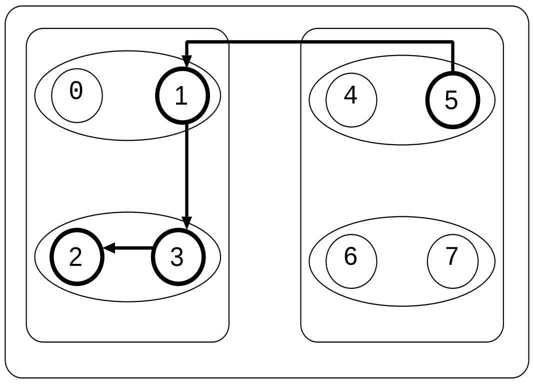

Figure 1.5: Longest diagnostic path.

A simulation of Hi-ADSD was conducted using the discrete-event simulation language SMPL. For a network of 512 nodes it took an average of 16 tests for all nodes to diagnose an event. The algorithm was implemented using SNMP on an Ethernet LAN with 37 Sun workstations. Some of the events and their diagnosis are presented in the figure above. Each event took an average of 427.1 seconds to be diagnosed, with a testing interval of 40 seconds.

Hi-ADSD and LAN diagnosis are presented in detail in chapter 4.

1.6.3 WAN Fault Diagnosis

We have proposed a new algorithm for diagnosis of non-broadcast networks, applied to WAN fault management.

In previous algorithms for system-level diagnosis on networks of general topology, fault-free nodes determine if nodes/links are faulty/fault-free. We claim this approach is unlikely to work, for in a non-broadcast network it is impossible to distinguish between the following two fault situations: (1) a node is faulty, and (2) all links to that node are faulty.

We proposed an algorithm in which nodes test links periodically, and and disseminate *link time-out information* to all its fault-free neighbors in parallel. Upon receiving link time-out information a node computes which *nodes* are *unreachable.* The diagnosis latency of the algorithm is optimal, proportional to the network diameter.

An SNMP MIB was devised for the algorithm. *Two-way* tests are employed, such that for each link only one of the nodes acts as tester, but each node detects if the other is faulty. The evaluation of the algorithm's impact on network performance shows that the amount of bandwidth required is less than 0.1% for popular link capacities.

The new algorithm for WAN diagnosis is presented in detail in chapter 5.

1.7 Overview of the Thesis

Chapter 2 presents further concepts of network management systems based on SNMP, allowing the reader to understand the implementation of the tools and algorithms described in later chapters. Chapter 3 presents the usage of proxies to allow a manager to reach agents even if there is a fault along the route determined by the network layer. Chapter 4 presents the Hierarchical Adaptive Distributed System-level Diagnosis algorithm and its application to LAN fault diagnosis. In chapter 5 a new algorithm for WAN fault diagnosis is presented. Chapter 6 concludes the thesis.

Chapter 2

Practical Network Management Based on SNMP

In this chapter we present an overview of the Internet-standard Network Management Framework, also known as the SNMP (Simple Network Management Protocol) framework [14, 15, 16]. SNMP has had a huge success, it is the *de facto* standard for network management today. A large number of organizations, both academic and business-related, have adopted SNMP. A vast number of network devices, routers, bridges, hubs, and operating systems offer support for SNMP. We used SNMP to deploy the algorithms and tools introduced in later chapters of this work.

As networks expand and become mission critical, the need for an integrated system to allow network monitoring and control becomes critical. Networks are made up of heterogeneous elements, being based on computers and communication technologies produced by different organizations and vendors. Thus, it is important that network management systems be based on international open standards, shared by all technologies. SNMP is an open framework developed by the TCP/IP community to allow the integrated management of highly heterogeneous internets.

Besides the TCP/IP community, the OSI community has also worked on a set of open standards for network management. It includes the Common Management Information Protocol (CMIP) standard. Although the OSI framework has also received attention, its complexity and the slow pace in which documents were produced have influenced in its low acceptance and scarce deployment compared to SNMP. There is little hope that CMIP will ever become more popular than SNMP.

Although SNMP is a simple protocol, its huge success is not due to a lack of more complex alternatives. SNMP's simplicity is, to the opposite, one of the reasons the protocol has been so widely deployed. As the impact of adding network management to managed nodes must be minimal, avoiding complicated approaches is a basic requirement of any network management model. The simplicity of SNMP has guaranteed its efficiency and scalability. Nevertheless, there is a number of areas in which SNMP has shown deficiency. However, the set of standards has been evolving: new versions and new solutions are being developed uninterruptedly.

In this chapter we give an overview of the SNMP management architecture, including the Structure of Management Information (SMI), the Management Information Base (MIB), the management protocol, and conclude examining the framework's evolution trends.

2.1 SNMP Architecture

SNMP is based on the *manager-agent paradigm*, in which the network is monitored and controlled from a Network Management Station (NMS). Managed nodes and devices are called agents. Each agent keeps management information stored at a local Management Information Base (MIB). The NMS also keeps a MIB. The MIB may include information about, for example, the number of packets received by the agent and the status of its interfaces. The NMS and the agents communicate using a network management protocol. Figure 2.1 illustrates the paradigm.

The NMS runs a collection of management applications, which allows fault, performance, and configuration management, besides security and ac-

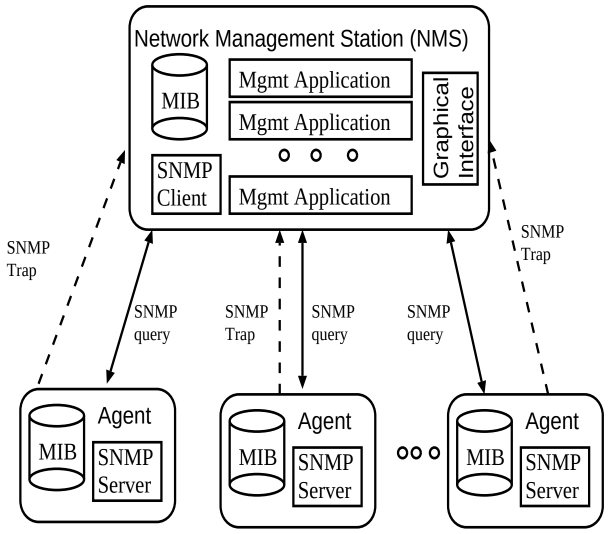

Figure 2.1: The Manager/Agent Paradigm.

counting management [17]. The NMS must have a proper interface, as it is the station from which the human manager accesses the network management system. Graphical interfaces are recommended [18, 2], and a number of current systems have Web-based interfaces.

Agents run on computers and network devices, including routers, bridges, hubs, that are equipped with SNMP so that they can be managed by the NMS. Each agent replies to SNMP queries, and may issue asynchronous alarms, called traps, to the NMS reporting exceptions, for example, when management objects indicate that an error has occurred. Agents run an SNMP server, and the NMS runs SNMP client applications.

An SNMP MIB is a collection of *management objects*. Each object is, essentially, a data variable that represents one aspect of the managed agent. The MIB's are standard, and different types of agents have MIB's containing different objects. The NMS can cause an action to take place at an agent or can change an agent's configuration by modifying the value of specific variables.

The NMS and the agents communicate using a network management protocol. SNMP contains three general types of operations: *get*, with which the NMS querries an agent for the value of a given management object; *set*, with which the NMS writes the value of a given management object at a given agent; and *trap*, with which an agent notifies the NMS about a significant event.

SNMP is an application layer protocol of the TCP/IP protocol suite. It runs on top of UDP (User Datagram Protocol), a connectionless transport protocol, that runs on top of IP. Two port numbers have been assigned to SNMP: agents listen for incoming get or set requests on port 161, managers listen to incoming traps on port 162.

The reason a connectionless protocol was chosen is that network management must be very resilient to faults over the network. If there is a fault, a connection may have problems to be established. Furthermore, a connection-oriented approach masks a number of network problems for the application, because it does retransmission and flow control automatically. Network management cannot have these problems hidden from it.

If a computer or network device does not support SNMP, UDP or IP, it can still be managed. An SNMP *proxy* is an agent that acts as a delegate for one or more devices. So when the NMS wants information about those devices, it queries the proxy, that is responsible to get the required information from the actual device and reply the querry.

There are two strategies usually employed by the NMS to monitor the network: polling and alarm management. The NMS polls agents regularly at specific time intervals querying for management objects. The interval may vary depending on the object or the network state. On the other hand, alarm management is based on traps sent by agents to the NMS when threshold conditions are reached. In general, network management systems employ a combination of polling and alarm management. Request for Comments 1224 presents strategies to blend polling and traps [19].

If the NMS monitors a large number of agents and each agent maintains a large number of objects, then it may become impractical to regularly poll all agents for all of their data [3]. However, trap processing also has problems of its own. As agents must generate events when thresholds are achieved, they must continuously monitor the value of the management objects. This process may have a bad impact on the performance of the agent. Furthermore, when there is a fault in the network, the NMS is usually flooded with alarms sent from agents that have different perspectives of the problem, and diagnosis can be difficult in those circumstances. If the network problem is congestion, traps will make it even worse. The SNMP community favors polling over traps [4].

2.2 Structure of Management Information

Management information is a key component of any network management system. In the SNMP framework, information is structured as a collection of management objects (MO's) stored at the MIB. Each managed device in the system keeps a MIB that has information about the managed resources of that device. The NMS monitors a resource by reading its corresponding MO's current value, and controls the resource by writing a new value to the MO. Although the word "object" is used, it simply refers to a MIB variable. In SNMP, the only data types that are supported are simple scalars and one-dimensional data arrays.

The Structure of Management Information (SMI), defined in Request for Comments 1155 [14], defines the rules for describing management information. The SMI defines the data types that can be used in the MIB, and how resources within the MIB are represented and identified. It was built to emphasize simplicity and extensibility. It allows the description of management information independently of implementation details.

The SMI is defined using a restricted subset of ASN.1 (Abstract Syntax Notation Language One). ASN.1 is an ISO formal language that defines the abstract syntax of application data. Abstract syntax refers to the generic structure of data, as seen by an application, independent of any encoding techniques used by lower level protocols. An abstract syntax allows data types to be defined and values of those types to be specified independently of any specific representation of the data. ASN.1 is used to define not only the management objects, but also the Protocol Data Units (PDU's) exchanged by the management protocol.

There must be a mapping between the abstract syntax and an encoding, which is used to store or transfer the object. The encoding rules used by SNMP are called BER (Basic Encoding Rules) which, like ASN.1, is also an ISO standard. BER describes a method for encoding values of each ASN.1 type as a string of octets. The encoding is based on the use of a type-length-value structure to encode any ASN.1 definition. The type includes the ASN.1 type plus its class, the length indicates the length of the actual value representation, and the value is a string of octects. The structure is recursive, so that complex types are also represented using this basic rule.

2.2.1 Object Identifiers

One of the most important tasks of the SMI is to define unique identifiers for management objects. There must be a consensus throughout systems of what each object is used to represent, and how objects are accessed.

Each object identifier is a sequence of labels, which translates to a sequence of integers. All objects are organized in a tree, such that the sequence of integers corresponds to the path from the root of the tree up to the point where the object is defined. It is important to remember that these objects and their identifiers are standard, defined by the authority who is responsible for the management framework.

The virtual root of the tree of objects is assigned to the ASN.1 standard. In the first level, there are three possible subtrees: `iso`, `ccitt`, and `joint-iso-ccitt`. Each SNMP MIB is defined under the `internet` subtree, which is under the `iso` subtree, and is referred to as:

```
iso.org.dod.internet
```

Which translates to:

```
1.3.6.1
```

Under the `internet` node, the SMI defines, among others, the following nodes:

- `mgmt`: used for standard Internet management objects. The `mgmt` subtree contains the standard MIB, under the subtree `mib-2`.
- `experimental`: used to identify experimental objects; Experimental MIBs may at some time be added to the standard MIB.
- `private`: used by vendors and organizations to attend particular needs. This subtree contains one child, the `enterprises` subtree, under which a subtree is allocated to each organization that registers for an enterprise object identifier. These MIBs can also eventually become part of the standard MIB.

2.2.2 MIB-2

The current standard MIB is MIB-2 defined in Request for Comments 1213 [16]. The `mib-2` subtree is under the `mgmt` subtree, i.e.:

```
iso.org.dod.internet.mgmt.mib-2
```

or

```
1.3.6.1.2.1
```

Within the MIB, objects are defined in groups, each of them having a specific purpose. Following is a summary of the groups in MIB-2:

- `system`: overall information about the managed node itself;

  For example, the variable `sysUpTime` identifies how long ago the agent (re-)started.

- `interfaces`: information about each of the interfaces from the agent to the network;

  For example, the variable `ifNumber` gives the number of interfaces of the agent to the network.

- `at` (address translation): contains address resolution information, used for mapping IP addresses into media-specific addresses.

  For example, the variable `atPhysAddress` gives the media address for a given interface, while the `atNetAddress` variable gives the IP address.

- `ip, icmp, tcp, udp, egp,` and `snmp`: Each group has information about that protocol's implementation and behavior on this system.

  For example, the variable `ipInReceives` gives the number of packets received by the IP protocol entity from lower layers.

- `transmission`: provides information about the transmission schemes and access protocols at each system interface, used for media-specific MIBs, like for X.25, Token Ring, FDDI, among others.

2.2.3 Defining a MIB

A collection of ASN.1 descriptions relating to a common theme, e.g. a MIB, is called a *module*. A module has the following basic form:

```
<<module>> DEFINITIONS ::=
  BEGIN
   EXPORTS
   IMPORTS
   <<declarations>>
  END
```

The `<<module>>` term is the name of the module. `EXPORTS` indicates which definitions used by the module can be used by other modules. `IMPORTS` indicates which type and value definitions from other modules are used by this module. The `<<declarations>>` consists of three types of assignments: type assignments, value assignments, and macro definitions.

Type assignments define new data structures. Value assignments associate instances to a type. The macro notation allows the user to extend the syntax of ASN.1 to define new types and their values. SMI defines and uses macros extensively.

Before presenting an example module, it is important to understand that ASN.1 adopts the following lexical conventions: layout is not significant, multiple spaces and blank lines are considered a single space; comments are delimited by a pair of hyphens at the beginning of the comment, and this can be ended either by another pair of hyphens, or a newline character.

ASN.1 identifiers consist of upper and lower case letters, digits and hyphens. A built-in data type identifier consists of all upper case letters. Other data types's identifiers just start with an uppercase letter. A macro identifier consists also of uppercase letters, and a value identifier of lowercase letters.

The following module is an example used to group the management objects that are required by the diagnosis algorithms presented later in this thesis. The `<<declarations>>` part is presented later.

```
DIAGNOSIS-MIB DEFINITIONS ::= BEGIN

--- The MIB for diagnosis algorithms
--- September 21, 1995

EXPORTS -- everything --;

IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, OBJECT-GROUP, enterprises, IpAddress
        FROM   RFC1155-SMI ---- SMP-SMI
    DisplayString
        FROM   RFC1158-MIB;

   <<declarations>>
```

```
END
```

DIAGNOSIS-MIB is an ASN.1 module that imports from the RFC1155-SMI module the macros MODULE-IDENTITY, OBJECT-TYPE, OBJECT-GROUP. It also imports the types enterprises and IpAddress from RFC1155-SMI, and type DisplayString from RFC1158-MIB.

A MIB module defines a collection of related management objects. Each module begins with an indication of the module's identity and its revision history. The SMI defines a special ASN.1 macro, MODULE-IDENTITY, used to define MIB modules. Below there is an example of how to define a new MIB called diagnosis MIB.

```
DiagnosisMIB MODULE-IDENTITY
   LAST-UPDATED "9303040000Z"
   ORGANIZATION "Tokyo Institute of Technology"
   CONTACT-INFO
           "     Elias Procopio Duarte Jr.

                 Titech - Dept. Computer Science
                 Nanya Lab.
                 Ookayama 2-12-1 Tokyo 152 Japan

                 Tel: +81-3-5734-3041
                 Fax: +81-3-5734-2817

                 E-mail: elias@cs.titech.ac.jp
                         elias@inf.ufpr.br"
   DESCRIPTION
          "A MIB that implements objects for diagnosis
           algorithms."

   ::= { enterprises 200 }
```

When a MIB module is defined, it includes information about when it was LAST-UPDATED, which ORGANIZATION produced the module, CONTACT-INFO and a `DESCRIPTION`. The `LAST-UPDATED` field is of a type defined by the ASN.1, `UTCTime` [4]. There are also optional fields, like `REVISION` and `DESCRIPTION`.

Defining Objects

Each type of object in a MIB has an identifier of the ASN.1 type `OBJECT IDENTIFIER`. The SMI defines the `OBJECT IDENTIFIER`'s which are used by the management framework.

The `OBJECT-TYPE` macro is used to define each managed object. It allows the following key entries:

- `SYNTAX`: abstract syntax;
- `MAX-ACCESS`: the object may be "read-only", "read-write", "write-only", "not-accessible";
- `STATUS`: may be "mandatory" or "optional" depending on the implementation support required for this object. May also be "obsolete" meaning that objects are not implemented any more, or "deprecated" meaning that in future MIB versions they are likely to be removed;
- `DESCRIPTION`: a textual description of the object;
- `REFERENCE`: an optional cross-reference to an object defined in other MIB module;
- `INDEX`: used when the object corresponds to a row of a table;
- `DEFVAL`: defines an acceptable default value that may be used when an object instance is created, it is optional.

Data Types

Each management object may be of the native ASN.1 types `INTEGER`, `OCTECT STRING`, `OBJECT IDENTIFIER` and `NULL`. Furthermore it allows the definition of new types derived from the previously defined types. Some of the types defined by the SMI are:

- `Network Address` for data-link layer addresses;
- `IpAddress` for IP addresses;
- `Counter`: non-decreasing, wraps around when reaches $2^{32} - 1$;
- `Gauge` may increase or decrease, remains at $2^{32} - 1$ until reset. A gauge can also be used to store the difference in the value of some entity from the start to the end of a time interval;
- `TimeTicks` counts the time in hundredths of a second since some initial time;
- `Opaque` supports the capability of passing arbitrary data, used as octet string for transmission. The data themselves may be used in any format defined by ASN.1 or some other syntax.

Furthermore, there are two native ASN.1 constructed types:

- `SEQUENCE`: which defines a collection of elements grouped into a structure, usually used by the SMI to define rows of tables.
- `SEQUENCE OF TYPE`: defines a table, i.e. a one-dimensional array of elements of a given ASN.1 type.

Each table must have an `INDEX` component that determines which object value(s) will be used to distinguish and index the table. It is possible that not one, but a set of objects be used as index.

An example of object definitions is given below, corresponding to the last part of the `<<declarations>>` for the DIAGNOSIS-MIB.

```
testedUP OBJECT-TYPE
    SYNTAX  SEQUENCE OF TestedUPEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
```

```
      "Tested_UP is the main data structure used
      by the Adaptive Distributed System-level
      algorithm (Adaptive-DSD or aDSD)."
    ::= { DiagnosisMIB 1 }

testedUPEntry OBJECT-TYPE
    SYNTAX  TestedUPEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
      "Each entry of testedUP identifies
      which node the testing station recognized
      as up in the last testing round."
    INDEX   { testingID }
    ::= { testedUP 1 }

TestedUPEntry ::=
    SEQUENCE {
       testingID
         INTEGER,
       testingAD
         DisplayString,
       testedID
         DisplayString
}

testingID OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
       "The integer unique identifier of all the
        nodes participating in Adaptive-DSD.
        Also indexes the table."
    ::= { testedUPEntry 1 }

testingAD OBJECT-TYPE
    SYNTAX  DisplayString
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
      "IP-address of all the nodes participating in
       Adaptive-DSD, uniquely mapped to testingID's."
    ::= { testedUPEntry 2 }

testedID OBJECT-TYPE
    SYNTAX  DisplayString
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
      "Index of the workstation tested on the last
       testing round as up or _x_ if unknown"
    ::= { testedUPEntry 3 }
```

`DiagnosisMIB` contains the definition of the table `testedUP`, of which a row, `testedUPEntry` is made up of the fields `testingID`, `testingAD` and `testedID` which are all of the type `DisplayString`.

The example given in this chapter up to this point is the whole MIB used for the implementation of the Hi-ADSD algorithm discussed in chapter 4.

2.2.4 Object Instantiation

Every object in the MIB has a unique object identifier that is defined by the position of the object in the MIB, which forms a tree.

To access objects that appear in tables, the identifier alone is not enough, for the `INDEX` object of the table is also necessary. This `INDEX` object distinguishes rows in a table, and it is not necessarily a scalar integer. The index must be a set of objects that uniquely distinguishes all the rows.

Thus, given an object whose identifier is y, in a table with INDEX objects i_1, i_2,..., i_N, the instance identifier for a particular row is the concatenation $y.i_1.i_2...i_N$.

A simple example is of the `ifTable` object in the interfaces group, which has one `INDEX` object, the `ifIndex`. Now suppose we want to know the interface type of the fourth interface of the system. The identifier of `ifType` is:

```
1.3.6.1.2.1.2.2.1.3
```

The value of `ifIndex` for the fourth interface is 4, thus to access the object, the following identifier is used:

```
1.3.6.1.2.1.2.2.1.3.4
```

An example of an `INDEX` object made up of a collection of objects is the `tcpConnTable`, which has four objects as index. Thus, an instance identifier for any of the five columnar objects in the table consists of the object identifier of that object (call it `x`) concatenated with the four indexing objects as follows:

```
x.I.(tcpConnLocalAddress).(tcp.ConnLocalPort).
(tcpConnRemAddress).(tcp.ConnRemPort)
```

Where `I` is an identifier for a column, i.e., a field withing each row.

A convention used to refer to scalar objects is to concatenate a ".0" to the identifier of the object. So, for example, the `sysDescr` object has identifier:

```
1.3.6.1.2.1.1.1
```

and is be referred to as:

```
1.3.6.1.2.1.1.1.0
```

2.3 SNMP: The Protocol

SNMP is the protocol employed by managers and agents to communicate management information. It is used by managers to query and control agents and by agents to issue traps and reply to queries. Version 2 of SNMP also allows managers to communicate among themselves. SNMP became a full Internet standard in 1990, and is described in Request for Comments 1157 [15]. The protocol has since evolved, but "basic" SNMP is in widespread use, having been adopted by dozens of organizations worldwide. For the implementations described in later chapters, basic SNMP version 1 was used, and is described here.

SNMP provides three basic operations, always performed on scalar objects:

- *Get*: to read a given management object;
- *Set*: to write a new value for given management object;
- *Trap*: for the agent to send the value of a given management object to a manager;

SNMP doesn't allow one to "create" new variables on the fly: management objects available are those previously defined in the MIB.

2.3.1 A Powerful Operation: *getnext*

As discussed before, all objects in SNMP follow a lexicographical order. This allows the manager to easily traverse the whole MIB. One SNMP command that allows this traversal is *getnext*. Given the identifier one a given object, the agent is queried for the next one in the lexicographical order. This operation can be used also to traverse a table. Using getnext on the result of a getnext the whole MIB can be traversed, until a well defined error message is got for the end of MIB. A popular application for traversing the whole MIB is *snmpwalk*.

2.3.2 SNMP Messages

In this section we examine the SNMP message format, and the Protocol Data Units (PDU's) that can be carried in a message.

SNMP Message:

version	community	SNMP PDU

GetRequest PDU, GetNext PDU, and SetRequest PDU:

PDU Type	request-id	0	0	variable-bindings

GetResponse PDU:

PDU Type	request-id	error-status	error-index	variable-bindings

Trap PDU:

PDU Type	enterprise	agent-addr	generic-trap	specific-trap	time-stamp	variable-bindings

Variable Bindings:

name1	value1	name2	value2	....	name N	value N

Figure 2.2: SNMP packet formats.

As shown in figure 2.2, each message includes a version, community, and a PDU. Communities can be seen as passwords, they are explained in the next subsection. There are five different types of PDU's: GetRequest, GetNextRequest, SetRequest, GetResponse, and the Trap PDU. Except for the trap, all PDU's contain a request-id field. It is very important to keep track of replies to a given query, especially when the query is retransmitted for some reason. The response also contain fields for error status and index. The variable-bindings field is used to allow a single packet to carry information about a number of SNMP variables. Figure 2.3 show how PDU's are exchanged. Except for the trap, each request should be followed by an appropriate response.

To transmit a message, first the PDU is constructed using ASN.1; then the PDU is passed to the authentication service, together with a source and

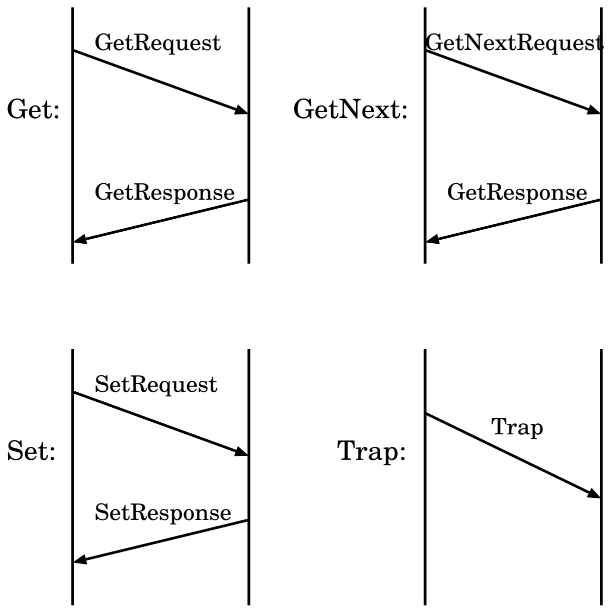

Figure 2.3: SNMP packet sequences.

destination transport addresses, and a community name. The result of this phase, which may be also encrypted, is then sent to the protocol entity, which constructs the message, including version field, community name. The result is then encoded, using BER, and passed to the transport service.

When the SNMP entity receives a message, it performs the following actions. First it does a basic syntax check, and discards the message if it fails to parse. Then it verifies the version number. The rest of the message is passed to the authentication service, which may do some decryption among other security checks, and produces an ASN.1 PDU. The protocol entity then does a syntax check of the PDU, and prepares for processing.

There is a number of PDU's specific for traps, for example informing that a machine has been started, that links are down or up, that neighbors have become unreachable, among others.

2.3.3 Communities: Passwords for Accessing Objects

It is important to provide security mechanisms that ensure only authorized access to management information. In SNMP version 1, the only security mechanism available is the *community*.

An SNMP community is a password, that defines for each agent a set of access policies for the MIB. The access mode may be "read-only" or "read-write". For example, a given manager when querying an agent, may send the community "public", which means that the manager can get values of certain objects, cannot get values of some other objects, and cannot set any object. As another example, the community "private" may allow read and write access to a limited number of objects.

An SNMP MIB view defines a subset of the objects within a MIB. Different MIB views may be defined for each community. An SNMP community profile is a combination of a MIB view and an access mode. A community profile is associated with each community defined by an agent. The combination of an SNMP community and an SNMP community profile is called an SNMP access policy.

The community concept is also important for proxies, which must keep an SNMP access policy for each proxied device.

As the only security mechanism of SNMP was the communities, vendors were reluctant to implement network configuration capabilities in SNMP, restricting their products to network monitoring. Nevertheless, later versions of SNMP have incorporated more sophisticated security procedures.

2.4 SNMP Evolution

With the huge success that followed the initial specification, SNMP problems became more evident. Nevertheless, new versions of the protocol have appeared, and important additions have been made. Some of the most significant have been the RMON MIB and SNMP version 2, described below.

RMON is the Remote Monitoring MIB. It defines a set of objects and a set of functions to monitor a remote network as a whole. Typically an agent has information only about its own devices. RMON is different. Basically it is an agent device connected to a broadcast network, the agent collects statistics concerning traffic on that network. RMON makes it possible for an NMS to collect information of a remote network as a whole.

An important deficiency of basic SNMP is its lack of effective security mechanisms. Enhancements were proposed and were called "Secure SNMP", but very soon these were included in the specifications of SNMP version 2 (SNMPv2) [20]. SNMPv2 allows distributed management, providing manager-to-manager communication capabilities. Besides that, it provides dynamic tables, in which rows can be created or deleted; a new macro do define object types; two new PDU's, one for manager-to-manager communications, the other for retrieving large blocks of data, GetBulkRequest.

The work on the framework has not stopped, and new results are being constantly discussed and produced.

Chapter 3

Fault-Tolerant SNMP Query Routing

The usual network management system is comprised of a network management station (NMS) which communicates with agents using a network management protocol [21, 1]. The NMS queries the agents for management information describing the state of links, devices, protocol entities and nodes [22]. The NMS takes decisions related to fault diagnosis, performance management, and network configuration, among others, based on the collected information [23, 24].

There is a pressing need for network management systems capable of handling errors [25]. Although network management systems are in principle responsible for fault diagnosis and management, current systems often become partially non-operational as a consequence of the faults they should instead be helping to solve [26, 22].

If a communication link along the path from the NMS to an agent or to a managed network is down, there will be a collapse of network management, as the NMS won't be able to determine the state of part of the managed network [27]. To make the network management system resilient to network failures there has to be alternative means of accessing agents. The network is in general a mesh-type structure, there are multiple potential paths between two communication nodes. However, since network management systems are application layer entities, these have little or no control over the paths that will be chosen by the network layer for routing the management queries. So, alternative paths for management communication have to use application layer entities which relay the management query and the replies along adequate communication routes [26].

In [27] Norton identifies the problem that occurs when the NMS looses the ability to monitor part of the network. This chapter presents a different approach: using the concept of a *proxy*, the NMS has a simple application routing engine to implement a fault tolerant routing system. An SNMP proxy is an entity used by the NMS to access another device, i.e., the proxy receives the query, transmits it to the agent, gets the reply and sends it back to the NMS. We present an efficient deployment strategy of proxy agents. An algorithm for determining the position of a set of proxies on a given network is presented. The solution is modeled as a Markov process, and the impact of application routes on the steady-state availability of the network management system is presented.

The rest of the chapter is organized as follows. Section 3.1 motivates the desired features of application routes, and provides precise definitions. Section 3.2 describes the proposed algorithm that uses network configuration information to determine the set of proxies and their position throughout the managed network. Section 3.3 provides the dependability evaluation of the solution, through the impact of application routes on the steady-state availability of the network management system as well as other measures of interest. Section 3.4 discusses practical aspects of deploying proxies on the managed network, including a discussion of Internet routing protocols. Section 3.5 examines how to obtain network topology information. Section 3.6 presents a network state determination strategy that uses application routes as alternative tools to diagnose faults. Section 3.7 shows an effective MIB to implement the proxies, that allows their deployment at virtually no cost. Section 3.8 presents a case study, locating proxies on a large network.

Section 3.9 contains concluding remarks.

3.1 Application Routes

Consider the simple network topology in figure 3.1, where the NMS is connected to an agent (Ag) and also to two gateways, G1 and G2. Considering communications involving the NMS and the Ag, suppose that routing is such that the direct link is used to communicate the queries and replies, as shown in part A of the figure. If the link between the NMS and agent fails, network management queries will be delayed until the network layer recovers from the failure. The delay may be significant as a new route for the agent must be discovered. During this delay, the NMS won't be fully operational, and is not able to manage the whole network. This problem could be solved if a management entity could relay the queries from NMS to AG and the corresponding replies from AG to NMS, as shown in part B of the figure. The condition to obtain this solution is that the routes used by the proxy be available when a failure occurs in the network route between manager and agent. In the example, G2 can be used as proxy in such situation.

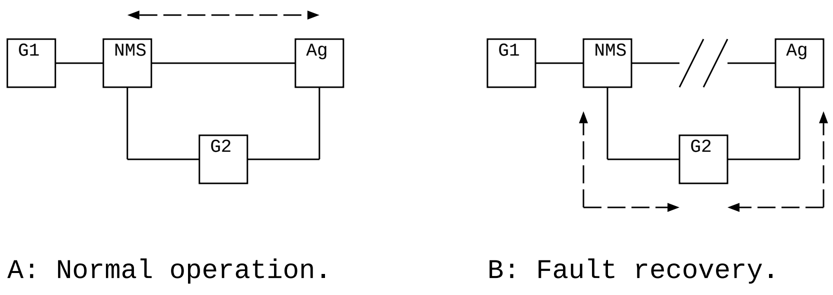

Figure 3.1: Management communication routes.

In this work the term *link* should be clearly defined. It models a bidirectional transmission channel between two nodes, and also includes the network interfaces at those nodes. It contains two queues of packets, each representing the packets in transit in one direction. A link can be subject to failure and repair [28], with up-times and down-times described by probability distributions. When a link fails, all packets that are in transit are lost.

Although management communication happens between the NMS and the agents, the paths used also employ ordinary communication nodes, that are not necessarily agents nor managers. A node or vertex may be an NMS, an agent or an ordinary communication node. Even if between a pair of vertices there are multiple different paths, the communication occurs along a precisely defined route. The route selection is carried out by the network layer protocol entites. This path is called the *network route* between two points, and is decided by the network layer. In figure 3.2 the network routes are depicted using dashed lines. If there is a failure along any component of a network route, communication between the end points is breached until the network adapts itself to the failure [21] and sets up a new network route.

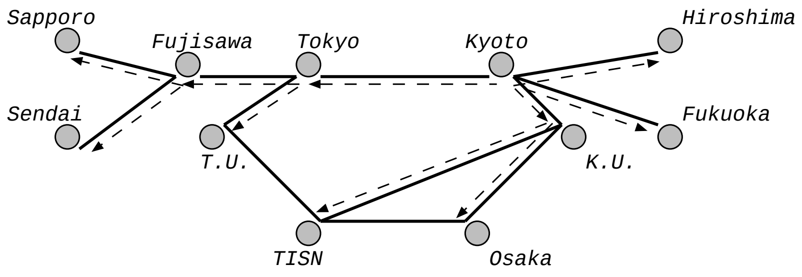

Figure 3.2: Node labels (α, β) represent the application routes' sizes from the NMS to agent1 and agent2 respectively.

An *application route* is a concatenation of one or more *network routes*, which are joined by an application. For example, the network route from the NMS to the proxy and the network route from the proxy to the agent result in an application route from the NMS to the agent when concatenated. It is important to understand that network routes are not transitive, so if there is a network route from node A to node B, and another network route from node B to node C, the concatenation of these two network routes may be different from the network route from node A to node C. Thus, the application route can be used as an alternative when there is a fault along the corresponding network route.

Application routes serve the purpose of bridging or relaying messages from one network route to another. In the case of a failure along a network route used for manager-agent communication, the agent will become unreachable and the network management system won't be able to monitor part of the network. What is worse, there is generally no simple way to find out where the failure occurred - in the agent, the connecting link or in the NMS node itself, which is a clearly undesirable situation.

For a simple network topology like that of figure 3.1 the position of the proxy is quite obvious, but for a more complex network, like that of figure 3.2 it is not a simple decision. Considering that the NMS is attached to Kyoto and there are agents attached to all other nodes. If any network route from the NMS to an agent is not available, the agent will become unreachable to the NMS. A set of proxies should be determined such that whenever an agent becomes unreachable an application route will be established to reach that node.

The configuration of the network as seen by the network and application layers must be taken into account in selecting the application routes to be employed, i.e. in positioning and configuring the proxy agents. The next section introduces an algorithm for selecting the application routes for a given network.

3.2 An Algorithm for Locating Proxies

In this section, we present an algorithm to place a set of proxies that allows application routes to be activated whenever network routes between the NMS and an agent is faulty. A graph depicting the topology of the network is obtained using network configuration information. In this graph [29], nodes represent the manager station, the agents and ordinary communication gateways. Edges represent communication links, of any physical structure.

3.2.1 Algorithm Specification

The algorithm begins considering the network route from the manager to each agent. In the first step, the algorithm marks all nodes that can reach both manager and agent if one link is removed from the graph, i.e., if one link is not operational, it finds which nodes can reach both agent and manager through a route that does not employ the removed link.

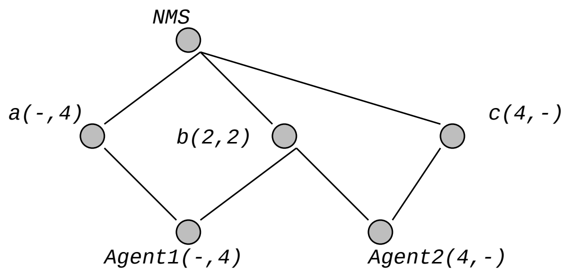

Figure 3.3: Node labels represent the application routes' sizes from the NMS to the two agents.

If a node has an alternative route from manager to agent, the entry for the node in a table called *Number of Paths* is incremented. Each entry in this table is a counter of the number of alternative routes that a node provides, and there is one entry for each node in the graph. For each link considered, each node may provide only one alternative route, namely, the network route for manager and agent. The criterion to select among nodes that provide the same number of alternative routes is the length of the routes, which are recorded in a table called *Hops*, from which the average number of hops in the alternative routes will be calculated.

The algorithm is as follows:

```
Algorithm
   REPEAT for each agent
      Get_Network_Route(NMS, agent);
      REPEAT for each link in the network route
        FOR each node in the network
           IF it can reach both NMS and Agent
```

```
            THEN record the sizes of the paths;
        UNTIL all the links have been verified;
        SELECT the nodes that provide:
              the maximum number of routes;
              the shortest average sized routes;
    UNTIL all the agents have been examined;
End Algorithm.
```

3.2.2 Algorithm Complexity

Let N be the number of nodes of the network, and L the number of links. The algorithm is polynomial, as the main loop refers to verifying the existence of network routes from the manager to each agent, taking out each link of the network. In the algorithm, the first loop is executed $N-1$ times, one for each agent. It contains a loop that is executed L times, for each link. And this runs the shortest path algorithm $2*(N-2)$ times, computing the path to the NMS to the node and from the node to the agent. Thus the total complexity is $O(N^3 * L * \log N)$.

3.2.3 A Small Example

The example of figure 3.3 is a simple one to illustrate the basic idea of the algorithm. There is one NMS, two agents and three ordinary gateways. For this specific example, network routes are shortest paths, management queries follow the network routes NMS-a-Agent1 and NMS-c-Agent2. The objective is to determine a set of proxies for both agents, that will be used when the network routes suffer one link failure. The node labels indicate the average sizes of application routes for Agent1 and Agent2, respectively; when the node cannot be a proxy for an agent, the size is omitted. For example, node a cannot be used as a proxy for agent1, because node a is part of the network route from the NMS to agent1.

Considering that the network route between the NMS and agent1 is (NMS-a-Agent1), there are three possible application routes, namely (NMS-b, b-Agent1), (NMS-c, c-Agent2-b-Agent1) and (NMS-c-Agent2, Agent2-b-Agent1). The three possible candidates both offer alternative routes when link NMS-a fails and when link a-Agent1 fails. Then the proxy with shortest application route is chosen, b. A similar process is done with Agent2, and b is selected as its proxy.

After the best candidates are selected for each agent, the minimum set that covers all agents may be selected, if the price of placing proxies in the network is significant. For our purposes, it is not necessary to calculate this minimum set. Later we present a MIB to implement the proxies that offers virtually no cost for an agent to become a proxy. For the example shown, the optimal set is node b, which serves as proxy for both agents.

3.2.4 Network Vulnerability

To allow an evaluation of the impact of using proxies on network management, we define a measure called *vulnerability*.

Def.: Link Vulnerability , v_i, for a given link l_i, is the number of nodes that become unreachable to the NMS if l_i is faulty.

Table 3.2 shows link vulnerabilities for all links of the network of figure 3.2.

Def.: Network Vulnerability , V, for a given network is the summation of link vulnerabilities, for all links in that network, i.e.: $V = \sum_{i=1}^{L} v_i$.

The last row of table 3.2 shows the vulnerability of the network of figure 3.2.

Def.: Risky nodes are those for which it is not possible to determine an alternative route to reach. These nodes are connected to the NMS through a path that, if faulty, will make the node unreachable. Table 3.1 shows the risky nodes for the network of figure 3.2.

Agents	*Candidate Proxies*
ku	tokyo, tu, tisn
sapporo	ku, fujisawa, tokyo, tu, tisn
fujisawa	ku, tokyo, tu, tisn
tohoku.u	ku, fujisawa, tokyo, tu, tisn
tokyo	ku, tu, tisn
tu	ku, tisn
hiroshima	*risky node - no candidates*
fukuoka	*risky node - no candidates*
osaka	tisn
tisn	tokyo, tu

Table 3.1: Proxies selected for the example network.

3.2.5 A Larger Example

To better illustrate the algorithm, and understand the impact on network vulnerability, a larger example is given. Consider the network of figure 3.2. Table 1 gives the results after the candidate proxies are selected based on the number of alternative paths and on the sizes of these paths. For example, for node *ku* there are three possible proxies *tokyo*, *tu*, *tisn*. In the second step the algorithm deals with the selection of one of these three candidates. For each candidate there is a counter of the number of agents for which it is a candidate, the one that may be a proxy for the largest number of agents is selected. In this case, *Tisn* is a candidate for 7 agents; *tokyo* is a candidate for 5 agents; *tu* is a candidate for 6 agents; *tisn* is then selected as proxy for ku.

It is interesting to notice that with only two proxies, *tisn* and *tu*, it is possible to have alternative routes for all agents. The improvement in the vulnerability, is shown in table 3.2. The impact on the steady-state availability of the system is given in next section along with other measures of interest.

Links	*V without proxies*	*V with proxies*
Kyoto-Tokyo	5	0
Tokyo-TU	1	0
TU-TISN	0	0
Kyoto-KU	3	0
KU-TISN	1	0
KU-Osaka	1	0
TISN-Osaka	5	0
Hiroshima-Kyoto	1	1
Fukuoka-Kyoto	1	1
Tokyo-Fujisawa	3	3
Fujisawa-Sapporo	1	1
Fujisawa-Sendai	1	1
TOTAL	23	7

Table 3.2: Improvement in the vulnerability.

3.3 Dependability Evaluation

This section introduces a general function that provides an evaluation of the impact of the proposed scheme on the steady-state availability of the system. The basic parameter sought is the dependability of the network management system, which in a repairable system is best described by its availability [30, 31].

At a given time, the system is in one of two states: *operational* and *failed.* It should be clear that *the system is operational when network management queries between the NMS and agents are properly delivered.* When there is a network route or an application route available for the NMS to reach the agents, the system is operational. Otherwise, when an agent is unreachable from an NMS the system is in the failed state.

The coverage, c, of a system gives the probability that the system will recover given the occurrence of a failure in the network. In this context it refers to the probability that the network management system will stay operational if one link fails throughout the network. It can be directly obtained

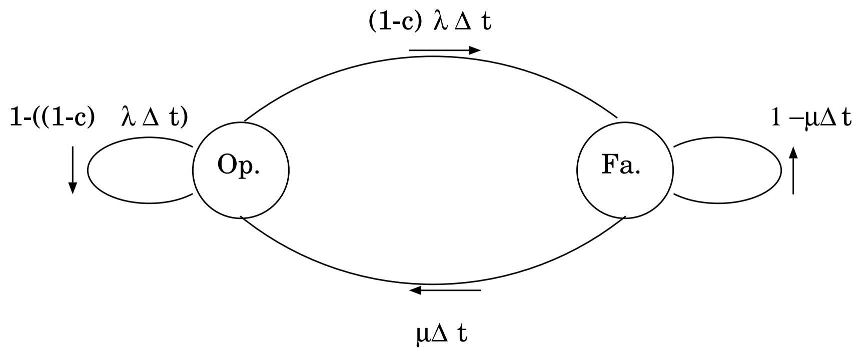

Figure 3.4: Markov process describing the behavior of the system.

from the previously introduced vulnerability. For the example of figure 3.2, if one link fails, the probability that the management queries are delivered is approximately 70%. The coverage must be calculated for each case [31, 32]. For the network management case, whenever an alternative route exists as an option for the communications that use a given link, the coverage of the system is improved, as the system remains operational.

To calculate the steady-state availability of the proposed solution the system is modeled as a simple Markov process with the two states previously defined, operational and failed. The probability that there is a failure in the system is the product of the failure rate (λ), an interval of time (δt), and the probability that the fault won't be instantly repaired with an application route. The same approach applies to the repair rate (μ) of the system.

After the fault-tolerant scheme is implemented, the probability that the

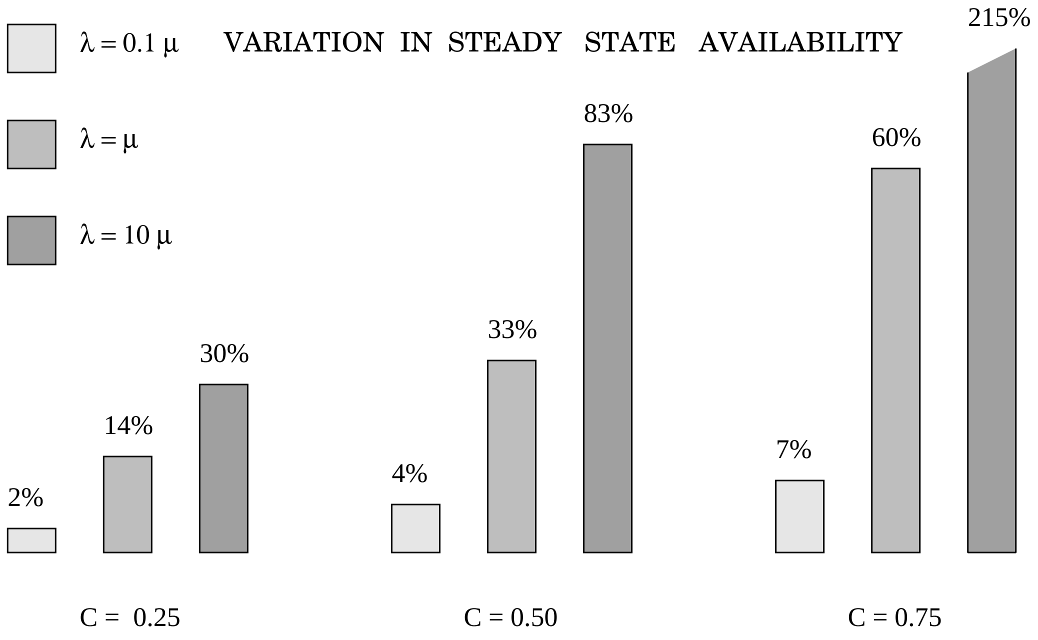

Figure 3.5: Analysis of the improvement on the steady-state availability, given different values of coverage.

system will remain operational is increased by a factor correspondent to the coverage c of the system. The Markov chain is depicted in figure 3.4.

The steady-state availability of this system is the probability that it stays operational when time approaches infinity, and it is given by the following formula [31, 32]:

$$p_{op}(\infty) = \frac{\mu}{(1-c)\lambda + \mu}$$

Our purpose is to obtain a general function that permits the analysis of the method under all possible conditions. This function is the ratio of the steady-state availability after and before the fault tolerant scheme is implemented, considering possible values of the coverage c. The ratio of the steady-state availability of fault-tolerant systems (Av_{ft}) in relation to current systems (Av_{cur}) gives this function:

$$\frac{Av_{ft}}{Av_{cur}} = \frac{\lambda + \mu}{\lambda + \mu - \lambda c}$$

The graphs in figure 3.5 gives the improvement in the steady state availability for various values of coverage.

As it is difficult to obtain real values for λ and μ, we performed the evaluation considering the relation between them. When λ and μ are equal, and coverage is high, the improvement in the steady-state availability is significant. When the repair rate is high compared with the error rate there is not much room for improvement in the dependability of the system. But, on the other hand, when the repair rate falls, the improvement is significant, making the solution especially justified for critical failures over the network.

3.4 Routing on the Internet & Proxy Placing

The Internet is partitioned into a disjoint set of autonomous systems that use an IGP (Internal Gateway Protocol) to propagate routing information inside each autonomous system. RIP (Routing Information Protocol), HELLO and especially OSPF (Open Shortest Path First) are among the commonly used IGP's. Besides these, EGP (External Gateway Protocol) and BGP (Border Gateway Protocol) are used by gateways from different autonomous systems to exchange routing information among themselves. Currently BGP has replaced EGP. All traffic routed from one autonomous system to a network in another will traverse one path, even if multiple physical connections exist. As Comer points in [21], it is difficult to switch to alternate physical paths if one fails, especially when the paths cross two or more autonomous systems.

An interesting work studying the behavior of BGP is described in [33]. This paper shows how unpredictable Internet routing is as seen by a network application. They found that the likelihood of encountering a major routing pathology more than doubled between the end of 1994 and the end of 1995, rising from 1.5% to 3.4%. The second parameter studied was *route stability*, or how frequently a given route change. It was shown that although Internet routes, in general, don't change frequently, it does occur. The third parameter, *route symmetry*, shows that 50% of the two routes between two nodes are different.

How does IP routing affect the proxies? The answer is that as network (IP) routes are used to compute the application routes, and the IP routes change with time, the algorithm must be run periodically, so that the proxy candidates for a given agent are kept correct. Besides that, a routing pathology may affect an application route.

To solve this problem a possible solution is to use source routing, i.e. to make all decisions related to routing of management packets at the NMS, instead of depending on the network layer. This solution is not recommended, for it is a universal police not to use source routing, unless strictly necessary, for it may interfere with other network algorithms, for example the one that controls congestion.

Recently we started the work on a new algorithm that solves this problem [34]. Instead of selecting a set of candidate proxies based on current network routes, the algorithm uses the unreachable agent's physical neighbors as proxies. If a proxy turns out to be also unreachable, their neighbors are used as proxies for proxies, and so on, until problems are identified.

Another solution to this problem is to use distributed monitoring, and this approach is shown in chapter 5.

3.5 Obtaining Network Topology

It is clear that network topology and configuration information is essential for determining application routes. It must be understood that a managed network is at most a collection of autonomous domains, whose topology and configuration can and must be available to the human manager. Network configuration information may be obtained from a number of sources, as pointed below.

- Static network maps: Network managers/operators may have access to such maps. But information in such maps is, more often than not, out-of-date.
- Directly from the network: Several tools and management devices exist by which some configuration information may be extracted by querying the elements that comprise the network. The disadvantage is that the network configuration itself is dynamic and it is generally expensive to query the network to keep track of these changes.
- Network configuration information servers: These servers may collect and update configuration information as the real network changes. A new approach is proposed in [35, 36], in which a communication network with related details and descriptions is represented in the X.500 directory. Applications may access the directory to obtain configuration information. Although this approach constitutes an effective way to maintain network configuration information [37, 38], it is not yet widely deployed.

Any of these approaches or their combination provides effective means by which a management entity may extract the topology of the network for the algorithm to calculate a set of application routes that will be used to achieve a fault tolerant management framework.

3.6 Using Proxies for Fault Management

An essential aspect of network management is to detect the presence of faults or to assure the absence of faults. There may be cases when the management system cannot say whether a network element is operational or not. The absence of response from a network element does not necessarily indicate the failure of the element. It may be due to a failed intermediate link.

In general, the NMS does not have any way of determining the state of network elements that lie beyond a failed network link. This is an undesirable

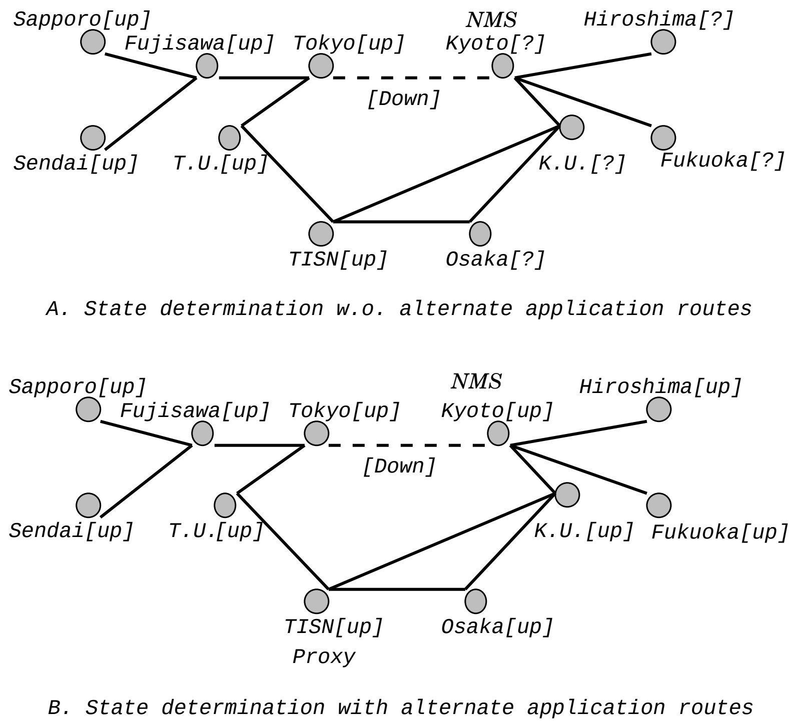

Figure 3.6: Proxy helps state determination.

state as it defeats the purpose of the NMS. However *application routes* do provide a convenient means by which one can bypass the failed link to gather vital information.

For example in figure 3.6, part A, the NMS is able to successfully poll the agents *Sapporo*, *Fujisawa*, *Sendai*, *T.U.*, and *TISN*. This is reflected in the state description *viz.[UP]* in the node labels. However due to a breached link indicated by the dotted line it is unable to poll the remaining agents. This is indicated by the state description *[?]*. The NMS is unable to determine the exact extent of the problem and may at best generate *alarms* for each of the unreachable nodes. These *alarms* can be deceptive and confusing as the real problem is camouflaged among the side effects. But in figure 3.6, part B, the Proxy agent relays the management queries, allowing the NMS to determine not only the state of the previously unreachable agents but also to detect the position of the fault, namely the broken link of the figure.

A significant benefit of using *application routes* is that it provides a convenient and simple way to improve the NMS capability to reliably monitor the network, allowing the system to bypass faults and reach agents that would otherwise be unreachable.

3.7 Proxy Implementation

The proxy was implemented as a conventional SNMP MIB: a simple and flexible approach that allows any agent to become a proxy at virtually no cost.

The MIB contains a table of which a row is made up of the following objects:

```
RproxyEntry ::= SEQUENCE {
  agentAD IpAddress,
  mgmtOBJ OBJECT IDENTIFIER,
  commPXY DisplayString,
  resultPXY DisplayString }
```

The NMS sets the address of the agent to be queried in variable *agentAD*, the object identifier to be queried in *mgmtOBJ*, and the community that should be used in *commPXY*. After that, by querying the *resultPXY* object, the proxy will issue an snmpget on the agent whose address is *agentAD*, for the object whose identifier is *mgmtOBJ* and using *commPXY* as the community. The result of the query is sent back to the NMS.

For example, if the NMS sets on the proxy:

```
agentAD <- 200.7.8.1
commPXY <- public
mgmtOBJ <- 1.3.6.1.2.1.1.1.0
```

After the settings are confirmed, the NMS can do an *snmpget* on the proxy's *resultPXY*. The result of the get is the reply to the query as received by the proxy from the machine whose IP address is "200.7.8.1".

By using *snmpwalk* the NMS is able to identify a row in the table that can be used for a new agent and object. It is the responsibility of the NMS to erase that row, after it is no longer necessary.

3.8 A Case Study

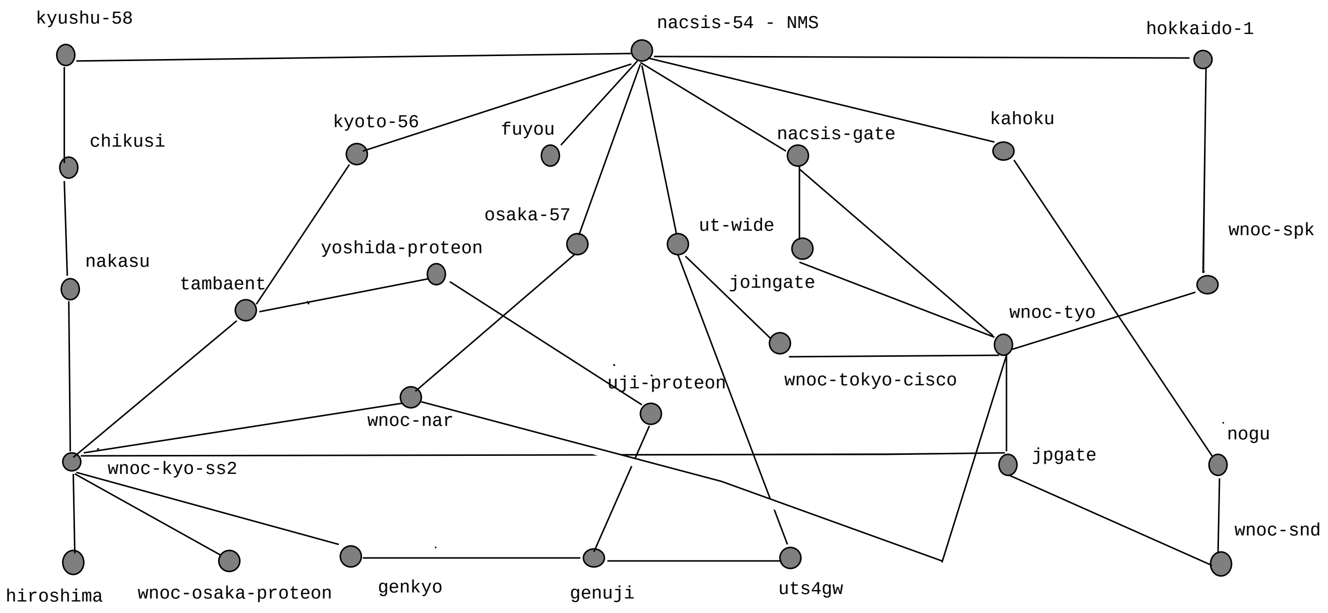

Figure 3.7: A large real wide-area network.

The algorithm was applied for a wide area network. The results are discussed in this section. The topology is shown in figure 3.7, with 28 nodes and 37 links. The NMS is located at *nacsis-54* and all the other nodes are agents. The physical network configuration was obtained by referring to the X.500 directory. The *network routes* were computed by running a series of *traceroute's*.

The results of the algorithm for this network follow the graph depicting the physical interconnections. The set of proxies selected was *tambaent*, *uji.proteon*, *wnoc.nar*, and *wnoc.tokyo*, figure 3.8. shows for which agents they can be activated. These four are the smallest set of proxies that covers all agents. Evidently, for each agent the set of candidate proxies contains more candidates. Using the MIB presented in the previous section, all the selected candidates can deploy proxies at virtually no cost.

The vulnerability before having proxies had a value of 60, and the vulnerability after proxies is merely 3. This huge increase in the dependability of the system is justified by the fact that few link failures have the effect of putting the NMS and agents on disjoint connected components, and thus allows the usage of proxies to reach the agents in such cases.

3.9 Conclusion

The major contribution of the results presented in this chapter is that they introduce a practical way to improve the reliability of current centralized network management systems. The presented approach provides the management application with application routes, that are made up with some nodes that act as management proxies, i.e., if an agent becomes unreachable through the network route, the NMS can use a proxy to reach the agent through an alternative path. Different approaches to obtain network topology information are discussed. An algorithm is employed to find alternative paths from the NMS to the agents. The algorithm selects the proxies based on the number of alternative paths they provide, and the sizes of these paths. Eventually, the topology of the network dictates if it is possible to have a proxy for a given node. In some cases it is not possible to find alternative paths completely disjunct from the usual path. In other cases no alternative paths exist at all, and this shows a risky portion of the network. The use of proxies for network management has a direct benefit in its capability of performing network state determination. The solution was modeled as a Markov process and a function that gives the improvement on the steady-state availability of the system was analyzed for different values of fault/repair rates as well as of the fault coverage of the system, showing that the proposed approach is specially effective for critical failures over the network. To illustrate the solution, a case study of a large network was carried out. The proxies were implemented as an SNMP MIB which allows their deployment with virtually no cost.

```
Thu Jun 30 14:12:50 JST 1994
*******************************************************
NetConf.                            [Program Output]

  The best set of proxies for NMS on nacsis-54
  is..........:

    Proxy                      For Agents on
 <=========>   <==========================================>
 tambaent      kyushu.58           wnoc.kyo.ss2      hiroshima
               wnoc.osaka.proteon  uji.proteon       genkyo
 uji.proteon   yoshida.proteon
 wnoc.nar      chikusi             nakasu            kyoto.56
               genuji              jp.gate           uts4gw
               wnoc.tyo            nacsis.gate       tambaent
 wnoc.tyo      osaka.57            wnoc.nar          wnoc.snd
               nogu                hokaido           wnoc.spk
               ut.wide             wnoc.tokyo.cisco  joingate
               kahoku

 Risky Nodes:
       fuyou
********************************************************
Thu Jun 30 14:12:51 JST 1994
```

Figure 3.8: Results of the algorithm for the Japanese Internet.

Chapter 4

System-Level Diagnosis for Fully Connected Networks

Consider a system consisting of N units, also called nodes or processors, that can communicate with each other. Each of these units can be in one of two states: *faulty* or *fault-free*.

The goal of a fault tolerant system is to make a faulty unit invisible to the application, so that the rest of the system performs the activities that are necessary. This goal implies that it is fundamental for the system to detect which units are working properly, and which are not. In other words, each system component must detect and diagnose the state of the other components.

A possible solution to the problem of diagnosis is to use a brute-force algorithm, in which every node tests all others. It is easy to see that this approach is not sound. At every testing interval every node has to issue $N-1$ tests and reply to $N-1$ tests. This testing strategy which may present a substantial overhead for both individual processors and the communication network. Other solutions must be examined.

In this chapter we review system-level diagnosis for fully connected systems, and introduce the *Hi-ADSD* algorithm.

4.1 The PMC Model

The goal of system-level diagnosis is to determine the state of a units of the system. For almost 30 years, researchers have worked on this problem, and the first model of diagnosable systems was introduced by Preparata, Metze, and Chien, the *PMC Model* [8]. In the PMC model, units are able to *test* other units and determine their status. Each unit is assigned a subset of the other units to test, and fault-free units are able to accurately assess the state of the units they test. Faulty units may report incorrect test resuls.

The set of all tests makes up a testing graph, i.e., a directed graph in which vertices represent the system's units and an edge from vertex i to vertex j corresponds to a test performed by unit i on unit j. For example, figure 4.1 shows a testing assignment on a system of five nodes, where node 1 is faulty.

The collection of all test results is called the *syndrome* of the system. The problem of diagnosis is to obtain the state of the system from a given syndrome. The PMC model assumes the existence of a *central observer* that, based on the syndrome, can diagnose the state of all the units.

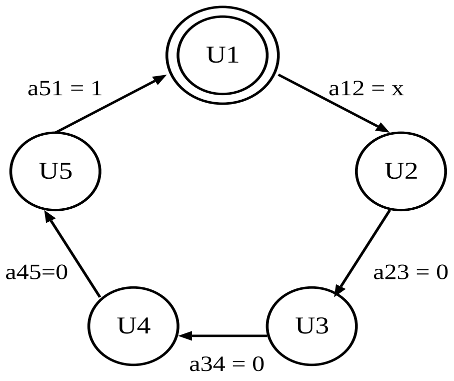

Figure 4.1: An example of PMC model's testing assignment.

For a given testing assignment, the diagnosability of a system may be limited by the number of faulty units, and determining this number is called the *diagnosability problem.* Preparata *et al.* showed that for a system to be t-diagnosable, it is necessary that $N \geq 2t + 1$, and that each unit is tested by at least t other units. Later, Hakimi and Amin [39] proved that if no two units test each other these conditions are sufficient for t-diagnosability.

For many years the research on system-level diagnosis concentrated on what is called *one-step* diagnosis, i.e., finding testing-graphs that provided enough information for the central observer to perform diagnosis efficiently. The concept of a fixed testing assignment changed when *adaptive* diagnosis was introduced, as shown in the next section.

4.2 Adaptive System-Level Diagnosis

Early system-level diagnosis algorithms assumed that all the tests had to be decided in advance. The tests were then executed, and from the obtained results, it was determined which units were faulty. Those algorithms focused on finding properties of the testing graph which would allow the observer to identify the faulty units from the tests corresponding to the testing graph's edges.

An alternative approach, which requires fewer tests, is to assume that each unit is capable of testing any other, and to issue the tests adaptively, i.e., the choice of the next tests depends on the results of previous tests, and not on a fixed pattern. Hakimi and Nakajima called this approach *adaptive* [9]. Early adaptive system-level diagnosis results assumed the existence of the previously mentioned central observer. Furthermore, a bound on the number of faulty nodes was imposed for the system to achieve correct diagnosis.

Adaptive system-level diagnosis algorithms proceed in testing rounds, i.e., the period of time in which each unit has executed the tests it was assigned. To evaluate adaptive algorithms two measures are normally used: the total number of tests required per testing round and the diagnosis latency, or delay, i.e., the number of testing rounds required to determine the state of the units.

4.3 Distributed System-Level Diagnosis

Previously, Kuhl and Reddy [40, 41], introduced *distributed* system-level diagnosis, in which fault-free nodes reliably receive test results through their neighbors, and each node independently performs consistent diagnosis. They proposed the SELF distributed system-level diagnosis algorithm, that although fully distributed, is non-adaptive, i.e., each unit has a fixed testing assignment, and the number of faulty units in the system cannot exceed t. We will use alternatively the word node for unit, and network for system.

Later, Hosseini, Kuhl and Reddy, [10] extended the SELF algorithm, introducing the NEW-SELF algorithm, which also has a fixed inter-node test assignment, but is executed on-line, permitting faulty nodes to reenter the network after being repaired. NEW-SELF ensures the accuracy of test-results by restricting the forwarding of test results to fault-free nodes. For correct diagnosis, NEW-SELF requires that every fault-free node receive all test results from all other fault-free nodes. To reduce the amount of network resources required for diagnosis, the EVENT-SELF algorithm was proposed by Bianchini *et.al.*[42] This algorithm uses event-driven techniques to improve both the diagnosis latency and the impact of the algorithm on network performance.

4.4 The Adaptive-DSD Algorithm

The Adaptive Distributed System-level Diagnosis algorithm, *Adaptive-DSD*, was introduced by Bianchini and Buskens [11, 12]. Adaptive-DSD is at the same time distributed and adaptive. Each node must be tested only one time per testing interval. All fault-free nodes achieve consistent diagnosis in at most N testing rounds. There is no limit on the number of faulty nodes for fault-free nodes to diagnose the system.

Adaptive-DSD is executed at each node of the system at predefined *testing intervals*. Each time the algorithm is executed on a fault-free node, it performs tests on other nodes until it finds another fault-free node, or it runs out of nodes to test. A *testing round* is defined as the period of time in which all nodes of the system have executed Adaptive-DSD at least once. After one testing round, if there are at least two fault-free units, the testing graph has the format of a ring, as shown in figure 4.2. In the example shown in figure 4.2, node 1, node 4, and node 5 are faulty, and the rest are fault-free. Node 0 tests node 1 and finds it faulty; so it goes on and tests node 2, which is fault-free, and then stops testing. Node 2 then tests node 3 as fault-free, and so on.

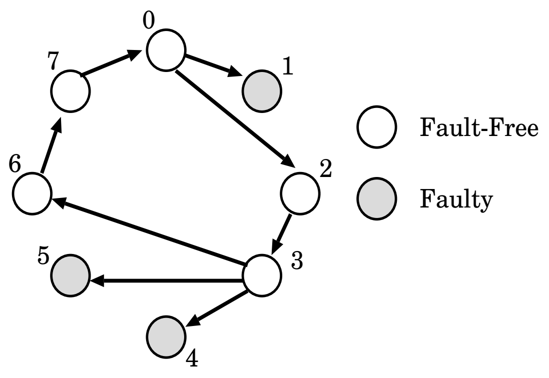

Figure 4.2: Example of test assignment in Adaptive-DSD.

Each node i that executes the algorithm has an array called TESTED-UP$_i$, that contains N entries, indexed by the node identifier. The entry TESTED-UP$_i[k] = j$ means that the node i has received diagnostic information from a fault-free node specifying that node k has tested j to be fault-free. An entry TESTED-UP$_i[j]$ is "arbitrary" if node j is faulty.

When node i finds node j to be fault-free, it saves this information in TESTED-UP$_i[i]$. In the next testing round, this test data of i is taken by its first fault-free predecessor, and so on, until all nodes get the information. In this way, the diagnostic information in the TESTED-UP array is forwarded to nodes in the reverse direction of the testing network. Using the information in TESTED-UP$_i$ a node i has to diagnose the state of all nodes in system, for this task, another algorithm, called *Diagnose*, is employed.

Adaptive-DSD has a diagnosis latency of N testing rounds. It is desirable to reduce this latency. In the original papers, Bianchini and Buskens use event-driven mechanisms to reduce the latency, like employing multicast or broadcast just after a new situation is identified. As none of the suggested event-driven mechanisms are fault-tolerant themselves, there is no proof that they can reduce the latency of Adaptive-DSD.

From next section, we present the Hierarchical Adaptive Distributed System-level Diagnosis (Hi-ADSD) algorithm. Hi-ADSD is hierarchical in the sense that it employs a divide-and-conquer testing strategy [43]. Hi-ADSD is the first hierarchical diagnosis algorithm that is at the same time adaptive and distributed. Previous hierarchical approaches include [44], [45], [46] and [47]. Hi-ADSD has diagnosis latency of log^2N rounds in the worst case, without employing extra event-driven mechanisms, and requiring less diagnostic information than Adaptive-DSD.

The results discussed here assume a fully connected network, no link faults and the PMC fault model. Besides the PMC fault model, many other fault models have been proposed. For example, a survey of probabilistic diagnosis is presented in [48]. Diagnosis of link faults were treated in [49]. Diagnosis on networks of general topology has received a great deal of attention recently, e.g. [50], [51], [52], and [53].

4.5 Adaptive DSD with Intersections

In this section we briefly introduce *ADSD with Intersections*, an adaptive distributed system-level algorithm that groups nodes in clusters to reduce the latency of Adaptive-DSD, the algorithm is presented in detail in [43, 54]. In ADSD with Intersections, nodes are grouped in logical clusters, and within each cluster nodes execute tests forming a ring, like in Adaptive DSD. But all clusters have a point of intersection, which is a node that tests nodes in all clusters. If there is a total of p clusters, it takes N/p testing rounds to diagnose the state of all N nodes. At each testing interval, all nodes still test one fault-free node, but the node at the intersection tests p nodes, one at each cluster.

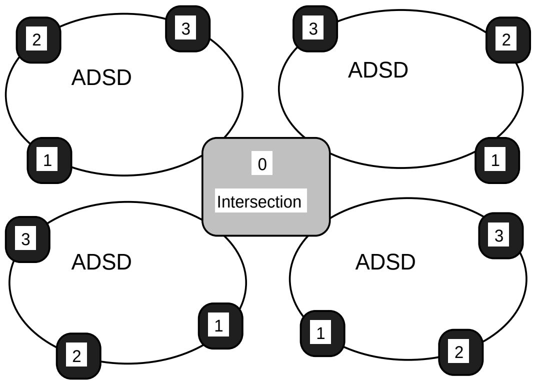

Figure 4.3: Multiple simultaneous clusters run Adaptive-DSD with intersection.

In ADSD with Intersections the intersection node presents a number of challenges. Not only does this node execute more tests, but also there should be a procedure to provide an intersection fault-tolerance. An election scheme is proposed in [54].

An interesting result is the best latency that this approach can give. For a given system of N nodes, one must determine which number of clusters, p, of which size, optimize the diagnosis latency of the system, while keeping the number of messages as low as possible. The total number of messages in the system per testing round is given by function $f(p) = p + N/p$. To find the number of clusters that minimize the number of messages we have to find a minimum of $f(p)$. This minimum is reached when $p = \sqrt{N}$, i.e., the best organization is to organize nodes in $\sqrt{N}$ clusters of size $\sqrt{N}$. Which gives an algorithm of diagnosis latency on the order of $O(\sqrt{N})$ testing rounds.

4.6 Hierarchical Adaptive System-Level Diagnosis

In this section the Hierarchical Adaptive Distributed System-Level Diagnosis *(Hi-ADSD)* algorithm is presented, its correctness is formally proved, and it is compared to the Adaptive-DSD algorithm. Hi-ADSD maps nodes to clusters, which are sets of nodes, and employs a divide-and-conquer testing strategy to permit nodes to independently achieve consistent diagnosis in at most $log^2 N$ testing rounds.

Before the algorithm is specified, it is important to recall the concepts of *test* and *testing round*, to avoid confusions. These concepts are the same used by Bianchini and Buskens for Adaptive-DSD in [11, 12]. At specified time intervals, for example 30 seconds, each fault-free node in the system executes *tests* on other nodes of the system, until the testing node finds another node that is fault-free, or tests all other nodes as faulty. For instance, if the first node tested is fault-free, the tester stops testing; otherwise, it will test another node, and so on, until a fault-free node is found. A *testing round* is defined as the period of time in which every fault-free node in the system has tested another node as fault-free, and has obtained diagnostic information from that node, or has tested all other nodes as faulty. The *diagnosis latency* of Hi-ADSD is defined as the number of *testing rounds* required for all fault-free nodes in the system to achieve diagnosis.

4.6.1 Algorithm Specification

Consider a system S consisting of a set of N nodes, n_0, n_1, ..., n_{N-1}. In this paper we alternatively refer to node n_i as *node i*. The system is assumed to be fully connected, i.e., there is a communication link between any two nodes (n_i, n_j). Each node n_i is assumed to be in one of two states, *faulty* or *fault-free*. A combination of the state of all nodes constitutes the system's fault situation. Nodes perform tests on other nodes in a testing interval, and fault-free nodes report test results reliably.

In Hi-ADSD, nodes are grouped into *clusters* for the purpose of testing. Clusters are sets of nodes. The number of nodes in a cluster, its size, is always a power of two. Initially, N is assumed to be a power of 2, and the system itself is a cluster of N nodes.

A cluster of n nodes $n_j, ..., n_{j+n-1}$, where $j \,\mathrm{MOD}\, n = 0$, and n is a power of

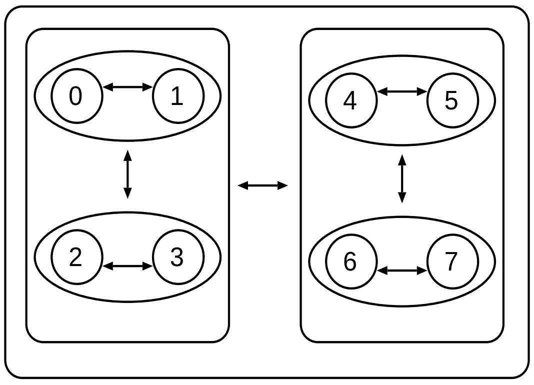

Figure 4.4: A hierarchical approach to test clusters.

two, is recursively defined as either a node, in case $n = 1$; or the union of two clusters, one containing nodes $n_j, ..., n_{j+n/2-1}$ and the other containing nodes $n_{j+n/2}, ..., n_{j+n-1}$. Figure 4.4 shows a system with eight nodes organized in clusters.

In the first testing interval, each node performs tests on nodes of a cluster that has one node, in the second testing interval, on nodes of a cluster that has two nodes, in the third testing interval, on nodes of a cluster that has four nodes, and so on, until the cluster of 2^{logN-1}, or $N/2$, nodes is tested. After that, the cluster of size 1 is tested again, and the process is repeated.

The lists of ordered nodes tested by node i in a cluster of size 2^{s-1}, in a given testing interval, are denoted by $c_{i,s}$. The following is an expression that completely characterizes list $c_{i,s}$, for all $i = 0, 1, ..., N-1$, and $s = 1, 2, ..., logN$. In the expression, a DIV b is the quotient of the integer division of a by b, and a MOD b is the remainder of the same integer division.

$$c_{i,s} = \begin{array}{l} \{n_t \,|\, t = (i \text{ MOD } 2^s + 2^{s-1} + j) \text{ MOD } 2^{s-1+a} + \\ (i \text{ DIV } 2^s) * 2^s + b * 2^{s-1} \,;\, j = 0, 1, ..., 2^{s-1} - 1\} \end{array}$$

Where:

$$a = \begin{cases} 1 & \text{if } i \text{ MOD } 2^s < 2^{s-1} \\ 0 & \text{otherwise} \end{cases}$$

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2,3	3,2	0,1	1,0	6,7	7,6	4,5	5,4
3	4,5,6,7	5,6,7,4	6,7,4,5	7,4,5,6	0,1,2,3	1,2,3,0	2,3,0,1	3,0,1,2

Table 4.1: $c_{i,s}$, for the system in figure 2.

$$b = \begin{cases} 1 & \text{if } a = 1 \text{ AND } (i \text{ MOD } 2^s + 2^{s-1} + \\ & j) \text{ MOD } 2^{s-1+a} + (i \text{ DIV } 2^s) * 2^s < i \\ 0 & \text{otherwise} \end{cases}$$

When node i performs a test on nodes of $c_{i,s}$, it performs tests sequentially, until it finds a fault-free node, or all other nodes are faulty. Supposing a fault-free node is found, from this fault-free node, node i copies diagnostic information of all nodes in $c_{i,s}$. For the system in figure 4.4, for all i and s, $c_{i,s}$ is listed in table 4.1.

If all nodes in $c_{i,s}$ are faulty, node i goes on to test $c_{i,s+1}$ in the same testing interval. Again, if all nodes in $c_{i,s+1}$ are faulty, node i goes on to test $c_{i,s+2}$ and so on, until it finds a fault-free node, or all nodes are found to be faulty. For example, figure 4.5 shows the testing hierarchy for 8 nodes, from the viewpoint of node 0. When node 0 tests a cluster of size 2^2, it first tests node 4. If node 4 is fault-free, node 0 copies diagnostic information regarding nodes 4,5,6 and 7. If node 4 is faulty, node 0 tests node 5, and so on.

Hi-ADSD uses a tree to store information about the tests in all clusters. To effectively diagnose the state of all nodes, it is sufficient to list all nodes in the tree. Figure 4.6 shows the tree for node 0, for the case that all nodes are fault-free.

A description of the algorithm in pseudo-code is given in figure 4.7.

It is important to observe that the system is asynchronous, i.e., at any time, different nodes in the system may be testing clusters of different sizes.

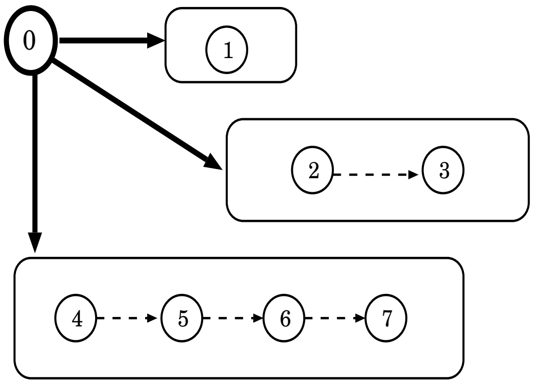

Figure 4.5: Each node adaptively tests all clusters.

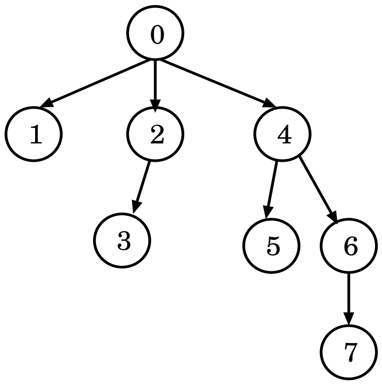

Figure 4.6: A tree keeps all testing information.

In other words, a node running Hi-ADSD does not know which tests are being performed by other nodes at any time. Even if nodes could be initially synchronized, after some of them become faulty and recover, the system would lose the initial synchronization. If there are at least two fault-free nodes in the system, in a testing round of Hi-ADSD, each node has tested at least one other fault-free node in c_{i,s_t}, but the other nodes don't know which s_t. This fact has major consequences on the performance of the algorithm, as will be seen in the next subsection.

It is assumed that a node cannot fail and recover from that failure during the time between two tests by another node. In Hi-ADSD this time may be of up to $logN$ testing rounds, in the worst case. This assumption can be enforced by, for example, recording and storing fault events, or by reducing

```
Algorithm Hi-ADSD;
{ at node i }
{ please refer to the text for c(i,s) }
{ j indexes the nodes of a given c(i,s) }
REPEAT
  FOR s := 1 TO logN DO
    REPEAT
     node_to_test := next in c(i,s);
     IF "node_to_test is fault-free"
     THEN "update cluster diagnostic information"
    UNTIL ("node_to_test is fault-free") OR
        ("all nodes in c(i,s) are faulty");
    IF "all nodes in c(i,s) are faulty"
    THEN "erase cluster diagnostic information";
  END FOR;
FOREVER
```

Figure 4.7: The Hi-ADSD algorithm.

the testing interval between consecutive tests [11].

In Hi-ADSD, whenever a faulty node becomes fault-free, it doesn't have complete diagnostic information. The tester of such a node must not get diagnostic information from this node, for it can be incorrect. An enough amount of time, at most $log^2 N$ testing rounds, should be allowed before diagnostic information can be obtained from that node.

However, during the algorithm initialization, every node has incomplete diagnostic information, for they have been fault-free for less than $log^2 N$ testing rounds. To guarantee the correct initialization, it is sufficient to have all nodes diagnostic information initialized as fault-free. The nodes will have the correct diagnostic information after the initial $log^2 N$ testing rounds.

An alternative approach, is to use a third state, call it *unknown*. This state is used whenever a node doesn't know the state of another node. Whenever a tested fault-free node has unknown information about nodes in the cluster being tested, the tester continue testing.

4.6.2 Correctness Proof

We now proceed to prove the correctness and the worst case of the diagnosis latency of the algorithm. For this proof we assume a system fault situation that doesn't change for an enough amount of time, until all fault-free nodes achieve diagnosis. The correctness proof of Adaptive-DSD also carried this assumption.

We begin by defining the *Tested-Fault-Free* graph, $T(S)$.

Definition 1 *The Tested-Fault-Free graph $T(S)$ is a directed graph whose nodes are the nodes of S. For each node i, and for each cluster $c_{i,s}$, there is an edge (i, t), directed from i to $t \in c_{i,s}$ if i has tested t as fault-free in the most recent testing interval in which it tested $c_{i,s}$.*

In $T(S)$ for each node i and each $c_{i,s}$, there is an edge directed from node i to the last node that node i tested as fault-free in that $c_{i,s}$. If, in the most recent testing interval in which node i tested $c_{i,s}$, all nodes in $c_{i,s}$ were tested as faulty, then $T(S)$ doesn't contain an edge from node i to any node in that $c_{i,s}$.

For example, consider a system of 16 nodes. Figure 4.8 shows the Tested-Fault-Free graph of that system, if all nodes are fault-free. It can be seen that it is a hypercube. It contains a directed edge from any node i to the last node that i tested as fault-free in $c_{i,1}$, another edge to the last node that i tested as fault-free in $c_{i,2}$, another edge to the last node that i tested as fault-free in $c_{i,3}$, and another edge to the last node that i tested as fault-free in $c_{i,4}$.

Lemma 1 *For any node i, any given s, and at any given instant of time t_i, it takes at most $logN$ testing rounds for node i to test $c_{i,s}$.*

Proof: *This follows from the definition of the algorithm, i.e., at a given testing interval node i tests a cluster, and looks for a fault-free node in that cluster. In one testing round, by definition, each fault-free node tests at least another fault-free node, if there is one. There may be at most $logN$*

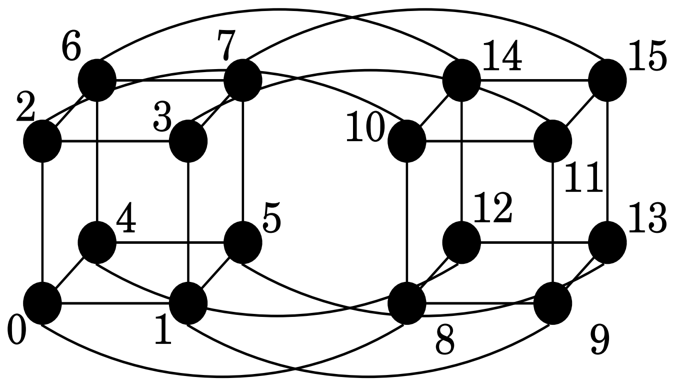

Figure 4.8: The Tested-Fault-Free graph for a system of 16 fault-free nodes.

clusters for node i to test. In logN consecutive intervals, at each interval a different cluster is tested. Thus, if node i executes exactly one successful test per testing round, it will take logN testing rounds for it to test all clusters. Therefore, in the worst possible case, for t_i immediately after a given cluster is tested, it will take up to logN testing rounds for that cluster to be tested again. □

Theorem 1 *The shortest path between any two fault-free nodes in $T(S)$ contains at most logN edges.*

Proof: *We will conduct an induction on t, for a system of 2^t nodes.*

First, consider a system of 2^1 nodes; each node tests the other, thus the shortest paths in $T(S)$ contain one edge.

Next, assume that for a system of 2^t nodes, a shortest path between any two nodes in $T(S)$ contains at most t edges. Then, by definition, in the system of 2^{t+1} nodes there are two clusters of 2^t nodes. Consider a subgraph of $T(S)$ that contains only the nodes in one of these clusters. By definition, this subgraph is isomorphic to the Tested-Fault-Free graph of a system of 2^t nodes. So, by the assumption above, the shortest path between any two nodes in this subgraph has at most t edges. Consider any two nodes, i and j. If i and j are in the same cluster of 2^t nodes, the shortest path between them in T(S) has at most t edges. Now, consider the case in which i and j are *in different clusters of 2^t nodes. Without loss of generality let's consider the shortest path from i to j. Node i tests one node in the cluster in which j is contained, call this node p. In T(S), the shortest distance from i to p contains one edge, and the shortest distance from p to j contains at most t edges. Thus the shortest distance from i to j contains at most $t + 1$ edges.* □

As an example, consider a system of size 2^2; this system has size four, and each node tests two other nodes, and gets information about the fourth node indirectly, through the tested nodes. This makes up a path of length two. Now consider a system of size 2^3, there are two clusters of size 2^2, and each node in one cluster tests one node in the other, thus, in T(S), there is an edge from each node in one cluster to the other. Therefore, the paths from a node in one cluster to the nodes in the other have lengths of the paths within the cluster which are at most of length 2, plus 1, for the edge linking the two clusters. Thus, in a system of size 2^3, the shortest path has length at most 3. For example, look at node 5 and node 2 in figure 4.9. For node 5 to get information about node 2, node 5 tests node 1, which tests node 3 which tests node 2. In this system of 8 nodes, the maximum path has size $log 8$.

Now let's consider each test in this worst case shortest path. How many testing rounds does it take to execute one test, in the worst case? Consider figure 4.9 again. If node 3 has tested node 2 just before it became faulty, then only after three testing rounds node 3 will discover that node 2 is faulty. Then, in the worst case, if node 1 tests node 3 just before node 3 tests node 2, it will take other three testing rounds for node 1 to discover that node 2 is faulty. If we are very unlucky and node 5 tested node 1 just before node 1 tested node 3, then it will take other three testing rounds for node 5 to discover that node 2 is faulty. In other words, there are three tests in the shortest path of longest length, and each one takes three testing rounds to be executed in the worst case, thus, in total, it may take up to nine testing rounds to execute all three tests.

Theorem 2 *Consider the system fault situation at a given time. After at*

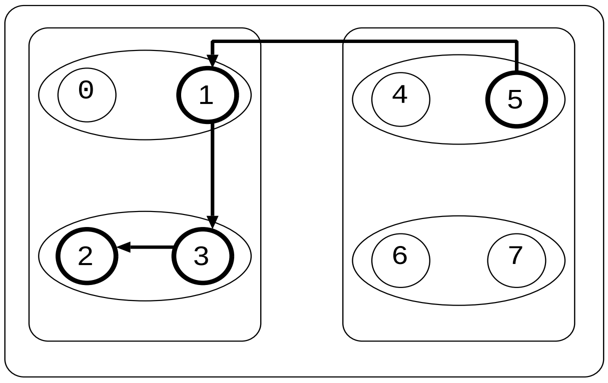

Figure 4.9: The shortest path from node 5 to node 2 has $log8 = 3$ edges.

most log^2N testing rounds, each node that has remained fault-free for that period correctly determines that fault situation.

Proof: *It was proved in theorem 1 that the shortest path between any two nodes in $T(S)$ has at most $logN$ edges. But, from lemma 1, each of the tests corresponding to an edge in $T(S)$ can take up to $logN$ testing rounds to be executed in the worst case. In other words, there are up to $logN$ different tests to execute, and each may take up to $logN$ testing rounds to be executed. So, in total, they may take at most $logN$ *$logN$ testing rounds to be executed. Thus, it may take up to log^2N testing rounds for a fault-free node to obtain diagnostic information about an event in S.* □

We believe that, in average, nodes running Hi-ADSD achieve diagnosis in less than log^2N testing rounds, and our experimental results confirm this fact. If nodes are roughly synchronized they will run the algorithm in $O(logN)$ testing rounds. If extra synchronization mechanisms are introduced better bounds can be guaranteed.

It should be clear that in Hi-ADSD, like in Adaptive-DSD, there is no limit on the number of faulty nodes for fault-free nodes to perform consistent diagnosis. In the worst case, when $N - 1$ nodes are faulty, the number of tests required is still N. For example, if $N - 1$ nodes are faulty, the fault-free node must test all other nodes to diagnose the system.

It is not necessary that the number of nodes, N, be a perfect power of 2.

In this case, testing nodes must skip the $2^{\lceil logN \rceil} - N$ non-existing nodes during test and diagnostic information transfer. For instance, the implementation discussed in section 4 was done on a 37-node system.

Nevertheless the worst possible latency is $\lceil log^2 N \rceil$, as there are at least *some* nodes for which the longest path in the Tested-Fault-Free graph have length $logN$. For example, consider the system of eight nodes in figure 4.9. If that system had six nodes instead of eight, i.e. if it didn't have node 6 and node 7, the length of the longest path would still be $log8 = 3$.

4.6.3 Comparison of Adaptive-DSD and Hi-ADSD

To compare Hi-ADSD and Adaptive-DSD we begin comparing the number of testing rounds required by both algorithms. We then compare the number of tests required, and conclude with the amount of diagnostic information that must be exchanged by nodes in the system until the fault situation is diagnosed.

The first difference between the two algorithms is their worst case diagnosis latencies, in terms of testing rounds. While Adaptive-DSD's diagnosis latency is N testing rounds, Hi-ADSD's is $log^2 N$.

Table 4.2 lists the diagnosis latency in terms of testing rounds for both algorithms, for networks having from 4 to 1024 nodes. The figures in this table should be clearly understood. They show the number of testing rounds that are needed for all nodes in the system to diagnose one event in the fault situation. For example, if all nodes are fault-free, and one node becomes faulty, that diagnostic information will take N testing rounds in Adaptive-DSD, being transferred sequentially through N nodes until all nodes diagnose the situation. In Hi-ADSD, the diagnostic information will be transferred through a tree of depth $logN$ and to reach all nodes it takes at most $log^2 N$testing rounds. For networks of 4 and 16 nodes, the algorithms present the same worst case latency. In one case, for a network of 8 nodes, Adaptive DSD presents better latency than Hi-ADSD, but this changes quickly as the number of nodes grows.

N	Hi-ADSD	Adaptive-DSD
4	4	4
8	9	8
16	16	16
32	25	32
64	36	64
128	49	128
256	64	256
512	81	512
1024	100	1024

Table 4.2: Examples of diagnosis latency.

To compare the number of tests required by both algorithms we show the number of tests required in one testing round. When all nodes are fault-free, both algorithms employ exactly the same number of tests per testing round, for each fault-free node executes tests until it finds another fault-free node. However, if there are faulty nodes in the system, Adaptive-DSD needs N tests per testing round, while Hi-ADSD may need more tests, depending on which nodes are faulty, and which clusters are being tested in the testing round.

These extra tests correspond to the situation in which two or more nodes test a given faulty node in the same testing interval. In this case, those nodes will run more tests. The lists of nodes to be tested in each cluster ($c_i s$) described previously in this section make this situation unlikely, as all entry points are specific for each node to its clusters. However, in the worst possible case, if $N/2$ nodes are faulty, and they are all in the same cluster, and all testers test this cluster in the same testing round, the total number of tests is $N^2/4$. It should be clear that this case is very rare, for even if $N/2$ are faulty, the probability that they are all in the same cluster is not large.

Now consider the total number of diagnostic messages transferred from fault-free nodes required by the algorithms. Adaptive-DSD requires a total of N^2 messages for all nodes to achieve diagnosis, while Hi-ADSD requires

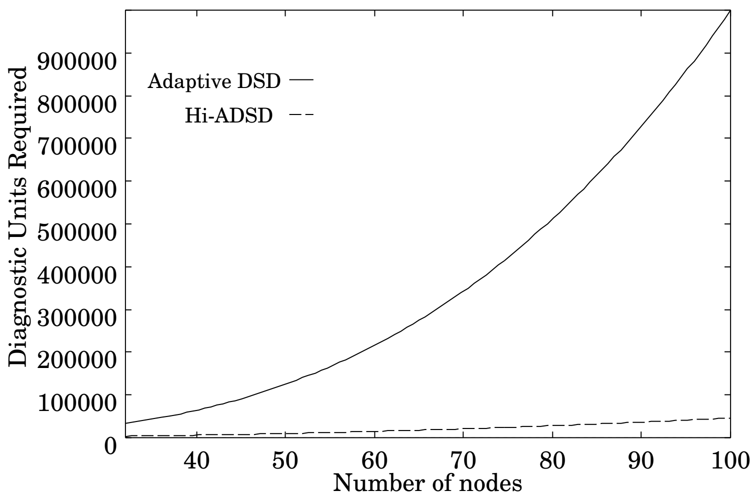

Figure 4.10: Comparison of the amount of diagnostic units required.

$N\ log^2N$ messages in the worst case.

There is also a major difference in the size of diagnostic messages in Adaptive-DSD and Hi-ADSD. Nodes running Adaptive-DSD get messages with diagnostic information concerning all nodes in all testing intervals, in contrast, Hi-ADSD's diagnostic messages only contain information about the nodes in each cluster being tested. Let's call the information about one node a *diagnostic unit.* Consider $logN$ consecutive testing intervals, during this period, a node running Adaptive-DSD requires $NlogN$ diagnostic units, while a node running Hi-ADSD requires only $2^0 + 2^1 + ... + 2^{logN-1} = N - 1$ units during the same period.

Figure 4.10, compares the total number diagnostic units required by both algorithms, for all nodes to achieve diagnosis. It can be seen that Hi-ADSD brings a significant improvement in terms of network bandwidth utilization.

The comparison shown in figure 4.10 is not meaningful if extra mechanisms, like timestamps, could be employed to avoid transferring diagnostic messages unless strictly necessary. Using these mechanisms, only information regarding a new event is transferred. However, to use any mechanism like this it is necessary to prove its correctness and impact on the algorithm.

Number k of Tests Executed	Number of Nodes that Diagnosed the Event
1	1
2	3
3	7
4	8
5	10
6	14
7	21
8	35
9	63

Table 4.3: Number of tests required for 63 nodes to diagnose one event.

4.7 Simulation

In this section we present experimental results of diagnosis on large networks using Hi-ADSD, obtained through simulation. The simulation was conducted using the discrete-event simulation language *SMPL* [55]. Nodes were modeled as SMPL facilities, and each node was identified by a SMPL token number. Three kinds of events were defined: (1) test, (2) fault, and (3) repair. Tests were scheduled for each node at each $30 \pm \sigma$ units of time, where σ is a random number between 0 and 3.

We modeled the fault as the facility being reserved, and the repair as the facility being released. During each test, the status of the facilities are checked and, if the node is fault-free, diagnosis information regarding the cluster is copied to the testing node. If the tested node is faulty, the testing nodes proceed testing as in the algorithm.

We conducted several experiments with networks of different sizes. In this paper we present results of two experiments: in the first experiment, on a network of 64 nodes, after a node becomes faulty, a second node also becomes faulty, and after that they are sequentially repaired. These four events were scheduled for times 100, 1000, 2100 and 3000, respectively. The

Number k of Tests Executed	Number of Nodes that Diagnosed the Event
1	1
2	3
3	7
4	15
5	31
6	63
7	64
8	66
9	70
10	77
11	91
12	119
13	175
14	287
15	511

Table 4.4: Number of tests required for 511 nodes to diagnose one event.

second experiment was conducted on a network of 512 nodes, a fault occurs at time 100, and the node is repaired at time 1100. Results of diagnosis presented here are representative from the large set of simulation runs executed for each experiment.

Table 4.3 and table 4.4 show the number of tests it takes for fault-free nodes to diagnose the first event of each experiment.

Table 4.3 shows that for the first event in the 64-node system, the 63 fault-free nodes take up to $k = 9$ tests to successfully diagnose the event. For example, there is one node that successfully diagnoses the event after one test, this node tested directly the faulty node.

Table 4.4 shows that for the first event in the 512-node system, the 511 fault-free nodes take up to $k = 15$ tests to successfully diagnose the event. To compare with Adaptive-DSD, without extra event-driven mechanisms, we point out that Adaptive-DSD would take 511 testing rounds for all fault-free nodes to diagnose any event in this 512-node system.

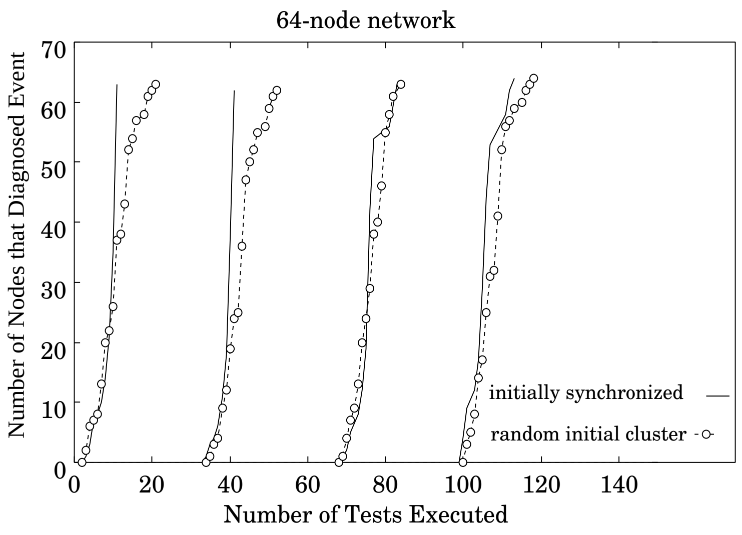

Figure 4.11: Simulation of Hi-ADSD on a 64-node network.

As we discussed before, nodes running Hi-ADSD run tests asynchronously, with consequences on algorithm performance. For each of experiments, we ran a second simulation, in which each node starts testing from a random cluster, as opposed to starting synchronized by testing cluster 1. The graphs on figures 4.11 and 4.12 show results from both types of simulation.

Both graphs have the number of testing rounds as the x-axis and the number of nodes that diagnosed the event as the y-axis. For the first event of the 64-node system, the original simulation took up to 9 tests, while the random version took up to 21 tests. For the second event, they took 8 and 18 tests respectively.

For the first event on the 512-node system, the first experiment took up to 15 tests, the random experiment took 52 tests. The second event took 17 tests, and the random experiment took 50 tests.

These experiments confirm the impact of the asynchronous execution of tests on Hi-ADSD's performance.

As a final comment, the graph of diagnosis for the initially synchronized 512-node system, shown in figure 4.12, slow during some periods. During

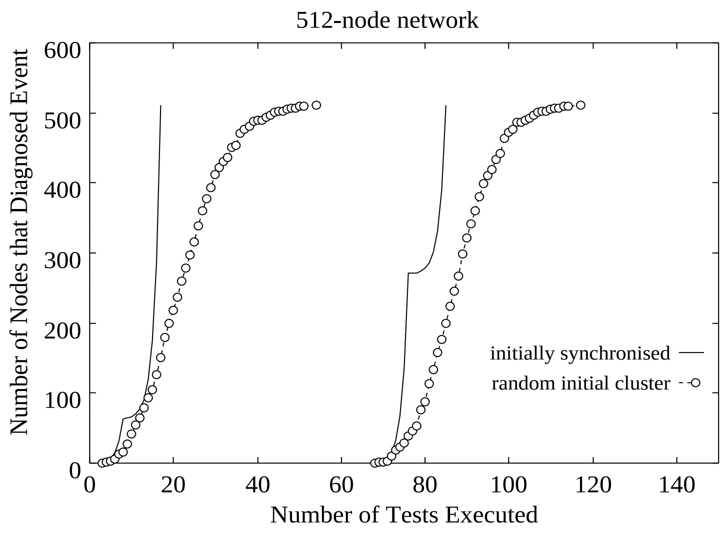

Figure 4.12: Simulation of Hi-ADSD on a 512-node network.

those periods, nodes are running tests on the faulty node as a small cluster. After those slow periods, diagnosis speeds up again, in result of the propagation of diagnostic messages to larger clusters.

4.8 Practical Implementation

In this section we present the application of Hi-ADSD to SNMP-based LAN fault management. Initially we describe the role of the NMS (Network Management Station) when Hi-ADSD is used for fault management. This is followed by the description of an approach to include monitoring of network devices, the description of our implementation and finally the experimental results obtained.

4.8.1 The Role of the Network Management Station

To apply system-level diagnosis to network fault management, it must be taken into account that the primary goal of SNMP-based fault management is to permit a central NMS to determine the state of all nodes in the network, in a reliable and efficient way. By reliable, we mean that if any node fails in the network, the diagnosis process continues, even if the faulty-node is the current NMS itself. By efficient, we mean that the diagnosis is accomplished within a small delay, and the overhead imposed by diagnosis messages requires a reasonable percentage of network bandwidth.

One of the goals of network management systems is to provide network state information to the human manager at the NMS. The concept of a central observation point is not contradictory with the previously presented distributed approach: the NMS can be seen now as a *management interface*, and not as the single monitor. This approach gives a number of advantages to the human manager, as she/he has a choice of workstations to monitor the network. Furthermore, there are obvious advantages in terms of the reliability of the network monitoring system itself, as fault-free nodes achieve correct diagnosis for any number of faulty nodes.

It has been shown that Hi-ADSD has a diagnosis latency of at most $log^2 N$ testing rounds. To further reduce this latency at the NMS, a feasible solution is to employ SNMP traps, i.e., an agent reports any new state information as soon as it is discovered. This combination of distributed monitoring and traps gives the system high resilience over errors, while keeping delays conveniently short. The NMS receives all changes in state information as soon as they are discovered. Using a simple configuration mechanism, all stations know the current NMS identity. Nevertheless, this event-driven approach is not fault-tolerant. There is no assurance that traps will be correctly delivered. However, even if the NMS is changed (or becomes faulty) soon after receiving and acknowledging the trap, by the time another node assumes the role of NMS, the information is delivered to this new NMS through the testing network.

4.8.2 Network Device Fault Management

To permit Hi-ADSD to monitor the state of network devices, each unit is classified into a *testing* node or a *tested-only* node. *Testing* nodes are usually workstations, which are not only subject to tests, but are also capable of testing. In contrast, *tested-only* nodes are only tested, and don't perform any testing on other elements. A number of managed devices, like printers, modems, terminals, among others are *tested-only.* Furthermore, to improve the diagnosis delay, some workstations may be *tested-only.*

There are two possible approaches to include tested-only nodes in the algorithm. In the first approach, each *testing* node has some associated *tested-only* nodes, that are tested at each testing interval. Whenever a *testing* node finds another *testing* node to be faulty it must test all *tested-only* nodes associated with that faulty *testing* node. In the other approach, tested-only nodes are simply tested as normal testing nodes, the only difference is that they don't carry diagnostic information. Thus a MIB variable identifies of which class a given node is part. If the second approach is used, it is interesting to distribute the tested-only nodes wisely through the network, to avoid that specific nodes execute large number of tests.

We are working on a Java interface for the algorithm, allowing the network to be monitored using any WEB browser. We expect it to be ready soon. The current interface is based on log files.

4.8.3 Experimental Results

The implementation of Hi-ADSD was run on a 10 Mbps Ethernet LAN (Local Area Network) that consisted of 37 Sun workstations, SPARCstation 20. Several experiments were conducted. In this section we describe a representative set of experiments and diagnosis results.

The CMU SNMP public-domain packet [4] was used as a base to implement the diagnosis agent, in which we coded the *Diagnosis MIB* variables. From the SNMP toolkit of the WILMA project [56] we used client programs to access and update MIB variables.

The ASN.1 coding of the Hi-ADSD MIB, as implemented, is shown in the Appendix.

The program that implements Hi-ADSD runs on top of SNMP, using its services. In each test, initially an SNMP query is issued, and the correct reply is expected. As SNMP is an application layer entity, a correct reply implies that the node is fully fault-free, except possibly for other applications.

However, as SNMP itself is not fault-tolerant, and uses the UDP (Internet's User Datagram Protocol) unreliable transport protocol, a timeout may be caused by the SNMP server being faulty, or a lost message, not necessarily by the tested node being faulty itself. To handle this situation, in the second part of the test, a *ping* query is issued, and if there is a correct reply it is concluded that the tested node is *partially faulty*, or that SNMP is not replying to querries. If there is a ping timeout, it is concluded that the tested node is faulty.

This strategy was also used for fault-injection. A specific MIB variable was introduced, that, when querried, made the SNMP server "sleep" for a specified amount of time. In that period the remaining nodes in the network diagnosed that the node was not replying to SNMP, but also not faulty.

As SNMP tables indexes entries from 1, the nodes were assigned identifiers from 1 to 37.

The testing interval was set at 40 seconds, for all nodes.

Figure 4.13 shows how diagnosis progressed for the first three initial events. Initially, node 6 was actually faulty. Before the algorithm was initialized we didn't know that, but all remaining 36 nodes diagnosed the fault situation in 198 seconds, from the time they started testing.

After that, we did the first fault injection, and node 16's SNMP server stopped replying to queries. But, as the graph in figure 4.13 shows, at roughly the same time, node 6 was repaired, and started replying to ping. This was also an unexpected occurrence. From figure 4.13 it can be seen that although the pace of diagnosis was different for both events, they were diagnosed in roughly the same amount of time: 575 seconds for node 6's fault, and 562 seconds for node 6's partial recovery, considering for the latter case the time since the first node diagnosed the event.

Next we proceeded to inject faults on 3 nodes, first at node 35, and after some time, at nodes 20 and 21 simultaneously. The diagnosis of these events

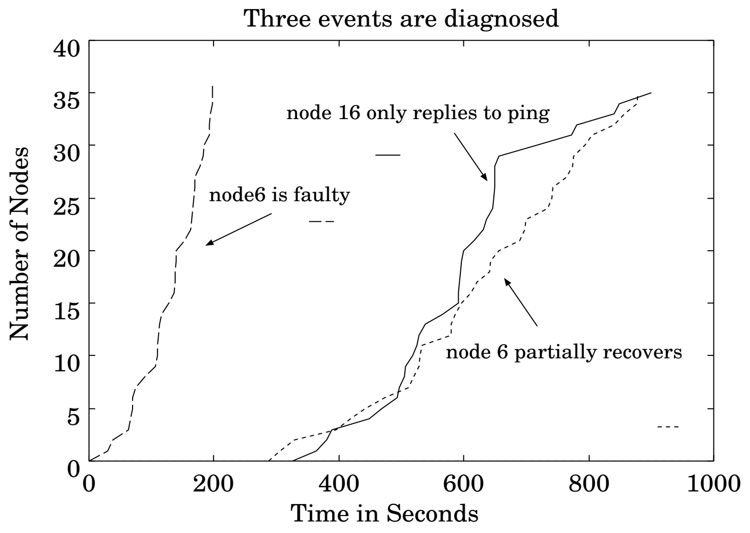

Figure 4.13: First three events of the experiment.

is shown in the graph of figure 4.14. The event at node 35 was diagnosed in 346 seconds. The events at node 20 and node 21 were diagnosed almost simultaneously, in 421 seconds, and 416 seconds. Not only the total amount of time, but also the pace in which both events were diagnosed was very similar, as can be confirmed by the fact that figure 4.14 seems to show the diagnosis of two events and not the real three.

Now all four nodes in which faults were injected are repaired. For the repair of node 16, the remaining nodes take 524 seconds, for the repair of node 35, they take 241 seconds, the second best latency we got. The remaining two events are the repair of node 20 and node 21, which are also diagnosed almost simultaneously, but not as much as was their previous fault diagnosis. The time was, respectively, 474 seconds and 514 seconds. We believe this slight difference is due to the fact that other nodes had been faulty and then repaired and were issuing testing not synchronously with other nodes.

The average latency for all the ten experiments above is 427.1 seconds. With a testing interval of 40s, latency could have been up to 1440 seconds. These results, together with previously shown simulation results, might confirm our belief that in average Hi-ADSD's latency is less than $log^2 N$ testing

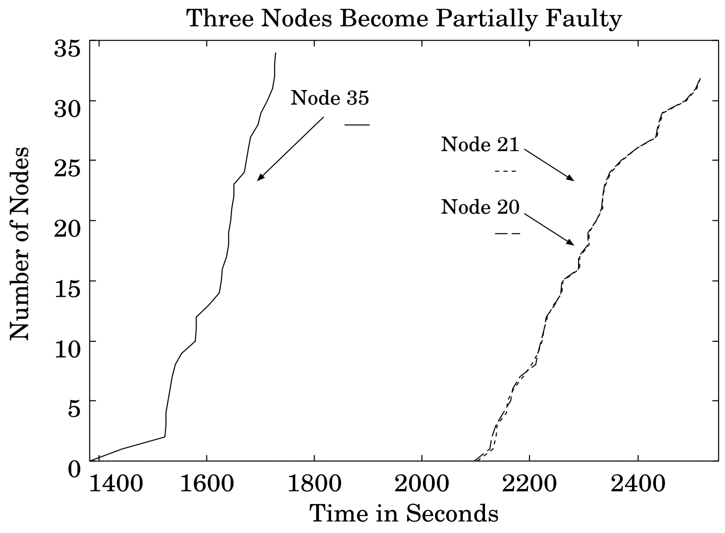

Figure 4.14: Next three events, but second and third are diagnosed almost simultaneously.

rounds.

4.9 Conclusions

In this chapter we introduced the application of distributed system-level diagnosis to SNMP-based fault management. We presented the Hierarchical Adaptive Distributed System-level Diagnosis algorithm. Hi-ADSD maps nodes to clusters, and uses a divide-and-conquer testing strategy to achieve diagnosis in at most $log^2 N$ testing rounds. In this way Hi-ADSD improves the diagnosis latency of previous algorithms, while keeping the number of tests conveniently low. The correctness and worst-case latency of the algorithm were formally proved. Simulation results of diagnosis on large networks of 64 and 512 nodes, obtained using simulation, were shown.

Hi-ADSD was implemented integrated to an SNMP-based network management system on a 37-node Ethernet LAN. Issues regarding the actual deployment of the algorithm were discussed, experimental results of fault and repair diagnosis were presented. As SNMP applications are currently

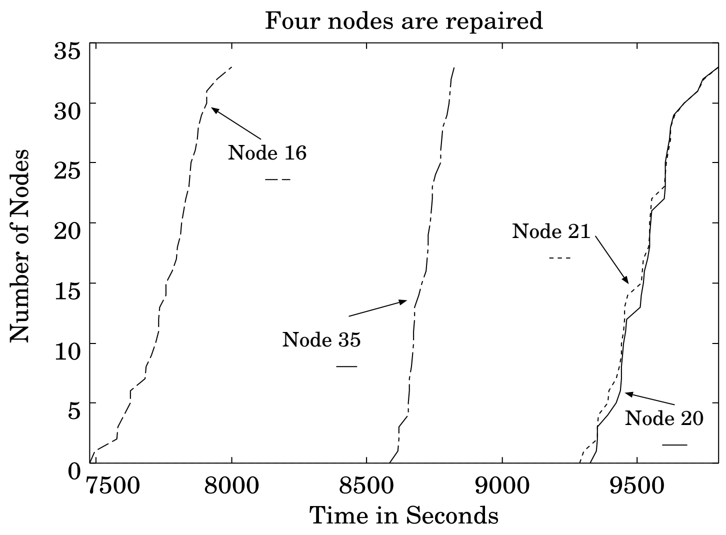

Figure 4.15: Simulation of Hi-ADSD on a 64-node network.

widely deployed, but fault management is still based on rudimentary procedures, this implementation by itself is also a significant contribution to the field of network management.

The next step of our research is to work on Hi-ADSD for a dynamic fault situation, in which any number of nodes become faulty and are repaired at any time. Other important issues include checking if synchronization mechanisms can guarantee a $logN$ diagnostic latency, fault-tolerant mechanisms for event-driven dissemination of events and for timestamps, that would guarantee the minimal amount of diagnostic information exchange.

Chapter 5

Non-Broadcast Network Fault-Monitoring Based on System-Level Diagnosis

Fault management is the set of activities required to guarantee network availability, even in the presence of network faults and performance degradation. Fault management must thus be fault tolerant, for network faults should not impair the system that is meant to solve them. Management can be broadly subdivided into monitoring and control. Monitoring is the process employed for obtaining information required about the components of a network, in order to make management decisions and subsequently control their behavior. In the previous chapter we introduced system-level diagnosis for fully connected networks applied for LAN fault management. In this chapter we present a fault-tolerant approach for non-broadcast network fault-monitoring also based on distributed system-level diagnosis.

As we have discussed, current SNMP-based fault-management systems are based on the manager-agent model, in which a fixed manager station queries a set of agents for management information. This centralized scheme is inherently unreliable, for if the manager becomes faulty, network management stops on the entire network.

System-level diagnosis offers a theoretically sound and practical framework for fault-tolerant network monitoring: even if any part of the network becomes faulty, fault-free nodes are able to diagnose the system.

In the previous chapter the Hi-ADSD algorithm for LAN fault-diagnosis was introduced, and also its implementation based on SNMP. In this algorithm, the number of tests is the same as in ADSD, but the testing topology is initially a hypercube, and diagnosis is reduced to $\log^2 N$ testing rounds. Those algorithms employ a distributed strategy for fault management, in which a collection of network nodes perform network diagnosis, and the human manager may attach an interface to any of these nodes to receive diagnostic information.

In this chapter we expand those results by introducing an algorithm for diagnosis in non-broadcast networks, applied to point-to-point network fault management. In the algorithm, a node tests links periodically, and disseminates link time-out information to all its fault-free neighbors in parallel. Upon receiving link time-out information a node computes which portion of the system has become unreachable. This new approach to diagnosis, based on *link time-out* and *node unreachability* is closer to reality than previous approaches. There are two reasons for this improvement: (1) it is impossible to distinguish a node fault from the fault on all the paths to that node; (2) in previous algorithms, two fault-free nodes in disconnected components keep the old status for each other, which may not correspond to reality.

A node joining the algorithm disseminates information about itself, and collects diagnostic information from its neighbors. The diagnosis latency of the algorithm is optimal, as nodes report events in parallel, and latency is proportional to the diameter of the network. The dissemination step includes mechanism to reduce the number of redundant messages introduced by the parallel strategy. We present a MIB for the algorithm, and a SNMP-based implementation. The evaluation of algorithm's impact on network performance shows that the amount of bandwidth required is less than 0.1% for popular link capacities. We conclude demonstrating the integration of LAN and WAN fault diagnosis into a unified framework.

The rest of the chapter is organized as follows. Section 5.2 reviews system-level diagnosis, including algorithms for LAN fault management. Section 5.3 reviews algorithms for diagnosis on networks of general topology, and includes the specification of the new algorithm for non-broadcast networks. In section 5.4 we present a MIB and a SNMP-based implementation of the algorithm. In section 5.5 we evaluate its impact on network performance. Section 5.6 concludes the chapter, showing the integration of LAN and WAN fault diagnosis.

5.1 System-Level Diagnosis for Networks of General Topology

In this section we review previously published algorithms for diagnosis in non-broadcast networks, which can be applied for point-to-point network fault management. We introduce our new algorithm in the next section.

In [51] Bagchi and Hakimi introduced an algorithm for system-level diagnosis in networks of general topology. Initially each fault-free node knows only about its own state, and of its physical neighbors. Fault-free processors form a tree-based testing graph. Diagnostic messages are sent along the tree. The number of messages required by this algorithm to achieve diagnosis is shown to be optimum. Unfortunately the algorithm is not executed *on-line*, i.e., no processor can become faulty or be repaired during the execution of the algorithm. This characteristic rules out the application of the algorithm for WAN fault diagnosis.

In [57, 52] Bianchini *et.al.* introduced and evaluated through simulation the Adapt algorithm. The Adapt algorithm can be executed on-line: when a given node becomes faulty, a new phase begins in which other nodes reconnect the testing graph. The underlying testing assignment of Adapt is a minimally strongly connected digraph over the physical network. To build the testing graph, Adapt employs a distributed procedure that requires massive amounts of large diagnostic messages to be exchanged among the nodes.

Recently Rangarajan *et.al.* [50] introduced another algorithm for system-level diagnosis for networks of arbitrary topology that can be executed on-line. The algorithm, which we call here *RDZ*, for the author's initials, builds a testing graph that guarantees the optimal number of tests, i.e., each node has one tester. Furthermore it presents the best possible diagnosis latency by using a parallel dissemination strategy. Whenever a node detects an event, it sends diagnostic information to all its neighbors, which in turn send it to all their neighbors, and so on.

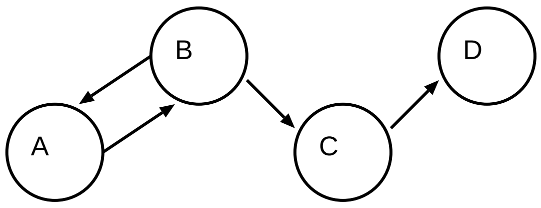

Figure 5.1: A *jellyfish* fault configuration.

Although the RDZ algorithm presents the best possible diagnosis latency, and the best possible number of testers per node, it does not diagnose link faults and also a node fault configuration which the authors call *jellyfish faulty node configuration*. In this fault configuration, between two connected components there is a set of nodes such that part of those nodes test each other in a cyclic fashion, and other tests emanate from the cycle. If all nodes in the jellyfish become faulty simultaneously, nodes in the connected components won't diagnose that situation. It should be noted that a jellyfish may involve from one to an arbitrary number of nodes.

Consider figure 5.1. All nodes form a jellyfish, in which there is a cycle (node A and node B) and tests emanating from the cycle (from node B to node C to node D). If both nodes A and B become faulty, nodes C and D won't be able to diagnose the fault. The same is true if nodes A, B, and C become faulty, i.e., node D doesn't detect the event. The RDZ algorithm cannot be applied for network fault monitoring, for it is unacceptable to have an arbitrarily large portion of the network to become faulty in an undetected fashion.

5.2 A New Algorithm for Diagnosis of Non-Broadcast Networks

In this section, we introduce a new algorithm that diagnoses link time-outs, and node reachability, using the minimum number of tests, i.e. one per link, and also presenting the optimal latency. Before introducing the algorithm, consider figure 5.2. In fault situation A the node is fault-free, but all links leading to that node are faulty, in fault situation B, the node itself is faulty. From test results it would be impossible for any other node in the system to determine which is the actual situation. Our algorithm is based on this fact: a link may *time-out* to a test, and if all links to a given node have timed-out, then the node is *unreachable.* Thus links may be in one of two states *fault-free, timed-out* and nodes may be *fault-free* or *unreachable.* This approach to fault diagnosis on wide-area networks is closer to reality, for links are usually made up of not only wires but may also involve a number of network devices, hubs and gateways.

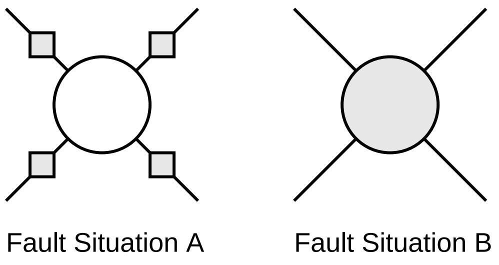

Figure 5.2: Ambiguous fault configurations.

To keep the number of tests minimum, there is one tester per link. As a link always connects two nodes, and nodes have unique identifiers, the node with the highest identifier tests the link at each testing interval. If the link *times-out*, i.e., the neighbor doesn't reply to the test, and in the past testing interval it did, then there is a new *fault event.* Analogously, if the link has timed-out in the past testing interval, and it does carry a reply this time, then there is a *repair event*.

The algorithm employs a *two-way test*. This guarantees that the *jellyfish* fault configuration is always detected, even keeping the minimum number of tests. When node A is testing the link to node B, node A gets the local time at B, and stores that result on B. In this way, not only node A knows about the state of B, but also node B can monitor the tester activity. If a threshold is decided for the maximum interval between link tests, then a node can time-out the tester whenever the threshold is exceeded. When a node detects a link time-out or a tester fault, it starts or continues testing the link until it ceases to time out, such that, when the link recovers again, only the node of highest identifier tests the link.

Each node keeps a state counter for each link in the system, which is initially zero, and is incremented at each new event information received for that link. This permits a node to identify redundant messages. After a new event is discovered, each node propagates event information to all neighbors. This parallel dissemination strategy is the same employed by the RDZ algorithm. Besides the nodes identifier, and the status counter, each diagnostic message carry information about which nodes have processed the message. In this way, the number of redundant messages is reduced, and messages do not cycle in the network. After receiving a message, each node appends its own identifier to the list of nodes that has processed the message. Furthermore it appends the identifiers of the neighbors to which the message was already sent. For a full discussion and evaluation of this approach please refer to [50]. It should be clear that, as messages are short, the impact of this strategy on network performance is small. In section 5 we evaluate the percentage of link bandwidth required to run the algorithm.

After a node receives information about a link event, it runs an algorithm (like the breadth-first tree) to compute the system connectivity, thus discovering which portions have become reachable or unreachable.

The data structures of the algorithm are thus:

- A Link table indexed by link identifier, containing a status counter for the link, and the last time the link was tested. The counter is initially zero, and an even value indicates a fault-free state; The last-test-time is updated only on nodes connected to the link and such that the node doesn't test the link, but its neighbor;
- A Link-Events table, containing at each entry the link identifier, the state counter of the link, and a list of nodes that have already processed the message as seen by the sending node.

The algorithm in pseudo-code is:

```
BEGIN
 /* at node i */
 DO FOREVER
  FOR each link i-j, that connects node i to node j
    IF (i > j) OR (node j is faulty)
    THEN test link i-j; /* get local time at node j */
         IF link i-j is fault-free
         THEN set last-time-tested on j;
         IF there is a new event
         THEN add event to new-event table;
    ELSE /* check link tested by neighbors */
    IF last-time-tested > testing interval threshold
    THEN add event to Link-Events table;

  FOR each entry in Link-Events table
       IF entry carries new information
       THEN update link counters;
            FOR each neighbor k of node i
                 IF k has not received the message
                 THEN set event information to k's new-event table;
  compute node reachability;
  SLEEP(testing interval)
END;
```

5.2.1 An Example Execution

Consider the example system in figure 5.4. Initially all links and nodes are fault-free. Each node starts testing links as depicted by arrows, and exchange test information with neighbors. Eventually each node receives information about all links.

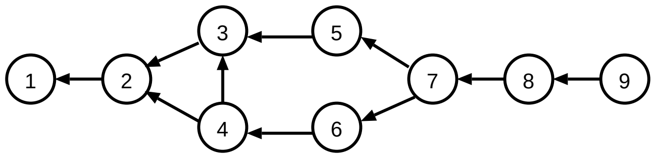

Figure 5.3: The testing assignment on an example non-broadcast network.

Now consider the first event depicted in figure 5.4, in which link 3-5 is faulty and times-out. This time-out will be detected by node 5, and immediately disseminated to node 7. This in turn will disseminate to node 8 (and from there to node 9), and node 6. Node 6 disseminates information to node 4, and from there to node 3, node 2 and node 1. Node 2 disseminates the information to node 3. Now, if node 3 timed-out out the tester (link 3-5) before the information arrives from node 2, then node 3 will also disseminate information on the time-out. If a node, say node 2, receives two diagnostic messages about the same event it will only disseminate the first of them, because the second is recognized as old information. Thus, the highest number of messages per event per link is two. After all nodes receive and process diagnostic messages, they run an algorithm to compute system connectivity, and conclude that all nodes are still connected.

In the second event depicted in figure 5.4, node 4 became faulty. Node 6 detects a time-out on link 6-4, and after the testing threshold expires, node 2 and node 3 detect time-outs on links 4-2 and 4-3 respectively. The system now is divided into two connected components, one consisting of node 1, node 2 and node 3, the other consisting of node 5, node 6, node 7, node 8 and node 9. As on each component a node detects and disseminates the event, diagnostic information will eventually reach every fault-free node in

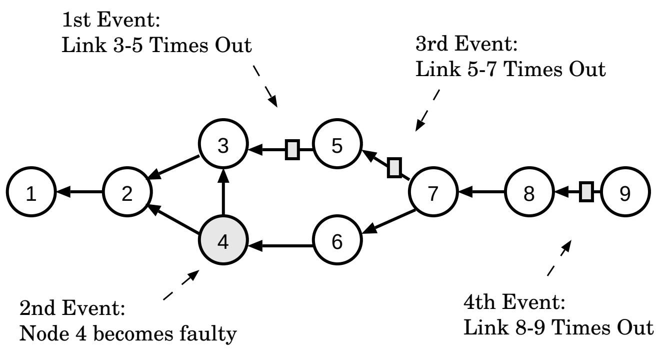

Figure 5.4: A series of events occur in the network.

the system.

Now consider the third event, link 7-5 becomes faulty and times-out. The resulting system has 3 connected components, the first consisting of node 1, node 2, and node 3; the second of node 5 alone; and the third of node 6, node 7, node 8, and node 9. In the first component not one node detects the event, because it is already disconnected from the rest of the system. In the second component, node 5 eventually times out on the test of link 7-5 and realizes it is disconnected from the system, i.e., every other node is unreachable. At the third component, node 7 initially detects link 7-5 time-out and the event is disseminated to the other nodes in the component.

If still another link, 9-8, becomes faulty and times-out, node 9 detects the event and recognizes it is disconnected from the system. Node 8 times-out on the testing threshold of link 9-8, and disseminates event information to node 7 and node 6. The other nodes in the network are already in disconnected components.

After these events, when faults are repaired, nodes testing corresponding links will detect the events, and disseminate the information to other nodes in their connected components. Eventually the whole system becomes a unique connected component, and every node receive diagnostic information about all links.

Correctness

Here we give an informal discussion of the correctness of the algorithm. Consider a connected component of the system, made up of fault-free nodes and such that between any pair of those nodes there is a fault-free path. The neighborhood of the component is defined as the set of links that timed-out in the previous testing-round. Clearly, any new event in the component or in its neighborhood is detected by nodes in the component. This is assured by the testing strategy, in which there is a two-way test on each link from any node of the component. Now consider that one event has occurred. If a fault-free node or link has become faulty, then one node in the component will detect the fault, and forward it to other neighbors. As each node forwards new information to all neighbors, information will eventually reach all nodes in the component. If the fault breaks the component in two, then nodes on both components will detect a link time-out, and disseminate the information on their respective components. Now consider a repair event: if a test succeeds on a link that had been timing-out, the two nodes (tested and tester) exchange diagnostic information, and disseminate this information to their neighbors. Thus event information is always disseminated to every fault-free node in the component.

Event counters guarantee that old information is recognized as such. Furthermore, those links that have odd event-counters are timed-out links and those that have even-counters are fault-free links. This is guaranteed because a counter is only incremented when a new event happens, from timed-out to fault-free or opposite. As the counter is initially zero, for a fault-free initial status, and when it times-out it is increased to 1 and so on, an even value will always indicate a fault-free state, and an odd value a faulty state.

5.3 SNMP MIB and Implementation

In this section we present an implementation of the algorithm for non-broadcast network fault management based on SNMP. Each node running the algorithm keeps two tables. The first table keeps information about each link in the network: its identifier, the state counter, and the time it was tested. The time field is only used by nodes that test a link to implement the two-way testing strategy. We give below the corresponding ASN.1 table.

```
LinkState OBJECT-TYPE
    SYNTAX  SEQUENCE OF LinkStateEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
        "Array that contains link status information."
    ::= { diagnosis 1 }

linkStateEntry OBJECT-TYPE
    SYNTAX  LinkStateEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
        "Each entry of shows if a link is timing-out
         or fault-free, according to the status counter"
    INDEX   { linkID }
    ::= { LinkState 1 }

LinkStateEntry ::=
    SEQUENCE {
        linkID         DisplayString,
        StatusCounter  Counter,
        TestedTime     TimeTicks  }
```

The second table, LinkEvents, is a dynamic table, in which event information is added by the local agent and its neighbors. After each testing interval, all entries in the table are processed. Each entry contains the identifier of the link that suffered the event, the timestamp for that event, and a string containing the identifiers of all the nodes that have already processed the message. The ASN.1 table is given below.

```
LinkEvents OBJECT-TYPE
    SYNTAX  SEQUENCE OF linkEventsEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
        "This is a dynamic table to which information
         about new link events are added."
    ::= { diagnosis 2 }

linkEventsEntry OBJECT-TYPE
    SYNTAX  LinkEventsEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
         "Each entry carries the link identifier,
          the status counter for a new event, and
          the identifiers of nodes that know the event"
    INDEX   { linkID }
    ::= { LinkEvents 1 }

LinkEventsEntry ::=
    SEQUENCE {
        LinkID         DisplayString,
        StatusCounter  Counter,
        Path           DisplayString  }
```

In our implementation nodes set neighbors tables, and thus security measures must be taken, specifically assignment of restricted access permission. It should be clear that from the LinkState table that the complete network configuration is available to each node, which can calculate the system connectivity at any time. Works on generating network configuration information automatically have been reported [37], and can be employed to build the LinkState table.

5.4 Impact on Network Performance

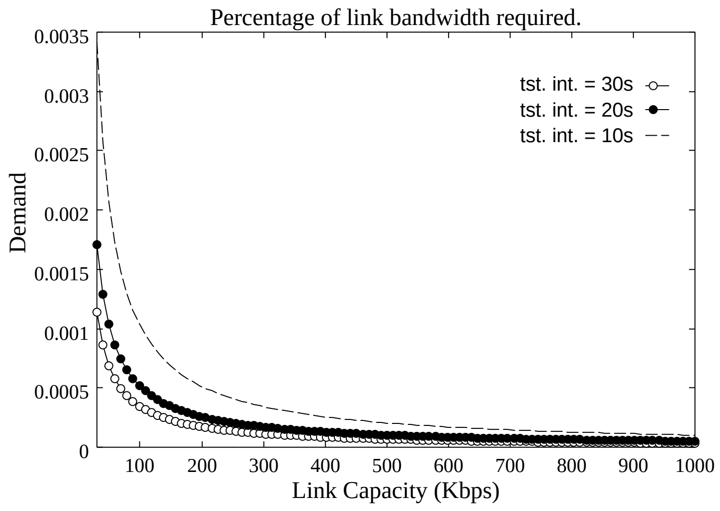

Figure 5.5: Amount of link bandwidth required to run diagnosis.

At each testing interval, each link carries one message from the tester to the neighbor. Furthermore, for any new event in the network, each link will carry usually one, and at most *two* messages about the event. The reason is that after updating the state counter, a node does not forward any other message that contains known information. The link will carry two messages only if both nodes send information at the same time. Thus, the total number of messages per event required by the algorithm is at most $2 * L$, where L is the number of links.

The graph in figure 5.5 shows the impact of the algorithm on network performance, by showing the percentage of link bandwidth required by diagnostic messages. The graph shows links of different capacities, and results are shown for different testing intervals, of 10 seconds, 20 seconds, and 30 seconds. We consider a fault rate λ of 0.001. The size of SNMP messages assumed is 128 bytes. Results show the percentage of bandwidth required is always less than 0.1%, on links from 28.8Kbps to 1Mbps.

5.5 Conclusions

In this chapter we presented a new distributed algorithm for system-level diagnosis on non-broadcast networks. The purpose of the algorithm is to allow each node to independently detect which portions of the network are faulty or unreachable. We show that in some cases it is impossible to distinguish between the two cases.

A node running the algorithm executes link tests at each testing interval. The algorithm employs the minimum number of tests, i.e., one per link. Of the two nodes connected by a link, the one with highest identifier is the link tester. We assume nodes have local memory, and tests are built in such a way that both ends of a link detect a link time-out in case of link or one node failure.

Upon detecting a new event, diagnostic information is disseminated in parallel, and the algorithm has the minimum diagnosis latency, i.e., proportional to the diameter of the network. Mechanisms are included to reduce the amount of redundant messages. As each message is small, containing information about one event, and any link carries at most two messages, the impact of the algorithm on network performance is small. A MIB and SNMP implementation were presented.

As future work, there is a pressing need for integrated approaches to do internet fault monitoring. This approach can be achieved by running specific algorithms for diagnosis on broadcast networks (LAN's), like Hi-ADSD, together with the algorithm introduced in this paper. Consider the small internet in figure 5.6. Nodes with identifiers from 1 to 9 are connected to broadcast networks. Node A, node C, and node D have a link to a broadcast network, and to a non-broadcast network. Node B takes part only in the non-broadcast network. For the two algorithms to run cooperatively, it

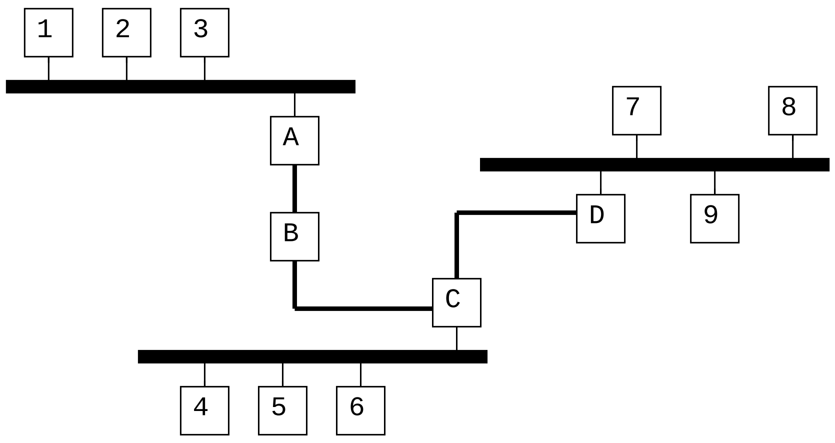

Figure 5.6: A small internet.

is sufficient that nodes only on a broadcast network run an algorithm for diagnosis on the LAN to which it belongs; nodes not on a non-broadcast network run the algorithm for diagnosis on that network; nodes that are on a broadcast network, but also have a link to another network must run both a LAN diagnosis algorithm, and a WAN diagnosis algorithm. This means these nodes execute tests according to the two algorithms, and also carry the necessary data structures to hold information about the entire system. In this way, a truly fault-tolerant network management system can be deployed, in which any fault-free node can diagnose the whole system.

Chapter 6

Conclusion

There is a pressing need for dependable network management systems. In current systems, a fault in the network may cause a partial collapse of the management entity. Considering that fault management is a key functional area of network management systems, this situation constitutes a paradox: the system is meant to solve faults, but those same faults impair the system. To tackle this problem, we have worked on algorithms and tools for fault-tolerant network monitoring. All theoretical work developed has been implemented in the SNMP framework, which was briefly described in chapter 2. SNMP was selected because it is currently widely available and deployed. However, the proposed solutions are in no way limited to this framework.

In chapter 3, we presented an approach to improve the dependability of centralized systems based on the manager-agent paradigm. When there is a fault on the route from the NMS to an agent or to a managed network, the manager station is unable to determine the state of part of the network. As network management is an application layer entity, it depends on the network layer for all routing decisions. To solve this problem, we proposed that the NMS use SNMP proxies to reach an agent, whenever it gets unreachable through the network layer route. In this way, a proxy can be used to bridge communications between the manager and agents. Algorithms were developed to locate proxies for each agent. The impact of this solution on the steady-state availability of the system was shown. An SNMP MIB implementation of the proxy was proposed that allows any agent to become a proxy at virtually no cost. For the future, we plan to work on expanding the concept of the proxy as the basis for a distributed network management system.

As network management is mission-critical, network monitoring must be fault tolerant. In other words, no matter how many nodes in the network were faulty, management would still be running on the fault-free nodes. To achieve this objective we proposed the application of distributed system-level diagnosis to network fault management. In chapter 4, we introduced system-level diagnosis and the Hierarchical Adaptive Distributed System-Level Diagnosis (Hi-ADSD) algorithm. Hi-ADSD is a fully distributed algorithm that has diagnosis latency of at most $(log_2 N^2)$ testing rounds for a network of N nodes. Nodes are mapped into progressively larger logical clusters, so that each node executes tests in a hierarchical fashion. The algorithm assumes no link faults and a fully-connected network, i.e. a LAN, in which all nodes share the broadcast medium to communicate. There are no bounds on the number of faults. Both the worst-case diagnosis and correctness of the algorithm were formally proved. Experimental results were given through simulation of the algorithm for large networks. Practical results were given from an implementation of the algorithm on a 37-node Ethernet LAN using SNMP. For the future, we plan to study the behavior of Hi-ADSD under a dynamic fault situation. Another interesting extension is the study of a synchronous version of Hi-ADSD.

In chapter 5, we proposed a new algorithm for system-level diagnosis of non-broadcast networks. This algorithm can be applied for on-line diagnosis of a WAN. In the algorithm, nodes test links periodically, and disseminate link time-out information to all its fault-free neighbors in parallel. Upon receiving link time-out information, a node computes which portion of the network has become unreachable. This approach is closer to reality than previous algorithms, for it is impossible to distinguish a faulty node from a node to which all routes are faulty. The diagnosis latency of the algorithm is optimal, as nodes report events in parallel, and latency is proportional to the diameter of the network. The dissemination step includes mechanisms to reduce the number of redundant messages introduced by the parallel strategy. We present a MIB which can be used to implement the algorithm using SNMP. The evaluation of the algorithm's impact on network performance shows that the amount of bandwidth required is less than 0.1% for popular link capacities. We proposed a MIB to implement the algorithm. The combination of LAN and WAN diagnosis on a solution for diagnosis of internets, which are a collection of LAN's connected through a WAN is left as future work.

Bibliography

[1] Y.Yemini. The OSI network management model. *IEEE Communications Magazine*, pages 20–29, May 1993.

[2] K. Terplan. *Communication Networks Management.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[3] W. Stallings. *SNMP, SNMPv2, and CMIP. The Practical Guide to Network Management Standards.* Addison Wesley, Reading, MA, 1993.

[4] M.T. Rose. *The Simple Book.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 1994.

[5] P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[6] S. Katker and M. Paterok. Fault isolation and event correlation for integrated fault management. In *Proc. IFIP/IEEE Integrated Network Management V (IM'97)*, pages 583–596, 1997.

[7] M.A. Rocha and C. Westphall. Proactive management of computer networks using artificial intelligence agents and techniques. In *Proc. IFIP/IEEE Integrated Network Management V (IM'97)*, pages 610–621, 1997.

[8] F. Preparata, G. Metze, and R.T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, 16(6):848–854, 1968.

[9] S.L. Hakimi and K. Nakajima. On adaptive system diagnosis. *IEEE Transactions on Computers*, 33(3):234–240, 1984.

[10] S.H. Hosseini, J.G. Kuhl, and S.M. Reddy. A diagnosis algorithm for distributed computing systems with dynamic failure and repair. *IEEE Transactions on Computers*, 33(3):223–233, 1984.

[11] R.P. Bianchini and R. Buskens. An adaptive distributed system-level diagnosis algorithm and its implementation. In *Proc. FTCS-21*, pages 222–229, 1991.

[12] R.P. Bianchini and R. Buskens. Implementation of on-line distributed system-level diagnosis theory. *IEEE Transactions on Computers*, 41(5):616–626, 1992.

[13] G. Berthet. *Extension and Application of System-Level Diagnosis Theory for Distributed Fault Management in Communication Networks.* PhD thesis, Ecole Polytechnique Federale de Lausanne, 1996.

[14] M.T. Rose and K. McCloghrie. Structure and identification of management information for TCP/IP-based internets. RFC 1155, PSI Inc., May 1990.

[15] J.D. Case, M.S. Fedor, M.L. Schoffstall, and J.R. Davin. A simple network management protocol. RFC 1157, SNMP Research Inc., May 1990.

[16] K. McCloghrie and M.T. Rose. Management information base for network management of TCP/IP-based internets. RFC 1213, Hughes LAN Systems, March 1991.

[17] W. Stallings. *Network Management.* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[18] M. Sloman. *Network and Distributed Systems Management.* Addison Wesley, Reading, MA, 1994.

[19] L. Steinberg. Techniques for managing asynchronously generated alerts. RFC 1224, IBM Corporation, May 1991.

[20] J.D. Case, K. McCloghrie, M.T. Rose, and S.L. Waldbusser. Introduction to version 2 of the internet-standard network management framework. RFC 1141, SNMP Research Inc., April 1993.

[21] D. Comer. *Internetworking with TCP/IP - Vol.1; Principles, Protocols and Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 1991.

[22] E.P. Duarte Jr., G. Mansfield, and et al. Reliable network management systems. In *Proc. 9th IEEE International Conference on Information Networking*, 1994.

[23] A. Ben-Ari, A. Chadna, and U. Warrier. Network management of TCP/IP networks: Present and Future. *IEEE Network Magazine*, July 1990.

[24] G. Mansfield and et al. An SNMP-based expert network management system. *The Institute of Eletronics, Information and Communication Engineers Transactions*, August 1992.

[25] C. Wang and M. Schwartz. Fault detection with multiple observers. *IEEE/ACM Transactions on Networking*, 1(1), February 1993.

[26] G. Mansfield, E.P. Duarte Jr., and et al. Vines: Distributed algorithms for a Web-based distributed network management system. In *Proc. ACM WWCA97 Lecture Notes in Computer Science.* Springer Verlag, 1997.

[27] W. Norton. Network discovery algorithms for the NSFNET. *ConneXions*, 1993.

[28] C. Alaettinoglu, A. Shankar, K. Dussa-Zieger, and I. Matta. Design and implementation of mars: a routing testbed. *Internetworking: Research and Experience*, 5:17–41, 1994.

[29] R. Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, MA, 1990.

[30] C. Das, T. Kreulen, M. Thazhuthaveethil, and L. Bhuyan. Dependability modeling for multiprocessors. *IEEE Computer*, 23(10), November 1990.

[31] B. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, Reading, MA, 1989.

[32] A. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, San Diego, CA, 2^{nd} edition edition, 1990.

[33] V. Paxson. End-to-end routing behavior in the Internet. In *Proc. ACM SIGCOMM*, 1996.

[34] E.P. Duarte Jr., G. Mansfield, T. Nanya, and S. Noguchi. WAN event-driven diagnosis based on SNMP delegates. FTS Technical Report, IEICE, 1997.

[35] G. Mansfield, T. Johannsen, and M. Knopper. Charting networks in the x.500 directory. RFC 1609, AIC Labs, 1994.

[36] T. Johannsen, G. Mansfield, M. Kosters, and S. Sataluri. Representing ip information in x.500 directory. RFC 1608, AIC Labs, 1994.

[37] G. Mansfield G., K. Jayanthi, and et al. Techniques for automated network map generation using SNMP. In *Proc. INFOCOM'96*, 1996.

[38] G. Mansfield and et al. Mapping communication networks in the directory. *Computer Networks and ISDN Systems*, 26(3), November 1993.

[39] S.L. Hakimi and A.T. Amin. Characterization of connection assignments of diagnosable systems. *IEEE Transactions on Computers*, 23(1):86–88, 1974.

[40] J.G. Kuhl and S.M. Reddy. Distributed fault-tolerance for large multiprocessor systems. In *Proc. 7th Annual Symp. Computer Architecture*, pages 23–30, 1980.

[41] J.G. Kuhl and S.M. Reddy. Fault-diagnosis in fully distributed systems. In *Proc. FTCS-11*, pages 100–105, 1981.

[42] R.P. Bianchini, K. Goodwin, and D.S. Nydick. Practical application and implementation of system-level diagnosis theory. In *Proc. FTCS-20*, pages 332–339, 1990.

[43] E.P. Duarte Jr. and T. Nanya. Multi-cluster adaptive distributed system-level diagnosis algorithms. Technical Report FTS 95-73, IEICE, 1995.

[44] M. Malek and J. Maeng. Partitioning of large multicomputer systems for efficient fault diagnosis. In *Proc. FTCS-12*, pages 341–348, 1982.

[45] A. Bagchi. A distributed algorithm for system-level diagnosis in hypercubes. In *Proc. 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 106–113, 1992.

[46] M. Barborak and M. Malek. Partitioning for efficient consensus. In *Proc. 26th Hawaii International Conference on System Sciences, Vol. II*, pages 438–446, 1993.

[47] J. Altman, F. Balbach, and A. Hein. An approach for hierarchical system-level diagnosis of massively parallel computers combined with a simulation-based method for dependability analysis. In *Proc. 1st European Dependable Computing Conference, LNCS 852*, pages 371–385, 1994.

[48] G. Masson, D. Blough, and G. Sullivan. *Fault-Tolerant Computer System Design*, chapter System Diagnosis. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[49] C.-L. Yang and G.M. Masson. Hybrid fault-diagnosability with unreliable communication links. In *Proc. FTCS-16*, pages 226–231, 1986.

[50] S.Rangarajan, A.T. Dahbura, and E.A. Ziegler. A distributed system-level diagnosis algorithm for arbitrary network topologies. *IEEE Transactions on Computers*, 44:312–333, 1995.

[51] A. Bagchi and S.L. Hakimi. An optimal algorithm for distributed system-level diagnosis. In *Proc. FTCS-21*, June 1991.

[52] M. Stahl, R. Buskens, and R. Bianchini. Simulation of the adapt on-line diagnosis algorithm for general topology networks. In *Proc. IEEE* 11^{th} *Symp. Reliable Distributed Systems*, October 1992.

[53] E.P. Duarte Jr., G. Mansfield, T. Nanya, and S. Noguchi. Non-broadcast network diagnosis based on system-level diagnosis. In *Proc. IFIP/IEEE Integrated Network Management V*, pages 597–609, 1997.

[54] E.P. Duarte Jr. and T. Nanya. Application of distributed system-level diagnosis for SNMP-based internet fault management. In *Proc. IEEE ICOIN'95*, pages 474–481, 1995.

[55] M.H. MacDougall. *Simulating Computer Systems: Techniques and Tools.* The MIT Press, Cambridge, MA, 1987.

[56] J. Swoboda and et al. *WILMA - Knowledge-Based LAN Management.* Technische Universitadt Munchen, http://www.ldv.e-technik.tu-muenchen.de/dist/WILMA/.

[57] M. Stahl, R. Buskens, and R. Bianchini. On-line diagnosis on general topology networks. In *Proc. Workshop Fault-Tolerant Parallel and Distributed Systems*, July 1992.