INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

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

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

MULTILEVEL DISTRIBUTED DIAGNOSIS AND THE DESIGN OF A DISTRIBUTED NETWORK FAULT DETECTION SYSTEM BASED ON THE SNMP PROTOCOL

A Dissertation

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirement for the

degree of

Doctor of Philosophy

By

MING-SHAN SU
Norman, Oklahoma
2002

UMI Number: 3038031

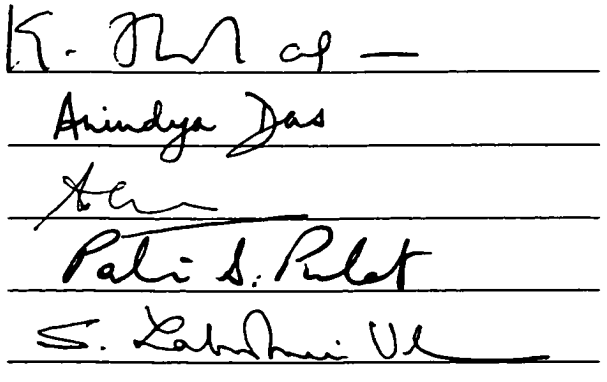
UMI Microform 3038031

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

MULTILEVEL DISTRIBUTED DIAGNOSIS AND THE DESIGN OF A DISTRIBUTED NETWORK FAULT DETECTION SYSTEM BASED ON THE SNMP PROTOCOL

A Dissertation APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY



ACKNOWLEDGEMENT

"Love the Lord your God with all your heart and with all your soul and with all your mind and with all your strength. ..., Love your neighbor as yourself."
[Mark 12:30-31]

The LORD is my shepherd, I shall not be in want. He makes me lie down in green pastures, he leads me beside quiet waters, he restores my soul. He guides me in paths of righteousness for his name's sake. Thanks to the LORD for his guidance while I struggled for the dissertation solution. Without HIM, my life will be meaningless.

I would like to express my deepest appreciation to my mentor Dr. K. Thulasiraman, for his inspiration, support, and encouragement during my graduate study. Especially, I am grateful for his patience and philosophical teaching, "*The goal of research is to find out the truth hidden by God.*" Also, without two coffees a day with him, this work would have taken more time to complete.

I also want to thank Dr. Anindya Das for his suggestions and tennis games while I needed relaxation. I also appreciate the discussions and advice from my committee members, Dr. S. Lakshmivarahan, Dr. Sudarshan Dhall in the School of Computer Science, and Dr. Simin Pulat in the School of Industrial Engineering.

I am thankful to Dr. Le Greuenwald and Dr. Sridhar Radhakrishnan for their excellent advice and guidance during the early period of my study at OU. In addition, I am grateful to many people that provided help and inspiration for this work in the department.

This work would not have been completed without the support of my lovely wife An-Chen Lai and I am deeply grateful for her encouragement. Without her, I would have dropped out of the Ph.D. program. Finally, I would like to thank the LORD for giving me a wonderful family member, my parents and my parents-in-law, and two precious gifts, my lovely children Joyce and Bertram.

Table of Contents

List of Figures

List of Tables

ABSTRACT

Advances in semiconductor technology have made possible design of large computer systems containing hundred of thousands of processing elements. As the complexity and computing power of these systems increase, fault tolerance and reliability have become important areas of concern. Yet, it is impossible to build systems without defects. Testing of such systems becomes extremely difficult due to their large sizes and possible geographical distribution of units. Therefore, it is important for computing systems to have the capability to automatically detect and identify faulty components. In 1967, Preparata, Metze and Chien proposed a model and framework, called system level diagnosis, to deal with this problem

In the two decades following Preparata, Metze and Chien's pioneering work, a number of issues arising from the application of their framework were investigated and resolved. All these works assumed the existence of a single highly reliable supervisory node to do the diagnosis. A single supervisory node is a bottleneck in a system with a large number of processing nodes. Distributed diagnosis algorithms which exploit the inherent parallelism available in a multiprocessor system would be desirable. With this in view, Kuhl and Reddy, in 1981, pioneered the area of distributed system level diagnosis.

Distributed diagnosis has been the focus of research in this thesis. There are two aspects to the contributions in this thesis: Design and performance evaluation of a new distributed diagnosis algorithm, and the design of a distributed network fault detection system based on the SNMP protocol.

In 1991, Bianchini and Buskens proposed an adaptive distributed algorithm to diagnose fully connected networks. This algorithm called the ADSD algorithm has a diagnosis latency of $O(N)$ for a network with N nodes. With a view to improving the

diagnosis latency of the ADSD algorithm, in 1998 Duarte and Nanya proposed a hierarchical distributed diagnosis algorithm for fully connected networks. This algorithm called the Hi-ADSD algorithm has a diagnosis latency of $O(log^2N)$. The Hi-ADSD algorithm can be viewed as a generalization of the ADSD algorithm. In this thesis, we propose a new distributed diagnosis algorithm using the multilevel paradigm. This algorithm is a generalization of the ADSD algorithm. We present all details of the design and implementation of this multilevel adaptive distributed diagnosis algorithm called the ML-ADSD algorithm. We also present extensive simulation results comparing the performance of these three algorithms.

The primary application of our research is to develop and implement a prototype network fault detection/monitoring system by integrating the ML-ADSD algorithm into a SNMP-based (Simple Network Management Protocol) fault management system [RM90] [MR90] [CFSD90]. We report the details of the design and implementation of such a distributed network fault detection system.

SNMP was developed by IETF in 1988 for the purpose of managing the network devices over a computer network and has been widely adopted by industry on network applications. The major drawback of SNMP-based fault management is its centralized nature. The resulting problems include a single point of failure, lack of scalability, and high communication costs around the central manager. Through our application, we demonstrate that some of the above problems can be solved and that the improvement of fault management through distributed fault location is feasible.

Chapter 1

Introduction

Continuing advances in semiconductor technology have made possible the development of large computer systems comprising hundreds of thousands of processors or units. As the complexity and the computing power of these systems increase, fault tolerance and reliability become acute areas of concern. Yet it is impossible to build such systems without defects. As the size of a system grows, it is more likely to develop faults both in the manufacturing process and during the operation period. Testing of such systems becomes extremely difficult due to their large sizes. First, the complexity of test generation for such large systems is overwhelming. Second, the application of test data, and observation and analysis of test responses are extremely difficult and costly, even if test data could be generated. This problem may be further aggravated by possible geographical distribution of units. Testing of such systems with the traditional stimuli-supplying and responses-observing philosophy has become virtually impossible. Therefore, it is important for computing systems to have the capability to automatically detecting and identifying faulty components.

In 1967, Preparata, Metze and Chien [PMC67] proposed a model and a framework, called ***System-Level Diagnosis***, for dealing with the above problem. In the more than three decades following this pioneering work, several issues arising from the application of this framework have been investigated and resolved. Many of these results have profound theoretical and practical implications. Most of the recent research efforts

in system-level diagnosis have focused on enhancing the applicability of system-level diagnosis based approaches to practical scenarios. Specifically, the focus has been on:

1) Probabilistic diagnosis and application to VLSI testing and

2) On-line distributed diagnosis of a network of processors.

The main theme of our research is on-line distributed diagnosis. The primary application of this research is in designing a distributed network fault detection/monitoring system based on the widely used SNMP (Simple Network Management Protocol) protocol.

This chapter is organized as follows. First, in section 1.1 we present a review of literature of most of the fundamental results in system level diagnosis using a central observer. In section 1.2, we review literature on distributed diagnosis. This is followed by a statement of the scope of this thesis.

1.1 System Level Diagnosis: Review of Literature

1.1.1 Models of System Level Diagnosis

In System-Level Diagnosis and the PMC model proposed by Preparata, Metze and Chien [PMC67] for diagnosis of large systems, the units are made to test each other through the interconnects instead of having a centralized tester to test the whole system. The result of such an inter-unit test may be unreliable since the testing unit may be faulty itself. Therefore, the whole set of test outcomes must be analyzed to locate the real faulty units. No postulate is to be made in the course of test outcome analysis either on the status (fault-free or faulty) of any of the units or on the correctness of any of the test

outcomes produced by the testing units. In the following, we will use units and nodes, system and network interchangeably.

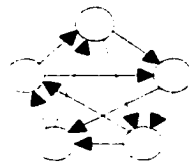


Fig. 1.1 A system with five units

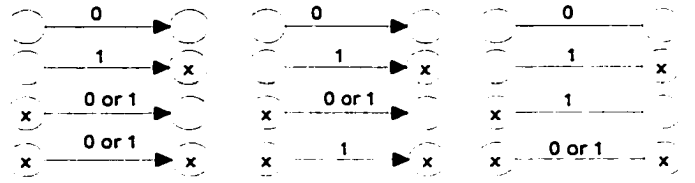


Fig. 1.2 Test outcomes under the PMC, BGM, and Comparison models

Legend: x faulty unit fault-free unit

Figure 1.1 shows an example of inter-node testing, where each node is represented by a vertex and each test by an arc. An arc from vertex u to vertex v means that u tests v. Test outcomes are classified as fault-free or faulty. The set of test outcomes is called the ***syndrome*** of the system. Nodes can test others or can be tested by others. It is assumed that test outcomes produced by fault-free testing nodes are always correct while those produced by faulty testing nodes can be anything (fault-free or faulty), irrespective of the status of the tested nodes. This kind of test outcome interpretation has since been known as the ***PMC model***. The PMC model is described in Figure 1.2(a). The labels on the arcs represent the possible test outcomes. The labels 0 and 1 correspond to the outcomes fault-free and faulty, respectively. Preparata, Metze and Chien also introduced the concept of t-diagnosable systems. A system is said to be ***t-diagnosable*** (or one-step t-diagnosable) if all faulty nodes can be identified from any syndrome produced by the system as long as the number of faulty nodes present does not exceed t. The ***degree of diagnosability*** of a system is the maximum number of faulty nodes that can be diagnosed correctly.

There are three major issues associated with system-level diagnosis: the characterization problem, the diagnosability problem, and the diagnosis problem. The

characterization problem is to find necessary and sufficient conditions to achieve a given degree of diagnosability in terms of test assignment, which specifies who tests whom. The diagnosability problem is to determine the degree of diagnosability (i.e., the largest value of t) for a given test assignment. Finally, the diagnosis problem is to develop an algorithm to identify the fault set from the test outcomes. Hakimi and Amin [HA74] presented the first full characterization of t-diagnosable systems. Sullivan [S84] solved the diagnosability problem giving a polynomial-time algorithm to determine the largest value of t for which a given system is t-diagnosable. Dahbura and Masson [DM84] solved the t-fault diagnosis problem. They presented an $O(n^{2.5})$ diagnosis algorithm for t-diagnosable systems. Other works on t-fault diagnosis on the PMC model include [AKT75] [KTA75] [K78] [NN86] [S88] [SAA89] and [DMY85].

In addition, several variations of the PMC model such as the BGM model (as in Figure 1.2(b)) have been proposed in the literature arising from different considerations of fault types, ways of testing, test invalidation, etc [BGM76] [CH81] [MM81]. Chwa and Hakimi [CH81], and Maeng and Malek [MM81] suggested that the stimuli-supplying and response-observing type testing schemes be replaced by comparison of computed results. This is known as the ***comparison model***. This model is shown in Figure 1.2(c). The outcome, for each pair of nodes whose outputs are compared, is labeled 0 (1), if the outputs agree (disagree). It is assumed that the outputs of a fault-free node and a faulty node always disagree and the outputs of faulty nodes may or may not disagree. For a broad description of the early works in the theory of system-level diagnosis can be found in the survey paper [KH87].

1.1.2 Diagnosis of Large Fault Sets

In multiprocessor systems, such as those implementable in VLSI and Wafer Scale Integration (WSI), the number of nodes - in this context we use nodes and processors interchangeably - in a system can be very large. Moreover, the commonly used interconnection networks such as the rectangular grids and the hypercubes are very symmetrical and sparse. If the testing links are the same as the communication links between the processors, the degree of t-diagnosability of such systems is very small. To address this issue, Somani, Agarwal and Avis [SAA87] proposed a generalized theory of diagnosis providing necessary and sufficient conditions for fault pattern of any size to be diagnosable. Motivated by the need to be able to diagnose large fault sets in sparse systems Das et al. [DTAL93] introduced the concept of ***local diagnosis*** and proposed to place reasonable local constraints to achieve a higher overall diagnosability degree. They also showed that many regular interconnected structures such as the hypercube and the rectangular grid are locally diagnosable. They also presented a simple algorithm for diagnosis of such systems. This algorithm is also amenable for a distributed implementation. In [DTLA93] a distributed diagnosis algorithm for a ring of processors in presented. A more recent work on local diagnosis may be found in [L97]. However, much work remains to be done with regard to the complete characterization of locally diagnosable systems and their diagnosis.

Sequential t-fault diagnosis and t/s-diagnosis allow for more nodes to be faulty in sparsely connected systems at the cost of prolonging diagnosis time or of misidentifying some fault-free nodes. A system is ***sequentially t-diagnosable*** if and only if, given a syndrome, at least one faulty node can be correctly identified, provided that the number of

faulty nodes in the system does not exceed t. A system is ***t/s-diagnosable*** if and only if, given a syndrome, the set of faulty nodes can be isolated to within a set of s nodes, provided that the number of faulty nodes in the system does not exceed t. With the sequential diagnosis approach up to the square root of the number of nodes in the system can be diagnosed on a single loop architecture [PMC67]. Das et al [DTA91] and Raghavan [R89] have given characterizations of t/s-diagnosable systems. Das et al [DTA91] have also given a diagnosis algorithm for t/s-diagnosable systems. Raghavan and Tripathi [RT91] showed that sequential t-diagnosability is Co-NP-Complete for both PMC as well as BGM models. Kavianpour and Friedman [KF78] considered a very interesting special case of t/s-diagnosability, the ***t/t-diagnosability***. They showed that with the same degree of connection the degree of t/t-diagnosability might double the degree of t-diagnosability. An $O(n^{2.5})$ diagnosis algorithm for t/t-diagnosable systems was given by Yang, Masson and Leonetti [YML86]. Das et al. [DTA94] presented an $O(n^{3.5})$ diagnosis algorithm for $t/t+1$-diagnosable systems.

More recently, Somani and Peleg [SP96] introduced a new measure of diagnosability, called ***t/k*-diagnosability**. This is similar to t/s-diagnosability except that there is an upper bound on the number of incorrectly diagnosed nodes regardless of the number of actual faulty nodes in the system. They have analyzed the t/k- diagnosability of hypercubes, star graphs and two dimensional meshes and have demonstrated that for these systems, a substantial increase in the degree of diagnosability is achieved at the cost of a small number of incorrectly diagnosed nodes.

1.1.3 Adaptive System-Level Diagnosis

In adaptive system-level diagnosis schemes, tests are assigned dynamically, instead of assigning all of them at the outset and decoding the test outcomes [HN84]. So, adaptive diagnosis requires fewer tests. In [N81], Nakajima proposed an adaptive diagnosis scheme. Here, a completely connected system is assumed which restricts its applicability. This approach is further studied in [NN86]. Vaidya and Pradhan [VP94] proposed a new adaptive scheme called ***safe system-level diagnosis***. The safe-diagnosis approach ensures that up to t faulty nodes can be located and up to u faulty nodes, where $u > t$, can be detected. In this approach, a minimal amount of fault location capability is sacrificed to attain a large degree of fault detection capability. More recently, Fang, Bhuyan and Lombardi [FBL96] proposed an adaptive diagnosis algorithm for hypercube systems. The diagnostic cost (measured in terms of the number of test links and diagnosis time) is very low for this scheme.

1.1.4 Probabilistic Diagnosis

Probabilistic diagnosis is yet another approach to allow diagnosis of large fault sets. The emphasis here is to identify all faulty nodes with a very high probability. This approach was initiated by Maheswari and Hakimi [MH76]. Dahbura, Sabnani and King [DSK87] considered probabilistic diagnosis with comparison testing. Scheinerman [S87] gave a probabilistic diagnosis algorithm which correctly identifies every node as n tends to infinity, as long as each node compares with slightly more than $log\ n$ nodes. Blough [B88] showed that correct diagnosis with high probability was impossible if each node was tested by only $O(logn)$ other nodes. Further results were presented by Blough,

Sullivan, and Masson [BSM92]. Fussell and Rangarajan [FR89] considered performing multiple tests to achieve correct diagnosis of constant degree connection structures. Slightly more than *log n* tests are performed with respect to each test link. They showed that the probability of correctly identifying every node approaches one as n tends to infinity. They further showed [RF92] that the number of test links per node and the number of tests per test link can be traded off as long as the product of these two parameters grows as $O(logn)$ as n tends to infinity. Laforge et al [LHA94] presented another approach to diagnosing constant degree systems. An extensive review of probabilistic diagnosis results may be found in Lee and Shin [LS94]. Applications of probabilistic approaches to VLSI testing may be found in [FR89], [RFM90], and [HAT98].

1.1.5 Algorithm-Based Fault Tolerance

In several computation intensive applications (such as signal processing), multiprocessor architectures are commonly used. To improve the reliability of such systems, it is desirable to provide them with concurrent error detection capability. Algorithm-based fault tolerance (ABFT) is one such technique [HA84]. There has been an extensive literature on the design and analysis of algorithm-based fault tolerant systems. In a pioneering work, Banerjee and Abraham [BA86a] proposed a graph-theoretic model to represent ABFT systems. They also showed in [BA86b] how the ABFT approach can be used for fault diagnosis in multiprocessor systems. Some recent works in this area are [BP94] [YJ97].

1.2 Distributed System-Level Diagnosis

Most diagnosis algorithms based on the PMC model are assumed to be executed on a single highly reliable supervisory node. A single supervisory node is a bottleneck in a system with a large number of processing nodes. Distributed diagnosis algorithms which exploit the inherent parallelism available in a multiprocessor system would be desirable. A detailed review of research results in distributed system-level diagnosis are presented next. The approaches reviewed use one of two fault models: the ***Byzantine failure*** model and the ***stopping failure*** model [J94] [L96]. In the case of a stopping failure, a node ceases to function without warning. Stopping failures are intended to model unpredictable node crashes. In the case of a Byzantine failure, a node may exhibit completely unconstrained behavior. Byzantine failures are intended to model any arbitrary node malfunction, including, for instance, failures of individual subcomponents.

1.2.1 On-line Distributed System-Level Diagnosis: SELF and Related Algorithms

Distributed system-level diagnosis was first considered in the early works by Kuhl, Reddy and Hosseini [KR80][KR81][HKR84] in which each fault-free node in a distributed system reliably receives test results through its neighbors to perform diagnosis. In this work the Byzantine failure model was used. It was assumed that the total number of faulty nodes is restricted to t or fewer nodes, and that the test assignment graph is fixed, i.e. each node tests a fixed set of neighboring nodes. In the SELF distributed algorithm [KR81] fault-free nodes forward test results to neighboring nodes

which are then propagated to other nodes. No assumption is made regarding faulty nodes which can propagate erroneous test results. Each node collects the test information and independently determines the status of all the nodes in the system. In the NEW_SELF distributed algorithm [HKR84] the key idea is that a fault-free node accepts test information from one of its neighbors only if it has tested that neighbor and determined it to be fault-free. This ensures that test result reports are propagated reliably along fault-free testing chains. For correct diagnosis, the NEW_SELF algorithm requires that every fault-free node receives all the tests results of every fault-free node in the system. This condition is satisfied if every node in the system is tested by $t+1$ other nodes. These algorithms allow both link and node failures.

1.2.2 Event-Driven Technique for Distributed System-Level Diagnosis

In 1990, Biancini et al [BGN90] proposed an event-driven technique to adapt Kuhl and Reddy's approach for an Ethernet-based network of workstations. To reduce the communication overhead required by Kuhl and Reddy's approach, they used an event-driven technique wherein only when a node is first detected as faulty or when a newly repaired node rejoins the network is the new information forwarded in the system. Test results are forwarded by a node only if it differs from the information stored at the node. The test assignment graph is such that each network node tests $t+1$ of its next logical neighbors, where t is the maximum number of faulty nodes that can be tolerated. This strategy significantly reduces the number of messages required to arrive at a diagnosis for

systems where the test assignment given above can be applied. These works allow both link and node failures. They also permit repairs during the execution of the algorithm.

1.2.3 Adaptive Distributed System-Level Diagnosis

A further refinement of the approach of Biancini et al was to replace single-step diagnosis by an adaptive strategy wherein the test assignment, instead of being fixed, is determined by the fault situation [BB91] [BB92]. This adaptive distributed system-level diagnosis approach also removes the bound on the number of faulty nodes in the system. This results in a sparse test assignment topology, a logical ring of fault-free nodes in a connected network. On occurrence of a fault, the information is forwarded in the network and the fault-free nodes rearrange the test assignment topology to preserve the ring structure. More recently Duarte and Nanya [DN98] proposed a hierarchical adaptive distributed diagnosis algorithm for fully connected networks. This algorithm has better diagnosis latency than Bianchini and Buskens' algorithm. More detailed descriptions of these two algorithms will be given in chapter 2.

1.2.4 Adaptive Distributed System-Level Diagnosis for Arbitrary Networks

In [BSB92][SBB92] the adaptive strategy given in [BB92] was extended to arbitrary networks. In [BSB92] the underlying test assignment topology is strongly connected among all the fault-free nodes. On the occurrence of an event, search and destroy phases are added to modify the test assignment topology so that the strong connectivity requirement is maintained. In [SBB92], the test assignment topology is a tree

wherein each node in the tree is tested by its designated parent and the root is tested by one of its children. On occurrence of an event, a new tree rooted at the node that detects the event is created to become the new test assignment topology. The path taken by the forwarding of information determines the new tree.

In [RDZ95], Rangarajan, Dahbura and Ziegler presented a distributed diagnosis algorithm for an arbitrary network in which each fault-free node ensures that exactly one fault-free neighbor - if it exists - is testing it. Nodes perform their tests periodically and if a failure event is detected then the information is propagated using validating transactions. The fault model for nodes considered in this case is the stopping failure model where a node simply ceases to operate without alerting other nodes and a bounded delay is assumed for communicating links. This work allows node failures and repairs to occur during the execution of the algorithm.

1.2.5 Gossiping and Consensus in a Distributed Environment

In [BH94] Bagchi and Hakimi presented a distributed algorithm for the gossiping problem in a faulty environment and demonstrated its application in distributed system level diagnosis. They assumed the Byzantine failure model and used a tree testing topology. The system is required to be t-diagnosable if t faults are to be permitted. In this work link failures are not considered. Also it is assumed that no processor can become faulty and that no processor is repaired during the execution of the algorithm. Bagchi and Hakimi pointed to "a growing overlap" between the field of fault diagnosis and the field of consensus in distributed systems. Barborak, Malek and Dahbura [BMD93] described results of interest in these fields.

1.2.6 Broadcast Model for Distributed Diagnosis

In a recent work Blough and Brown [BB99] proposed a new comparison-based model for distributed diagnosis. This model is a combination of distributed diagnosis and the generalized comparison model of Sengupta and Dahbura [SD92]. In the broadcast comparison model, a distributed diagnosis procedure is used, which is based on comparisons of redundant task outputs and has access to a weak broadcast protocol.

In this model, a task is assigned to a pair of distinct processors with the same input. These two processors perform this task and broadcast their outputs to all processors in the system. Every fault free processor in the system compares the two outputs received. Note that comparisons are made on every fault free processor including the processors being compared. Once a processor produces a sufficient number of comparison outcomes, it executes a diagnosis algorithm to determine the status of all processors in the system. In other words, the diagnosis algorithm is executed in a distributed fashion. This algorithm has been implemented in the COSMOS operating system. The authors have produced simulation results to show that the algorithm diagnoses all fault situations with low latency and very little overhead.

1.3 Scope of the Thesis

There are two aspects to this thesis. In chapter 2, we first review two distributed diagnosis algorithms due to Bianchini and Buskens [BB91] and Duarte and Nanya [DN98]. Both these algorithms are designed to diagnose fully connected networks. The ADSD algorithm of Bianchini and Buskens has a diagnosis latency of $O(N)$ for a network

with N nodes. The Hi-ADSD algorithm of Duarte and Nanya may be a viewed as a generalization of the ADSD algorithm, and has diagnosis latency of $O(log^2N)$. After a review of these two algorithms, we propose a new algorithm based on the multilevel paradigm. The multilevel paradigm can be applied to both the ADSD and the Hi-ADSD algorithms. We establish the diagnosis latency of this algorithm to be called the ML-ADSD algorithm and present extensive simulation results comparing the diagnosis latencies of these three algorithms.

The primary application of our research is to develop and implement a prototype network fault detection/monitoring system by integrating the ML-ADSD algorithm into a SNMP-based (Simple Network Management Protocol) fault management system [RM90] [MR91] [CFSD90]. Design and implementation of such a distributed network fault detection system is discussed in chapter 3. SNMP was developed by IETF in 1988 for the purpose of managing the network devices over a computer network and has been widely adopted by industry on network applications. The major drawback of SNMP-based fault management is its centralized nature. The resulting problems include a single point of failure, lack of scalability, and high communication costs around the central manager. Through our application, we demonstrate that some of the above problems can be solved to some extend and that the improvement of fault management through distributed fault location is feasible. Chapter 4 concludes the thesis with a summary of research and suggestions for future work.

Chapter 2

Multi-Level Adaptive Distributed Diagnosis for Fault Detection in a Network of Processors

In this chapter we present and discuss a multilevel adaptive distributed diagnosis algorithm for fully connected networks. This is a generalization of Bianchini and Buskens' [BB92] ADSD algorithm. The multilevel paradigm also helps design an algorithm with diagnosis latency smaller than that of the ADSD. As mentioned in Chapter 1, this was also the motivation of Duarte and Nanya's [DN98] hierarchical distributed diagnosis algorithm.

We first present in sections 2.1 and 2.2 the essential features of the diagnosis algorithms of Bianchini and Buskens, and Duarte and Nanya. Several concepts and ideas used in these algorithms are also relevant to the design of our new algorithm to be discussed in the following section. We present the new multilevel diagnosis algorithm and several aspects of this algorithm in section 2.3. In section 2.4, we present simulation results comparing these three algorithms.

A preliminary version of the work in this chapter has been reported in [STD01].

2.1 The ADSD Algorithm

The adaptive-distributed system-level diagnosis algorithm (to be called ADSD) proposed by Bianchini and Buskens [BB91][BB92] assumes the existence of a logical fully connected network and does not permit link failures. It is distributed, adaptive on testing set, and imposes no limit on the number of faulty nodes. It is assumed that there

are no links failures. The PMC fault model [PMC67] is used. Also, during the testing process a node cannot fail and recover from that failure during the time between two tests by another node.

The ADSD algorithm is the first practical application of system level diagnosis theory and has been implemented to run on an Ethernet network of over 200 workstations at the Carnegie Mellon University.

Before we discuss the specification of the algorithm, we have to clarify the concepts of "*test*" and "*testing round*" and "*diagnosis latency*" used in distributed system-level diagnosis literature.

A "*test*" could be just simply a node i sending a message to node j to ask for some information. If the response is proper and on-time, then node i evaluates node j as fault-free. Otherwise, node j is faulty.

The concept of testing round plays a very important role in expressing the diagnosis latency (or capturing the time complexity) of a distributed diagnosis algorithm.

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 [BB91]. In other words, the duration of a "*testing round*" includes the time taken by a node i to find a fault-free node j or evaluate all the nodes as faulty. For example, assume a node i at time t_1 starts its sixth test execution, and finds nodes i+1, i+2 as faulty and i+3 as fault-free at time t_4. At this time, node i stops testing. On the other hand, node i+3 at time t_2 starts its sixth test execution and finds node i+4 as fault-free at time t_3, and then stops testing. Although the times and the number of tests for node i and node i+3 to find a fault-free node are

different, we still say that nodes i and i+3 performed their tests in the same testing round, that is, sixth testing round.

The "*Diagnosis latency*" is defined as the time from the detection of a fault event to the time when all the fault-free nodes correctly diagnose the event. In the following our interest is in diagnosis latency after the last fault event has occurred.

Algorithm Specification:

In the ADSD algorithm, a node i uses an array called TESTED_UP$_i$ to update the testing results and to respond to the request of its tester. An example of the data structure for node 2 in an eight node system is shown in Figure 2.1.

The TESTED_UP$_i$ array contains N entries, and the array indices and the values of the entries are node identifiers. For example, entry TESTED_UP$_i[u] = v$, means that node i has received a diagnostic message from a neighbor node (which it has tested as fault-free) indicating node u has tested node v and found node v fault-free. Also an entry of TESTED_UP$_i[i] = u$ means that node i itself has tested node u as fault-free. If the value of an entry is "x", it means that the entry is arbitrary. Figure 2.1 shows the values kept at node 2 for an eight node network with nodes 1, 3, and 4 faulty.

TESTED_UP$_2[0] = 2$
TESTED_UP$_2[1] = x$
TESTED_UP$_2[2] = 5$
TESTED_UP$_2[3] = x$
TESTED_UP$_2[4] = x$
TESTED_UP$_2[5] = 6$
TESTED_UP$_2[6] = 7$
TESTED_UP$_2[7] = 0$

Fig. 2.1 The TESTED_UP information stored in node 2

A special property of this array is that in one testing round after the last fault event has occurred a "*fault-free ring*" will be formed if we start from a fault-free node i and connect the fault-free paths from node i to other fault-free nodes. Using the above as an example, if we start from node 2, then the fault-free tests are 2 to 5, 5 to 6, 6 to 7, 7 to 0, and 0 to 2. By viewing these fault-free tests as paths and connecting them together, we will have a "*fault-free ring*" (e.g., 2→5→6→7→0→2). This property plays an important role in the proof of correctness of the ADSD algorithm as well as the algorithm we shall propose in the following section.

During the process of diagnosis, each node, in a testing round, executes the ADSD algorithm to completion and resumes the testing after a predefined interval. Each node tries to find a fault-free node and uses the information from that fault-free node to update its local diagnosis information in the TESTED_UP array. In at most N testing rounds after the last fault event, each fault-free node will have consistent diagnosis of the fault status of the nodes in the network, thereby resulting in a diagnosis latency of $O(N)$ testing rounds.

Before the execution of the algorithm, all the nodes are ordered sequentially in a list, as $(n_0, n_1, ..., n_{N-1})$. Thus, a node i will test nodes $i+1$, $i+2$, ..., etc., sequentially until a fault-free node is found, and then acquires the diagnosis information from that node. Since all the additions are modulo N, we will find that once we connect the testing paths of all the fault-free nodes, the testing paths will form a ring. Therefore, in each testing round, a node i will perform a test in a "*forward*" manner from node i to node $i+1$ as in Figure 2.2(a), but will get the diagnosis information in a "*backward*" manner from node $i+1$ to node i as in Figure 2.2(b). It takes one testing round for the diagnostic information,

TESTED_UP$_{i+1}$, to be propagated between fault-free nodes i+1 and i. Likewise, it will take two testing rounds for fault-free node i-1 to get TESTED_UP$_{i+1}$ from node i. At the end of at most N testing rounds, all the fault-free nodes will have the same fault status information of all the nodes in the network. Based on the information in TESTED_UP, an algorithm called "*Diagnose*" is used to determine all the fault-free nodes in the network.

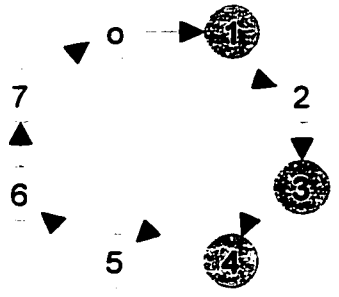


Fig. 2.2(a) An adaptive testing topology for an eight node network with nodes 1, 3, 4 faulty

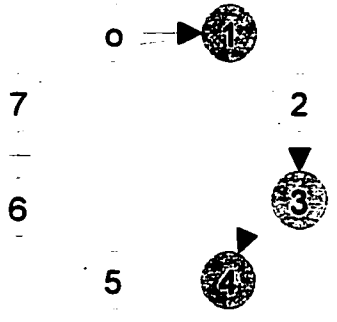


Fig. 2.2(b) Information propagation along fault-free nodes

Summarizing, an informal description of the ADSD algorithm and the *Diagnose* algorithm are given Figures 2.3 and 2.4 respectively.

ADSD algorithm (informal)

- List the nodes in sequential order, as ($n_0, n_1, \ldots, n_{N-1}$).

Testing round for node n_x

1. Node n_x identifies the next sequential fault-free node in the list
   - sequentially **testing** consecutive nodes $n_{x+1 \bmod n}$, $n_{x+2 \bmod n}$, ..., etc.,
   - until a fault-free node is found.
2. Diagnostic information received from the tested fault-free node is utilized to **update** local information.

- Repeat steps 1 and 2 in subsequent testing rounds.

Fig. 2.3 ADSD algorithm

/* **Diagnose algorithm***/
/* The following is executed at each n_x, $0 \leq x < N$ when n_x desires diagnosis of the systems. */

1. **For** $i = 0$ **to** $N - 1$
1.1. $\text{STATE}_x[i]$ = faulty;

2. node_pointer = x;

3. **repeat** {
3.1. STATE_x[node_pointer] = fault-free;
3.2. node_pointer = TESTED_UP_x[node_pointer]
3.3.} **until** (node_pointer == x);

Fig. 2.4 Diagnose algorithm

2.2 Hierarchical Adaptive Distributed System-Level Diagnosis

The hierarchical adaptive distributed system-level diagnosis (Hi-ADSD) algorithm of Duarte and Nanya [DN98] uses a divide-and-conquer testing strategy to reduce the diagnosis latency. For testing, Hi-ADSD divides nodes into clusters of various sizes (from small to large), and then collects the diagnosis information in each cluster to accomplish the diagnosis of the system.

Hi-ADSD has a diagnosis latency of at most $O(log^2N)$ testing rounds where N is the number of nodes in the network. It is both adaptive and hierarchical on the testing nodes, distributed on execution, and imposes no limit on the number of faulty nodes. It is also claimed that it has less communication overhead in terms of packet size. Additionally, it has been integrated with the simple network management protocol (SNMP) [RM90] [CFSD90] [MR91] and applied in a 37-node Ethernet network platform.

Algorithm Specification:

Consider a network (system) S of N nodes with each node in one of two states, faulty or fault-free. The system is assumed to be a logically complete network. It is assumed that there are no links failures. Again the PMC fault model [PMC67] is used. Also, during the testing process a node cannot fail and recover from that failure during the time between two tests by another node, and that the time could be as long as $logN$ testing rounds in the worst case.

Firstly, the Hi-ADSD algorithm at each node divides the remaining nodes into various sizes of clusters for testing, and the size of the cluster tested will vary at different

testing rounds. In the following discussion, the network size N and all the cluster sizes are assumed to be a power of 2. In general, a cluster of m nodes will contain nodes u_i, ..., u_{i+m-1} with i MOD $m = 0$, and m is a power of 2. If $m = 1$, then the cluster has only one node. If $m = 2$, then the cluster is the union of two smaller clusters, one containing nodes, u_i, ..., $u_{i+m/2-1}$ and the other containing nodes $u_{i+m/2}$, ..., u_{i+m-1}. As an example, an eight node network with clusters of different sizes is illustrated in Figure 2.5. Duarte and Nanya have developed a formula to identify the clusters with respect to each node. This helps reduce the complexity of the algorithm.

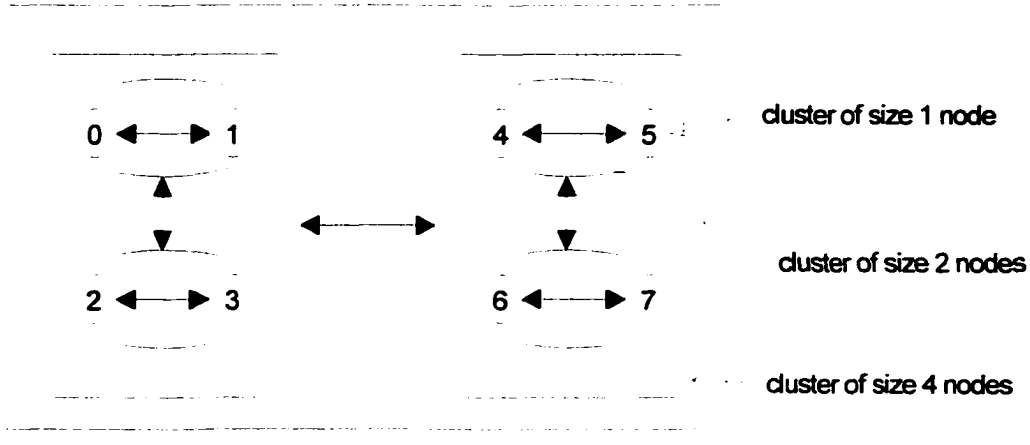


Fig. 2.5 An eight node network with clusters of different sizes

Next, during the execution of the algorithm, in any testing round, each node starts by testing a cluster of size one, then a cluster of size two, and so on, ..., up to a cluster of size of 2^{logN-1} or $N/2$ nodes, and then repeats this testing process. Accordingly, after the continued execution of the Hi-ADSD in at most log^2N testing rounds after the last fault event, each fault-free node will have the same fault status information of the system.

Basically, in one testing round, a node i will sequentially test the nodes in cluster $C_{i,s}$. Function $C_{i,s}$ is used to generate the list of nodes in a cluster as shown below.

$$C_{i,s} = \{n_t \mid t = (i \text{ MOD } 2^s + 2^{s-1} + j) \text{ MOD } 2^{s-1+a} + (i \text{ DIV } 2^s) * 2^s + b * 2^{s-1}; j = 0, 1, \ldots, 2^{s-1}-1\},$$

where

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

For example, the table generated by function $C_{i,s}$ which contains the lists of nodes in various clusters to test for an eight nodes network is shown in Table 2.1.

Table 2.1
Table generated by function $C_{i,s}$ which contains the lists of nodes in various clusters to test for an eight nodes network is shown below.

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

During execution of the Hi-ADSD, if node i finds a fault-free node j in the cluster, then node i will stop testing and copy the diagnosis information regarding the nodes in $C_{i,s}$ from node j. If node i can not find a fault-free node in $C_{i,s}$, then node i will continue to test the next cluster $C_{i,s+1}$. This testing process at cluster s+1 will stop when node i finds a fault-free node in some cluster or all the nodes in all the clusters are tested as faulty. In the next testing round, node i will start testing the cluster next to the one where it stopped in the previous testing round. However, regardless of the number of clusters that node i has to test in order to find a fault-free node or find the rest of the nodes as faulty, we define the time interval node i spends in doing this as belonging to one testing round.

Summarizing, an informal presentation of the Hi-ADSD algorithm is given Figure 2.6 along with an illustration in Figure 2.7.

Hi-ADSD algorithm (informal)

- Divide nodes into *logN* clusters of various sizes, (from small to large), with cluster index $s = 0, 1, \ldots, logN-1$.
- Cluster s will contain an ordered list of size 2^s nodes. Initially, set $s = 0$.

Each testing round for node n_x

1. Node n_x identifies a fault-free node in cluster s
   - sequentially testing list nodes in cluster s
   - until a fault-free node is found or all the nodes in cluster s are faulty.
2. If a tested fault-free node is found, then
   - copy diagnostic information of all nodes in cluster s from this fault-free node.
   - Set $s = (s + 1)$ modulo *logN*

   Else goto Step 1 with $s = (s + 1)$ modulo *logN*

Fig. 2.6 Hi-ADSD algorithm

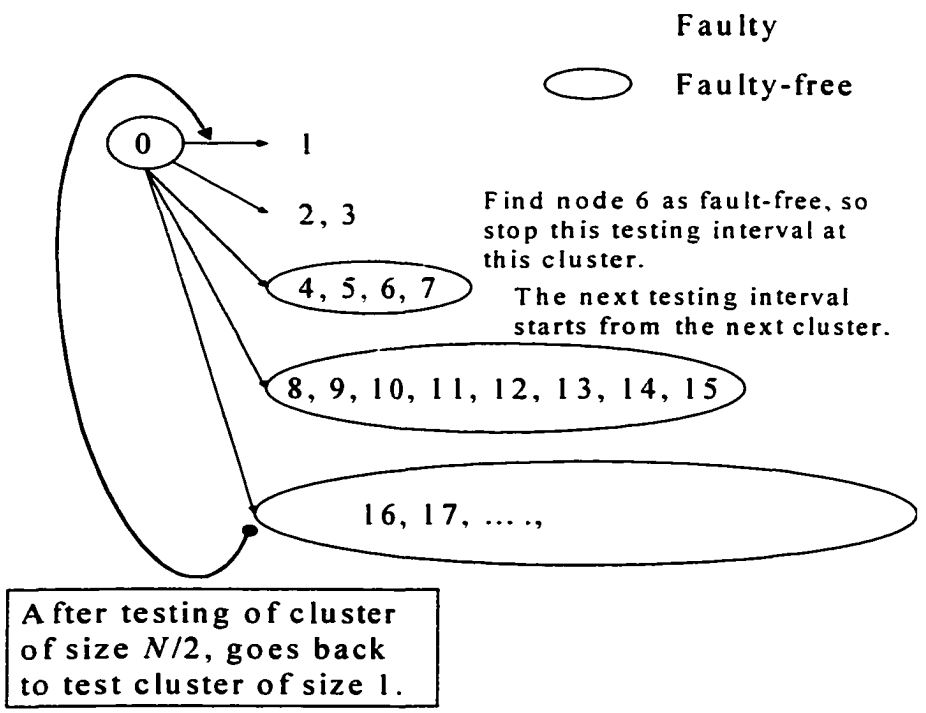


Fig. 2.7 Illustration of Hi-ADSD algorithm

We conclude this section by pointing out that the ADSD algorithm can be viewed as a special case of the Hi-ADSD algorithm.

In the Hi-ADSD algorithm each node i views the remaining nodes as partitioned into $\log N$-1 clusters of varying sizes. In the first testing round it starts testing a cluster of size one, then a cluster of size 2, and so on until it finds a cluster with a fault free node. In the subsequent testing round, node i starts testing with the cluster next to the one where it stopped in the previous testing round.

On the other hand, node i in the ADSD algorithm views the remaining nodes as one single cluster of N-1 nodes: i+1, i+2,..., i-1. So, in every testing round node i tests its only cluster starting each time testing node i+1.

Thus the ADSD algorithm can be viewed as a 1-level hierarchical algorithm whereas the Hi-ADSD algorithm is a $\log N$-1 level hierarchical algorithm.

2.3 ML-ADSD: Multi-Level Adaptive Distributed System-Level Diagnosis

In this section we propose the design of a multi-level adaptive distributed diagnosis algorithm for fully connected networks, which generalizes the ADSD algorithm of Bianchini and Buskens. This algorithm, to be called the **ML-ADSD algorithm**, has been motivated by the need to reduce the diagnosis latency of the ADSD algorithm as well as the message transmission overhead. The main features of the ML-ADSD algorithm are: multi-level divide-and-conquer partition strategy; adaptive on the next testing assignment; distributed on execution; no upper bound on the number of faulty

nodes; autonomous leader election; easy control on synchronization; and less message transmission overhead.

For the sake of simplicity in explanation, the following discussion of the ML-ADSD algorithm assumes a two-level scheme. The details of the algorithm for more than two levels will be presented in the following section.

2.3.1 Level-1 Clusters

Again, we assume a logically complete network G of N nodes, $n_0, n_1, \ldots, n_{N-1}$. The nodes are first partitioned into p clusters of equal size. Thus each cluster has N/p nodes. For reasons which will become clear later we shall call these clusters as **level-1 clusters**. To make the discussion and the algorithm simpler to present, we assume that both N and p are powers of two. Also it is assumed that each node is able to correctly test and determine the state of other nodes based on the PMC fault model. Link failures are not permitted. An example of a Bus/Ethernet Network of eight nodes and its logical fully connected network is shown in Figures 2.8(a) and (b). Figures 2.9(a) and (b) also shows the re-mapping of old node ids to new node ids and the partition of a network of eight nodes into four clusters.

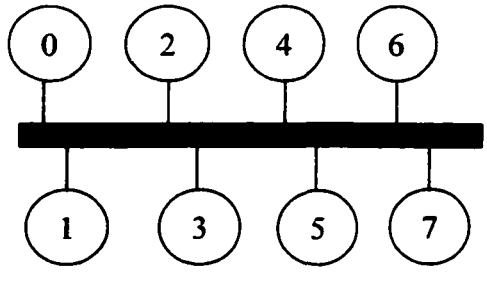


Fig. 2.8(a) An eight node Ethernet network

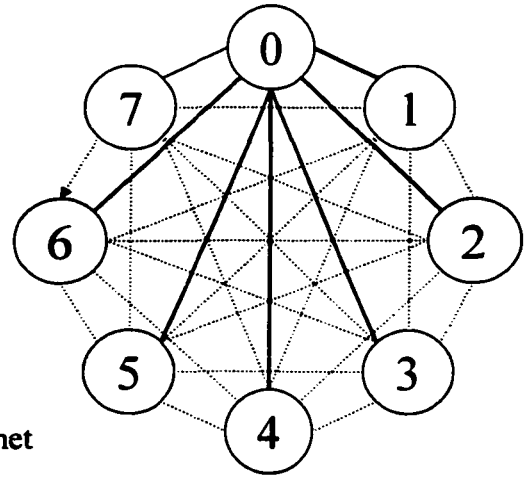


Fig. 2.8(b) A logical fully connected network

k	0	1	2	3	4	5	6	7
i	0	1	0	1	0	1	0	1
j	0	0	1	1	2	2	3	3

n_k = old node id

n_k => $n_{i,j}$, i: new node id,
j: cluster id

$i = k$ MOD (N/p),

$j = k$ DIV (N/p)

e.g., n_2 => $n_{0,1}$, n_5 => $n_{1,2}$ where $N = 8, p = 4$.

Fig. 2.9(a) Node IDs re-mapping table

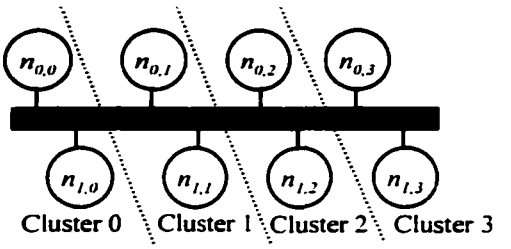


Fig. 2.9(b) Re-map and partition of the nodes into four clusters

2.3.2 Informal Description of the Two-Level Diagnosis Algorithm

Consider Figure 2.10 which gives a pictorial view of our two-level algorithm. In this tree description of the algorithm we have the p original clusters at level 1. Whenever possible, we associate a leader node with each cluster. The fault free node with the smallest id in a cluster is selected as the **leader** of that cluster. If a node is not a leader node, then it is called a **regular node**. Note that if all the nodes in a level-1 cluster are faulty, then this cluster has no leader. At level 2 there is a single **level-2 cluster** consisting of the leaders from the different level-1 clusters. Thus the size of the level-2 cluster is at most p nodes.

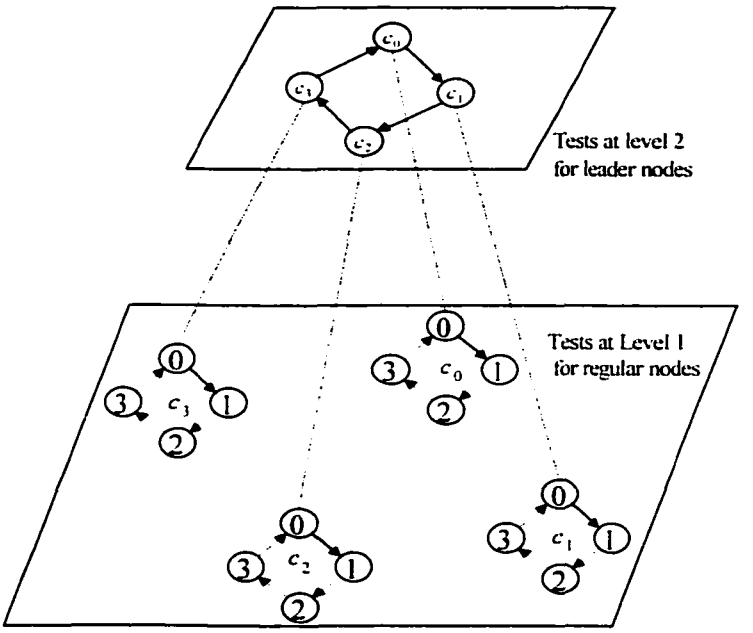


Fig. 2.10 A two-level scheme

Initially, all nodes are regular nodes. Each node begins the diagnosis process by testing, in its first testing round, only the nodes in its cluster. This testing round is called a **level-1 testing round**. The actions taken during this testing round are almost the same as in the testing round of the ADSD algorithm. Specifically, during a level-1 testing round each fault free node identifies a unique fault free node, if it exists, and updates the local diagnosis information. Moreover, in at most N/p testing rounds after the occurrence of the last fault event these fault free nodes will form a cycle and a node will be able to detect all the nodes in this cycle. A subroutine "***Cluster-leader***" (in Figure 2.12) to be discussed later is used to detect this cycle. If a node finds itself to have the smallest id among the ids of all the nodes in this cycle, it elects itself as leader of its cluster. After completing

the testing of nodes in a testing round, the node changes its status to "leader", if necessary, and ends the testing round.

On the other hand, a leader node executes a **level-2 testing round** to be defined in the following and also updates its status to "regular", if necessary.

Note that in a level-1 testing round a node $n_{i,j}$ will try to find a fault-free node sequentially in cluster j and uses the diagnosis information from that fault-free node to update its local diagnosis information. On the other hand, in a level-2 testing round a leader node $n_{i,j}$ will try to find a fault-free node starting from $n_{0,j+1}$ in cluster j+1 (modulo p). The leader will then use the diagnosis information from that fault-free leader node in cluster j+1 to update the status of the nodes in clusters other than its own. If all the nodes in cluster j+1 are found faulty, the leader node in cluster j will continue to look for a fault-free node starting from $n_{0,j+2}$ in cluster j+2, ..., and so on, if necessary. As in the case of level-1 testing rounds, in each level-2 testing round each leader identifies a unique leader from another cluster and these leaders will form a cycle. But a leader will be able to detect the nodes in this cycle only after performing at most p level-2 testing rounds. Also, during a level-2 testing round, in some cases, a leader node may be required to do a level-1 testing if it continues to be the fault free node with the smallest id in its cluster and take appropriate actions, if necessary.

We next present the details of the data structures and algorithms used in implementing the above diagnosis scheme.

2.3.3 Data Structures

In the ML-ADSD algorithm, a node $n_{i,j}$ uses an array called TESTED_UP$_{i,j}$ to update the testing results. TESTED_UP$_{i,j}$ is a two-dimensional array of size (N/p x p)

where N is the number of nodes of the network and p is the number of clusters defined by the user. Also the row index i represents the re-mapped node id and the column index j represents the cluster id. Additionally, the values in the array are the re-mapped node identifiers. In addition, entry $TESTED\_UP_{i,j}[u][k] = v$ at node $n_{i,j}$ means that node i in cluster j has received a diagnosis message from a neighbor node (which it has tested as fault-free) indicating node u in cluster k has tested node v in cluster k and found node v as fault-free Also, an entry $TESTED\_UP_{i,j}[i][j] = u$ means that node i itself has tested node u and found node u as fault-free in cluster j. If the value of an entry is "x", it means that the entry is arbitrary. Figure 2.11 shows that the values kept at node $n_{0,1}$ for an eight node network of four clusters with nodes $n_{1,1}$, and $n_{0,2}$ (where the old node ids are n_3 and n_4) faulty.

$TESTED\_UP_{0,1}$ [0][0] = 1
$TESTED\_UP_{0,1}$ [1][0] = 0
$TESTED\_UP_{0,1}$ [0][1] = 0
$TESTED\_UP_{0,1}$ [1][1] = x
$TESTED\_UP_{0,1}$ [0][2] = x
$TESTED\_UP_{0,1}$ [1][2] = 1
$TESTED\_UP_{0,1}$ [0][3] = 1
$TESTED\_UP_{0,1}$ [1][3] = 0

Fig. 2.11 Values kept at node $n_{0,1}$

2.3.4 Cluster Leader Election

The subroutine "*Cluster_leader*" in Figure 2.12 identifies the smallest node id along a fault-free ring (cycle) in a cluster as the leader in that cluster. Recall that one of the special properties of the one-dimensional $TESTED\_UP_i$ array used in the ADSD algorithm is that a fault-free ring will be formed in at most N testing rounds after the last

fault event if there exists at least one fault-free node in the network. A fault-free ring can be formed at a node i by connecting all the fault-free tests in the TESTED_UP$_i$ array starting from the value in TESTED_UP$_i[i]$. To find out a leader, a node i just goes along the members of the fault-free ring and compares its own node id with other members' node ids. If node i has the smallest node id among the members in the ring, then node i is the leader in the fault-free ring. Otherwise, node i is not the leader. If a node is not in a ring, then it is not a leader node.

```
/* Cluster_leader election subroutine at node n_ij */

1. MARK n_ij = True; //Mark node n_ij visited
2. next_node = TESTED_UP_ij[i][j]; //get the next tested node id

3. While (next_node ≠ 'x') do
4.    If (has_marked(next_node) == True)
5.        If (next_node == i)
6.            stop, i is the leader;    //cycle back to myself
7.        Else          // cycle detected but i is not in the cycle
8.            stop, i is not the leader;
9.    Else
10.       If (next_node < i)
10.           stop, i is not the leader;  //someone's id smaller than mine
11.       Else
12.           MARK next_node = True;
13.           next_node = TESTED_UP_ij[next_node][j];
14. End_While

15. Stop;// i is not leader or the fault-free ring has not formed yet
```

Fig. 2.12 Cluster leader election subroutine

Similarly, we will apply the same fault-free ring concept in the two-dimensional TESTED_UP$_{i,j}$ array in this section by viewing each cluster's values in TESTED_UP$_{i,j}$ as one smaller one-dimensional TESTED_UP$_i$. To find out the cluster leadership in cluster j,

a node $n_{i,j}$ can start from the node id value in TESTED_UP$_{i,j}$[i][j] and go along the fault-free ring to compare the rest of node ids in TESTED_UP$_{i,j}$. The pseudo code for the routine "*Cluster_leader*" is shown in Figure 2.12.

2.3.5 Cycle Detection at Level 2

To detect the cycle at level 2 we use a data structure similar to the TESTED_UP array. With each node $n_{i,j}$, we associate a one-dimensional leader array LEADER$_{i,j}$.

The LEADER$_{i,j}$ array contains p entries. The array indices represent the cluster ids and the values of the entries are node identifiers. For example, entry LEADER$_{i,j}[u] = n_{x,y}$, means that node $n_{i,j}$ has received an information from a neighbor node (which it has tested as fault-free) indicating a node in cluster u has tested node $n_{x,y}$, in cluster y and found node it fault-free. Also an entry of LEADER$_{i,j}[j] = n_{x,y}$ means that the leader node itself in cluster j has tested node $n_{x,y}$, in cluster y as fault-free. If the value of an entry is "x", it means that the entry is arbitrary.

As in the case of cluster leader election, the LEADER$_{i,j}$ array can be used to determine if node $n_{i,j}$, is in a cycle at level 2. The subroutine for cycle detection is given in Figure 2.13. Once a node detects that it is in a cycle its status is changed to "regular".

If a faulty node at level 2 recovers it may detect a fault free cycle at level 2, but may not find itself in the cycle. In this case also we change the status of the node to "regular". Certain actions need to be taken if the node does not detect a cycle. See the algorithm description given below.

Now we are ready to present our two-level adaptive distributed diagnosis algorithm.

```
/* Cycle Detection subroutine at node n_{i,j}*/

1.  MARK j = True;  //Mark cluster id j Visited
2.  n_{a,b} = LEADER_{i,j}[j];  //get the next tested node id

3.  While n_{a,b} ≠ 'x' do
4.    If ( has_marked(b) == True )  // Has cluster b been visited?
5.        If ( a == i ) //cycle back to myself
6.            Stop, n_{i,j} is in a cycle;
7.        Else
8.            Stop, n_{i,j} detects a cycle but
                it is not in the cycle;
9.    Else
10.       MARK b = True;  // Mark cluster id b visited
11.       n_{a,b} = LEADER_{i,j}[b];
12. End_While

13. Stop; // n_{i,j} does no detect a cycle
```

Fig. 2.13 Cycle detection subroutine

2.3.6 Two-Level Algorithm Description

2-Level ADSD Algorithm

- Initially, all the nodes are regular nodes.

During a testing round each node v performs the following:

CASE 1: If v is a regular node, then execute the "Level-1 Testing Algorithm" of Figure 2.14, which includes leader election.

- Change the status of v to "leader", if it becomes a leader.
- End testing round.

CASE 2: If v is a leader, then execute the "Level-2 Testing Algorithm" of Figure 2.15.

- Execute the Cycle detection algorithm of Figure 2.13.
- If v detects a cycle at level 2, then
    - change the status of v to "regular".

- End testing round.

- If v does not detect a cycle, then
  - if v is not the fault free node with smallest id in its cluster at level-1, then change the status of v to "regular".
  - Otherwise, execute the "Level-1 Testing Algorithm."
  - End testing round.

The *Diagnose* algorithm is very similar to that used in the ADSD algorithm. As in the diagnosis algorithm in Figure 2.4, the STATE information is completed at each node $n_{i,j}$ by using the node identifiers in TESTED_UP$_{i,j}$ and LEADER$_{i,j}$ arrays.

```
/* Regular node, Level-1 Testing Algorithm at node n_{x,a} */
1.  y = x          //assign my node id
2.  repeat {
3.      y = (y + 1) mod N/p
4.      request n_{y,a} to forward TESTED_UP_{y,a} to n_{x,a}
5.  } until (n_{x,a} tests n_{y,a} as "fault-free")
6.  for node = 0 to (N/p – 1)          // update local cluster diagnosis information
7.      TESTED_UP_{x,a}[node][a] = TESTED_UP_{y,a}[node][a]
8.  TESTED_UP_{x,a}[x][a] = y          // x itself tests y as fault free

/* Level-1 Cluter Leader Election or Update */
9.  If ( Cluter_leader( ) == "leader")        // check the leadership
10.     status(x) = leader
11. Else        // update diagnosis information regarding other clusters
12.     for cluster = 0 to (p – 1)
13.       for node = 0 to (N/p – 1)
14.           if (cluster ≠ a)
15.               TESTED_UP_{x,a}[node][cluster] = TESTED_UP_{y,a}[node][cluster]
16. Stop.       // end of Level-1 testing round
```

Fig. 2.14 Level-1 testing algorithm

```
/* Leader node, Level-2 Testing Algorithm at node n_{x,a}*/
1.   j = a; /* assign my cluster id*/
2.   repeat {
3.        j = (j + 1) mod p;
4.        u = N/p - 1;          // always start to test node id 0 in a new cluster
5.        repeat {
6.             u = (u + 1) mod N/p;
7.             request n_{u,j} to forward TESTED_UP_{u,j} and
                  LEADER_{u,j} to n_{x,a};
8.        } until (n_{x,a} tests n_{u,j} as "fault-free") or ("all the nodes in cluster j are
               faulty");
9.   } until (n_{x,a} tests n_{u,j} as "fault-free")
10.  for cluster = 0 to (p – 1)          // update info. regarding other clusters
11.      for node = 0 to (N/p – 1)
12.           if (cluster ≠ a)
13.              TESTED_UP_{x,a}[node][cluster]= TESTED_UP_{u,j}[node][cluster]
14.  Stop.     // end of Level-2 testing round
```

Fig. 2.15 Level-2 testing algorithm

2.3.7 Proof of Correctness and Diagnosis Latency

Note that our ML-ADSD algorithm is a generalization of the ADSD algorithm of Bianchini and Buskens [BB92]. We can view the ADSD algorithm as a level-1 algorithm. Since all the nodes are at level 1 in the ADSD algorithm, in at most N testing rounds after the last fault event every fault free node will have correct and consistent fault status information of all the nodes in the network. But in the case of the multilevel algorithm some of the faulty nodes could be at level 2 and may recover to become fault free nodes while being at level 2. Until they return to level 1, the information they contain is not reliable and may not be correct. Also, they may not perform level-1 testing, thereby

preventing the election of the leaders at level 1. So, in proving the correctness of the two-level algorithm we need to ensure that a faulty node after recovery does not prevent the election of leader at level 1 and also eventually returns to level 1. The actions taken by the ML-ADSD algorithm achieve this.

Consider again the two-level scheme show in Figure 2.10. Without loss of generality, we assume that initially all nodes are fault free and are regular nodes. Our proof of correctness involves establishing that after executing a certain number of testing rounds after the occurrence of the last fault event, all the nodes will have the correct view of the fault status of all the nodes in the system.

Note that a testing round for a node includes the time taken by the node to perform the actions specified by the 2-level ADSD algorithm.

The algorithm may be viewed as consisting of three phases.

Phase 1: In this phase all the nodes identify their respective leaders. The nodes in each level-1 cluster acquire correct view of the fault status of all the nodes in that cluster. In order to elect the leader each node must identify, using the data in the TESTED_UP array, the cycle of fault free processors at level 1. To do so, each node must execute certain number of level-1 testing rounds after the last fault event.

We need to consider several cases.

Case 1: No faults occur.

In this case, all nodes are regular nodes and as in the ADSD algorithm, in at most

***N/p* level-1 testing rounds** after the last fault event, all nodes in each level-1 cluster will get consistent and correct fault status information of all nodes in that cluster, and the leader of each cluster will be selected.

In the following, cluster refers to a level-1 cluster.

Case 2: A faulty node can recover only when it is at level 1.

Without loss of generality, let us assume that the last fault event is the recovery of a faulty node, say node k, in cluster 1. Let C_1, the cycle of fault free processors in cluster 1 just before the last fault event be as in Figure 2.16(a). Let i be the fault free node with the smallest id in C_1. In the worst case node k may also be in a cycle C_2 of faulty processors before its recovery. Cycle C_2 is shown in Figure 2.16(b). Since node k recovers in the last faulty event, the cycle of fault free processors after the recovery of node k will be as in Figure 2.16(c). This cycle needs to be identified by all the fault free nodes in cluster 1.

- Let us first assume that node k is less than i.

In the first testing round after the recovery of node k, node i will execute a level-1 testing (Case 1 in the 2-level ADSD algorithm) and identify node i_1 as the next node in the cycle of fault free processors. Node i being the leader of cluster 1 before the recovery, the TESTED_UP$_i$ will indicate that node i is a leader and so it will change its status to "leader". This is because, in the TESTED_UP$_i$ array of node i, node k will not be identified as a fault free processor until after a certain number of testing rounds. Other nodes including node k also will execute one level-1 testing in this first round, but will not identify themselves as leaders.

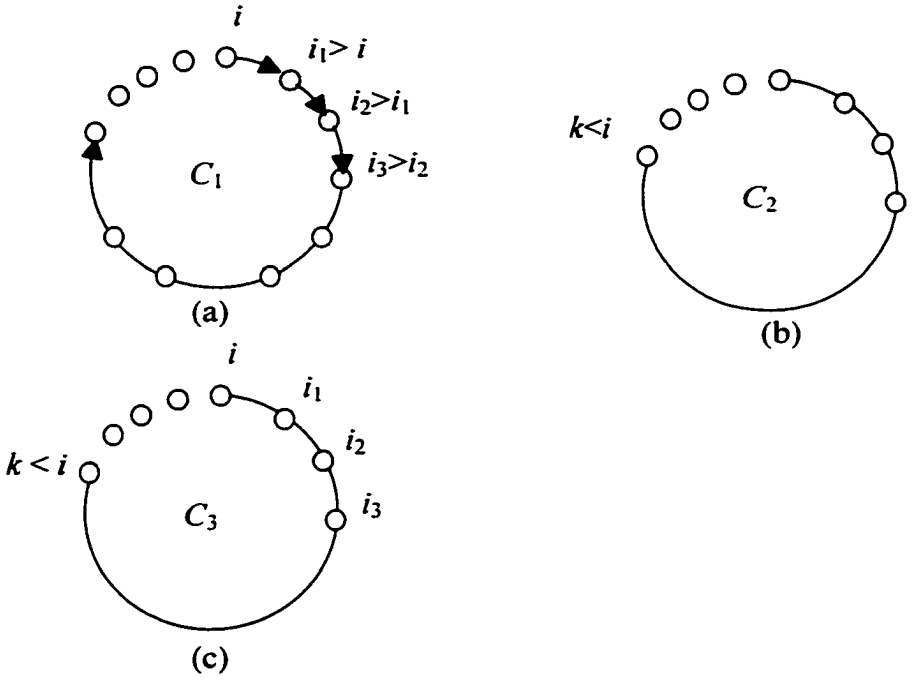


Fig. 2.16 Illustration of proof of correctness

In the second testing round, being a leader node, node i will execute a level-2 testing (Case 2 in the 2-level ADSD algorithm). Node i will not find itself to be the fault free node with the smallest id in cluster. So, it will set its status to "regular". This completes the actions taken by node i in the second testing round after the recovery of node k. Other nodes in cluster 1 will execute one level-1 testing and remain as regular nodes. These nodes also will identify the next two nodes in the fault free cycle C_3. In particular, node i_1 will identify i_2 and i_3 as the next two nodes in C_3.

In the third testing round, node i will execute one level-1 testing and identify itself as leader. After testing i_1 fault free and updating its TESTED_UP array, it will also identify nodes i_1, i_2, and node i_3 as the next three nodes in the fault free cycle C_3.

Thus in an even testing round node i will execute one level-2 testing and in an odd testing round it will execute one level-1 testing. This will continue until node i identifies

node k as the smallest fault free node in C_3 and node i recognizes that it is no longer the leader of its cluster. On the other hand, all other nodes will execute only level-1 testings.

Thus, if node k is xth one after node i in the fault free cycle C_3, then these x nodes will be identified by node i in x or $x+1$ testing rounds after the recovery of node k, and the cycle C_3 of fault free processors will be identified by all the nodes in at most ***N/p* or *N/p*+1 testing rounds** after the last fault event. Also, at the end of these testing rounds, node k will be identified as the leader of cluster 1.

The above reasoning is applicable even if more than one node recovers in the last fault event.

- Next consider the situation when k is greater than i.

In this case, in the first testing round after the last event, node i will execute a level-1 testing round and may find itself as leader. In the second and subsequent testing rounds, being a leader and the fault free node with the smallest id, node i will execute a level-1 testing as well as a level-2 testing and will remain at level 2. On the other hand, all other nodes in cluster 1 will execute only level-1 testings. Thus, in at most ***N/p* testing rounds** after the last event, node i and all other nodes in cluster 1 will identify node i as the leader of cluster 1.

Case 3: Faulty nodes at level 2 may recover

Let the last fault event be the recovery of node k at level 2.

- Let us first consider the situation when node k is not the fault free node with the smallest id in cluster 1 after the last fault event.

  In the first testing round after its recovery, node k will execute a level-2

testing round and will return to "regular" node status. In the second testing round, it will execute one level-1 testing and may find itself as a leader because, possibly, it was in a cycle of faulty nodes before the last fault event. In the third testing round it will execute one level-2 testing and again return to "regular" node status. In the fourth testing round it will execute one level-1 testing and may again find itself to be a leader. This sequence of alternation between "regular" and "leader" status will continue until at most ***N/p* testing rounds** are executed. At the end of these testing rounds, node k will identify the cycle C_3. This is also true of all other nodes in the cycle C_3.

- Let us next consider the situation when node k is the fault free node with the smallest id in cluster 1 after it recovers in the last fault event.

  In this case, in each testing round after the last fault event, the node k will execute both a level-1 testing and a level-2 testing and will remain at level 2. All other nodes will execute level-1 testings in these testing rounds. So, in at most ***N/p* testing rounds** after the last event, every node in cluster 1 will identify node k as the leader.

Summarizing, in **at most *N/p* or *N/p* +1** testing rounds after the last fault event all nodes in all clusters will identify the leader nodes and acquire correct view of the status of all the nodes in their respective clusters.

Phase 2: During this phase, in **at most *p* testing rounds** the fault free leaders of the clusters identify the ring of leader nodes at level 2 using the LEADER array, update

their TESTED_UP arrays with the status information of all the nodes in clusters other than their own.

Phase 3: During this phase consisting of **at most *N/p* Level-1 testing rounds**, the information at the fault-free leader nodes will be propagated to the remaining nodes in their respective clusters.

Thus at the end of at most $2(N/p) + p + 1$ testing rounds after the last fault event all the fault free nodes in the network will have consistent and correct status information of all the nodes in the network. Thus we have the following theorem.

> **Theorem 1:** The diagnosis latency of the two-level adaptive distributed diagnosis algorithm is **at most $2(N/p) + p + 1$ testing rounds**

We now draw attention to certain issues we encountered while developing the two-level algorithm.

Suppose no faults occur (Case 1 in Phase 1 discussed above). The following simplified version of the 2-level ADSD algorithm will be adequate for all nodes to acquire a correct view of the status of all the nodes in the network.

Simplified 2-Level ADSD Algorithm

Initially, all the nodes are regular nodes.

During a testing round each node v performs the following.

CASE 1: If v is a regular node, then execute the "Level-1 testing algorithm" of Figure

2.14, which includes leader election.

- Change the status of v to "leader", if it becomes a leader.
- End testing round.

CASE 2: If v is a leader, then execute the "Level-2 testing algorithm" of Figure 2.15.

- Execute the "Cycle detection algorithm" of Figure 2.13.
- If v detects a cycle at level 2, then change the status of v to "regular".
- End testing round.

Consider Case 2 in Phase 1. Assume k is less than i (as in Figure 2.17). In this situation, if the above simplified algorithm is used, then in the first testing round node i will identify itself as leader of cluster 1 and move to level 2. In the second and subsequent testing rounds it will execute only level-2 testing. It will also identify the leader node q of cluster 2 and update its LEADER array appropriately. Since it does not execute any level-1 testings, the nodes in cluster 1 will never be able to identify their leader node k. But the node t will become a leader of cluster 4 at some point and will identify node k as the leader of cluster 1 and update its LEADER accordingly. Since node k has not been able to identify itself as leader of cluster 1, none of the leader nodes will identify the cycle of leaders at level 2. Thus, a deadlock situation results (as in Figure 2.17) and the nodes execute their respective actions without ever being able to acquire the correct view of the status of the nodes in the network.

Similar deadlock situations would occur in case 3 too. The additional actions taken in Case 2 of the 2-level ADSD algorithm ensure that such deadlock situations do not occur.

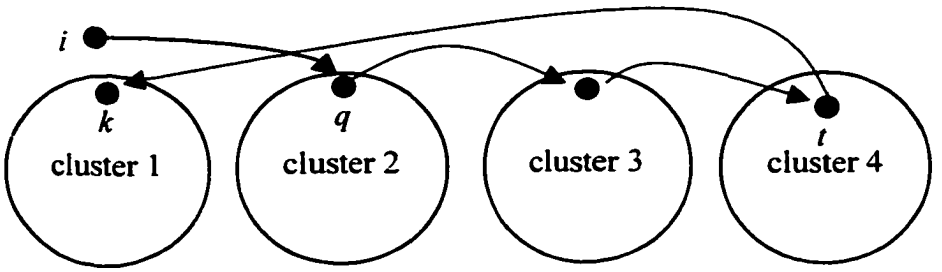


Fig. 2.17 Illustration of deadlock

2.3.8 ML-ADSD Algorithm for the General Case

We now present the details of the Multi-level Adaptive Distributed Diagnosis Scheme (ML-ADSD) for the general case of level greater than 2. A pictorial tree description of this algorithm is shown in Figure 2.18(a). To simplify the discussion and without loss of generality, we assume that each cluster at level 1 (the original p clusters) has at least one fault free node. We denote the last level by M.

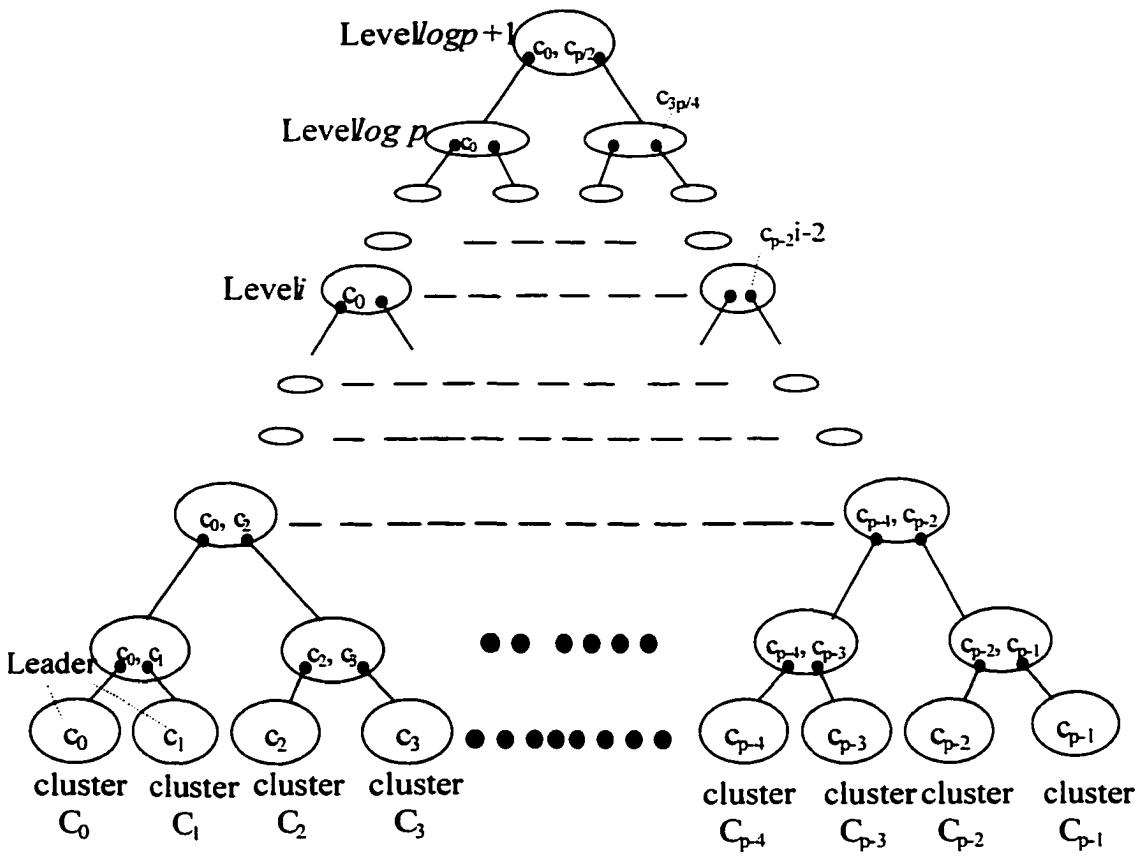


Fig. 2.18(a).Tree representation of ML-ADSD algorithm

Algorithm Specification

- **Level-*i* Clusters and Leaders**

At level 1 of the tree representation are the p original clusters, to be called the **level-1 clusters**. At level 2 there are $p/2$ clusters, to be called **level–2 clusters**. Each level-2 cluster consists of at most 2 nodes which are leaders of two level-1 clusters. In general, at level i there are $p/2^{i-1}$ clusters, each containing at most two nodes which are leaders of two level-(i-1) clusters. In view of our assumption that each cluster has at least one fault free node, all clusters at levels greater than one contain exactly two nodes. Note that, as before, in each cluster the node with the smallest id will be called the leader of that cluster. The leader will be called the left node and the other node will be called the right node of that cluster. Also, if $M < \log p + 1$, then at level M there will be one cluster of $p/2^{M-2}$ nodes, one for each cluster at level $M-1$.

Each cluster at level $i < M$ may be viewed as the root of the subtree with 2^{i-1} level-1 clusters as leaves. Specifically, consider the jth cluster (counted from left) in level i. This cluster may be viewed as representing the level-1 clusters numbered from $j2^{i-1}$ to $(j+1)2^{i-1}-1$. Each one of the two nodes in this cluster will then represent half of this group of the level-1 clusters. That is, the left node (the node with the smallest id) in this cluster will represent the level-1 clusters numbered from $j2^{i-1}$ to $2^{i-2}(2j+1)-1$, and the right node will represent the clusters numbered from $2^{i-2}(2j+1)$ to $(j+1)2^{i-1}-1$. We shall refer by $P_{l,j}(i)$ the set of level-1 clusters represented by the left node in cluster j at level i, and by $P_{r,j}(i)$ the set of clusters represented by the right node in jth cluster at level i. See Figure 2.18(b) for a pictorial explanation of these definitions.

If $M < \log p + 1$, then the level-1 clusters represented by the jth node at level M

are those at the leaves of the subtree rooted at this node and will be denoted by $P_j(M)$.

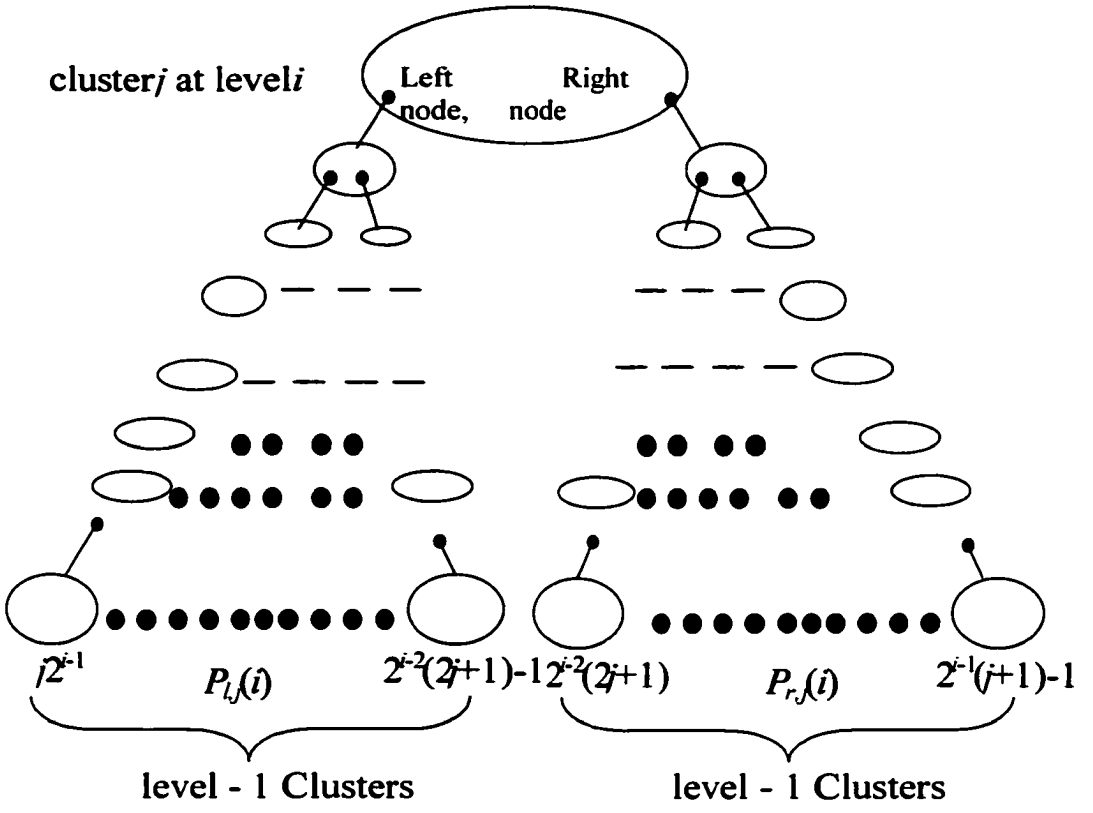


Fig. 2.18(b) Subtree representation at the node of cluster j at level i

Level-i Testing Rounds

If $M = log\ p + 1$, then nodes at level i perform level-i testing as follows. The node in the cluster j at level i representing $P_{l,j}(i)$ will test the nodes in the level-1 clusters in the set $P_{r,j}(i)$ until it finds the smallest fault free node, if such a node exists, and updates in its TESTED_UP array the status of the nodes in these clusters. The other node in the cluster j at this level representing $P_{r,j}(i)$ will test the nodes in the level-1 clusters in the set $P_{l,j}(i)$ until it finds the smallest fault free node , if it exists, and updates in its TESTED_UP array the status of nodes in these clusters.

If $M < log\ p + 1$, then nodes at level M perform the level-M testing as follows. The j^{th} node in this cluster will test the nodes not in $P_j(M)$ until it finds the smallest fault free node in a cluster greater than its cluster (modulo N) and updates the TESTED_UP array with the status of nodes in other clusters.. This is similar to the level-2 testing round definition in section 2.3.2.

With these definitions we are now ready to present the ML-ADSD algorithm for the general case of level greater than two.

- **ML-ADSD Algorithm**

Initially all nodes are regular nodes and are at level 1.

During each testing round each node v executes the following.

Let k be the current level of node v.

CASE 1: If $k == 1$, then do the following.

- Execute the level-1 testing algorithm of Figure 2.19
- Execute the leader election algorithm of Figure 2.12
- If node v is a leader, then set status(v) = *leader*
- If node v is not a leader then execute the algorithm in Figure 2.20 and update the TESTED_UP array with the status of nodes in other clusters.

CASE 2: If $k > 1$, then do the following.

- Execute the level-k testing algorithm of Figure 2.21.

CASE 2.1 If v detects a cycle (using the LEADER array) then do the following.

- If k is not the last level then do:

(a) If v is not a leader then change the level of v to 1 and End testing round.

(b) If v is a leader at level k, then change the level of v to k+1 and End testing round.

- If k is the last level, then change the level of v to 1 and End testing round.

CASE 2.2 If v does not detect a cycle at level k, then test

- If v is still the fault free node with the smallest id in the group of level-1 clusters at the leaves of the subtree rooted at v.
  - If not, change the level of v to 1.
  - Otherwise, execute the level-1 testing algorithm (see Figure 2.19) and End testing round.

```
/* Multi-level Level-1 testing algorithm at node n_{x,a}*/
1. y = x          //assign my node id
2. repeat {
3.     y = (y + 1) mod N/p
4.     request n_{y,a} to forward TESTED_UP_{y,a} to n_{x,a}
5. } until (n_{x,a} tests n_{y,a} as "fault-free")

6. for node = 0 to (N/p – 1)          // update local cluster info.
7.     TESTED_UP_{x,a}[node][a] = TESTED_UP_{y,a}[node][a]

8. TESTED_UP_{x,a}[x][a] = y          // x itself tests y as fault free

9. End of level-1 testing
```

Fig. 2.19 Multi-level level-1 testing algorithm

// **Update information regarding other clusters at node $n_{x,a}$**

1. **for** *cluster* = *0* to $(p - 1)$
2. **for** *node* = *0* to $(N/p - 1)$
3. **if** $(cluster \neq a)$
4. $\text{TESTED\_UP}_{x,a}[node][cluster] = \text{TESTED\_UP}_{y,a}[node][cluster]$

Fig. 2.20 Information update for non-leaders

/* Multi-level level *k* testing algorithm */

1. If $M = log\ p + 1$, then nodes at level i perform level-i testing rounds as follows.
   - The node in the cluster j at level i representing $P_{l,j}(i)$ will test the nodes in the level-1 clusters in the set $P_{r,j}(i)$ until it finds the smallest fault free node, if such a node exists, and updates in its TESTED_UP array the status of nodes in these clusters. The other node in the cluster j at this level representing $P_{r,j}(i)$ will test the nodes in the level-1 clusters in the set $P_{l,j}(i)$ until it finds the smallest fault free node , if it exists, and updates in its TESTED_UP array the status of nodes in these clusters.
2. If $M < log\ p + 1$, then nodes at level M perform the level-M testing rounds as follows.
   - The *jth* node in this cluster will test the nodes not in $P_j(M)$ until it finds the smallest fault free node in a cluster greater than its cluster (modulo p) and updates the TESTED_UP array with the status of nodes in other clusters.

Fig. 2.21 Multi-level level-*k* testing algorithm

2.3.9 Proof of Correctness and Diagnosis Latency of the ML-ADSD Algorithm

As in our discussion in the previous section on the proof of correctness of the 2-level algorithm, we view the ML-ADSD algorithm as consisting of three phases. We show that after executing certain number of testing rounds after the last fault event, all nodes acquire correct view of the status of all the nodes in the network.

First, we shall assume that the number of levels $M = \log p + 1$.

Case 1: No faults occur

Phase 1: In at most ***N/p* level-1 testing rounds** after the last fault event, all nodes in each level-1 cluster will get consistent and correct fault status information of all nodes in that cluster, and the leader of each cluster will be selected.

Phase 2: The leaders (left nodes of the two clusters at level $\log p$ of the first cluster (0^{th} cluster) and the $N/2\text{-}1^{th}$ cluster) will reach the last level in $2(\log p\text{-}1)$ testing rounds, and then perform two testing rounds. At the end of these ***2logp* testing rounds**, these two nodes will have correct fault status information of all the nodes in the network.

- While the left nodes of the two clusters at level $\log p$ move to the last level, the right nodes of these clusters move to level 1. After performing one testing round at level 1, these right nodes will move to level $\log p$ after $2(\log p\text{-}2)$ testing rounds and perform two additional testing rounds at that level to collect information from the left nodes in their respective clusters the correct fault status information of all

the nodes in the network. In all, they perform **$2(\log p - 1) + 1$ testing rounds** to collect correct fault status information of all the nodes in the network.

- Continuing as above, in general, the right nodes of the clusters at level i perform $2(i-1) + 1$ testing rounds to collect the correct fault status information of all the nodes in the network. Note that $2 \leq i \leq \log p$.

Phase 3: Finally, the nodes in all level-1 clusters will collect from their respective leaders the correct fault status information of all the nodes in the network in at most ***N/p* level-1 testing rounds**

Combining all these testing rounds, all the nodes will have correct fault status information of all the nodes in the network in at most

$$N/p + [\, 2 \log p + 2 (\log p - 1) + 2 (\log p - 2) + \ldots\ldots\ldots + 2 \,] + \log p - 1 + N/p$$

$$= 2\, N/p + (\log p + 1)(\log p) + \log p - 1$$

$$= 2\, N/p + (\log p + 2) \log p - 1$$

testing rounds.

Case 2:Only Faulty nodes at level 1 recover

As in Case 2 in the discussion in the previous section, all nodes will identify their cluster leaders in Phase 1 in at most N/p or $N/p + 1$ testing rounds. The other phases will proceed as in Case 1 above. So the diagnosis latency in this case is one more than that for the first case.

Case 3: Faulty nodes at levels greater than one may recover

In this case, an additional at most 2 $log\ p$ testing rounds will be required for all the fault free nodes to return to level 1. So, in this case the diagnosis latency is **2 N/p + ($log\ p$ +4) $log\ p$ testing rounds**.

Summarizing the above, we have the following:

> **Theorem 2:** If $M = log\ p + 1$, the diagnosis latency of the M-Level ML-ADSD algorithm is at most **2 N/p + ($log\ p$ + 4) $log\ p$ testing rounds**.

Proceeding as above, we can determine the diagnosis latency for the M-level ML-ADSD algorithm as in Theorem 3.

> **Theorem 3:** The diagnosis latency of the M-level algorithm (M > 2) is at most **2 N/p + (M-2)(M+4) + $p2^{(-M+3)}$ + 1 testing rounds**.

We wish to note that other implementations of our multilevel scheme are possible. For instance, at each level we can combine more than 2 clusters to form clusters for the next higher level. Diagnosis latency calculation and proof of correctness in these cases will proceed as above with some appropriate minor changes.

2.4 Simulation and Discussion

In this section, we present the results of our simulation of the ADSD [BB92], Hi-ADSD [DN98] and our ML-ADSD algorithms (for two-level and three-level schemes) for networks of various sizes. The algorithms were simulated using the discrete-event simulation language *SSS* [P96]. The fail-stop model was used. All of the network nodes are modeled as independent processes and each node is assigned a unique node identifier.

Three types of events are defined: test, fault occurrence and recovery.

Also, tests are scheduled for each node at each $30 \pm \sigma$ time units as in [DN98], where σ is a random number in the range of 0 and 3. This is the time interval between two consecutive testing rounds at a node.

The fault event is modeled as the process being in *faulty* state and the recovery as the process being in *recovery* state. During each test, the status of the node is checked and, if the node is fault-free, the whole diagnosis information stored in the tested fault-free node is copied to the testing node. If the tested node is faulty, the testing nodes proceed testing as in the algorithm.

Experiments are conducted for all three algorithms on networks of different sizes. In each simulation, we first made five percent of the nodes fail. We then allowed 60% of these failed nodes to recover. In all, there were $0.08*N$ event fault events. These events occurred about $x * (30 \pm \sigma)$ time units apart, where x is a random number between 0 and 5. There are $30 + \sigma$ time units between the last fault event and the first recovery event. The average diagnosis latency in terms of testing rounds and also in testing time units as well the total number of test messages exchanged from the last event and until all the

fault free nodes have the same correct diagnosis information were collected. The average was over 50 simulation runs. The results are presented in Table 2.2 and Figure 2.22.

Table 2.2 Simulation results for network sizes from 64 to 1024 nodes

Size		*pxN/p*	Round	Time	Test
N = 64	ML-2	8x8	12.2	405	1697
	ML-3	8x8	**11.9**	**399**	**1618**
	Hi-ADSD		**15.2**	**499**	**2056**
	ADSD		30.7	999	4051
N = 128	ML-2	16x8	18.2	596	4972
		8x16	18.3	602	5022
	ML-3	16x8	**16.4**	**540**	**4403**
		8x16	19.9	646	5264
	Hi-ADSD		**19.5**	**634**	**5228**
	ADSD		61.2	1978	16116
N = 256	ML-2	8x32	32.8	1070	17821
		16x16	26.0	852	14200
		32x8	34.5	1121	18676
	ML-3	8x32	33.2	1081	17657
		16x16	**24.1**	**790**	**12906**
		32x8	27.9	907	14813
	Hi-ADSD		**28.7**	**931**	**15351**
	ADSD		123.7	3988	65160
N = 512	ML-2	16x32	40.1	1309	43632
		32x16	41.9	1363	45419
	ML-3	16x32	37.8	1237	40418
		32x16	**35.1**	**1148**	**37508**
	Hi-ADSD		**38.7**	**1247**	**41135**
	ADSD		245.4	7949	259677
N = 1024	ML-2	32x32	55.2	1808	120533
	ML-3	32x32	**49.8**	**1636**	**106833**
	Hi-ADSD		**52.1**	**1674**	**110475**
	ADSD		488.8	15888	1037842

N: the number of network nodes. ML2: ML-ADSD with two-level scheme
p: the number of clusters ML3: ML-ADSD with three-level scheme

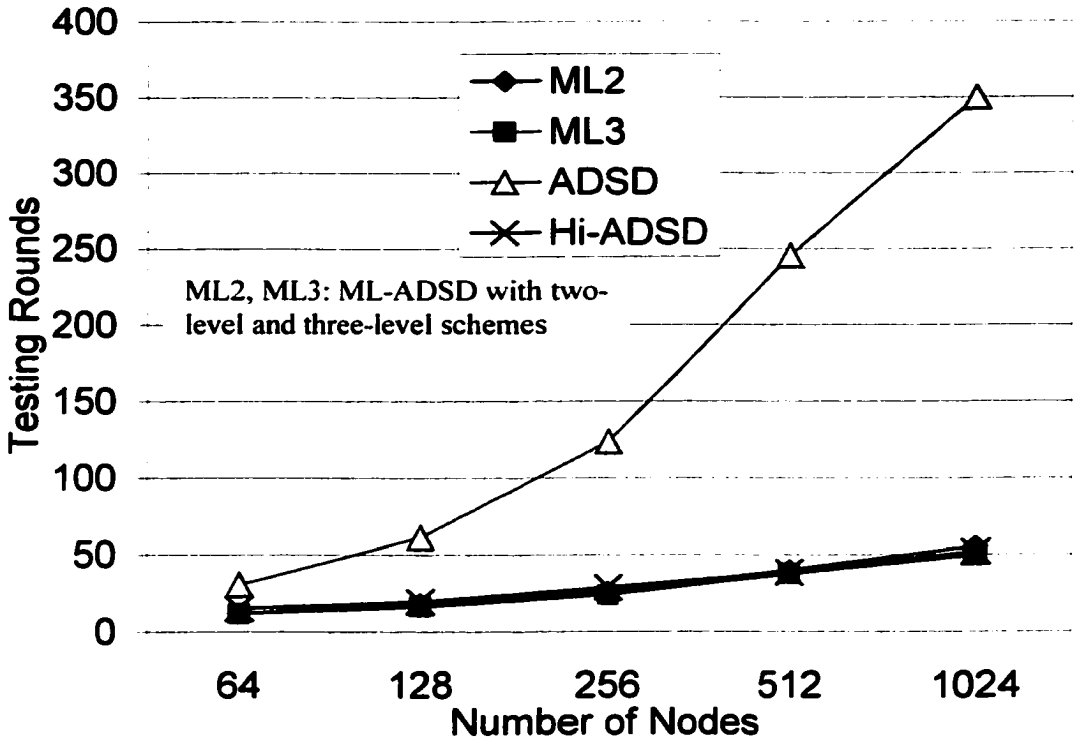


Fig. 2.22 (*a*) Comparison of diagnosis latencies in terms of testing rounds

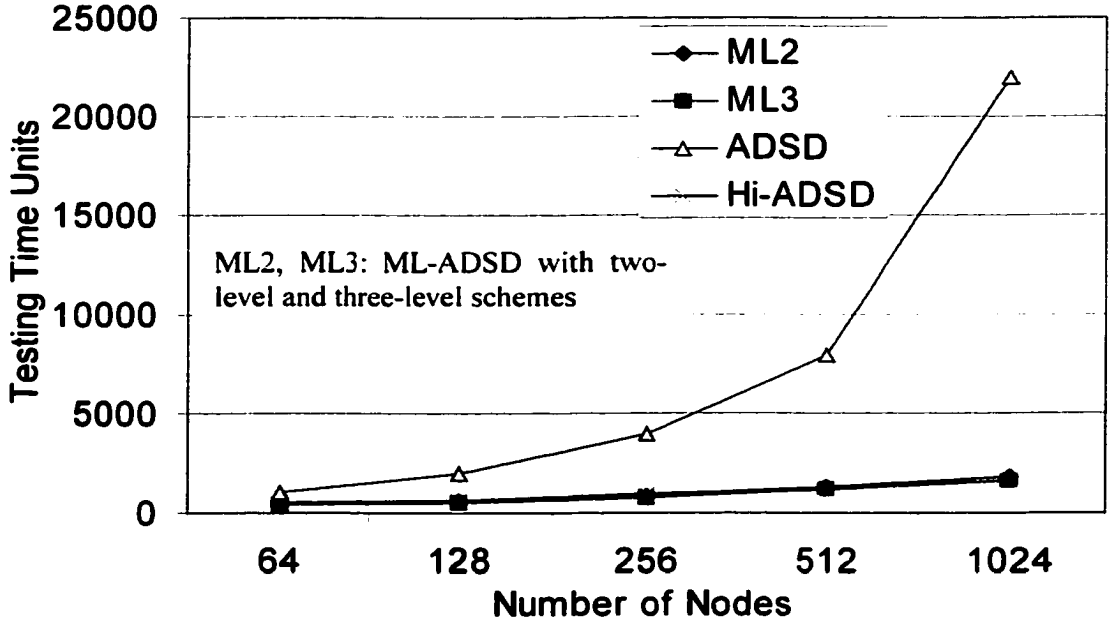


Fig. 2.22 (*b*) Comparison of diagnosis latencies in terms of testing time

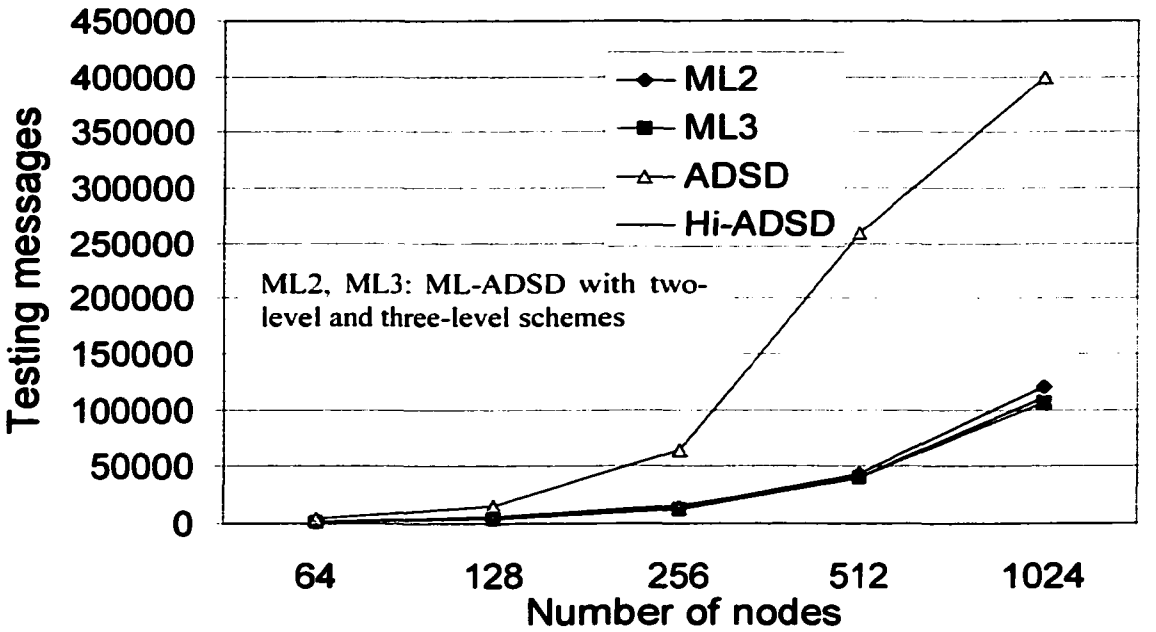


Fig. 22(*c*) Comparison of the total numbers of testing messages

Recall that the ADSD algorithm has a diagnosis latency of $O(N)$ and the Hi-ADSD algorithm has a diagnosis latency of $O(log^2 N)$. As we can see from Table 2.2 and Figure 2.22 and as expected, in all respects, the performance of the ML-ADSD algorithm is much better than that of the ADSD algorithm. In all cases, the number of tests (messages) used by the ML-ADSD algorithm is smaller than the number for the Hi-ADSD algorithm. In all cases, the time required by the ML-ADSD algorithm is better than or the same as for the Hi-ADSD algorithm. Note that the performance of the ML-ADSD algorithm can be improved by an appropriate choice of the number of clusters and the number of levels. We would also like to point out that the ML-ADSD algorithm is scalable in the sense that only some minor modifications will be required to adapt the algorithm to networks of varying sizes. This property is not shared by the Hi-ADSD algorithm.

Chapter 3

A Distributed Network Fault Detection System Based on the SNMP Protocol

In this chapter we describe the design and implementation of a distributed network fault detection system based on the SNMP (Simple Network Management Protocol) protocol. The fault detection system is implemented by integrating the ML-ADSD algorithm of the previous chapter with the SNMP protocol.

In section 3.1, we present a brief sketch of the evolution of the SNMP protocol. In section 3.2, we describe the motivation for distributed network fault detection. In section 3.3, we briefly summarize the five key fundamental areas of the network management and the details of the architecture of a network management system and SNMP. In section 3.4, we describe how we integrate our multi-level distributed algorithm into the SNMP-based fault management network.

3.1 Evolution of Network Management Technology

In this section, we present a brief overview of the Network Management Framework developed by International Organization for Standardization (ISO) also later known as the SNMP (Simple Network Management Protocol) [RM90] [CFSD90] [MR91] and why we need a framework to manage the network devices running over the Internet.

An Internet consists of different kinds of networks and they are connected together through the use of network devices (e.g., routers, switches, and bridges) and

some network protocols. In general these devices are able to implement TCP/IP protocol, but they have the interoperability problems because different vendors produce these devices. Therefore, the concept of an "open" network management architecture framework is proposed to manage these devices running over the networks. As a result, the SNMP is such a management framework and has been widely accepted and broadly deployed over the Internet today.

In 1987, three models of network management, namely, HEMS, CMOT, and SGMP were proposed. High-level Entity Management System (HEMS) was first proposed by researchers as an experimental framework to manage the networks. Later, the Common Management Information Protocol (CMIP) was proposed by Open System Interconnection (OSI) group of the International Organization for Standardization (ISO) and CMIP was designed to run on the OSI-based networks. So a new protocol CMOT (CMIP over TCP) was proposed by ISO to be used for the TCP networks. In November 1987, a simple design and easy to implement protocol Simple Gateway Monitoring Protocol (SGMP) [DCFS87] was proposed and soon gained it acceptance and wide deployment in the Internet community.

In February 1988, the Internet Activities Board (now Internet Architecture Board) decided to promote CMOT as the future model for Internet Network Management Framework, and use SGMP as the short-term solution before CMOT was accepted. For some reason, HEMS was not considered.

In April 1988, the Simple Network Management Protocol (SNMP) was proposed by IAB as the common network management model to be developed to allow the future transition of systems from SGMP to CMOT.

In June 1989, due to some disagreements between CMOT and SNMP groups, IAB decided to let CMOT and SNMP groups to develop independently. In May 1990, IAB promoted SNMP as the standard network management protocol and a recommended framework for use on the TCP/IP networks.

Soon after defining the format of Management Information Base (MIB) [RM91], and Trap [R91] message, and revision of MIB, SNMP version 1 (SNMPv1) [MR91] was issued in March 1991. SNMPv1's standardization, universal acceptance, independent operating systems and languages, and little demand on system resources have resulted in its wide deployment.

In the years that followed, SNMP went through several improvements. In early 1996, SNMPv2 (SNMP version 2) was finalized and issued. In April 1999, SNMPv3 [HPW99][CHPW99], the latest version, was proposed to emphasize the security and administration aspects of the earlier versions. SNMPv3's features include identification between users, verification of message for modification, protection of message from disclosure, and message authentication services, etc.

Although many versions of SNMP and different protocols (such as Remote Monitoring Protocol, RMON [W91]) have been proposed and implemented to monitor and control the networks, SNMPv1 is still the widely accepted and deployed protocol.

3.2 Motivation for Distributed Network Management

The purpose of network management is to manage (monitor and control) the local and remote network devices over a computer network through the exchange of messages between devices and hopefully to maintain the health of the network through such framework.

Base on this approach, in general one node (device) will be designated as a central network management station (NMS) or manager and the rest of the nodes as regular nodes (Agent). The manager then periodically polls information from the agents and then organizes those data into meaningful information to diagnose or prevent network problems. If the network size increases, then one or more nodes will be designated as new NMS with dual role, manager and agent, to balance loads and to void traffic congestion around the central network manager. Those new managers then poll information from agents under their respective domains and the central network manager

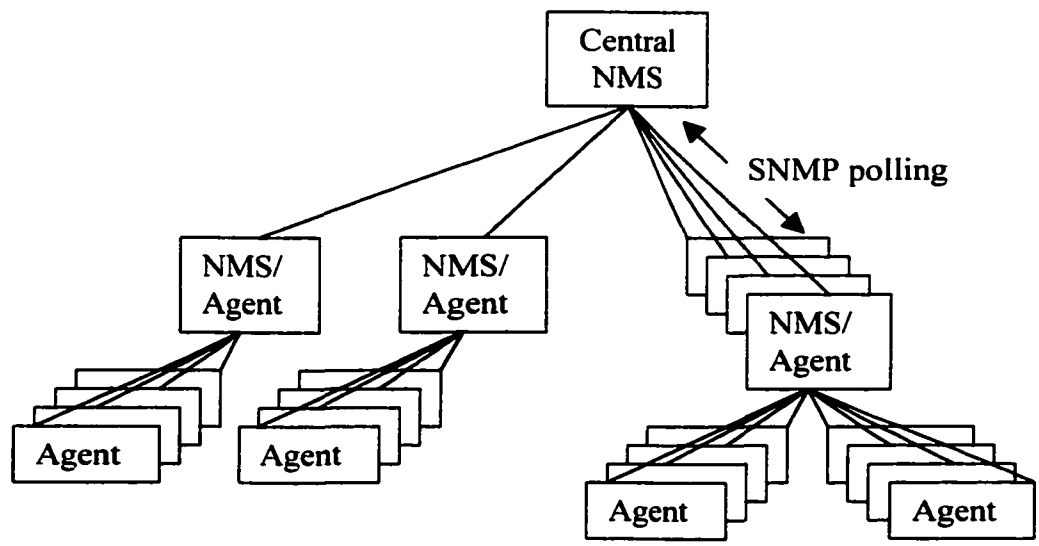


Fig. 3.1 A tree like structure of the interaction between manager and agent

polls information from those new managers. Figure 3.1 illustrates a tree like structure of the interaction between the managers and agents.

The drawbacks of such an approach include a single point of failure, lack of scalability, and high communication costs around the central manager. If the manager fails, the node information under its domain will be lost because there is no automatic substitution mechanism provided for by the network management system. Through our application, we want to show that some of the above problems can be solved to some extent and that the improvement of fault management through distributed diagnosis is feasible.

3.3 Functional Areas of Network Management

There are five key functional areas of network management as defined by the International Organization for Standardization (ISO). Although this functional classification was developed for the Open Systems Interconnect (OSI) environment, it has gained broad acceptance by vendors of both standardized and proprietary network management systems. These areas [M98] include:

- ***Fault management*** involves the detection of a fault, fault isolation, and possibly a correction operation on the abnormal situation and informing the problem to the management system. It plays a very important part of network management in that it keeps the network running under a healthy (correct) condition.
- ***Accounting management*** is to determine the charge for the use of network resources (managed devices), also called *chargeback management*, and identify the cost of operating and maintaining the network resources (or called *cost management*).

- ***Configuration management*** is to identify the physical or logical connection of network devices, collect information from and provide information to the managed devices, and to configure and coordinate those devices to provide uninterrupted interconnection services.
- ***Performance management*** is to evaluate and monitor the performance of the managed devices and modify those devices' settings to have a better performace, if necessary.
- ***Security management*** is to provide the security policies and actions to prevent unauthorized users from accessing, using, and changing the network devices.

Among these areas, our research focuses on fault management, that is, to quickly detect a service-affecting problem, and report it to a management device.

3.4 Network Management Architecture (Model)

In general, a network management system contains five components (as shown in Figure 3.2):

- one or more ***managed nodes*** (e.g., Desktop PCs, Workstations, Laser printer, and Tower Box) each containing an agent;
- at least one ***network management station (NMS)*** on which one or more network management applications (also termed managers) reside;
- perhaps one or more ***dual role entities*** which are able to act in both the agent and manager roles;

- ***a network management protocol*** (e.g., SNMP) which is used by the station and the agents to exchange management information; and
- ***management information*** (e.g., Management Information Base – MIB) which is used to reflect the current status of the managed devices.

Figure 3.2 is a snapshot of the model over an Internet. The Workstation can play the dual role entities, as a manager to poll management information from agents (e.g., Laser printer or Tower box) through SNMP protocol and as an agent to provide management information to other managers in the network.

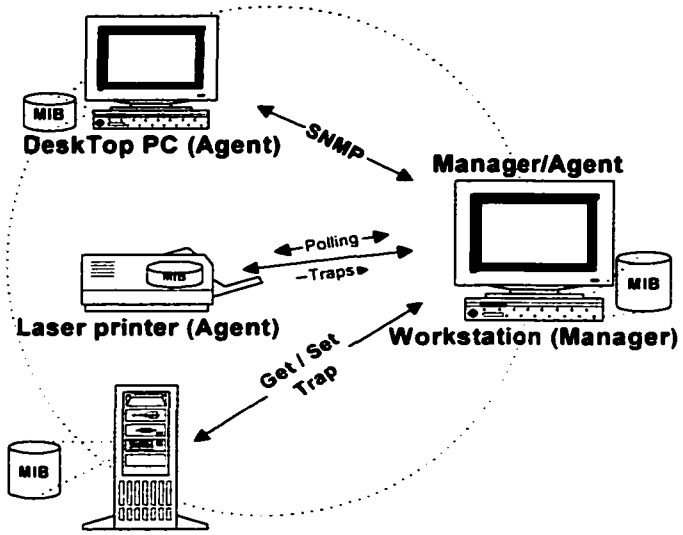


Fig. 3.2 Network management architecture

3.4.1 Managed Nodes

A managed node refers to any device that has some sort of network capability that allows being managed. A device can be a modem, printer, workstation, mainframe, bridge, repeater, hub, or router system. Also, we say that a node is managed when it is continuously monitored by an SNMP agent.

3.4.2 Network Management Stations (NMS)

A network management station (NMS or management node) refers to a workstation or server on a network which contains one or more network management applications and exchanges information with other workstation through some network management protocol. Those applications will periodically poll (collect) information from other devices, and present the meaningful information to the human managers with an interactive menu-driven or graphical user interface or web-based interface for analysis.

3.4.3 Dual Role Entities

As the term implies that a workstation can both have the role of a managed node and management node at the same time. It can have an agent application and a management application running at a node and doesn't intervene with each other's functionality.

3.4.4 Network Management Protocol

There are many network management protocols, such as SNMP, CMIP (Common Management Information Protocol), and TL1 (Transaction Language 1). Because of the wide acceptance and low complexity features of SNMP, we have decided to use SNMP in our research.

SNMP protocol defines the format of the SNMP messages exchanged between management and managed nodes, and uses four simple yet powerful operations (Get, GetNext, Set, and Trap) to read or change information in managed node or report events

between management and managed nodes. The features of SNMP include its ubiquity, standardization, broad acceptance and support, lightweight workload (small code) extendibility, and portability [M98].

3.4.5 Management Information

Management Information [RM90] reflects the current values and states of the managed nodes. Managed object is defined as a basic unit of management information. A collection of related managed objects defined in a document is called the Management Information Base (MIB) [MR90]. A managed object can be viewed as simple as the representation of the location of the machine it resides, the machine's IP address, the SNMP service uptime, a document, or as complex as the routing table of a router. Even more, the user can write his own codes (e.g., C, C++, or Java) after defining the managed object, such that when the managed object is queried it can interact with the local database and return the result.

In general, the network can be monitored and controlled by the management node by sending a query message (e.g., the SNMP Get or Getnext operation) or an alter message (e.g., the SNMP Set operation) to the managed objects of the remote managed node.

3.5 Structure of Management Information

In the SNMP management framework, the OSI uses the Structure of Management Information (SMI) language [RM90] to define the common data structures and identification scheme for the definition of management information to be exchanged and

understood between managed nodes and management stations. SMI is a subset of the formal language Abstract Syntax Notation One (ASN.1) [CCITT89a], and the goals of SMI is to achieve simplicity and extensibility. Management Information defined using ASN.1 is implementation-independent, well-defined, and unambiguous. ASN.1 also includes the rules called Basis Encoding Rules (BER) [CCITT89b] as to how instances of an object type are represented when being transmitted on the network.

In general, managed objects reside in a virtual information store, like a database, called the Management Information Base (MIB). The SMI also defines the schema of the database.

Each type of object (termed an object type) consists of five fields: a *name*, a *syntax*, *max-access*, *status* and *description*. The *name* is an unique OBJECT IDENTIFIER and should not be conflict with other type name. The *syntax* defines the abstract syntax for the object type (e.g., Integer, Octet String, Null, and TimeTicks*Access* defines the access rights, i.e., read-only, read-write, or not-accessible. *Status* shows one of the mandatory, optional, or obsolete states.). The *description* describes the meaning of the object type. An example of the object type definition is given below:

sysDescr OBJECT-TYPE
SYNTAX OCTET STRING
MAX-ACCESS read-only
STATUS mandatory
DESCRIPTION "A textual description of the entity. This value should include the full name and version identification of the system's hardware type, software operation-system, and networking software. It is mandatory that this only contains printable ASCII characters.
::= {system 1}

For example, this object type is named sysDescr (i.e., system description for short) and is assigned as the first object under *system* group. So its object ID (OID) is the concatenation of *system* OID (1.3.6.1.2.1.1) plus 1 which is equal to 1.3.6.1.2.1.1.1 (as shown in Figure 3.3). The construction of the system OID can be viewed as the path from root of the OID directory tree to the leave with a dot between each internal node. For example, the path of ***sysDescr* OID** is

iso(1).org(3).dod(6).internet(1).mgmt(2).mib-2(1).system(1).sysDescr(1) or

1.3.6.1.2.1.1.1 in numerical object ID string format.

3.5.1 Names

Managed objects are identified by names and each name is constructed in a hierarchical tree structure. To model the naming notion, the OBJECT IDENTIFIER (OID) concept is utilized. An OJBECT IDENTIFIER is a sequence of integers that traverse a reverse tree. The concept is similar to the computer directory tree structure. At the top of the reverse tree is the unlabeled virtual root and there are three branches which connect to three labeled nodes under the root. Each node can have its own branches and each branch may have its own children which in turn can have its own branches. The maximum number of levels of the tree (depth of the tree) is 128. An OID is like a "path" which is constructed by traversing from the root to the leaf nodes. Each branch is separated by a period. A label consists of a textual description and an integer.

At the first level, there are only three labeled nodes ccitt(0), iso(1), and joint-iso-ccitt(2). After that, there are many different nodes, as can be seen in Figure 3.3. However, most of the new MIB objects defined will be under the **internet** node.

For example, the path of the ***internet*** node is written as:

iso(1).org(3).dod(6).internet(1) or

1.3.6.1 in numerical object ID (OID) string format.

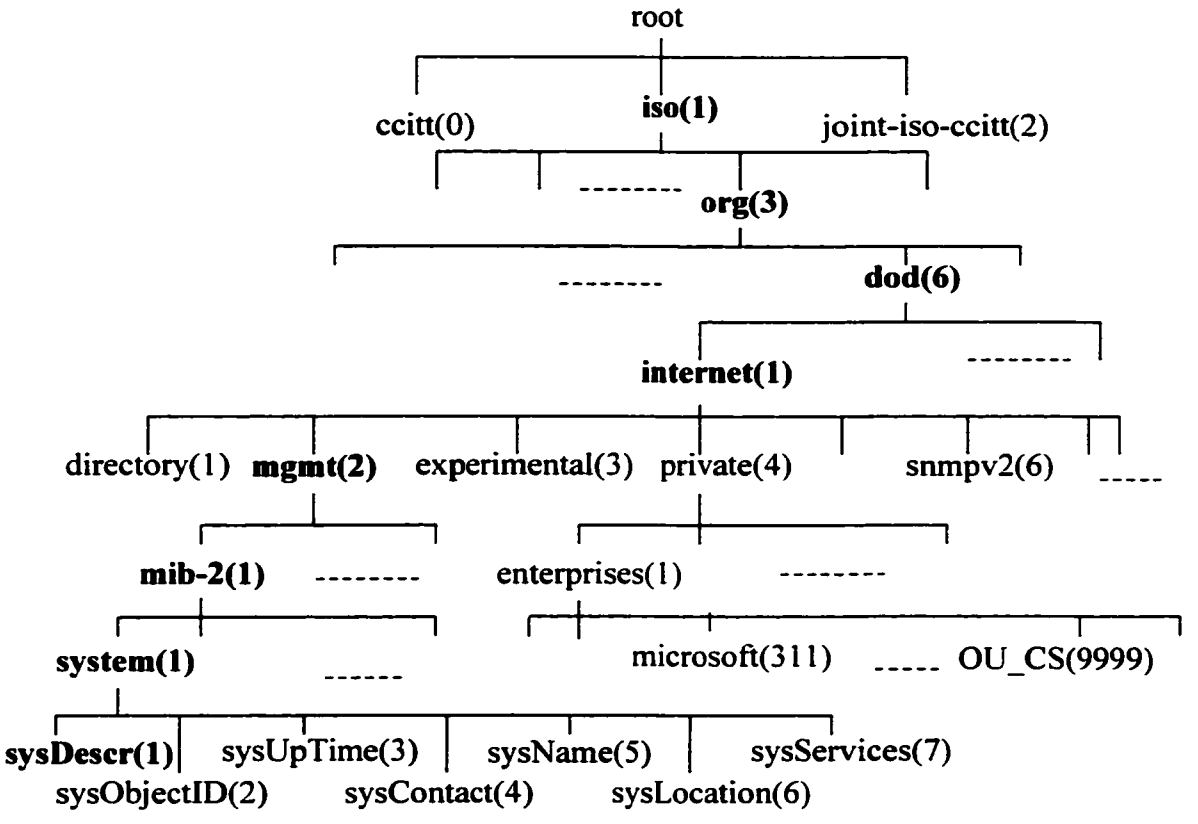


Fig. 3.3 OID directory tree structure and the managed objects under *system* group (1.3.6.1.2.1.1)

3.5.2 Management Information Base (MIB)

The MIB defines the management information to be monitored and controlled in the TCP/IP-based network. The first version MIB-I [MR90] is classified as *historic* and the current one is MIB-II [MR91]. In a MIB, related managed objects are organized into a group. Each group can be further subdivided into subgroup or subsubgroup, if necessary. The MIB-II contains the following essential groups and some of their managed objects:

Group	**Managed Objects**
System	sysDescr, sysObjectID, sysUpTime, sysContact, sysName, sysLocation, and sysServices
Interfaces	ifIndex, ifDescr, ifType, etc.
Address Translation	atTable, atEntry, atPhysAddress, and atNetAddress
IP	ipForwarding, ipInHdrErrors, ipInAddrErrors, etc,.
ICMP	icmpInMsgs, icmpInDestUnreachs, icmpInTimeExcds, etc.,
TCP	tcpRtoAlgorithm, tcpMaxConn, tcpActiveOpens, etc.,
UDP	udpInDatagrams, udpNoPorts, updInErrors, and udpOutDatagrams
EGP	egpInMsgs, egpInErrors, egpOutMsgs, etc.,

3.5.3 MIB module

A collection of ASN.1 descriptions relating to a common theme is named a *module*. A module has the following syntax [R94]:

<<module>> DEFINITIONS ::=

BEGIN

<<linkage>> -- some IMPORTS and/or EXPORTS

<<module identity definition>>

<<declarations>>

END

The <<module>> is the name of the module. Since some modules might be used in other MIB, the module name should be unique. Thus, modules can EXPORT definitions for use by other modules, which in turn IMPORT them. The <<declarations>> term contains the definitions of all node and leaf objects.

Three kinds of objects are defined using ASN.1:

- *types*, which define new data structures;
- *values*, which are instances (variables) of a type; and,
- *macros*, which are used to change the actual grammar of the ASN.1 language.

An example of the module identify is as follows.

```
mynewMibModule    MODULE-IDENTITY
        LAST-UPDATED "200204081300Z"
        ORGANIZATION    "Univ-Oklahome-CS-Dept"
        CONTACT-INFO    "Prof. K. T. Email: kt_example@ou.edu"
        DESCRIPTION     "Experimental MIB only"
::= {enterprises 9999}
```

The keywords of the ASN.1 language appear entirely in uppercase. Comments in ASN.1 start with two consecutive dashes ("--") and continue until reaching another two dashes or the end of the line A detail description of MIB file format, syntax, and how to write a MIB can be found in [R94] [PM97].

3.6 MIB for the ML-ADSD Algorithm

```
MLADSD-MIB-DEF DEFINITIONS ::=
BEGIN

IMPORTS     enterprise      FROM RFC1155-SMI

MLADSD-MIB      MODULE-IDENTITY
    LAST-UPDATED  "0106270000Z"
    ORGANIZATION  "University of Oklahoma"
    CONTACT-INFO
            "Ming-Shan Su, K. Thulasiraman and Anindya Das
             University of Oklahoma – School of Computer Science
             Phone: 2-405-325-0566
             Email: {mssu, thulasi, das}@ou.edu"

    REVISION        "0105120000Z"
    DESCRIPTION
            "The MIB module for mapping the testedUp array diagnosis information
            in the ML-ADSD algorithm."
    ::= {enterprise 9999}

testedUp  OBJECT-TYPE
    SYSTAX          SEQUENCE OF TestedUpEntry
    ACCESS          not-accessible
    STATUS          current
    DESCRIPTION
            "In the ML-ADSD algorithm, a node uses a two-dimensional array called
            TESTED_UP to update the testing results of the network."
    ::= {MLADSD-MIB 1}

testedUpEntry  OBJECT-TYPE
    SYNTAX          TestedUpEntry           -- this is a new type
    ACCESS          not-accessible
    STATUS          current
    INDEX           {testerD, clusterID}
    DESCRIPTION
            "Each entry testedUp[u][k] = v at a node n_{i, j} means that node i in cluster j
            has received a diagnosis message from a neighbor node (which it has tested
            as fault-free) indicating node u in cluster k has tested node v in cluster k and
            found node v as fault-free."
    ::= {testedUP 1}

TestedUPEntry ::= SEQUENCE {
    testerID      INTEGER,
    clusterID     INTEGER,
```

```
    testedID        INTEGER
}

testerID      OBJECT-TYPE
    SYNTAX          INTEGER
    ACCESS          read-only
    STATUS          current
    DESCRIPTION
        "The tester node ID index of the TestedUp array also indexes the table."
    ::= {testedUpEntry 1}

clusterID     OBJECT-TYPE
    SYNTAX          INTEGER
    ACCESS          read-only
    STATUS          current
    DESCRIPTION
        "The cluster index of the TestedUp array also indexed the table."
    ::= {testedUpEntry 2}

testedID OBJECT-TYPE
    SYNTAX          INTEGER
    ACCESS          read-write
    STATUS          current
    DESCRIPTION
        "If the node ID of a test result is -1, it means that the entry is arbitrary. "
    ::= {testedUpEntry 3}
END
```

3.7 SNMP Protocol

The SNMP protocol is a request-and-response protocol. It defines the functions of the base operations of SNMP and the format of the messages exchanged by management systems and agents.

3.7.1 Four Simple yet Powerful Operations

SNMP is based on a simple philosophy: it defines only four operations to perform all network functions between a management system and a managed node. These four

operations ***Get***, ***GetNext***, ***Set***, and ***Trap*** can monitor (read) and alter (write) the values of the managed objects which are maintained by the managed node.

The ***Get*** operation retrieves (i.e., read, fetch, or query) the value of a given managed object. For example, part of the managed objects in system group in my machine are:

Name	ID	Value
sysContact	1.3.6.1.2.1.1.4.0	Ming-Shan Su
sysName	1.3.6.1.2.1.1.5.0	KTGroup-WinPC1
sysLocation	1.3.6.1.2.1.1.6.0	KTGroup-Rm104

If we execute "***Get sysName***", then the value "***KTGroup-WinPC1***" will be received.

The ***GetNext*** operation retrieves the value of the managed object lexicographically next to the object specified in the GetNext operation. For example,

if we execute a "***GetNext sysName***", then the value "***KTGroup-Rm104***" will be received since sysLocation is lexicographically next to sysName.

The ***Set*** operation modifies the value of a managed object. For example,

if we execute a "***Set on sysLocation to KTGroup-Rm106***", then the new value stored in sysLocation will be "***KTGroup-Rm106***".

The ***Trap*** operation is an unsolicited notification operation. Once an event occurred, the agent will generate a trap message and inform one or more designated management stations about the event. For example, if a *warmstart* event occurred, then a trap message will be sent to the management station(s).

The wide popularity of SNMP is because of these four simple but powerful operations.

3.7.2 Message Exchange between Management System and Managed Node

Typically the message exchange between a management station and a managed node is as follows. A management station sends a Get (GetNext, or Set) request message to the managed node. The managed node responds by sending a GetResponse message back to the management station. As for the Trap message, it is a one-way trip message. Once an event occurs in a managed node, the node will send a Trap message to its designated management station and it is not required for the management station to send any message back to the managed node. One thing worth mentioning is that since SNMP uses UDP (User Datagram Protocol), there is no guarantee that the messages sent will reach the destination. It is up to the message sender's responsibility to handle this situation. A pictorial illustration of the message exchange is shown in Figure 3.4.

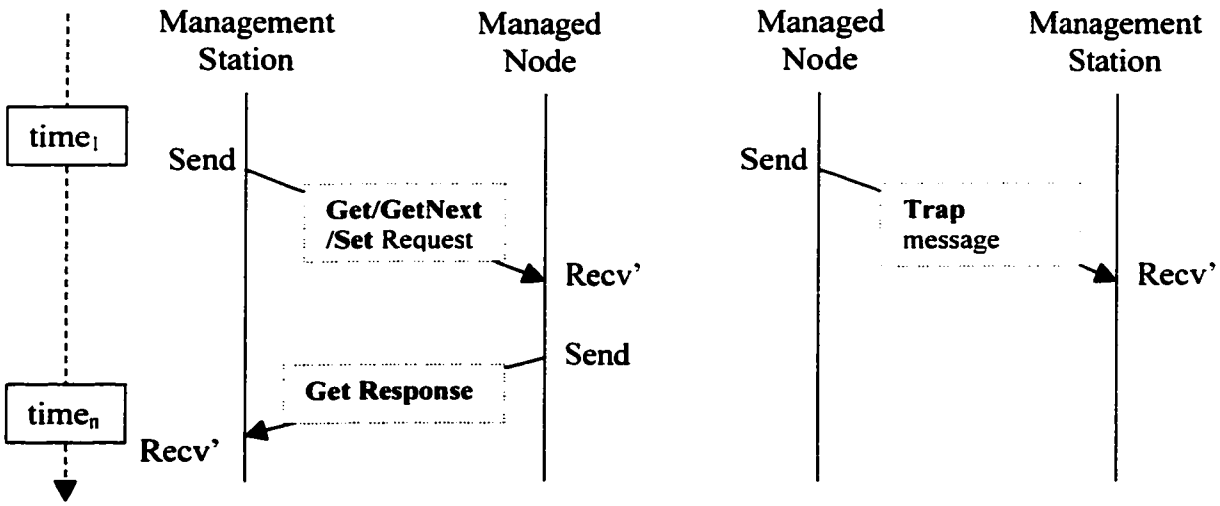


Fig. 3.4 Message exchange between management station and managed node

3.7.3 SNMP message

An SNMP message is encapsulated in the payload portion of an UDP (User Datagram Protocol) datagram (as in Figure 3.5) and then exchanged between Agents and Managers over a TCP/IP network. Each SNMP message also uses the Basic Encoding Rule (BER) to encode the message.

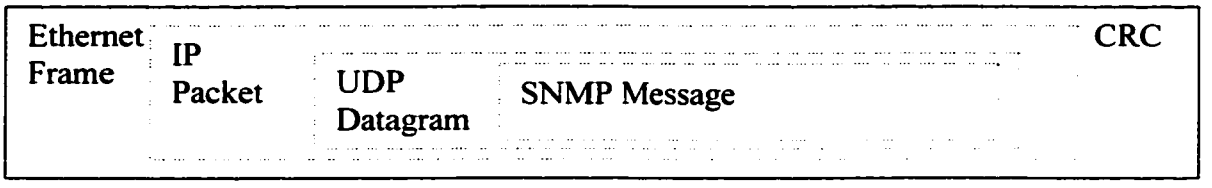


Fig. 3.5 Ethernet packet format

In general, an SNMP message format consists of two parts (as in Figure 3.6), the Preamble and PDU (Application Protocol Data Unit also termed SNMP PDU.)

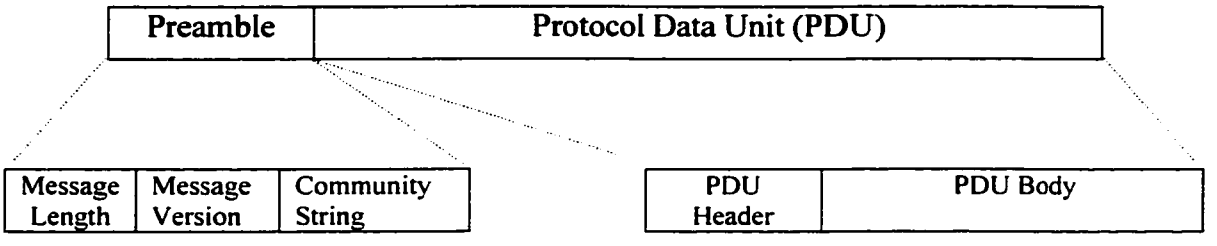


Fig. 3.6 Preamble and PDU format of SNMP message

SNMPv1 [CFSD90] defines five different PDUs: GetRequest, GetNextRequest, SetRequest, GetResponse, and Trap PDUs. The format of those PDUs can be categorized into two types as in Figures 3.7(a) and 3.7(b).

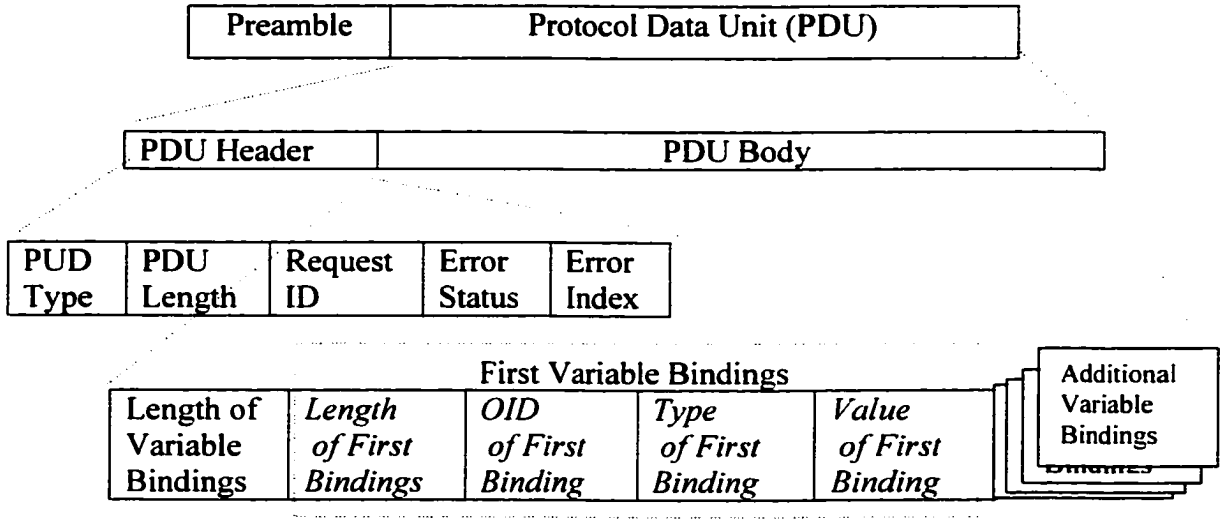


Fig. 3.7(a) GetRequest, GetNextRequest, SetReqeust, and GetRespose PDUs

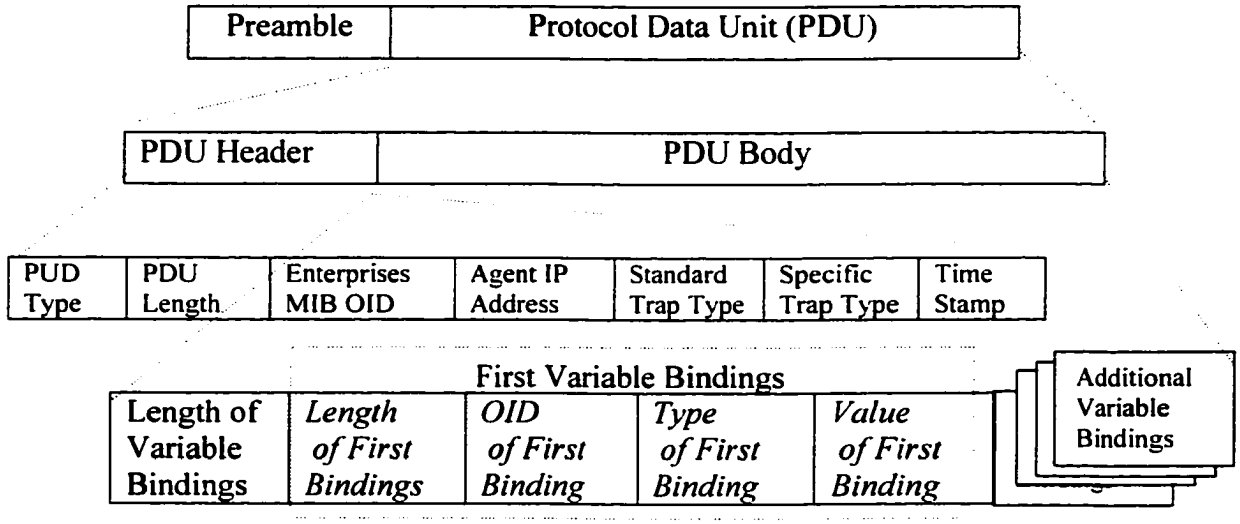


Fig. 3.7(b) Trap PDU

Some important fields in the message are explained below:

- Community String: serves security checking purpose
- PDU Header: used for the error checking
- PDU Body: the request or response data to and from agent.

An example of a GetRequest query from manager to agent is as follows:

SnmpUtil GetRequest midway.cs.ou.edu public 1.4.0

The manager sends a *GetRequest* (e.g., retrieve) query to an agent residing in machine named *midway.cs.ou.edu*. The password is *public* and the variable name is *system.sysContact* with alias *1.4.0*. The request and response data in the messages [M97] are as in Figure 3.8.

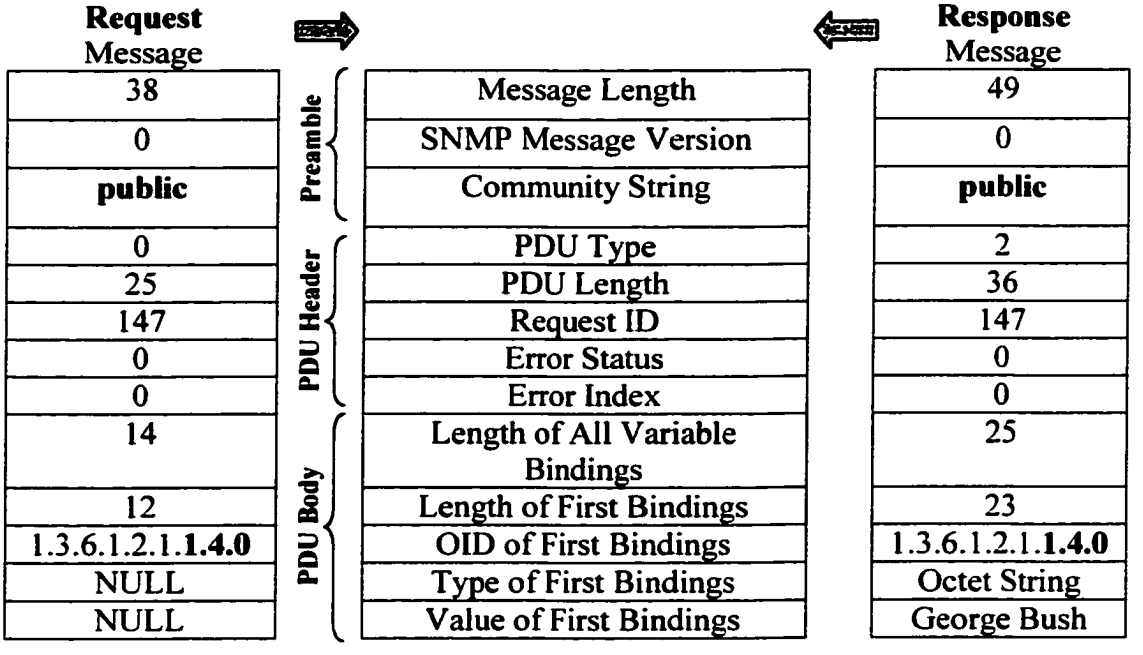


Fig. 3.8 Contents of a SNMP *Get* and its *Response* messages

3.7.4 Administrative Policy

SNMP defines a *community* to be a relationship between SNMP entities. The *community* also serve as a security mechanism.

When SNMP messages are exchanged, they contain two parts:

- a *community string*, sent in plain text; and,

- *data*, containing an SNMP operation and associated operands.

The agent will do the following checking before responding to the manager:

- first identify the collection of managed object resources to be monitored or controlled.
- then determine which SNMP operations may be performed on them. This is termed an *access policy*, and is used to control the flow of information between an SNMP agent entity and a given management application entity.

If the *community* name is correct and the operation is appropriate on the managed objects (e.g., a SET operation should issue on an object which its ACCESS attribute is Read-Write), then the proper response will be responded.

3.8 Distributed Fault Detection: Integration of the ML-ADSD Algorithm and SNMP

The primary application of this our effort in this thesis is to develop and implement a prototype network fault detection/monitoring system by integrating the ML-ADSD algorithm into the SNMP-based fault management network [RM90] [MR91] [CFSD90]. As mentioned earlier, the Simple Network Management Protocol was developed by IETF in 1988 for the purpose of managing the network devices over a computer network and has been widely adapted by industry on network applications. The major drawback of this SNMP-base fault management is the central management system. The problems include a single point of failure, lack of scalability, and high communication costs around the central manager. Through our application, we want to

show that some of the above problems can be solved to some extent and that the improvement of fault management through distributed diagnosis is feasible.

The operation platform we use to implement our integrated management software is Microsoft's Windows systems (including Windows 95, 98, NT, 2000, and ME). Three major tasks, **Generate the MIB, Setup the Agent, and Setup the Manager**, are required to implement the management software.

As mentioned in section 3.4, a node can have serve in dual roles, as a network management station (NMS or manager) as well as a managed node (agent). In our implementation, we set up each machine to have this property too. Since in SNMP applications, the manager and agent are two different software applications running independently, the software codes for manager and agent are written differently and they have to be compiled using different SNMP interfaces (libraries).

Figure 3.9 shows how a node i tests node i+1 and requests i+1 to forward the management information. The manager i in node i does not test the manager i+1 in node i+1. Instead manger i tests agent i+1 as in step 1 (see Fig. 3.9) and requests agent i+1 to forward the management information as in step 2 (see Fig. 3.9). Then manager i updates the information (e.g., TESTED_UP array) and stores the information in agent i as in step 3 (see Fig. 3.9). The whole distributed diagnosis process proceeds in this fashion.

In our implementation, the agent when queried just returns the information in TESTED_UP array. However, in some commercial software the function of an agent can be extended to perform some operation and then return the value stored in database (e.g., Oracle database) when queried.

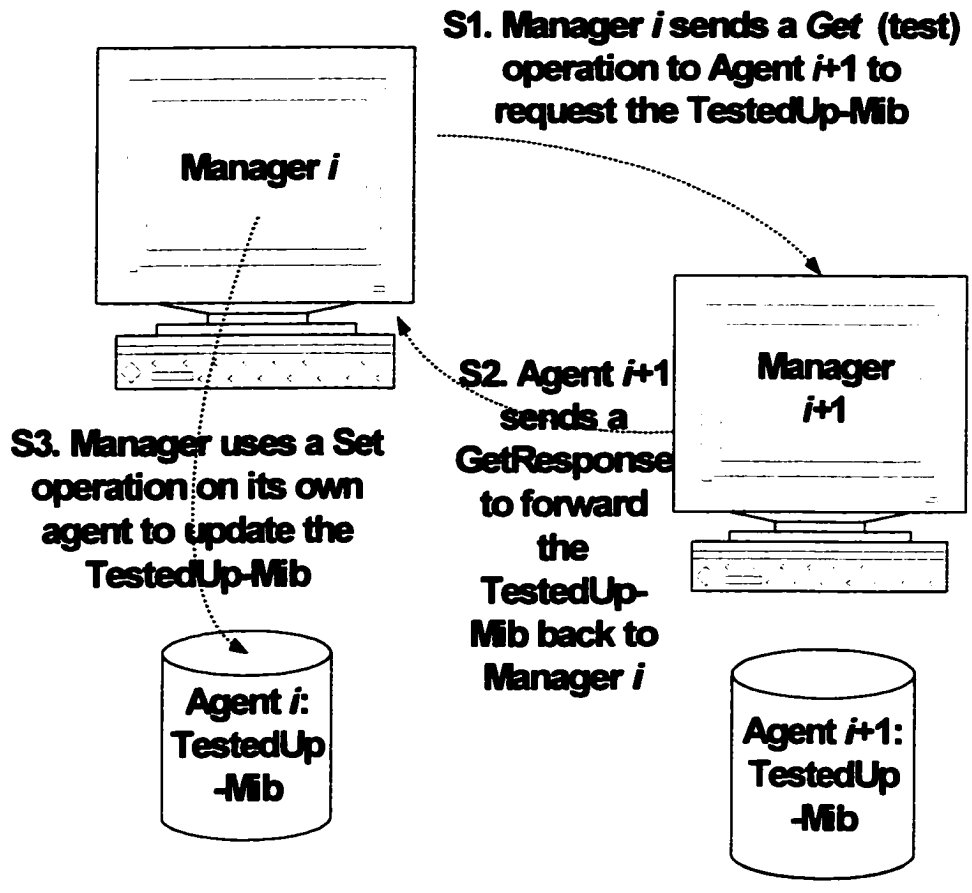


Fig. 3.9 Node *i* tests node *i*+1 and requests *i*+1 to forward the management information

3.8.1 How to Generate the Agent Program

- **Generation of the MIB**

The process to generate a user defined MIB is illustrated in Figure 3.10. Steps 1, 5, 6, and 7 are to be implemented by the user. The other steps employ the third party software.

Step 1. Use the SMI language syntax to define the managed object(s) (testedUp-MIB in our case).

Step 2. Use a MIB compiler (e.g., Microsoft MIBcc.exe) to compile the testedUp-MIB to make sure the syntax is correct. And if it is correct, a temporary code (e.g., testedUpMIBTemp) will be generated.

Step 3. and Step 4. Use a MIB C (can be C++, or Java) code generator toolkit to read the testedUpMib.

Step 5. Generate a Template C code (e.g., testedUpMIBTempC) for each managed object.

Step 6. Add or modify the testedUpMIBTempC in step 5 to fit our needs and call the codes as testedUpMIBFinalC. The reason for being called a Template code is that it is like a new user defined data type for each managed object defined. For example, we can define a new data type or called structure in *C* which includes name, student ID, major, and address, etc., for a student. But we can only specify how many such different types of structures are needed for each different managed object in step 6. There is no indication that how many managed objects will use that type.

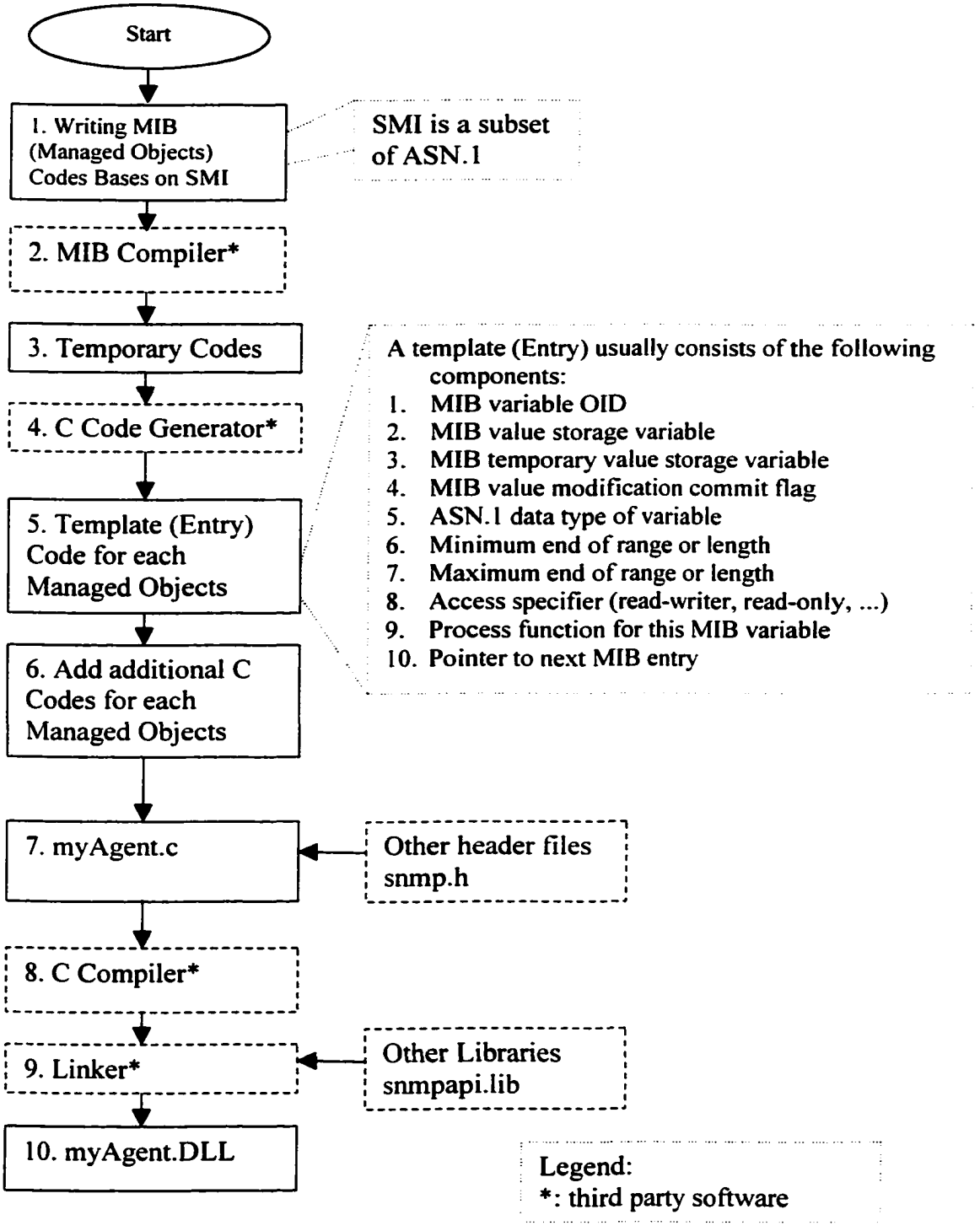


Fig. 3.10 Process to build an SNMP agent on Windows platform

- **Generation of the MIB Dynamic Link Library Code**

The process to generate the MIB dynamic link library is illustrated in steps 7 to 10 in Figure 3.10.

Step 7. Include the required snmp.h header file in our testedUpMIBFinalC code.

Steps 8 and 9. Use a C compiler and a snmpapi.lib to compile and link the testedUpMIBFinalC to generate the testedUpMIB.DLL dynamic library code. The next task is to setup the agent so it can run the testedUpMIB.DLL.

- **How to Setup the Agent Program**

The Agent in Windows system is called Extendible Agent and it is implemented by dynamically linking to Extension Agent DLLs [M98] that implement portions of the MIB. These Extension Agents are configured in the Windows NT Registration Database. When the Extendible Agent Service is started, it queries the registry to determine which

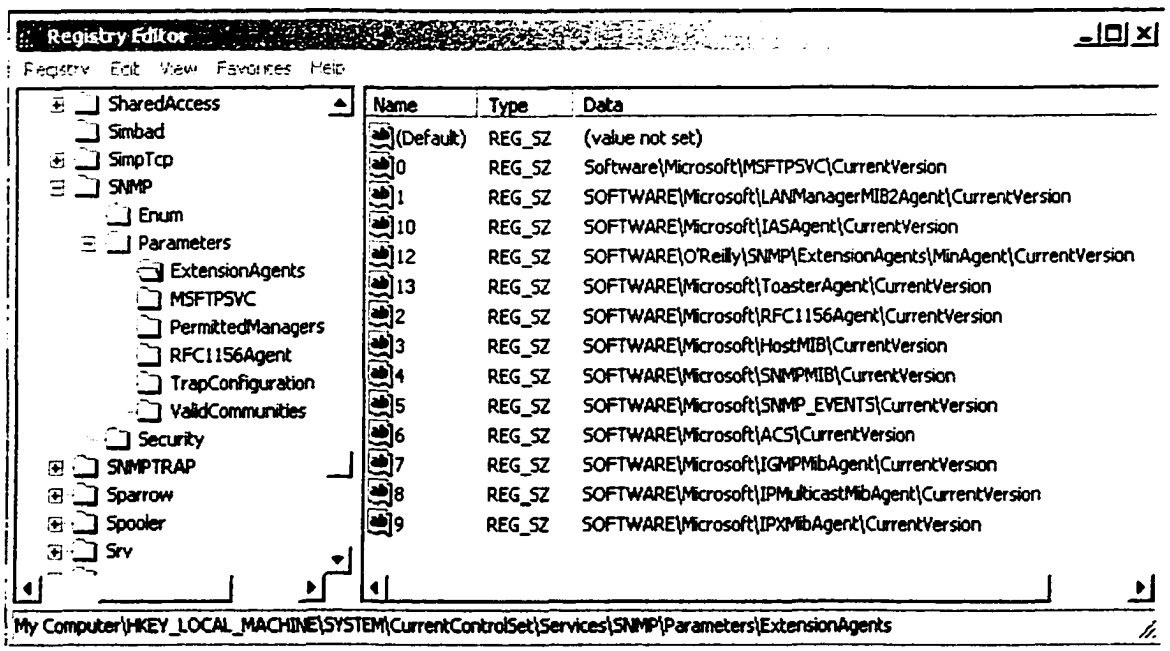


Fig. 3.11 Inclusion of testedUpAgent into Registry

Extension Agent DLLs have been installed (see Figure 3.11) and need to be loaded and initialized. The Extendible Agent invokes various DLL entry points (see Figure 3.12) to request MIB queries and obtains Extension Agent generated traps.

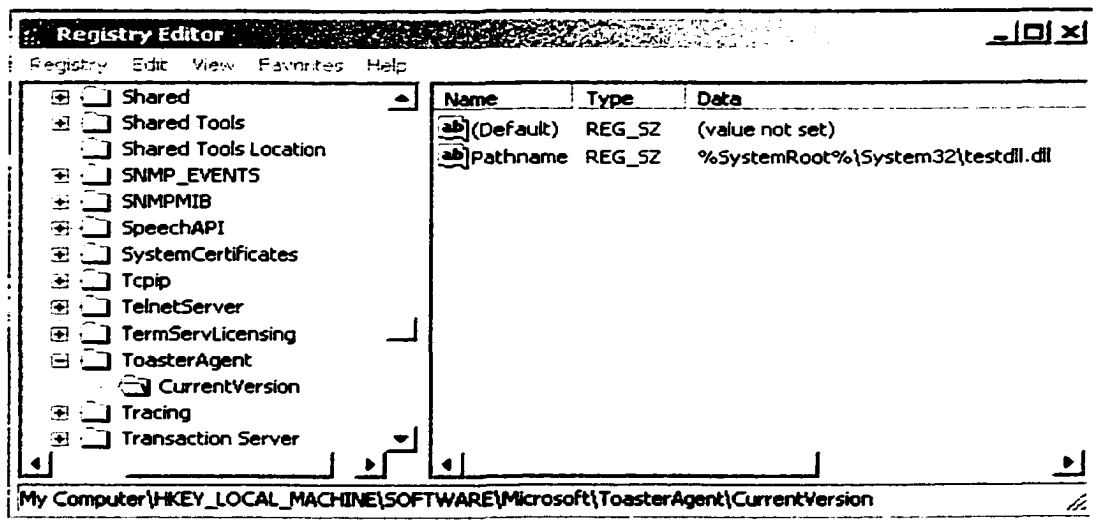


Fig. 3.12 Inclusion of the testedUpdll entry into Registry

- **Registering the testedUpMIB.dll in Windows Registry:**

Step 1. Use a mib compiler mibcc.exe to compile the required MIBs and create the new mib.bin file.

A MIB compiler converts a human-readable MIB module text file into a data format that can be read more easily by the SNMP Management API. Microsoft's mibcc.exe (in NT Resource Kits) is used to create the MIB.BIN file. The command to generate the new mib.bin is as follows:

MIBCC SMI.MIB MIB-II.MIB LMMIB2.MIB WINS.MIB DHCP.MIB INETSRV.MIB FTP.MIB GOPHERD.MIB HTTP.MIB **TestedUp.MIB**

The mib.bin file itself is a binary record format file that contains the compiled collection of MIBs. This file is read by the management API when it needs to use symbolic names (e.g., system.sysContact.0) instead of numbers (.1.3.6.1.2.1.1.4.0) used on the local machine.

Step 2. Stop the SNMP service ("net stop SNMP", or "SNMP -stop" in Win95).

Step 3. Rename %SYSTEMROOT%\SYSTEM32\MIB.BIN to MIB.OLD.

Step 4. Copy the following files to the %SYSTEMROOT%\SYSTEM32 folder:

- MIB.BIN (created after you ran MIBCC.EXE)
- testedUpMIB.DLL (This is the extension dll that will process the SNMP requests and return the testedUp array date stored in local machine).

Step 5. Register testedUpMIB in the Registry by adding the following entries to the registry manually (as in Figures 3.11 and 3.12).

```
\HKEY_LOCAL_MACHINE
    \Software
        \OU_CS
            \TestedUpAgent
                \CurrentVersion
                    \Pathname = REG_EXPAND_SZ
                        %SystemRoot%\System32\testedUpmib.dll

\HKEY_LOCAL_MACHINE
        \System
            \CurrentControlSet
                \Services
                    \SNMP
                        \Parameters
                            \ExtensionAgents
                \OU_CSAgent =
                  SOFTWARE\OU_CS\TestedUpAgent\CurrentVersion
```

Step 6. Restart the SNMP service ("net start SNMP" or "SNMP" in Win95).

Step 7. Now the management program snmpMgr.exe should be able to query (request or test) the extended agent agent.dll to get (or forward) the data stored in testedUpMIB.

Note: Once the testedUpMIB has been compiled in a NT machine, there is no need to repeat the process for other Windows machines. Just copy the required files to other machines and add new entries and values in the registry, and then stop and re-start the SNMP service.

- **Interaction between SNMP and Agents**

The interaction between SNMP service and Agent DLLs is illustrated in Figure 3.13. When a manager queries the SNMP agent, the query will go to SNMP service first, then SNMP will pass the query to the appropriate extended agent, e.g., testedUpMib.dll. The testedUpMib.dll then communicates with snmpapi.dll agent for some required library functions, retrieves the testedUp array data and gives to the SNMP service. The SNMP service then sends back the query to the manager.

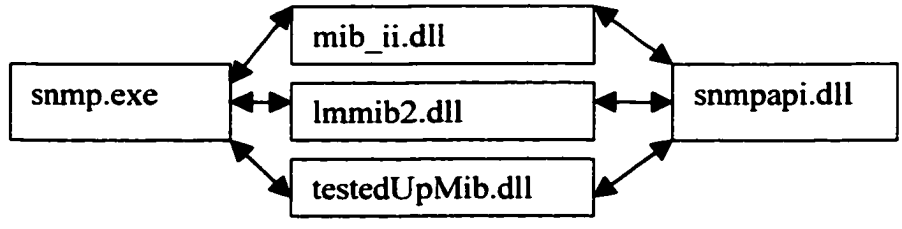


Fig. 3.13 Interactions between SNMP service and agent DLLs

3.8.2 Setting up the Manager

- **Generation of the Manager Executable Program**

The process to build a manager application program is illustrated in Figure 3.14.

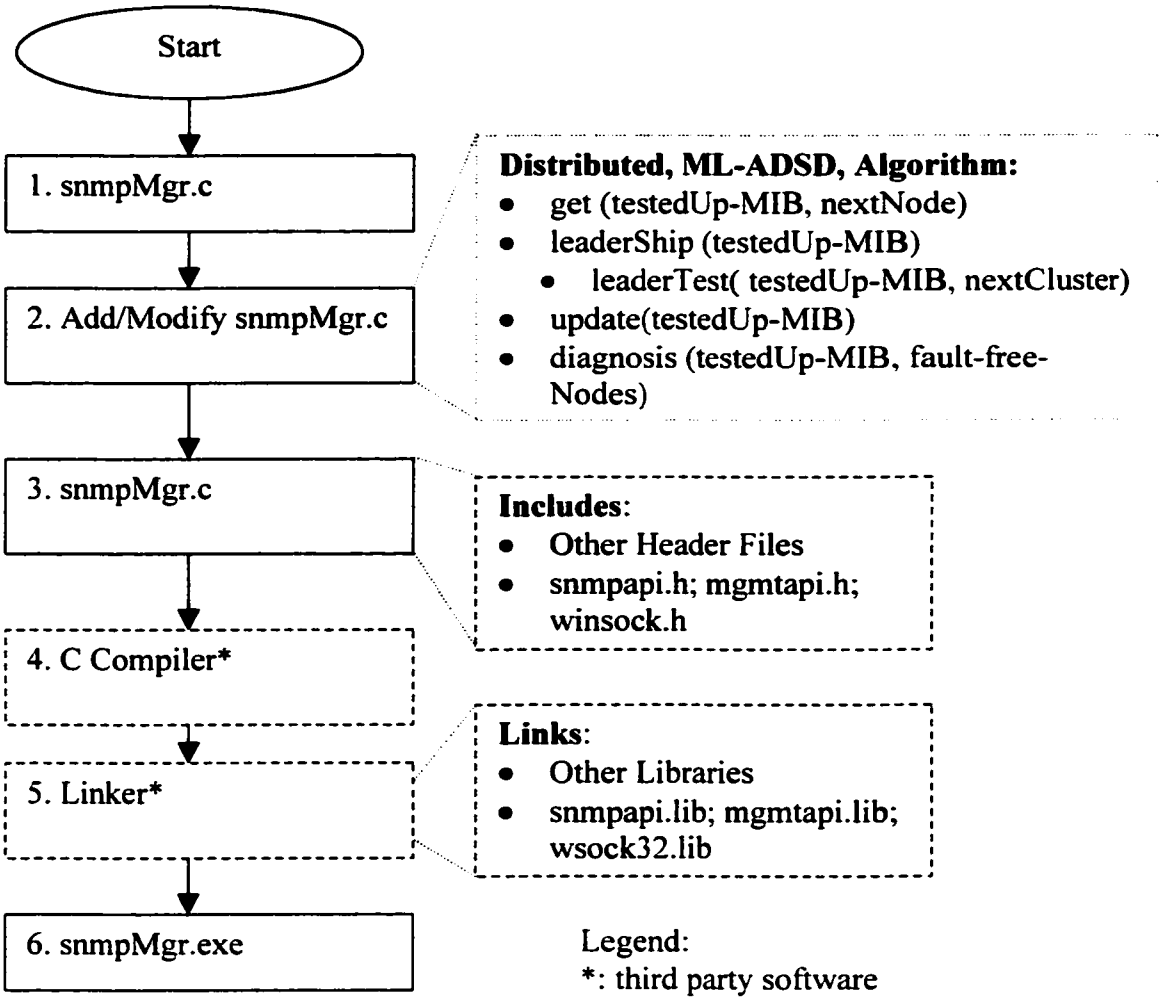


Fig. 3.14 Process to build an SNMP management application

Steps 1, 2, and 3 are to be implemented by the user. The other steps employ some third part software.

Step 1. Use a manager sample program, snmputil.c, from Microsoft's Visual C++ software and rename it as snmpMgr.c.

Step 2. Add our ML-ADSD algorithm into the program and modify the program to suit our needs. In snmpMgr.c, we first convert all the pseudo subroutines or algorithms of ML-ADSD into C codes, such as *leader Election, Cycle Detection, Level-1 and Level-2 Testing, and Diagnose Algorithms*. The update TestedUp array part has to be written by the user. (i.e., use the "*SET*" operation to write the local testedUpMIB).

Step 3.Include the necessary header files into snmpMgr.c, such as snmp.h, mgmtapi.h, winsock.h, so that it will not have compilation errors.

Steps 4 and 5. First add some required function libraries for the linker (as in Figure 3.15), such as snmpapi.lib, mgmtapi.lib, and wsock32.lib.

Step 6.Compile the program snmpMgr.c to generate snmpMgr.exe executable program.

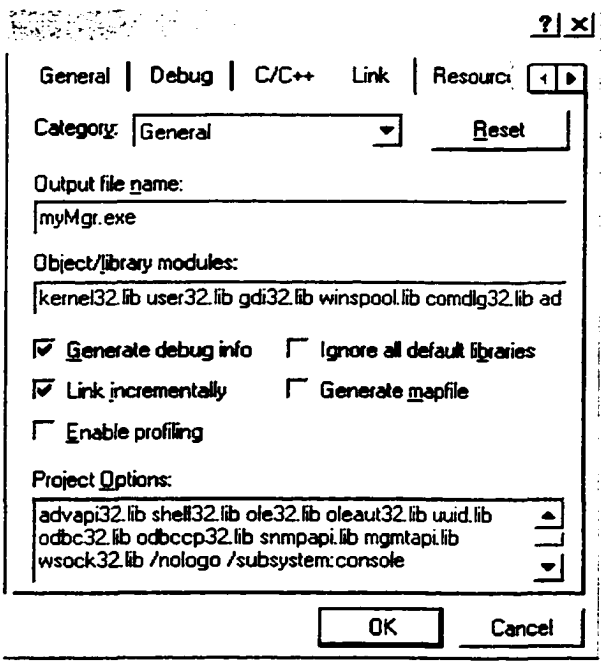


Fig. 3.15 Libraries for linker

In step 2, the ***request*** $n_{y,a}$ ***to forward TESTED_UP***$_{y,a}$ ***to*** $n_{x,a}$ (e.g, line 4 in Figure3.16) is similar to the following functions from MGMTAPI.LIB:

```
y = x;
repeat {
    y = (y + 1) mod N/p;
    n_y,a Session = SnmpMgrOpen( n_y,a IPaddress, CommunityName, Timeout, Retries);
    • • • // some user codes, e.g., SetUpVariableBindings( __, __, TestedUpOID, __);
    SNMPMgrRequest( n_y,a Session, GetRequest, VariableBindings, ErrorStatus, ErrorIndex);
    • • • // some user codes, e.g., n_y,a FaultFree = CheckVariableBindings( VariableBindings);
} until ( n_y,a FaultFree )
```

The ***Update*** process (e.g., lines 12-15 in Figure 3.16) is similar to the following functions:

```
n_x,a Session = SnmpMgrOpen( n_x,a IPaddress, CommunityName, Timeout, Retries);
• • • // some user codes, e.g., SetUpVariableBindings( __, __, TestedUpOID, __);
SNMPMgrRequest( n_x,a Session, SetRequest, VariableBindings, ErrorStatus, ErrorIndex);
```

```
/* Regular node, Level-1 Testing Algorithm at node n_x,a */

1.  y = x          /* assign my node id */
2.  repeat {
3.      y = (y + 1) mod N/p
4.      request n_y,a to forward TESTED_UP_y,a to n_x,a
5.  } until (n_x,a tests n_y,a as "fault-free")

6.  for node = 0 to (N/p – 1)          // update local cluster info.
7.      TESTED_UP_x,a[node][a] = TESTED_UP_y,a[node][a]

8.  TESTED_UP_x,a[x][a] = y            // x itself tests y as fault free

9.  If ( Cluster_leader( ) == True )   // check the leadership
10.     status(x) = Leader
11. Else          // update info. regarding other clusters
12.     for cluster = 0 to (p – 1)
13.       for node = 0 to (N/p – 1)
14.         if (cluster ≠ a)
15.                 TESTED_UP_x,a[node][cluster] =
                    TESTED_UP_y,a[node][cluster]

16. Stop. // end of Level-1 testing
```

Fig. 3.16 Level-1 testing algorithm

3.8.3 Installation of the Proper Dynamic Library to Run Manager Program

In Windows 95, SNMP service is designed as an Agent application. In order to run the manager application, the following programs have to be installed and updated.

- To Set up Windows 95:

1. Update the Windows 95 sockets to Winsock version 2 by running **W95ws2setup.exe.** (from Microsoft download web site)
2. Setup SNMP service by running the self-extracting file **Snmppz.exe** first, then follow the procedures in WinNT Book [M98].
3. Copy **mgmtapi.dll** (newer version of file size 18KB) to directory c:\windows\system
4. Copy any Mib.bin to c:\windows\system.
5. Run **snmpMgr.exe** to start the distributed diagnosis.

- **To Set up Win98/2000/ME,**

Use steps 2, 3, 4, and 5 as for Windows 95.

- **To Set up WinNT,**

Follow the WinNT Book [M98].

3.8.4 Interaction between Manager and SNMP Service

The interaction between manager program and snmp agents is illustrated in Figure 3.17. The Manager program, e.g., snmpMgr.exe sends the query to snmp service over the agent side through the application program interface libraries snmpapi.dll and mgmtapi.dll. The snmp service then sends the query to the appropriate agent (as in Figure 3.12) and sends back the answer once the agent returns the response.

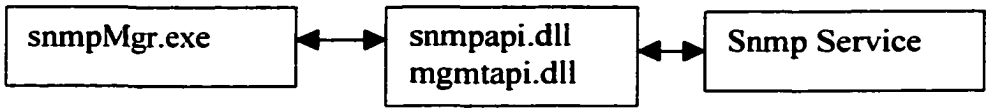


Fig. 3.17 Interaction between manager and SNMP service

3.9 Experimental Results

We developed the network fault detection system described in the previous section and implemented it on a 10/100 Mbps Ethernet Local Area Network consisting of 16 machines (see Fig. 3.18) running Microsoft Windows operating systems (Windows 2000, NT, 95, and ME). Since robustness and diagnosis latency of the ML-ADSD algorithm are our main focus, we tested the fault detection system extensively with different combinations of parameter settings as explained later. Since the total number of machines in our experimental system is small, we have used the 2-level diagnosis scheme of section 2.3 in the design of the fault detection system.

In our experiments we used two parameters:

- *Waiting time* (in seconds) between test intervals
- *Maximum number of faults* during each run of an experiment

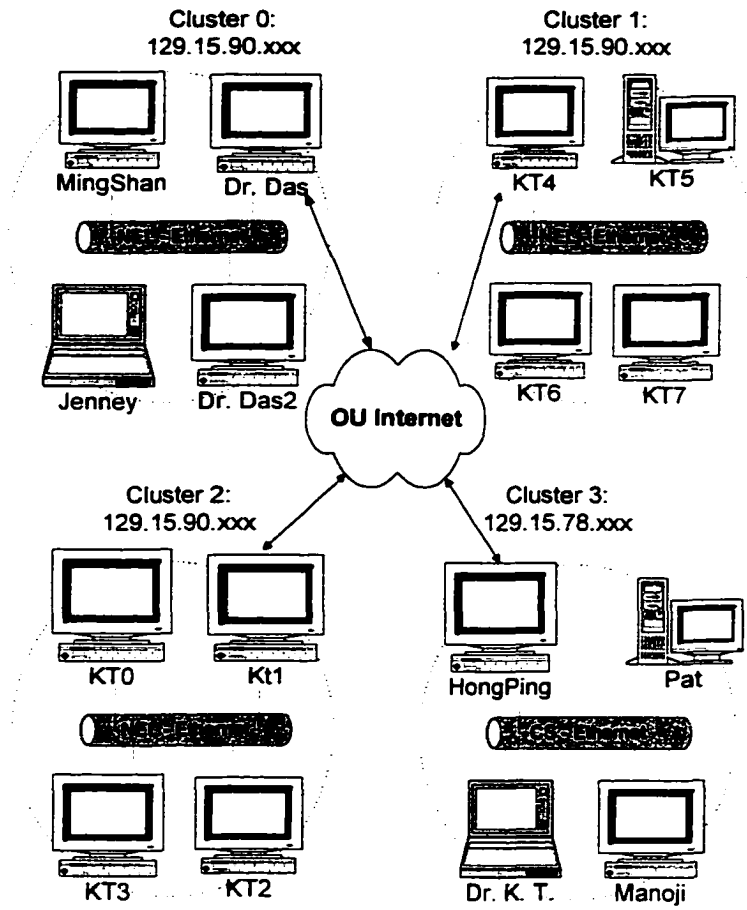


Fig. 3.18 An Ethernet Network with 16 machines

An experiment corresponds to a setting of the above parameters. Each experiment consists of 20 runs of the diagnosis algorithm with 20 randomly generated fault events injected during a run. Recall that diagnosis latency refers to the time taken for all fault free nodes to reach a consensus view of the fault/free-free status of the whole network. The diagnosis latency is recorded after the last fault event (failure or recovery) in each run. The average diagnosis latency over the 20 runs of an experiment gives the average diagnosis latency for that experiment.

Initially, for each run a file of 20 randomly faulty events are created. The file specifies the time and type of fault event. For example, a typical file can be as in Figure 3.19. The fault events as specified by the file are injected into the system in the course of a run of an experiment. In addition, among those 20 events in each run, many nodes can fail or recover at the same time (as long as the maximum number of faulty nodes allowed in the setting is not exceeded at any time). However, the time between the failure and recovery (or vice versa) of the same node has to be at least a few seconds since a node does not fail and then recover (or vice versa) at the same time in a computer network

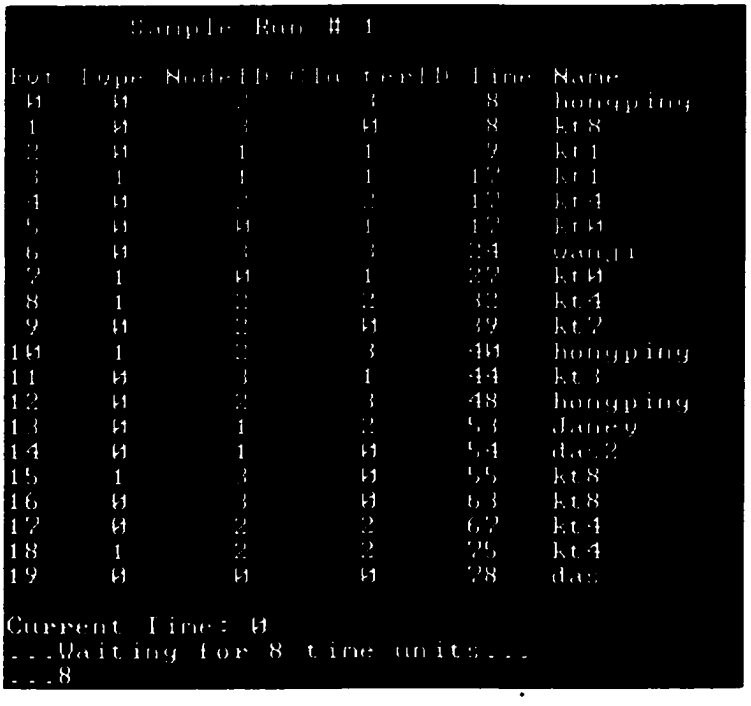


Fig. 3.19 Sample events file

Additionally, since all the machines are running in a distributed and independent manner, a virtual tester (see Fig. 3.20) is used to inject the events (faulty or recovery). The tester also compares the TestedUp-Mib in each fault free machine after the last event, and calculating and recording the diagnosis latency when all the fault free

machines have reached a consensus on the fault status of all the machines in the network. Note that a tester is not required in the real management system and we use it here just for the experiment purpose.

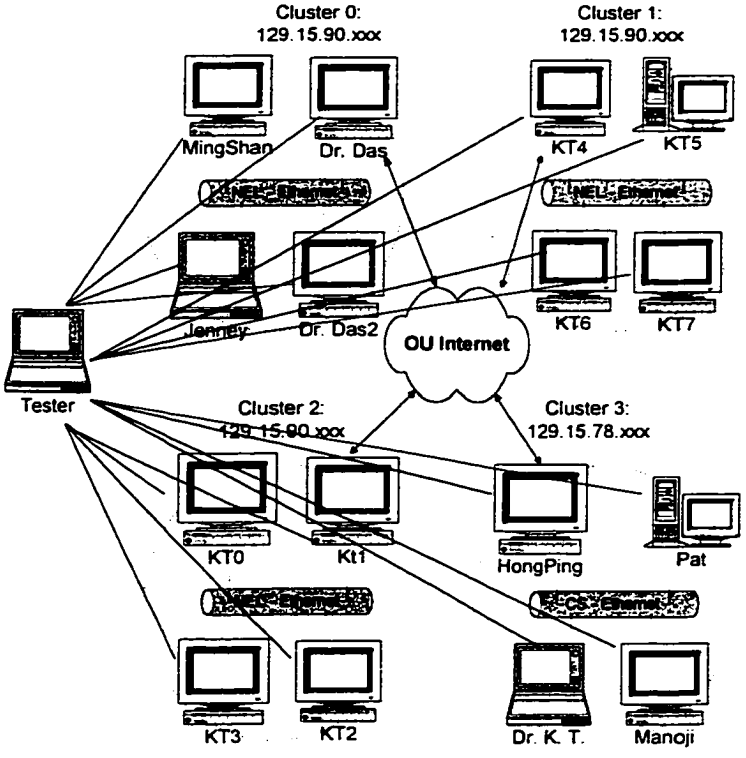


Fig. 3.20 A virtual tester is used to inject the events

The results of 30 experiments are reported in Table 3.1.

Table 3.1 Diagnosis latency (in seconds) of the 30 experiments on Two-Level ML-ADSD

		maximum number of faults / per run				
		4	5	6	7	8
waiting time (seconds)	3	20.45	19.20	19.85	18.45	18.35
	4	27.95	27.25	24.50	25.00	23.20
	5	32.50	33.60	29.20	31.00	29.20
	6	39.40	36.25	38.50	36.55	34.30
	7	46.90	45.70	43.70	41.90	39.15
	8	50.05	50.35	50.80	48.10	43.60

In Table 3.1, the diagnosis latency of 20.45 seconds at row 1 and column 1 of corresponds to the experiment with the waiting time set to 3 seconds and maximum number of faults / per run set to 4.

It can be seen from Table 3.1 and Figure 3.21 that the greater the number of faulty nodes in a system the smaller the diagnosis latency. We also observe that the diagnosis latency does not increase in proportion to the increase in waiting time. This could be because some of the faulty events may not be seen by a node, if the waiting interval is too large. In other words, the effect of some of these faulty events may not get propagated.

Diagnosis latency is an important piece of information for a network administrator. For example, if all the fault free nodes have to detect the last event within 30 seconds of the last event, then setting the waiting interval to 5 seconds is a good choice. Again, there is always a tradeoff between the diagnosis latency and the number of messages required by the network management system. Smaller diagnosis latency will require more testing messages which in turn would require more network bandwidth.

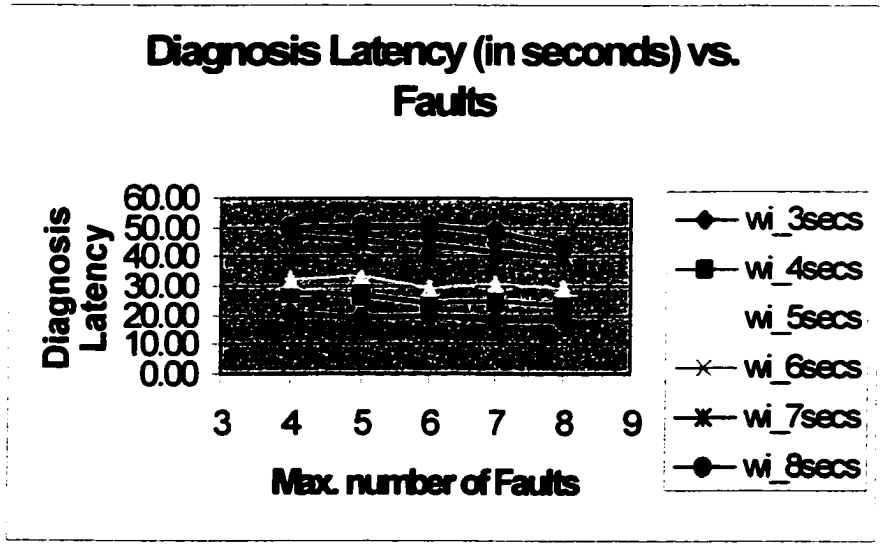


Fig. 3.21 Diagnosis latency (in seconds) vs. max. number of faults

Since ML-ADSD is a generalization of the ADSD algorithm, we also implemented and tested the ADSD algorithm on similar settings. The ADSD implementation results are shown in Table 3.2.

Table 3.2 Diagnosis latency (in seconds) of the 30 experiments on ADSD

Waiting Interval (seconds)	Maximum Number of Faults / run				
	4	5	6	7	8
3	21.65	21.10	20.55	18.70	19.00
4	27.45	25.35	24.75	24.85	21.25
5	33.95	32.55	30.95	29.50	27.45
6	41.35	39.10	37.75	35.95	33.30
7	48.05	44.30	36.40	37.10	36.95
8	54.50	51.60	51.90	43.90	42.95

Comparing the ML-ADSD and ADSD algorithms, for a network with 16 machines, it can be seen that if the number of faulty nodes in the system is small, then the ML-ADSD algorithm will have a better diagnosis latency. For example, with waiting interval of 8 seconds and a maximum of 4 faulty nodes, the diagnosis latency of ML-DSD is 50.05 while ADSD needs 54.5 seconds. However, if the number of faulty nodes is large in the system, such as 8 faulty nodes, then ML-ADSD needs 43.6 seconds while ADSD needs only 42.95. This result is expected since the ML-ADSD algorithm is designed to outperform ADSD for large networks. For networks of small sizes, the overhead involved in the ML-ADSD algorithm will result in a degradation of performance.

Chapter 4

Summary and Future Work

4.1 Summary of Research

Continuing advances in semiconductor technology have made possible the development of large computer systems comprising hundreds of thousands of processors or units. As the complexity and the computing power of these systems increase, fault tolerance and reliability become acute areas of concern. Yet it is impossible to build such systems without defects. As the size of a system grows, it is more likely to develop faults both in the manufacturing process and during the operation period. Testing of such systems becomes extremely difficult due to their large sizes. First, the complexity of test generation for such large systems is overwhelming. Second, the application of test data, and observation and analysis of test responses are extremely difficult and costly, even if test data could be generated. This problem may be further aggravated by possible geographical distribution of units. Testing of such systems with the traditional stimuli-supplying and responses-observing philosophy has become virtually impossible. Therefore, it is important for computing systems to have the capability to automatically detecting and identifying faulty components. In 1967, Preparata, Metze and Chien [PMC67] proposed a model and a framework, called ***System-Level Diagnosis***, for dealing with the above problem.

In the two decades following Preparata, Metze and Chien's pioneering work, a number of issues arising from the application of their framework were investigated and resolved. All these works assumed the existence of a single highly reliable supervisory

node to do the diagnosis. A single supervisory node is a bottleneck in a system with a large number of processing nodes. Distributed diagnosis algorithms which exploit the inherent parallelism available in a multiprocessor system would be desirable. With this in view, Kuhl and Reddy, in 1981, pioneered the area of distributed system level diagnosis.

Distributed diagnosis has been the focus of research in this thesis. There are two aspects to the contributions in this thesis: Design and performance evaluation of a new multi-level distributed diagnosis algorithm, and the design of a distributed network fault detection system based on the SNMP protocol.

In 1991, Bianchini and Buskens proposed an adaptive distributed algorithm to diagnose fully connected networks. This algorithm called the ADSD algorithm has a diagnosis latency of $O(N)$ for a network with N nodes. With a view to improving the diagnosis latency of the ADSD algorithm, in 1998 Duarte and Nanya proposed a hierarchical distributed diagnosis algorithm for fully connected networks. This algorithm called the Hi-ADSD algorithm has a diagnosis latency of $O(log^2 N)$. The Hi-ADSD algorithm can be viewed as a generalization of the ADSD algorithm.

In this thesis, we propose a new distributed diagnosis algorithm using the multilevel paradigm. This algorithm is a generalization of the ADSD algorithm. We present all details of the design and implementation of this multilevel adaptive distributed diagnosis algorithm called the ML-ADSD algorithm. We present extensive simulation results comparing the performance of these three algorithms.

Simulation results indicate that in all cases, the ML-ADSD algorithm is much better than the ADSD algorithm. In all cases the time required by the ML-ADSD algorithm is better than or the same as the time required by the Hi-ADSD algorithm. The

performance of the ML-ADSD can be tuned/improved, depending on the needs, by an appropriate choice of the number of clusters and the number of levels. The ML-ADSD algorithm is scalable in the sense that only some minor modifications will be required to adapt the algorithm to networks of varying sizes. This property is not shared by the Hi-ADSD algorithm.

The primary application of our research is to develop and implement a prototype network fault detection/monitoring system by integrating the ML-ADSD algorithm into a SNMP-based (Simple Network Management Protocol) fault management system [RM90] [CFSD90] [MR91]. We report the details of the design and implementation of such a distributed network fault detection system.

SNMP was developed by IETF in 1988 for the purpose of managing the network devices over a computer network and has been widely adopted by industry on network applications. The major drawback of SNMP-based fault management is its centralized nature. The resulting problems include a single point of failure, lack of scalability, and high communication costs around the central manager. Through our application, we demonstrate that some of the above problems can be solved to some extend and that the improvement of fault management through distributed fault location is feasible.

4.2 Future Work

Our research in this thesis suggests several directions for research in the area of system level diagnosis. Four of these are presented below.

- **Hybrid Multilevel Distributed Diagnosis**

In our work we have applied the multilevel paradigm to the ADSD algorithm. In other words, we have used the ADSD algorithm at all levels. Multilevel paradigm can also be applied to the Hi-ADSD algorithm. Since the Hi-ADSD algorithm requires a complex partitioning scheme, it is not suitable at levels greater than one. If we could implement the Hi-ADSD algorithm at level 1 and the ADSD algorithm at other levels, we would get a hybrid ML-ADSD scheme. The diagnosis latency of such a hybrid scheme can be obtained by simply replacing N/p by log^2N/p in Theorem 2. However, certain questions need to be resolved to achieve a successful design of such a hybrid scheme. One of them is the definition of actions to be taken during a testing round. In the Hi-ADSD algorithm, a node starts a new testing round from the cluster next to the one where it stopped in the previous round. It is not clear whether this would work in the case of the hybrid multilevel scheme. Research in this direction will be quite rewarding. It will also be an illustration of the power of the multilevel paradigm.

- **Multilevel Diagnosis for Networks of Arbitrary Topologies**

The diagnosis algorithm of Rengarajan, Dahbura and Zeigler [RDZ95] is the only distributed diagnosis algorithm in the literature applicable to networks of arbitrary topologies. An interesting direction of research is to see if we could apply the multilevel paradigm on top of this algorithm. This will not be as simple as extending the Hi-ADSD algorithm mentioned above. Several questions arise. It is not even clear how to define a testing round, because the algorithm of [RDZ95] is quite a complex one.

- **SNMP Based Distributed Fault Detection Systems for Networks of Arbitrary Topologies**

The SNMP based fault detection system discussed in chapter 3 assumes that the network is fully connected. But it will also work correctly if the network is logically fully connected. This would require a fault free communication path between every pair of nodes. In other words, the distributed fault detection system based on the SNMP protocol can be used to diagnose networks of arbitrary topologies. Designing, implementing and testing such a fault detection system in large networks will be a significant accomplishment. To make this useful in a multi-vendor environment, mangers and agents have to be designed to suit platforms other than the Windows platform we have considered. Such a system will demonstrate the applicability of our approach for networks of arbitrary topologies.

- **Distributed Testing and Diagnosis in a Mobile Computing Environment**

The rapidly expanding technology of cellular communication, wireless LANs and satellite services will make information available anywhere and at any time [IB94]. Soon, millions of people will carry portable computing devices. Regardless of the size of these mobile computers, they will be equipped with a wireless connection to the fixed part of the network. The resulting computing environment, often referred to as **mobile** or **nomadic computing**, does not require users to maintain a fixed and universally known position in the network and allows almost unrestricted mobility. Rising expectations from this environment require a rich set of computing and communication capabilities and services to be provided to the mobile user in a transparent, integrated and convenient

form [BCKP95] [IK96]. This new mobile computing environment has given rise to a host of new research challenges in areas such as address management, mobility management, data distribution, security and bandwidth management [AB94] [FZ94] [KCVP95] [BCKP95] [IK96] [CMB96] [S97].

Currently most studies regarding host mobility use the assumption that the mobile computing device is only "one hop" away from the fixed part of the network. We are seeing the emergence of multi-hop ad-hoc networks which have dynamically changing mesh topologies. These networks could be of interest in military or disaster situations, or even during the organization of a special event such as a scientific conference or a business meeting. These networks could also be seen to exist at the boundaries of a fixed network. Keeping in view the emergence of such networks and the need for mobile hosts to have a global picture of the network topology either for routing purposes or organizing distributed computations, a challenging research direction is to investigate distributed testing and diagnosis issues in a general mobile computing environment.

References:

[AB94] A. Acharya and B.R. Badrinath, "Delivering Multicast Messages in Networks with Mobile Hosts," Proc. 4th International Conference on Distributed Computing Systems, 1994.

[AKT75]F.J. Allan, T. Kameda, and S. Toida, "An Approach to the Diagnosability Analysis of a System," IEEE Trans. on Computers, vol. 24, 1975, pp. 1040-1042.

[B88] D.M. Blough, "Fault Detection and Diagnosis in Multiprocessor Systems," Ph.D. Thesis, The John Hopkins University, 1988.

[BA86a] P. Banerjee and J. Abraham, "Bounds on Algorithm-based Fault Tolerance in Multiprocessor Systems," IEEE Trans. on Computers, vol. 35, 1986, pp. 296-306.

[BA86b] P. Banerjee and J. Abraham, "Concurrent Fault Diagnosis in Multiprocessor Systems," Proc. 16th Intl. Symp. on Fault-Tolerant Computing, 1986, pp. 298-303.

[BB91] R. Bianchini, R. Buskens, "An Adaptive Distributed System-Level Diagnosis Algorithm and its Implementation", Proceedings of the 21st International Symposium on Fault Tolerant Computing, 1991, pp. 222-228.

[BB92] R. Bianchini, R. Buskens, "Implementation of On-Line Distributed System-Level Diagnosis Theory", IEEE Trans. on Computers, vol. 41, no. 5, May 1992, pp. 616-626.

[BB99] D. Blough, H. Brown, "The Broadcast Comparison Model for On-Line Fault Diagnosis in Multicomputer Systems: Theory and Implementation," IEEE Trans. on Computers, vol. 48, no.5, 1999, pp. 470-493.

[BCKP95] R. Bagrodia, W.W. Chu, L. Kleinrock, and G. Popek, "Vision, Issues, and Architecture for Nomadic Computing," IEEE Personal Communications, Dec. 1995, pp. 14-27.

[BGM76] F. Barsi, F. Grandoni, P. Maestrini, "A Theory of Diagnosability of Digital Systems", IEEE Trans. on Computers, vol. C-25, no. 6, June 1976, pp. 585-593.

[BGN90] R. Biancini, Jr., K. Goodwin, and D.S. Nydick, "Practical Application and Implementation of Distributed System-Level Diagnosis Theory," Proc. 20th Intl. Symp. on Fault-Tolerant Computing, 1990, pp. 332-339.

[BH94] A. Bagchi and S.L. Hakimi, "Information Dissemination in Distributed Systems with Faulty Units," IEEE Trans. on Computers, vol. 43, 1994, pp. 698-710.

[BMD93] M. Baborak, M. Malek, and A.T. Dahbura, "The Consensus Problem in Fault-Tolerant Computing," ACM Computing Surveys, vol. 25, no. 2, 1993, pp. 171-220.

[BP94] D.M. Blough and A. Pelc, "Almost Certain Fault Diagnosis through Algorithm Based Fault Tolerance," IEEE Trans. on Parallel and distributed Systems, vol. 5, 1994, pp. 532-539.

[BSB92] R. Biancini, Jr., M. Stahl, and R. Buskens, "The Adapt2 On-Line Diagnosis Algorithm for General Topology Networks," Proc. GLOBECOM, 1992, pp. 610-614.

[BSM92] D-M. Blough, G.F. Sullivan and G.M. Masson, "Efficient Diagnosis of Multiprocessor Systems under Probabilistic Models," IEEE Trans. on Computers, vol. 41, no. 9, 1992, pp. 1126-1136.

[CCITT89a] CCITT Recommendation X.208 (1989) | ISO/IEC 8824 : 1990, Information Technology - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).

[CCITT89b] CCITT Recommendation X.208 (1989) | ISO/IEC 8824 : 1990, Information Technology - Open Systems Interconnection - Specification of Basic Encoding Rules For Abstract Syntax Notation One (ASN.1).

[CFSD90] J.D. Case, M.S. Fedor, M.L. Schoffstall and J.R. Davin, "A Simple Network Management Protocol", RFC 1157, 1990.

[CH81] K. Chwa, S. Hakimi, "On Fault Identification in Diagnosable Systems", IEEE Trans. on Computers, vol. C-30, no. 6, June 1981, pp. 414-422.

[CHPW99] J. Case, D. Harrington, R. Presuhn, and B. Wijnen, "Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)", RFC 2572, 1999.

[CMB96] S. Corson, J. Macker, and S. Batsell, "Architectural Considerations in Mobile Mesh Networking,", tonnant.itd.nrl.navy.mil/mmnetRFC.txt, June 1996.

[DCFS87] J. Davin, J. Case, M. Fedor, and M. Schoffstall, "A Simple Gateway Monitoring Protocol", RFC 1028, 1987.

[DM84] A. T. Dahbura, G. M. Masson, "An $O(n^{2.5})$ Fault Identification Algorithm for Diagnosable Systems", IEEE Trans. on Computers, vol. C-33, no. 6, June 1984, pp. 486-492.

[DMY85] A.T. Dahbura, G.M. Masson, and C. Yang, "Self-Implicating structures for Diagnosable Systems," IEEE Trans. on Computers, vol. 34, 1985, pp. 718-723.

[DN98] E. P. Duarte Jr., T. Nanya, "A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm", IEEE Trans. on Computers, vol. 47, no. 1, Jan. 1998, pp. 34-45.

[DSK87] A.T. Dahbura, K.K. Sabnani and L.L. King, "The Comparison Approach to Multiprocessor Fault Diagnosis," IEEE Trans. on Computers, vol. 33, 1987, pp. 373-378.

[DTA91] A. Das, K. Thulasiraman and V.K. Agarwal, "Diagnosis of t/s-Diagnosable Systems", Journal of Circuits, Systems, and Computers, vol. 1, 1991, pp. 353-371.

[DTA94] A. Das, K. Thulasiraman and V.K. Agarwal, "Diagnosis of t/t+1-Diagnosable Systems", SIAM Journal on Computing, 1994, pp. 895-905.

[DTAL93] A. Das, K. Thulasiraman, V.K. Agarwal and K.B. Lakshmanan, "Multiprocessor Fault Diagnosis under Local Constraints", IEEE Trans. on Computers, vol. 42, 1993, pp. 984-988.

[DTLA93] A. Das, K. Thulasiraman, K.B. Lakshmanan, and V.K. Agarwal, "Distributed Fault Diagnosis of a Ring of Processors", Parallel Processing Letters, vol. 3, 1993, pp. 195-204.

[FBL96] C. Feng, L.N. Bhuyan, and F. Lombardi, "Adaptive System-Level Diagnosis for Hypercube Multiprocessors," IEEE Trans. on Computers, vol. 45, 1996, pp. 1157-1170.

[FR89] D. Fussell and S. Rangarajan, "Probabilistic Diagnosis of Multiprocessors with Arbitrary Connectivity," Proc. 19th Intl. Symp. on Fault-Tolerant Computing, 1989, pp. 560-565.

[FZ94] G.H. Forman, and J. Zahorjan, "The Challenges of Mobile Computing," IEEE Computer, 1994.

[HA74] S. L. Hakimi, A. T. Amin, "Characterization of Connection Assignment of Diagnosable Systems", IEEE Trans. on Computers, January 1974, pp. 86-88.

[HA84] K. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," IEEE Trans. on Computers, vol. 33, 1984, pp. 518-528.

[HALT95] K. Huang, V.K. Agarwal, L.E. LaForge, and K. Thulasiraman, "A Diagnosis Algorithm for Constant Degree Structures and its Application to VLSI Testing", IEEE Trans. on Parallel and Distributed Systems, vol. 6, 1995, pp. 363-372.

[HAT98]K. Huang, V.K. Agarwal, and K.Thulasiraman, "Diagnosis of Clustered Faults and Wafer Testing," IEEE Trans. on Computer Aided Design, vol. 17, no. 2, February 1998, pp. 136-148.

[HKR84] S. H. Hosseini, J. G. Kuhl, S. M. Reddy, "A Diagnosis Algorithm for Distributed Computing Systems with Dynamic Failure and Repair", IEEE Trans. on Computers, vol. C-33, no. 3, March 1984, pp. 223-233.

[HN84] S.L. Hakimi and K. Nakajima, "On Adaptive System Diagnosis", IEEE Trans. on Computers, vol. 33, 1984, pp. 234-240.

[HPW99] D. Harrington, R. Presuhn, and B. Wijnen, "An Architecture for Describing SNMP Management Frameworks", RFC 2571, 1999.

[IB94] T. Imielinski and B.R. Badrinath, "Mobile Wireless Computing: Challenges in Data Management," Communications of the ACM, Oct 1994.

[IK96] T. Imielinski and H.F. Korth, *Mobile Computing*, Kluwer Academic Publishers, 1996

[J94] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice-Hall, 1994.

[K78] T. Kohda, "On One-Step Diagnosable Systems Containing at most *t*-Faulty Units," Systems, Computers, and Control, vol. 9, 1978, pp. 30-37.

[KCVP95] P. Krishna, M. Chatterjee, N.H. Vaidya, and D.K. Pradhan, "A Cluster-based Approach for Routing in Ad-Hoc Networks," Proc. Mobile and Location-Independent Computing Symposium, 1995.

[KF78] A. Kavianpour and A.D. Friedman, "Efficient Design of Easily Diagnosable Systems," 3rd USA-Japan Comput. Conference, vol. 5, 1978, pp. 251-257.

[KH87] S.E. Kreutzer and S.L. Hakimi, "Microprocessing and Microprogramming 20 (1987), pp. 33-330.

[KR80] J.G. Kuhl, S. M. Reddy, "Distributed Fault-Tolerance for Large Multiprocessor Systems", Proceedings of the 7th Annual Symposium on Computer Architecture, 1980, pp. 23-30.

[KR81] J.G. Kuhl, S. M. Reddy, "Fault-Diagnosis in Fully Distributed Systems", Proceedings of the 11th International Symposium on Fault Tolerant Computing, 1981, pp. 100-105.

[KTA75]T. Kameda, S. Toida, and F.J. Allan, "A Diagnosing Algorithm for Networks," Information and Control, vol. 5, 1975, pp. 141-148.

[L96] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Francisco, 1996.

[L97] C. Lamb, "Graph Models and Systems Level Testing", Ph.D. thesis, Dept. of ECE, University of Oklahoma, 1997.

[LHA94] L.E. LaForge, K. Huang, and V.K. Agarwal, "Almost Sure Diagnosis of Every Good Element," IEEE Trans. on Computers, vol. 43, 1994, pp.295-305.

[LS94] S. Lee and K.G. Shin, "Probabilistic Diagnosis of Multiprocessor Systems," ACM Computing Surveys, vol. 26, 1994, pp. 121-139.

[M98] J. Murray, *Windows NT SNMP*, O'Reilly & Associated, Inc., 1998.

[MH76] S.N. Maheshwari and S.L. Hakimi, "On Models for Diagnosable Systems and Probabilistic Fault Diagnosis," IEEE Trans. on Computers, vol. 25, no. 3, 1976, pp. 228-236.

[MM81] J. Maeng, M. Malek, "A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems", Proceedings of the 11th International Conference on Fault Tolerant Computing, 1981, pp. 173-175.

[MR90] K. McCloghtie and M.T. Rose, "Management Information Base for Network Management of TCP/IP-based internets", RFC 1156, 1990.

[MR91] K. McCloghtie and M.T. Rose, "Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II", RFC 1213, 1991.

[N81] K. Nakajima, "A New Approach to System Diagnosis," Proc. 19th Annual Allerton Conf. Communication, Control, and Computers, 1981, pp. 697-706.

[NN86] J. Narasimhan and K. Nakajima, "An Algorithm for Determining the Fault Diagnosability of a System," IEEE Trans. on Computers, vol. 35, 1986, pp. 1004-1008.

[P96] M. Pollastshek, *Programming Discrete Simulations – Tools for Modeling the Real World*, R&D Books, 1996.

[PM97] D. Perkins, E. McGinnis, *Understanding SNMP MIBS*, Prentice-Hall, Inc. 1997.

[PMC67] F. P. Preparata, G. Metze, R. T. Chien, "On the Connection Assignment Problem of Diagnosable Systems", IEEE Trans. on Electronic Computers, vol. EC-16, no. 6, December 1967, pp. 848-854.

[R89] V. Raghavan, "Diagnosability Issues in Multiprocessor Systems," Ph.D. Thesis, University of Minnesota, 1989

[R91] M. Rose, "A Convention for Defining Traps for use with the SNMP," RFC 1215, 1991.

[R94] M. Rose, *The Simple Book An Introduction to Internet Management*, 2nd edition, PTR Prentice-Hall, Inc, 1994.

[RDZ95] S. Rangarajan, A.T. Dahbura and E.A. Ziegler, "A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies", IEEE Trans. on Computers, vol. 44, 1995, pp. 312-333.

[RF92] S. Rangarajan and D. Fussell, "Diagnosing Arbitrarily Connected Parallel Computers with High Probability," IEEE Trans. on Computers, vol. 41, 1992, pp. 606-615.

[RFM90] S. Rangarajan, D. Fussell, and M. Malek, "Built-In Testing of Integrated Circuit Wafers", IEEE Trans. on Computers, vol. 39, 1990, pp. 195-205

[RM90] M. Rose and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-Based Internets," RFC 1155, 1990.

[RM91] M. Rose and K. McCloghrie, "Concise MIB Definitions," RFC 1212, 1991.

[RT91] V. Raghavan and A. Tripathi, "Sequential Diagnosability is Co-NP-Complete," IEEE Trans. on Computers, vol. 40, 1991, pp. 584-595.

[S84] G.F. Sullivan, "A Polynomial Time Algorithm for Fault Diagnosability", Proc. 25th Symp. on Foundations of Computer Science, 1984, pp. 148-156.

[S87] E.R. Scheinerman, "Almost Sure Fault-Tolerance in Random Graphs," SIAM Journal on Computing, vol. 16, 1987, pp. 1124-1134.

[S88] G.F. Sullivan, "An $O(t^3+|e|)$ Fault Identification Algorithm for Diagnosable Systems," IEEE Trans. on Computers, vol. 37, 1988, pp. 538-546.

[S97] S. Sathyanarayanan, "Mobile Computing: Where is the Tofu?," Mobile Computing and Communications Review, vol. 1, 1997, pp. 17-21.

[SAA87] A.K. Somani, V.K. Agarwal, and D. Avis, "A Generalized Theory for System-Level Diagnosis," IEEE Trans. on Computers, vol. 36, no. 5, 1987, pp. 388-397.

[SAA89] A.K. Somani, D. Avis, and V.K. Agarwal, "On the Complexity of Single-Fault Diagnosability and Diagnosis Problems in System-Level Diagnosis," IEEE Trans. on Computers, vol. 38, 1989, pp. 195-201.

[SBB92] M. Stahl, R. Buskens, and R. Biancini, Jr., "On-Line Diagnosis in General Topology Networks," Proc. Workshop on Fault-Tolerant Parallel and Distributed Systems, 1992.

[SD92] A. Sengupta and A. Dahbura, "On Self-Diagnosable Multiprocessor Systems: Diagnosis by the Comparison Approach", IEEE Trans, on Computers, vol. 41, no. 11, 1992, pp. 1386-1396.

[SP96] A.K. Somani and O. Peleg, "On Diagnosability of Large Fault Sets in Regular Topology-based Computer Systems," IEEE Trans. on Computers, vol. 45, 1996, pp. 892-903.

[STD01] M.-S. Su, K. Thulasiraman, and A. Das, "A Multi-Level Adaptive Distributed Diagnosis Algorithm for Fault Detection in a Network of Processors", Proc. 39th Annual Allerton Conf. on Communication, Control, and Computers, 2001.

[VP94] N.H. Vaidya and D.K. Pradhan, "Safe System-Level Diagnosis," IEEE Trans. on Computers, vol. 43, 1994, pp. 367-370.

[W91] S. Waldbusser, "Remote Network Monitoring Management Information Base", RFC 1271, 1991.

[YJ97] S. Yajnik and N.K. Jha, "Graceful Degradation in Algorithm-Based Multiprocessor Systems," IEEE Trans. on Parallel and Distributed Systems, vol. 8, 1997, pp. 137-153.

[YML86] C. Yang, G.M. Masson, and R. Leonetti, "On Fault Identification and Isolation in t_1/t_1-Diagnosable Systems," IEEE Trans. on Computers, vol. 35, no. 7, 1986, pp. 639-643.