A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm

Elias Procópio Duarte Jr., *Member*, *IEEE*, and Takashi Nanya, *Senior Member*, *IEEE*

Abstract—Consider a system composed of N nodes that can be faulty or fault-free. The purpose of distributed system-level diagnosis is to have each fault-free node determine the state of all nodes of the system. This paper presents a Hierarchical Adaptive Distributed System-level Diagnosis (*Hi-ADSD*) algorithm, which is a fully distributed algorithm that allows every fault-free node to achieve diagnosis in, at most, $(log_2 N)^2$ testing rounds. Nodes are mapped into progressively larger logical clusters, so that tests are run in a hierarchical fashion. Each node executes its tests independently of the other nodes, i.e., tests are run asynchronously. All the information that nodes exchange is diagnostic information. The algorithm assumes no link faults, a fully-connected network and imposes no bounds on the number of faults. Both the worst-case diagnosis latency and correctness of the algorithm are formally proved. As an example application, the algorithm was implemented on a 37-node Ethernet LAN, integrated to a network management system based on SNMP (Simple Network Management Protocol). Experimental results of fault and repair diagnosis are presented. This implementation by itself is also a significant contribution, for, although fault management is a key functional area of network management systems, currently deployed applications often implement only rudimentary diagnosis mechanisms. Furthermore, experimental results are given through simulation of the algorithm for large systems of 64 nodes and 512 nodes.

Index Terms—System-level diagnosis, adaptive diagnosis, distributed diagnosis, network management, fault management, SNMP.

1 Introduction

As computer networks have grown into complex, enterprise-wide systems, management of operations and associated risks has become a critical task. The goal of Network Management Systems is to monitor, interpret, and control network operations, optimizing costs and reducing risks.

In the manager-agent paradigm, a Network Management System consists of a Network Management Station (NMS), also called monitor or manager, that queries a set of agents for information describing the state of links, devices, protocol entities, and nodes. Agents collect operational data (e.g., performance parameters) and detect exceptional events (e.g., error rates exceeding thresholds). This information is kept in the Management Information Base (MIB). Agents may issue alarms to inform the NMS about an exception. The NMS and the agents communicate through a network management protocol. Applications based on the Simple Network Management Protocol (SNMP) [1], [2], [3] are currently widely available.

Current network management systems often implement only rudimentary fault diagnosis mechanisms. Consider a Local Area Network (LAN). The traditional approach to monitoring [4] is to have a few managers, usually only one, organized in a tree, each of them responsible for querying a set of agents, and reporting to monitors in higher levels of the tree, as shown in Fig. 1. In these trees, agents are the leaves, and intermediate nodes are monitors that implement both an agent process (i.e., SNMP server) and a manager process (i.e., SNMP client). This approach presents two drawbacks:

1) If monitors become faulty or unreachable, diagnosis stops on an entire portion of the network;
2) All monitors are required to test a large number of network nodes.

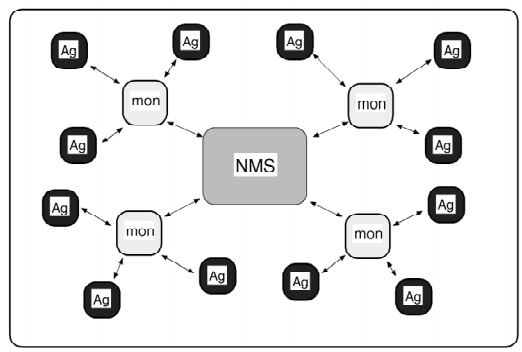


Fig. 1. A common approach to network management monitoring.

The field of distributed system-level diagnosis has flourished for years. Not only theoretical, but also practical implementations have been presented. In [5], [6], Bianchini and Buskens introduced the Adaptive Distributed System-level Diagnosis (Adaptive DSD) algorithm, and also its implementation in an Ethernet environment. Adaptive DSD has diagnosis latency of N testing rounds for a network of N nodes.

- *E.P. Duarte Jr. Is with the Department of Informatics, Federal University of Paraná, C.P. 19081 Curitiba PR, 81531-990, Brazil. E-mail: elias@inf.ufpr.br.*
- *T. Nanya is with the Research Center for Advanced Science & Technology, University of Tokyo, 4-6-1, Komaba, Meguro-ku, Tokyo 153, Japan. E-mail: nanya@hal.rcast.u-tokyo.ac.jp.*

Manuscript received January 1997; revised September 1997.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 105898.

In this paper, we present a new Hierarchical Adaptive Distributed System-level (*Hi-ADSD*) algorithm and its implementation integrated to a Network Management System based on SNMP (Simple Network Management Protocol). Hi-ADSD is a fully distributed algorithm that has diagnosis latency of, at most, $log^2 N$ testing rounds for a network of N nodes. As in Adaptive-DSD, each fault-free node executes tests until another fault-free node is found. Nodes are grouped in progressively larger logical clusters, so that each node executes tests in a hierarchical fashion. The algorithm assumes no link faults, a fully-connected network and imposes no bounds on the number of faults. All logarithms used in this paper are base 2.

The rest of the paper is organized as follows. Section 2 is a brief revision of system-level diagnosis. In Section 3, the Hierarchical Adaptive Distributed System-level (*Hi-ADSD*) algorithm is specified and its correctness is formally proved. Section 4 shows experimental results of diagnosis on large networks obtained through simulation. Section 5 presents the implementation of Hi-ADSD integrated to an SNMP-based Network Management System. This is followed by conclusions in Section 6.

2 System-Level Diagnosis

Consider a system consisting of N units, which can be faulty or fault-free. The goal of system-level diagnosis is to determine the state of those units [7]. For almost 30 years, researchers have worked on this problem. The first model of diagnosable systems was introduced by Preparata et al., the *PMC Model* [8]. In the PMC model, units are assigned a subset of the other units to test, and fault-free units are able to accurately assess the state of the units they test. The set of all tests makes up a testing graph, i.e., a directed graph in which vertices represent the system's units and an edge from vertex i to vertex j corresponds to a test performed by unit i on unit j.

The collection of all test results is called the *syndrome* of the system. The problem of diagnosis is to obtain the state of the system from a given syndrome. The PMC model assumes the existence of a *central observer* that, based on the syndrome, can diagnose the state of all the units. For a given testing assignment, the diagnosability of a system may be limited by the number of faulty units, and determining this number is called the *diagnosability problem*. Preparata et al. showed that, for a system to be t-diagnosable, it is necessary that $N \geq 2t + 1$, and that each unit is tested by at least t other units. Later, Hakimi and Amin [9] proved that if no two units test each other these conditions are sufficient for t-diagnosability.

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

An alternative approach, which requires fewer tests, is to assume that each unit is capable of testing any other, and to issue the tests adaptively, i.e., the choice of the next tests depends on the results of previous tests, and not on a fixed pattern. Hakimi and Nakajima called this approach *adaptive* [10]. Early adaptive system-level diagnosis results assumed the existence of the previously mentioned central observer. Furthermore, a bound on the number of faulty nodes was imposed for the system to achieve correct diagnosis.

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

Previously, Kuhl and Reddy [11], [12] introduced *distributed* system-level diagnosis, in which fault-free nodes reliably receive test results through their neighbors, and each node independently performs consistent diagnosis. They proposed the SELF distributed system-level diagnosis algorithm that, although fully distributed, is nonadaptive, i.e., each unit has a fixed testing assignment. We will alternatively use the word node for unit and network for system.

Later, Hosseini et al. [13] extended the SELF algorithm, introducing the NEW-SELF algorithm, which also has a fixed internode test assignment, but is executed on-line, permitting faulty nodes to reenter the network after being repaired. NEW-SELF ensures the accuracy of test-results by restricting the forwarding of testing results to fault-free nodes. For correct diagnosis, NEW-SELF requires that every fault-free node receives all test results from all other fault-free nodes. To reduce the amount of network resources required for diagnosis, the EVENT-SELF algorithm was proposed by Bianchini et al. [14]. This algorithm uses event-driven techniques to improve both the diagnosis latency and the impact of the algorithm on network performance.

The Adaptive Distributed System-level Diagnosis algorithm, *Adaptive-DSD*, was introduced by Bianchini and Buskens [5], [6]. Adaptive-DSD is, at the same time, distributed and adaptive. Each node must be tested only one time per testing interval. All fault-free nodes achieve consistent diagnosis in at most N testing rounds. There is no limit on the number of faulty nodes for fault-free nodes to diagnose the system. Practical results of Adaptive-DSD were presented of its implementation on an Ethernet environment.

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

Each node i that executes the algorithm has an array, called TESTED–UP$_i$, that contains N entries, indexed by the node identifier. The entry TESTED–UP$_i[k] = j$ means that node i has received diagnostic information from a

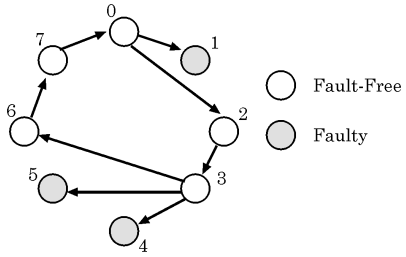


Fig. 2. Example of test assignment in Adaptive-DSD.

fault-free node specifying that node *k* has tested *j* to be fault-free. An entry TESTED–UP$_i$[*j*] is "arbitrary" if node *j* is faulty.

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

Adaptive-DSD has a diagnosis latency of *N* testing rounds. It is desirable to reduce this latency. In the original papers, Bianchini and Buskens use event-driven mechanisms to reduce the latency, e.g., employing multicast or broadcast just after a new situation is identified. There is no proof that the suggested event-driven mechanisms can reduce the latency of Adaptive-DSD.

In this paper, we present the Hierarchical Adaptive Distributed System-level Diagnosis (Hi-ADSD) algorithm. Hi-ADSD is a fully distributed algorithm that has diagnosis latency of at most $log^2 N$ testing rounds for a network of *N* nodes. The algorithm doesn't employ extra event-driven mechanisms, and requires less diagnostic information than Adaptive-DSD. Nodes execute tests asynchronously, and all the information that nodes exchange is diagnostic information. The algorithm assumes no link faults, a fully-connected network, and imposes no bounds on the number of faults. All logarithms used in this paper are base 2.

Hi-ADSD is hierarchical in the sense that it employs a divide-and-conquer testing strategy [20]. Nodes are grouped in progressively larger logical clusters, so that each node executes tests in a hierarchical fashion. Hi-ADSD is the first hierarchical diagnosis algorithm that is, at the same time, adaptive and distributed. Previous hierarchical approaches include [22], [23], [24], and [25].

As an example application, the algorithm was implemented on a 37-node Ethernet LAN, integrated to a network management system based on SNMP (Simple Network Management Protocol). Experimental results of fault and repair diagnosis are presented. Furthermore, experimental results are given through simulation of the algorithm for large systems of 64 nodes and 512 nodes.

The results discussed here assume a fully connected network, no link faults, and the PMC fault model. Besides the PMC fault model, many other fault models have been proposed. For example, a survey of probabilistic diagnosis is presented in [19]. Diagnosis of link faults was treated in [15]. Diagnosis on networks of general topology has received a great deal of attention recently, e.g., [18], [17], [16].

3 Hierarchical Adaptive Distributed System-Level Diagnosis

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

Before the algorithm is specified, it is important to recall the concepts of *test* and *testing round*, to avoid confusion. These concepts are the same used by Bianchini and Buskens for Adaptive-DSD in [5], [6]. At specified time intervals, for example, 30 seconds, each fault-free node in the system executes *tests* on other nodes of the system, until the testing node finds another node that is fault-free, or tests all other nodes as faulty. For instance, if the first node tested is fault-free, the tester stops testing; otherwise, it will test another node, and so on, until a fault-free node is found. A *testing round* is defined as the period of time in which every fault-free node in the system has tested another node as fault-free, and has obtained diagnostic information from that node, or has tested all other nodes as faulty. The *diagnosis latency* of Hi-ADSD is defined as the number of *testing rounds* required for all fault-free nodes in the system to achieve diagnosis.

Although it is possible to measure a testing round in terms of seconds or minutes, one should note that this is not always easy. For instance, one tester may test a fault-free node immediately, while another one may first have to test many faulty nodes until a fault-free node is finally found. Furthermore, if timeouts are used, it takes much longer to test a faulty node than to test a fault-free node. In spite of these problems, the notion of a testing round is a very convenient measure to determine and prove the latency of a diagnosis algorithm.

3.1 Algorithm Specification

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

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

A cluster of *n* nodes n_j, ..., n_{j+n-1}, where *j* MOD *n* = 0, and *n* is a power of two, is recursively defined as either a node, in case *n* = 1; or the union of two clusters, one containing

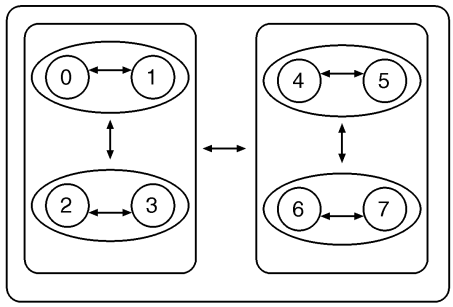


Fig. 3. A hierarchical approach to test clusters.

nodes n_j, ..., $n_{j+n/2-1}$ and the other containing nodes $n_{j+n/2}$, ..., n_{j+n-1}. Fig. 3 shows a system with eight nodes organized in clusters.

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

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

$$c_{i,s} = \left\{ n_t \,\middle|\, t = \left(i \text{ MOD } 2^s + 2^{s-1} + j\right) \text{ MOD } 2^{s-1+a} + \left(i \text{ DIV } 2^s\right) * 2^s + b * 2^{s-1}; j = 0, 1, \ldots, 2^{s-1} - 1 \right\},$$

where

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

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

When node i performs a test on nodes of $c_{i,s}$, it performs tests sequentially until it finds a fault-free node or all other nodes are faulty. Supposing a fault-free node is found; from this fault-free node, node i copies diagnostic information of all nodes in $c_{i,s}$. For the system in Fig. 3, for all i and s, $c_{i,s}$ is listed in Table 1.

Using function $c_{i,s}$, when two different nodes test the a given cluster, they will start testing different nodes.

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

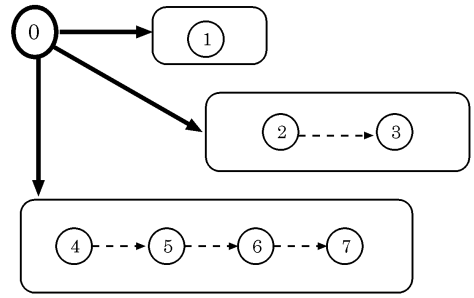


Fig. 4. Each node adaptively tests all clusters.

Hi-ADSD uses a tree to store information about the tests in all clusters. To effectively diagnose the state of all nodes, it is sufficient to list all nodes in the tree. Fig. 5 shows the tree for node 0, for the case that all nodes are fault-free.

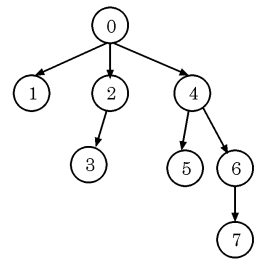


Fig. 5. A tree keeps all testing information.

A description of the algorithm in pseudocode is given in Fig. 6.

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

TABLE 1
$c_{i,s}$ FOR THE SYSTEM IN FIG. 2

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

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

Fig. 6. The Hi-ADSD algorithm.

It is assumed that a node cannot fail and recover from that failure during the time between two tests by another node. In Hi-ADSD this time may be of up to *log N* testing rounds, in the worst case. This assumption can be enforced by, for example, recording and storing fault events, or by reducing the testing interval between consecutive tests [5].

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

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

3.2 Correctness Proof

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

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

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

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

For example, consider a system of 16 nodes. Fig. 7 shows the Tested-Fault-Free graph of that system, if all nodes are fault-free. It can be seen that it is a hypercube. It contains a directed edge from any node i to the last node that i tested as fault-free in $c_{i,1}$, another edge to the last node that i tested as fault-free in $c_{i,2}$, another edge to the last node that i tested as fault-free in $c_{i,3}$, and another edge to the last node that i tested as fault-free in $c_{i,4}$.

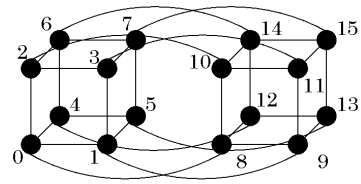


Fig. 7. The Tested-Fault-Free graph for a system of 16 fault-free nodes.

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

PROOF. This follows from the definition of the algorithm, i.e., at a given testing interval node i tests a cluster, and looks for a fault-free node in that cluster. In one testing round, by definition, each fault-free node tests at least another fault-free node, if there is one. There may be at most *log N* clusters for node i to test. In *log N* consecutive intervals, at each interval, a different cluster is tested. Thus, if node i executes exactly one successful test per testing round, it will take *log N* testing rounds for it to test all clusters. Therefore, in the worst possible case, for t_i immediately after a given cluster is tested, it will take up to *log N* testing rounds for that cluster to be tested again. □

THEOREM 1. *The shortest path between any two fault-free nodes in $T(S)$ contains, at most, log N edges.*

PROOF. We will use induction on t, for a system of 2^t nodes.

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

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

As an example, consider a system of size 2^2; this system has size four and each node tests two other nodes and gets information about the fourth node indirectly through the tested nodes. This makes up a path of length two. Now,

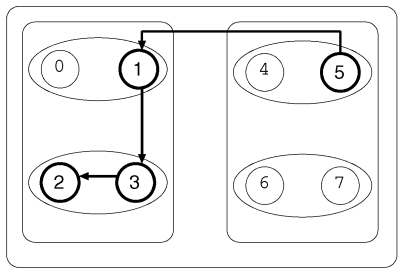


Fig. 8. The shortest path from node 5 to node 2 has *log* 8 = 3 edges.

consider a system of size 2^3: There are two clusters of size 2^2, and each node in one cluster tests one node in the other, thus, in T(S), there is an edge from each node in one cluster to the other. Therefore, the paths from a node in one cluster to the nodes in the other have lengths of the paths within the cluster which are at most of length 2, plus 1, for the edge linking the two clusters. Thus, in a system of size 2^3, the shortest path has length, at most, three. For example, look at node 5 and node 2 in Fig. 8. For node 5 to get information about node 2, node 5 tests node 1, which tests node 3, which tests node 2. In this system of eight nodes, the maximum path has size *log* 8.

Now let's consider each test in this worst case shortest path. How many testing rounds does it take to execute one test, in the worst case? Consider Fig. 8 again. If node 3 has tested node 2 just before it became faulty, then, only after three testing rounds, node 3 will discover that node 2 is faulty. Then, in the worst case, if node 1 tests node 3 just before node 3 tests node 2, it will take another three testing rounds for node 1 to discover that node 2 is faulty. If we are very unlucky and node 5 tested node 1 just before node 1 tested node 3, then it will take another three testing rounds for node 5 to discover that node 2 is faulty. In other words, there are three tests in the shortest path of longest length, and each one takes three testing rounds to be executed in the worst case, thus, in total, it may take up to nine testing rounds to execute all three tests.

THEOREM 2. *Consider the system fault situation at a given time. After, at most, $log^2 N$ testing rounds, each node that has remained fault-free for that period correctly determines that fault situation.*

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

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

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

It is not necessary that the number of nodes, *N*, be a perfect power of 2. In this case, testing nodes must skip the $2^{\lceil logN \rceil} - N$ nonexisting nodes during test and diagnostic information transfer. For instance, the implementation discussed in Section 4 was done on a 37-node system.

Nevertheless, the worst possible latency is $\lceil log^2\ N \rceil$, as there are at least *some* nodes for which the longest path in the Tested-Fault-Free graph have length *log N*.

For example, consider the system of eight nodes in Fig. 8. If that system had six nodes, instead of eight, i.e., if it didn't have node 6 and node 7, the length of the longest path would still be *log* 8 = 3.

3.3 Comparison of Adaptive-DSD and Hi-ADSD

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

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

Table 2 lists the diagnosis latency in terms of testing rounds for both algorithms, for networks having from four to 1,024 nodes. The figures in this table should be clearly understood. They show the number of testing rounds that are needed for all nodes in the system to diagnose one event in the fault situation. For example, if all nodes are fault-free, and one node becomes faulty, that diagnostic information will take N testing rounds in Adaptive-DSD, being transferred sequentially through N nodes until all nodes diagnose the situation. In Hi-ADSD, the diagnostic information will be transferred through a tree of depth *log N* and, to reach all nodes, it takes at most $log^2 N$ testing rounds. For networks of four and 16 nodes, the algorithms present the same worst case latency. In one case, for a network of eight nodes, Adaptive DSD presents better latency than Hi-ADSD, but this changes quickly as the number of nodes grows.

TABLE 2
EXAMPLES OF DIAGNOSIS LATENCY

N	Hi-ADSD	Adaptive-DSD
4	4	4
8	9	8
16	16	16
32	25	32
64	36	64
128	49	128
256	64	256
512	81	512
1,024	100	1,024

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

These extra tests correspond to the situation in which two or more nodes test a given faulty node in the same testing interval. In this case, those nodes will run more tests. The lists of nodes to be tested in each cluster ($c_{i,s}$) described previously in this section helps to make this situation unlikely, as all entry points are specific for each node to its clusters. However, in the worst possible case, if $N/2$ nodes are faulty, and they are all in the same cluster, and all testers test this cluster in the same testing round, the total number of tests is $N^2/4$. This situation is unlikely, not only because it requires a large number of faulty nodes, but also because it requires that specific nodes be faulty and tested in the same testing round.

Now, consider the total number of diagnostic messages required by the algorithms. Adaptive-DSD requires a total of N^2 messages for all nodes to achieve diagnosis, while Hi-ADSD requires $N \log^2 N$ messages in the worst case.

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

Fig. 9 compares the total number diagnostic units required by both algorithms for all nodes to achieve diagnosis. It can be seen that Hi-ADSD brings a significant improvement in terms of network bandwidth utilization.

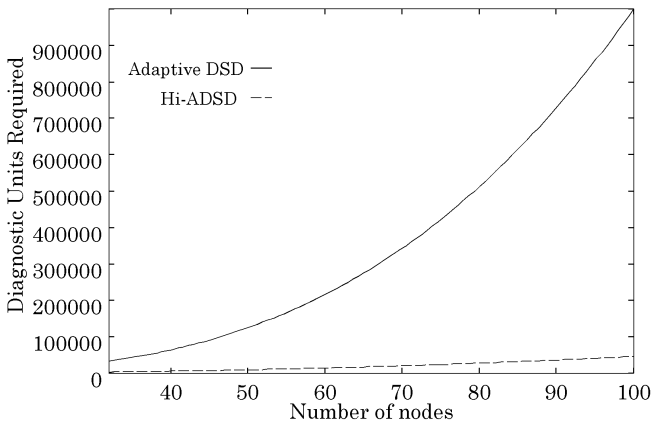


Fig. 9. Comparison of the amount of diagnostic units required.

The comparison shown in Fig. 9 is not meaningful if extra mechanisms, like timestamps, could be employed to avoid transferring diagnostic messages unless strictly necessary. Using these mechanisms, only information regarding a new event is transferred. However, to use any mechanism like this, it is necessary to prove its correctness and impact on the algorithm.

4 Simulation

In this section, we present experimental results of diagnosis on large networks using Hi-ADSD, obtained through simulation. The simulation was conducted using the discrete-event simulation language *SMPL* [21]. Nodes were modeled as SMPL facilities, and each node was identified by an SMPL token number. Three kinds of events were defined:

1) test,
2) fault, and
3) repair.

Tests were scheduled for each node at each $30 \pm \sigma$ units of time, where σ is a random number between 0 and 3.

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

We conducted several experiments with networks of different sizes. In this paper, we present results of two experiments: In the first experiment, on a network of 64 nodes, after a node becomes faulty, a second node also becomes faulty, and, after that, they are sequentially repaired. These four events were scheduled for times 100, 1,000, 2,100, and 3,000, respectively. The second experiment was conducted on a network of 512 nodes; a fault occurs at time 100, and the node is repaired at time 1,100. Results of diagnosis presented here are representative from the large set of simulation runs executed for each experiment.

Table 3 and Table 4 show the number of tests it takes for fault-free nodes to diagnose the first event of each experiment.

Table 3 shows that, for the first event in the 64-node system, the 63 fault-free nodes take up to $k = 9$ tests to successfully diagnose the event. For example, there is one node that successfully diagnoses the event after one test, this node tested directly the faulty node.

TABLE 3
NUMBER OF TESTS REQUIRED FOR 63 NODES TO DIAGNOSE ONE EVENT

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

Table 4 shows that, for the first event in the 512-node system, the 511 fault-free nodes take up to $k = 15$ tests to successfully diagnose the event. To compare with Adaptive-DSD, without extra event-driven mechanisms, we point out that Adaptive-DSD would take 511 testing rounds for all fault-free nodes to diagnose any event in this 512-node system.

TABLE 4
NUMBER OF TESTS REQUIRED FOR 511 NODES TO DIAGNOSE ONE EVENT

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

As we discussed before, nodes running Hi-ADSD run tests asynchronously, with consequences on algorithm performance. For each experiment, we ran a second simulation in which each node starts testing from a random cluster, as opposed to starting synchronized by testing cluster 1. The graphs in Figs. 10 and 11 show results from both types of simulation.

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

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

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

As a final comment, the graphs in Fig. 11 of the simulation of diagnosis on the initially synchronized 512-node system present some curves that deserve explanation. Those curves reflect a change of the number of nodes that learn about a new event. In other words, in Hi-ADSD, information flows from cluster to cluster, so there is a testing round when only one node learns about a new event, but, later, there is another testing round in which up to $N/2$ nodes diagnose that same event simultaneously.

5 PRACTICAL IMPLEMENTATION

In this section, we present the application of Hi-ADSD to SNMP-based LAN fault management. We describe the role of the NMS (Network Management Station) when Hi-ADSD is used for fault management. This is followed by the description of an approach to include monitoring of network devices that are not capable of testing. Finally, we present our implementation and experimental results obtained.

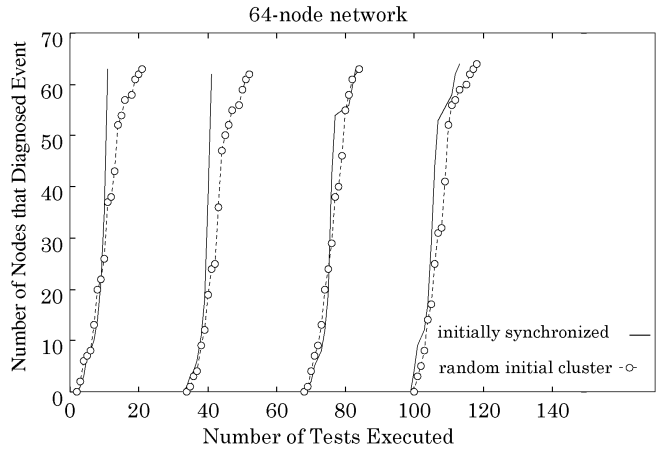


Fig. 10. Simulation of Hi-ADSD on a 64-node network.

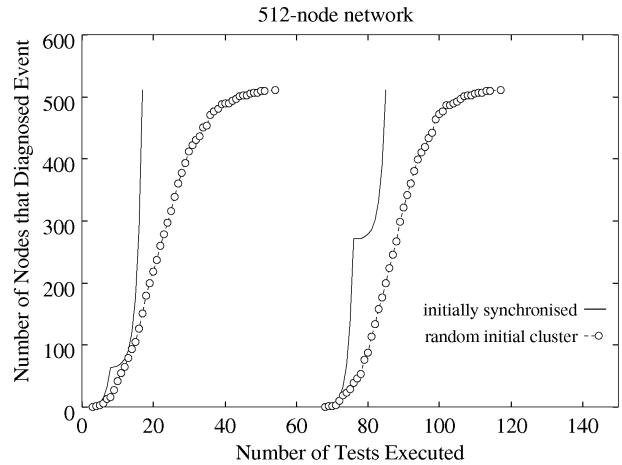


Fig. 11. Simulation of Hi-ADSD on a 512-node network.

5.1 The Role of the Network Management Station

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

One of the goals of network management systems is to provide network state information to the human manager at the NMS. The concept of a central observation point is not contradictory to the previously presented distributed approach: The NMS can now be seen as a *management interface* and not as the single monitor. This approach gives a number of advantages to the human manager, as she/he

has a choice of workstations to monitor the network. Furthermore, there are obvious advantages in terms of the reliability of the network monitoring system itself, as fault-free nodes achieve correct diagnosis for any number of faulty nodes.

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

5.2 Network Device Fault Management

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

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

It is important to provide a graphical interface for the fault management system and, currently, interfaces based on Web browsers are increasingly popular for this task.

5.3 Experimental Results

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

The implementation confirmed our expectation that even if the algorithm is run on a communication network based on a shared medium where collisions are possible, like the Ethernet, the probability of a collision of diagnosis packets is small. The reason is that the testing interval is large (e.g., 10s or 30s), diagnosis packets are small, and shared medium networks work at relatively high data rates (e.g., 10 Mbps). If there are a few collisions they will be handled by data link layer protocols. Furthermore, nodes are not synchronized, and are not expected to run tests at exactly the same time.

The CMU SNMP public-domain packet [26] was used as a base to implement the diagnosis agent in which we coded the *Diagnosis MIB* variables. From the SNMP toolkit of the WILMA project [27], we used client programs to access and update MIB variables.

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

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

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

It is important to realize that a test may have an impact on the performance of the tested node. This depends on how the test is implemented and on the characteristics of the tested nodes. In Hi-ADSD, this point is more critical than in Adaptive-DSD, because each node may be tested more than once every testing interval.

Faults were injected through the use of a specific MIB variable that, when queried, made the SNMP server "sleep" for a specified amount of time. In that period, the remaining nodes in the network diagnosed that the node was not replying to SNMP, but also was not faulty.

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

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

Fig. 12 shows how diagnosis progressed for the first three initial events. Initially, node 6 was actually faulty. Before the algorithm was initialized, we didn't know that, but all remaining 36 nodes diagnosed the fault situation in 198 seconds from the time they started testing.

After that, we did the first fault injection, and node 16's SNMP server stopped replying to queries. But, as the graph in Fig. 12 shows, at roughly the same time, node 6 was repaired, and started replying to ping. This was also an unexpected occurrence. From Fig. 12, it can be seen that, although the pace of diagnosis was different for both events, they were diagnosed in roughly the same amount of time: 575 seconds for node 6's fault, and 562 seconds for node 6's partial recovery, considering, for the latter case, the time since the first node diagnosed the event.

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

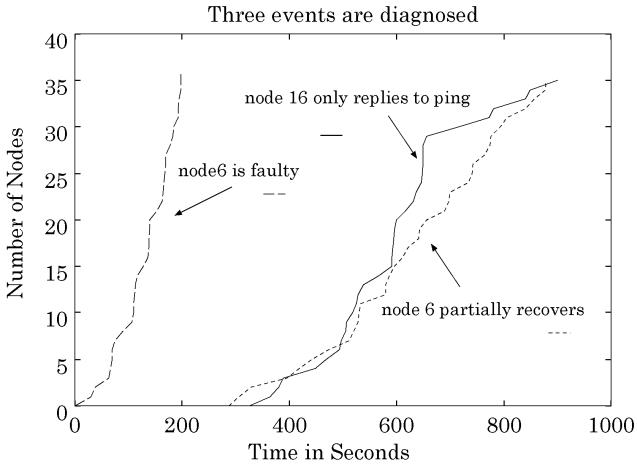


Fig. 12. First three events of the experiment.

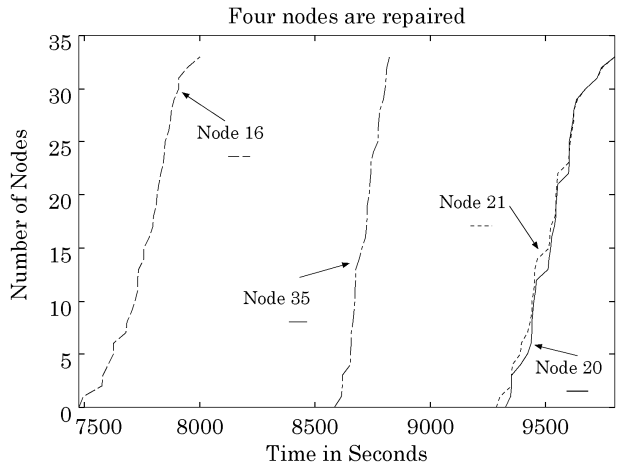


Fig. .14 Nodes are repaired and diagnosed.

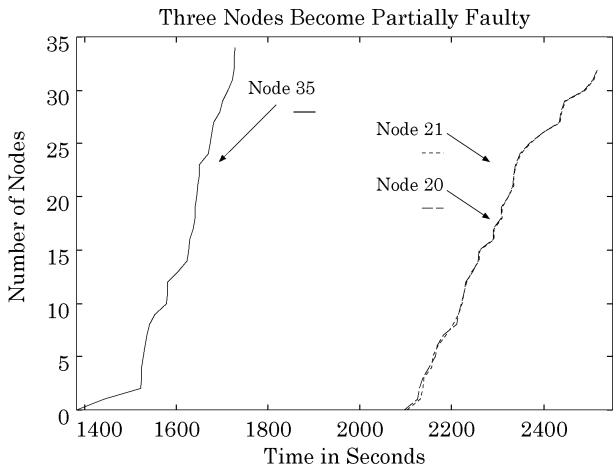


Fig. 13. Next three events, but second and third are diagnosed almost simultaneously.

graph of Fig. 13. The event at node 35 was diagnosed in 346 seconds. The events at node 20 and node 21 were diagnosed almost simultaneously, in 421 seconds, and 416 seconds. Not only the total amount of time, but also the pace with which both events were diagnosed, were very similar, as can be confirmed by the fact that Fig. 13 seems to show the diagnosis of two events and not the real three.

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

The average latency for all 10 experiments above is 427.1 seconds. With a testing interval of 40s, latency could have been up to 1,440 seconds. These results, together with previously shown simulation results, might confirm our belief that in average Hi-ADSD's latency is less than $log^2 N$ testing rounds.

6 Conclusions

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

Hi-ADSD was implemented, integrated to an SNMP-based network management system on a 37-node Ethernet LAN. Issues regarding the actual deployment of the algorithm were discussed, experimental results of fault and repair diagnosis were presented. As SNMP applications are currently widely deployed, but fault management is still based on rudimentary procedures, this implementation by itself is also a significant contribution to the field of network management.

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

Appendix

The ASN.1 coding of the SNMP Hi-ADSD MIB, as implemented, is shown below:

```
DIAGNOSIS-MIB DEFINITIONS ::= BEGIN

EXPORTS – everything –;

IMPORTS
  OBJECT-TYPE, OBJECT-GROUP
    FROM SMP-SMI
  enterprises, IpAddress
    FROM RFC1155-SMI
  DisplayString
    FROM RFC1213-MIB;
```

```
DiagnosisMIB   OBJECT IDENTIFIER ::=
                 {enterprises 200}
                 — this number is NOT official

HiADSDGroup OBJECT-GROUP
  OBJECTS {DiagTree, IsTestedOnly, NMSid,
                  HiADSDisUP}
  DESCRIPTION
    "The diagnosis group."
  ::= {DiagnosisMIB 1}

DiagTree OBJECT-TYPE
  SYNTAX  SEQUENCE OF diagTreeEntry
  ACCESS  not-accessible
  STATUS  mandatory
  DESCRIPTION
    "Table in which each testing node
    keeps network diagnostic info."
  ::= {HiADSDGroup 1}

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

DiagTreeEntry ::=
  SEQUENCE {
    NodeID       INTEGER,
    NodeAD       IpAddress,
    NodeStatus   INTEGER
}

NodeID OBJECT-TYPE
  SYNTAX  INTEGER
  ACCESS  read-only
  STATUS  mandatory
  DESCRIPTION
    "The integer unique identifier of all the
    nodes participating in HiADSD.
    Also indexes the table."
::= {DiagTreeEntry 1}

NodeAD OBJECT-TYPE
  SYNTAX  IpAddress
  ACCESS  read-write
  STATUS  mandatory
  DESCRIPTION
    "IP-address of all the nodes participating
    in Hi-ADSD"
::= {DiagTreeEntry 2}

NodeStatus OBJECT-TYPE
  SYNTAX  INTEGER
  ACCESS  read-write
  STATUS  mandatory
  DESCRIPTION
    "0 - node is fault-free;
    1 - node is faulty;
    2 - node replies to ping but not to SNMP"
::= {DiagTreeEntry 3}

IsTestedOnly OBJECT-TYPE
  SYNTAX      DisplayString (SIZE (0..1))
  MAX-ACCESS read-only
  STATUS      mandatory
  DESCRIPTION
    "If value is 0, then node is testing node.
    Otherwise tested_only node."
::= {HiADSDGroup 2}

NMSid  OBJECT-TYPE
  SYNTAX      IpAddress
  MAX-ACCESS read-write
  STATUS      mandatory
  DESCRIPTION
    "Used by the human manager to set the id
    of the machine from which she/he is
    monitoring the network"
::= {HiADSDGroup 3}

HiADSDisUP OBJECT-TYPE
  SYNTAX      DisplayString (SIZE (0..1))
  MAX-ACCESS read-write
  STATUS      mandatory
  DESCRIPTION
    "When queried, this variable checks if
    hiadsd process is up"
::= {HiADSDGroup 4}

END
```

The main portion of the *Diagnosis MIB* is the *Diag-Tree*, which is implemented as an SNMP table. *NodeID* is the identifier of each node; it varies from 1 to N. *NodeAD* is the ip-address of the each node. *NodeStatus* contains 0 if the node is known to be fault-free, and 1 if the node is faulty, and 2 if the node is replying to ping but not to SNMP.

IsTestedOnly identifies if the node has diagnostic information, i.e., is a testing node or a tested-only node.

NMSid is the IP address of the station from which the human manager is monitoring the network. Whenever a new event is discovered, a trap is sent to this address.

HiADSDisUP is an important MIB variable. In our implementation, Hi-ADSD was done on top of SNMP, so there is a possibility that SNMP is up, but Hi-ADSD is not, and the MIB doesn't have current information. This variable checks if Hi-ADSD is up.

ACKNOWLEDGMENTS

We wish to thank Prof. Doug Blough of the University of California at Irvine for his insightful comments on the algorithm, Fátima L.P. Duarte for her expert support in the SMPL simulation, and the many colleagues at Tokyo Institute of Technology who have helped with this project, especially Yoichiro Ueno for his UNIX administration support during the implementation of Hi-ADSD. Elias P. Duarte Jr. was partially supported by a scholarship from the Brazilian CNPq/Japanese Monbusho for his PhD course, during which this research was done. This work was done when both authors were at Tokyo Institute of Technology, Department of Computer Science, 2-12-1, Ookayama, Meguro-ku, Tokyo, 152, Japan.

REFERENCES

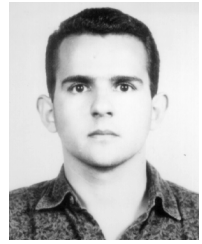
[1] M. Rose,and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-Based Internets," *RFC 1155*, 1990.

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

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

[4] L. Steinberg, "Techniques for Managing Asynchronously Generated Alerts," *RFC 1224*, 1991.

[5] R.P. Bianchini and R. Buskens, "An Adaptive Distributed System-Level Diagnosis Algorithm and Its Implementation," *Proc. FTCS-21*, pp. 222-229, 1991.
[6] R.P. Bianchini and R. Buskens, "Implementation of On-Line Distributed System-Level Diagnosis Theory," *IEEE Trans. Computers*, vol. 41, pp. 616-626, 1992.
[7] P. Jalote, *Fault Tolerance in Distributed Systems*. Englewood Cliffs, N.J.: Prentice Hall, 1994.
[8] F. Preparata, G. Metze, and R.T. Chien, "On The Connection Assignment Problem of Diagnosable Systems," *IEEE Trans. Electronic Computers*, vol. 16, pp. 848-854, 1968.
[9] S.L. Hakimi and A.T. Amin, "Characterization of Connection Assignments of Diagnosable Systems," *IEEE Trans. Computers*, vol. 23, pp. 86-88, 1974.
[10] S.L. Hakimi and K. Nakajima, "On Adaptive System Diagnosis" *IEEE Trans. Computers*, vol. 33, pp. 234-240, 1984.
[11] J.G. Kuhl, and S.M. Reddy, "Distributed Fault-Tolerance for Large Multiprocessor Systems," *Proc. Seventh Ann. Symp. Computer Architecture*, pp. 23-30, 1980.
[12] J.G. Kuhl and S.M. Reddy, "Fault-Diagnosis in Fully Distributed Systems," *Proc. FTCS-11*, pp. 100-105, 1981.
[13] S.H. Hosseini, J.G. Kuhl, and S.M. Reddy, "A Diagnosis Algorithm for Distributed Computing Systems with Failure and Repair," *IEEE Trans. Computers*, vol. 33, pp. 223-233, 1984.
[14] R.P. Bianchini, K. Goodwin, and D.S. Nydick, "Practical Application and Implementation of System-Level Diagnosis Theory," *Proc. FTCS-20*, pp. 332-339, 1990.
[15] C.-L. Yang and G.M. Masson, "Hybrid Fault-Diagnosability with Unreliable Communication Links," *Proc. FTCS-16*, pp. 226-231, 1986.
[16] S.Rangarajan, A.T. Dahbura, and E.A. Ziegler, "A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies," *IEEE Trans. Computers*, vol. 44, pp. 312-333, 1995.
[17] M. Stahl, R. Buskens, and R. Bianchini, "Simulation of the Adapt On-Line Diagnosis Algorithm for General Topology Networks," *Proc. IEEE 11th Symp. Reliable Distributed Systems*, Oct. 1992.
[18] A. Bagchi and S.L. Hakimi, "An Optimal Algorithm for Distributed System-Level Diagnosis," *Proc. FTCS-21*, June, 1991.
[19] G. Masson, D. Blough, and G. Sullivan, "System Diagnosis," *Fault-Tolerant Computer System Design*, D.K. Pradhan, ed. Prentice Hall, 1996.
[20] E.P. Duarte Jr. and T. Nanya, "Multi-Cluster Adaptive Distributed System-Level Diagnosis Algorithms," *IEICE Technical Report FTS 95-73*, 1995.
[21] M.H. MacDougall, *Simulating Computer Systems: Techniques and Tools*. Cambridge, Mass.: The MIT Press, 1987.
[22] M. Malek and J. Maeng, "Partitioning of Large Multicomputer Systems for Efficient Fault Diagnosis," *Proc. FTCS-12*, pp. 341-348, 1982.
[23] A. Bagchi, "A Distributed Algorithm for System-Level Diagnosis in Hypercubes," *Proc. 1992 IEEE Workshop Fault-Tolerant Parallel and Distributed Systems*, pp. 106-113, 1992
[24] M. Barborak and M. Malek, "Partitioning for Efficient Consensus," *Proc. 26th Hawaii Int'l Conf. System Sciences*, vol. II, pp. 438-446, 1993.
[25] J. Altman, F. Balbach, and A. Hein, "An Approach for Hierarchical System-Level Diagnosis of Massively Parallel Computers Combined with a Simulation-Based Method for Dependability Analysis," *Proc. First European Dependable Computing Conf., Lecture Notes in Computer Science*, vol. 852, pp. 371-385, 1994.
[26] M.T. Rose, *The Simple Book—An Introduction to Internet Management*, second ed. Englewood Cliffs, N.J.: Prentice Hall, 1994.
[27] J. Swoboda, et al. http://www.ldv.e-technik.tu-muenchen.de/dist/WILMA/.
[28] W. Stallings, *SNMP, SNMPv2, and RMON: Practical Network Management*, second ed. Reading, Mass.: Addison Wesley, 1996.



Elias Procópio Duarte Jr. received the BS and MSc degrees in computer science from Federal University of Minas Gerais, Belo Horizonte, Brazil, in 1988 and 1991, respectively. He also received an MSc degree in networks and communication systems from the Polytechnical University of Madrid, Spain, in 1991. He received his PhD in computer science from Tokyo Institute of Technology, Tokyo, Japan, in 1997. He is an associate professor in the Department of Informatics at the Federal University of Paraná, Curitiba, Brazil. His main research interests include distributed systems and computer networks, their dependability, management, performance evaluation, and algorithms. He is a member of the ACM and the IEEE.

Takashi Nanya received the BE and ME degrees in mathematical engineering and information physics from the University of Tokyo, Tokyo, Japan, in 1969 and 1971, respectively, and the DrEng degree in electrical engineering from the Tokyo Institute of Technology, Tokyo, Japan, in 1978. He worked on digital system design methodology at NEC Central Research Laboratories from 1971 to 1981. In 1981, he moved to the Tokyo Institute of Technology, where he was a professor of computer science. In 1995, he joined the University of Tokyo, where he is a professor at the Research Center for Advanced Science and Technology.

Dr. Nanya was a visiting research fellow at Oakland University, Michigan, in the fall quarter of 1982, and at Stanford University, California, in the 1986-1987 academic year. His research interests include fault-tolerant computing, computer architecture, design automation, and asynchronous computing. He received the Best Paper Award from IEICE in 1987, and the Okawa Prize for Publication in 1994. He served as program cochair of the 1995 IEEE International Symposium on Fault-Tolerant Computing, as the conference cochair of the 1996 IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems, and as guest editor of a special issue on asynchronous architecture in *IEE Proceedings, Computers and Digital Techniques*. He is a senior member of the IEEE.