

DIP : Distributed Diagnosis Protocol

Luís Moura Silva and João Gabriel Silva

Laboratório de Informática e Sistemas
Universidade de Coimbra
Urb. Boavista, Lt. 1-1
3000 Coimbra - Portugal
Fax : +351.39.701266
e-mail : lams@uc.pt

Abstract

This paper presents the DIP - a Diagnosis Protocol to check the availability of processors in a distributed system. Each node maintains a Network Status Table, and failure detection is achieved by alive messages. The overhead of such messages is minimal and the main virtue of the protocol is the lower diagnosis latency when a failure is detected. Since the unavailability of a node can be the result of a processor crash or a network partition, the DIP protocol is also able to cope with partitions. The protocol can diagnose fail-stop and omission failures. Moreover, it is robust against out-of-order messages, duplicates and false-alarms.

Keywords : *Distributed Diagnosis, Fault Tolerance, Membership Protocols, Network Monitoring.*

1. Introduction

One of the requirements of a fault-tolerant distributed system is the failure detection capability of remote sites. Recovery and reconfiguration are other desirable properties. Since the paper of Preparata, Metze and Chien [1] there were further theoretical developments [2] in the area of distributed diagnosis. As a result, several PMC-based diagnosis algorithms have been suggested [3-9]. Recently, Bianchini *et al.* [10,11] have conducted the first implementations of system diagnosis theory in a real distributed network environment. Nevertheless, all those PMC-based algorithms have serious drawbacks:

1) when the number of node failures exceeds the connectivity of the system digraph, those algorithms simply yield false diagnostic results. 2) The PMC-model is only valid for permanent processor failures and it assumes that a node cannot fail and restart in the period between two tests. 3) Most of the algorithms require a high number of diagnostic messages and, as a result, the execution time may exceed reasonable values.

All these shortcomings make the algorithms unsuited to local area networks. Moreover, if one considers that processors may exhibit byzantine faultiness [12], PMC-based algorithms may identify

none of the faulty processors [13]. Another class of diagnosis algorithms which identifies byzantine faults has been suggested [13-15]. These algorithms are able to diagnose intermittent and malicious faults. However, the diagnosis of byzantine faults becomes extremely difficult and all the proposed solutions present a great message overhead. As practitioners, we are not willing to pay such price and we simply assume a more restricted failure model: the crash failure model. In this model it is assumed that each node can only fail in "clean" well defined ways. During its functioning the node behaves correctly and if an internal failure occurs it halts automatically. Such nodes are called fail-silent or fail-stop. A lot of algorithms [18-27] have also assumed this failure model but no one seems to perfectly suit our goals and assumptions.

2. System Model and Assumptions

The system consists of a set of N stations interconnected by a LAN with broadcast capability. Each station has access to *stable storage* which can be used to store information that must survive to failures. In perfect conditions each processor can communicate with any other processor in the network. We assume that the relation "can-communicate" between nodes is symmetric and

transitive. Two kinds of failures are considered: processor failures and communication link failures. Processor failures are assumed to be clean. When a node fails it simply stops running and does not send any arbitrary or malicious messages. Thus, the universe of failures includes stop and omission failures and no assumption is made on the number of allowable processor failures.

Network failures can be classified in hard and soft failures [16]. An hard failure is characterized by the incapability of the network to deliver any bandwidth at all. Soft failures are characterized by performance degradations or deviations from the expected behaviour. In networks, such as the Ethernet [17], which do not provide any hardware level acknowledgment, the other nodes of a network cannot distinguish an isolated node from a crashed one, since in both cases the symptom is the same - the loss of communication with the node. When an active node is not available to any other node, it means that the network was partitioned (hard failure) and both nodes are in different partitions. However, crashed nodes can recover and partitions can be reconnected. For the sake of generality let us consider the existence of two main events : crashes and restarts. Each available node is marked as UP while an unavailable node is classified as DOWN. Each processor will have its own view of the network which must be consistent with that of other nodes belonging to the same partition.

We assume that the message delivery is not reliable since the network is subjected to soft failures. Messages can be lost, delayed or delivered out-of-order. However, a network soft failure cannot produce any malicious message.

We consider that there exists a maximum message delay ∂_{max} , which reflects the expected maximum transmission time of a single message between two nodes, including the processing delay. The minimum message delay is represented by ∂_{min} . Every node should have a physical clock in order to implement timers but we do not require clocks to be synchronized.

3. Design Guidelines

The DIP protocol should be able to perform three functions :

- (1) To detect node crashes.
- (2) To notify all the network nodes of the crash of a processor (Node Down event).

- (3) and also to notify all the nodes of the restart of a crashed node (Node Up event).

Each node maintains a table with the status of the other nodes, called the *Network Status Table* (NST). Each entry of the NST has three fields : the host name, its status and the *Incarnation Number* (IN). The IN number is helpful to distinguish duplicates and out-of-order messages, as will be explained in §4.4.

Since failures are the exception rather than the rule, the first imperative is to reduce the overhead during normal operation. The protocol must assure a minimal number of messages during steady-state operation. Another important goal is to reduce the diagnosis latency to a minimal value, i.e., the protocol must provide timely failure detection. This is a paramount figure of merit to allow an effective recovery/reconfiguration of distributed applications. When a node failure occurs all the other nodes must be acquainted with that as quickly as possible. As failures and restarts are asynchronous events, the protocol should be robust against their occurrence during its execution. Other important issue is the handling of network partitions. The other desired goals are efficiency, simplicity and correctness of the protocol.

4. Design Details

4.1 The Detection of Failures

As a fail-silent processor only stops when an internal fault occurs the failure can be simply detected by background `I_AM_ALIVE` messages. The network nodes are organized on a virtual ring and all the sites are numbered from 0 up to N-1, along the path of the virtual ring. It should be noted that the ring does not reflect the physical organization of the network. The order of the nodes in the ring follows the order of the Status Table, which is unique to all the processors in the network.

Each processor P_i sends periodically an `I_AM_ALIVE` message to the next neighbour $P_{((i+1) \bmod N)}$ while it is checking for the messages sent by the previous processor $P_{((i-1) \bmod N)}$. Each processor has two independent processes to perform that job.

- 1) The *Heartbeat* process is periodically awakened by a timer interrupt in order to send a message to the next neighbour. This timer is represented by T_H and is set to α .

2) The detection of failures is done by the Watchdog process, which is listening to the messages sent by the previous neighbour. Every time it receives an `I_AM_ALIVE` message it triggers a timer T_W with $((\Omega+1).\alpha + \partial_{\max} - \partial_{\min})$, where Ω is the number of omission failures or lost messages that can be tolerated. If the timer expires, this means that the previous node has crashed and the Node Down Protocol is started for that node.

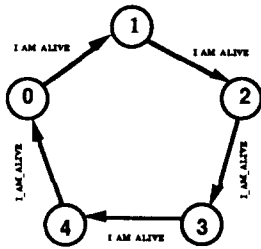


Figure 1 : Virtual Ring

4.2 Node Down Protocol

In order to reduce the diagnosis latency, the node should notify all the others immediately by a broadcast message to the network, announcing the `Node_Down` event of its neighbour.

Take as an example the network of figure 2. One can notice that node #2 has crashed and its tester (node #3) will detect that when its timer T_W expires. The status of node #2 is marked as `DOWN` in its Network Status Table, and it will send a broadcast message to the network, which includes the number of the node and the `NST(2).IN` : `NODE_DOWN(2, 1)`.

In a meantime, a timer T_{ack} will be set to $2.\partial_{\max}$. The node will inspect its status table to find the available nodes and waits for the acknowledge of every UP node. On the other hand, when a node receives a `NODE_DOWN` broadcast it replies with a `ND_Ack` message to node #3. It should be noted that each acknowledge is also a broadcast message. This may be useful if a particular node does not receive the broadcast message of #3 it may be informed of that by one of the acknowledge messages. When node #3 receives all acknowledges this phase is finished and the protocol returns to the steady state. However, if the timer T_{ack} expires and some reply is missing, the message `NODE_DOWN(2, 1)` is retransmitted again (until a maximum number of retries is reached).

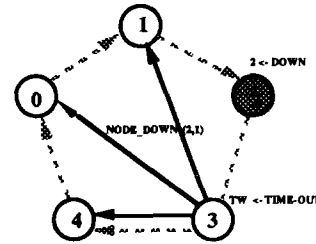


Figure 2 : Detection of a Failure

It may happen that the unavailability of node #2 is not the crash of a single processor, but instead the network may be partitioned. In this case, some nodes may not answer to the message and should also be classified as `DOWN`. In fact, if after the retransmission, node #3 does not receive any reply then it will start again this protocol announcing the unavailability of those nodes which remained in the other partition of the network.

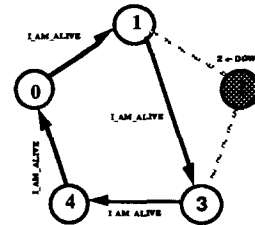


Figure 3 : Reconfiguration

After the notification phase of a node crash, the testing structure has to be adapted to a new configuration. Each node that had been notified that its tester neighbour has crashed must identify the next available node, by searching the Status Table (see example of node #1 in figure 3). The detector node (#3) will also have a new tested node (#1).

4.3 Node Up Protocol

When a node rejoins the network it must tell the others that it is back in operation and must also determine which nodes are UP and reachable. Therefore, when a node recovers it reads the NST from stable storage to main memory. The status of all the other nodes are set to `DOWN`. Then it increments its `IN` value, which is also stored in stable storage, and sets to zero all the counters of the other nodes. Finally, it broadcasts a `NODE_UP` message to notify all the active nodes in the

network. A timer T_{up} is activated. This message must be acknowledged by the other available nodes. The replies will be used by the restarted node to achieve its new network view. Furthermore, to guarantee a distributed consistency, node #2 must verify whether all the nodes it can reach share the same NST.

Let us consider the previous scenario : node #2 is DOWN and its IN is equal to 1. Now, suppose that it recovers and rejoins the network. First of all, its IN counter will be incremented, and it will send a `NODE_UP` broadcast message to the network, which includes the new IN (see figure 4).

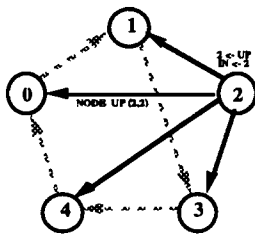


Figure 4 : Node #2 Rejoins the Network

When each of the other nodes receive the `NODE_UP(2,2)` message they check the 2nd entry of its `NST`, and the status of node #2 is marked as UP. Then, they send a `NU_Ack` message to node #2, which contains their own INs. Each node also appends a list of its DOWN nodes into that message, including the corresponding INs. When the timer T_{up} expires, node #2 will have a complete information about the network status. Likewise, to assure a distributed agreement, node #2 must check the information it had received from the `NU_Ack` messages. The DOWN lists which were piggybacked into the acknowledges must be all consistent. Otherwise, node #2 will have to broadcast the result of the comparison to force a distributed consensus.

4.4 Protocol Integration

As the two sub-protocols may execute simultaneously they must interact with each other. Let us consider the following situation : node #2 has crashed ; node #3 detects its failure and sends a `NODE_DOWN` broadcast message to the network. However node #2 is DOWN for only a short time, after that it comes up again and rejoins the network by sending a `NODE_UP` message. Yet, an

inconsistent scenario may occur: Node #0 receives the sequence (`NODE_DOWN`, `NODE_UP`) while node #1 receives the (`NODE_UP`, `NODE_DOWN`) sequence. Fortunately, the *Incarnation Number* can solve this inconsistency. Every message is tagged with the IN and each receptive node will compare the incoming IN with the value stored in its NST. When a host receives a `NODE_UP` message from node #K and the incoming IN exceeds `NST(K).IN`, the table is automatically updated. If the incoming IN is similar it is recognized as a duplicate message and no action is performed. In either cases a reply (`NU_Ack`) is sent. When a node receives a `NODE_DOWN` message from node #X notifying the crash of its neighbour (node #Y) it checks in its table the status and the current IN of node #Y. If #Y is UP and the incoming IN exceeds the `NST(Y).IN`, then #Y is marked as DOWN and the local IN is updated. In normal situations, the incoming IN is similar to the local value. In this case, only the status of node #Y is updated. On the other hand, if #Y is already DOWN the `NST(Y).IN` will only be updated if the incoming IN exceeds the local IN. In either case the receiving node will reply to each message with its own IN.

Taking the previous example, let us assume that node #2 was running with `IN=1`. By the time it had crashed, node #3 would have sent a `NODE_DOWN` message announcing this event (with `IN=1`). When the node recovers, it would increment IN to 2.

Node #0 would receive the messages in the correct order. However, when node #1 received the message `NODE_UP(2,2)` it would realize that node #2 had crashed and came up again without being aware of that. It would set `NST(2).IN` to 2 and send a `NU_Ack` message to node #2, which would include its own IN (`NST(1).IN`).

After having received the `NODE_DOWN(2,1)` message it would conclude that the message was out-of-date since the incoming IN was less than the stored one. In spite of that, a reply message (`ND_Ack`) would be sent to node #3.

4.5 False Alarms

A *false alarm* occurs when an active node receives a `NODE_DOWN` broadcast message notifying its own failure. It may happen if the number of lost messages exceeds the tolerated level. A network soft failure can be the cause of that. In this case, the node will react to that "injury" by broadcasting a

I'M_NOT_DOWN message to the network with its current IN. When the other nodes receive such message they undo the action taken by the reception of the *false alarm* message. This message is only taken into account if the incoming IN is equal to the stored value.

4.6 Coping with Network Partitions

Network partitioning is an event which causes the network to divide into two or more sub-networks. The worst case is when a single node is isolated from the rest of the network due to a crash of its communication processor. Partitioning should be detected during the execution of the Node Down Protocol.

To simplify the explanation let us consider the network represented in figure 5. Due to a communication link breakdown or to some other network component (e.g. gateway, bridge, router...) failure, the network was split into two sub-networks. The detection of this partitioning begins with the detection of two single node failures. Node #2 and node #0 will detect the crashes of node #1 and node #4, respectively. Each of the detectors will send a NODE_DOWN broadcast and then wait for the acknowledge messages. As Node #2 will note the lack of response from node #0 it will repeat the Down protocol for node #0. On the other hand, this node only receives the ND_Ack from node #1. Thus, it will broadcast to its sub-network the failure of node #2 and #3. Only node #1 will hear these messages.

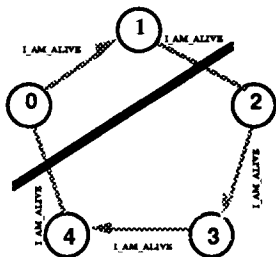


Figure 5 : Network Partitioning

The final result of this phase is represented in figure 6. Two partitions have been identified (P1 and P2) and the main graph of test has been divided into two independent graphs. To the nodes of partition P1, the nodes belonging to P2 are marked as DOWN, and vice-versa. The protocol guarantees that each time a network partitioning is detected this is assumed at

both sides of the "wall" between partitions.

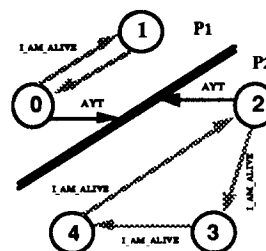


Figure 6 : Partitioning Reconfiguration

Furthermore, it is necessary to assure the detection of the possible merge of these partitions. This job is assigned to both *detector* nodes (#0 and #2). Each one will resume a *dummy* process that periodically sends a AYT² broadcast message to the network, which contains the identification of the sender and its IN. A rule is established to the nodes of the same partition : they do not answer to those messages. For instance, in partition P1 node #0 sends periodically an AYT broadcast message. Since node #0 is UP for node #1, it will not reply.

Those messages may pass through the partition "wall", and can be heard by the other nodes only when the two sub-networks get merged. Each node that receives an AYT message from a sender judged as DOWN, will acknowledge it with a YIA³ message. This reply contains the node identification, its current IN and is also broadcasted. A YIA works like a NODE_UP message with just one exception : if the incoming IN is similar to or exceeds the local IN, the message is accepted and the table is updated for the sender of the message. If the incoming IN is smaller, it is classified as obsolete. In both cases, a reply is sent to the network.

In the previous example, nodes #2 and #0 kept watching the other partition. They periodically send a AYT messages to the network. While the partition is maintained, those messages are not acknowledged. When the clusters get reconnected, node #1 and #0 hear the AYT(2) while nodes #2, #3 and #4 hear the AYT(0) message. They answer to those messages with a broadcast YIA and a special Node_Up protocol is started in each node. After that phase, the network recovers to the original configuration.

² AYT means "Are You There?"

³ YIA means "Yes, I Am !"

As it is not possible to distinguish a node failure from a network partition, every time a node detects a node crash it resumes that *dummy* process, which is suspended when all the failed nodes it has detected get up again. As it is represented in figure 7 it may happen that network partitioning does not affect a contiguous set of nodes (as in fig.6), but instead an alternate set along the path of the virtual ring. This is the worst case in terms of the number of messages, as will be seen in the next section.

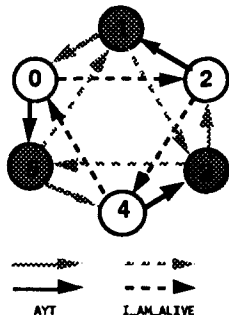


Figure 7 : Special Partitioning

5. Analysis of the Protocol

The original protocol uses 8 different messages: *I_AM_ALIVE*, *NODE_UP*, *NU_Ack*, *NODE_DOWN*, *ND_Ack*, *I'M_NOT_DOWN*, *AYT*, *YIA*.

In steady-state operation, the protocol requires the minimum number of diagnostic messages. In a network of N active processors, N messages are sent to the network between each testing interval. When a node failure occurs, it is detected by its neighbour in a finite interval called *failure detection latency*. This latency depends on how much time has elapsed between the time the last *I_AM_ALIVE* message was sent and the failure. In the worst case, the failure occurs immediately after the last message is sent out, while in the best case the node fails just before sending the message. So, the detection latency falls between $(\partial_{\max})$ and the period of the Watchdog Timer: $((\Omega+1).\alpha + \partial_{\max} - \partial_{\min})$.

Let us now refer to the time that it takes to notify the failure to all the other nodes the *diagnosis latency*. In the DIP approach, the path length of the diagnosis information is equal to 1 and the maximum diagnosis latency is just $\partial_{\max}$. The diagnosability of the protocol is equal to $(N-1)$, and the number of messages necessary to perform the Node_Down protocol is bound to $(N-1)$ (1 broadcast

and $(N-2)$ replies)⁴. To notify a Node_Up event it only requires N messages. However, if we have a network with (S) subnets connected through routers, the protocol has to send (S) *NODE_UP* messages, one for each subnet, and the diagnosis latency is also increased by the processing time of each intermediate router. The message count will be bound to $(S+N-2)$ for the Node Down Protocol, while the Node Up protocol will use $(S+N-1)$ messages.

Now, let us consider the partitioning of the network into P partitions. If the set of nodes of each partition is contiguous as in figure 6 the number of messages will be $(P+N)$: (P) *AYT* messages and (N) *I_AM_ALIVE* messages. In the worst case, as shown in figure 7, the message count will be bound to $(P.N)$. When only one node has become isolated, the number of messages will be $(N+1)$.

6. Relation to Previous Work

In this section, we will briefly discuss others algorithms found in the literature, and a comparison with the DIP will be made. The implementation of the NEW-SELF algorithm, conducted by Bianchini *et al.* [10] has shown that it requires a high number of diagnostic messages. It was concluded that such algorithm is not feasible to real systems. In the same paper, a new algorithm is proposed: the EVENT-SELF. This one reduces the message count during normal operation but if a failure occurs it produces a packet count higher than the one of NEW-SELF. Moreover, an experiment with 20 processors has shown that the complete diagnosis of one processor failure would take about 154 seconds. Both algorithms have a static testing graph, which means that the number of faulty units is bounded to a well established limit (t). If the number of processor failures exceeds that number (t), the network becomes undiagnosable.

In order to solve these serious problems, Bianchini proposed a new algorithm: the Adaptive DSD [11]. This algorithm can diagnose any number of failures since it has an adaptive testing graph. This represents some improvements, but it also has some shortcomings. Since it is a PMC-based algorithm it does not support that a node fails and recovers between two tests executed by another node. The algorithm is not able to diagnose link failures, and the path length of the diagnosis information is bound

⁴ Assuming no retransmissions.

to $\log_2 N$. An experiment in a network of 60 processors with a version that uses an asymmetric forwarding scheme has shown that the overall diagnosis latency was about 90 seconds for one processor failure. The version that assumes the original model of forwarding of the PMC-model has an overall diagnosis latency of 450 seconds.

If one wants an efficient algorithm the latency should be drastically reduced and the number of messages minimized. The algorithms that assume the crash failure model are the most suitable and can be easily applied in real systems since they require a lower message count and are much more simple.

A centralized approach was chosen in the Publishing system [18]. In this case, a dedicated node, called the recorder, has a watchdog process for each processor of the network. If no messages are received in a time limit, the processor is considered to have crashed. This approach is simple but has a weak point in the recording node. Another centralized approach was chosen in [19], which uses an election algorithm to choose the master node.

At the other extreme, we found the detection mechanism of the TANDEM NonStop system [20], where each node tests every other node in the network. The DIP approach falls between the centralized and the fully distributed approach.

A practical algorithm can be found in [21]. It has some similarities with the DIP. However, that proposal has some shortcomings. First, it leaves the failure detection job to the application. Second, when a node fails, the number of diagnosis messages is $2N^2 - 6N + 4$ (where N is the number of nodes). This means that the network will be flooded with messages, and the diagnosis latency will suffer from that.

The algorithm presented by Walter [22] is similar to the DIP - it also organizes the network as a virtual ring of processors. But, one may notice that it is not robust against lost messages, duplicates, out-of-order messages and false alarms, since it assumes a reliable network. The critical message path of the diagnosis is 3, while the critical path of the DIP is just 1, which means that the diagnosis latency is at least 3 times higher than the latency of DIP. During steady-state operation the RSC protocol always requires the same number of messages : $P.N$.

Another algorithm implemented in the SDD/1's RelNet [23] provides a systematic detection of failures and node recoveries. Nevertheless, that

algorithm is not robust against network partitions. Another set of algorithms has been proposed for synchronous distributed systems [24-27].

7. Implementation Features

The first release of the DIP was implemented on a set of SUN workstations which communicate through an Ethernet local area network. It was written in C using sockets UDP/IP. The protocol itself was running as a background process. The initialization module stores in stable storage the list of the processors in the network. This list is sorted by the Internet host address. As this number is unique to each processor it assures that all the hosts generate identical lists. All the crash and recovery events are stored in the *Crash_Log* file, which can be inspected for failure analysis. A query interface is also available to users asking for the network status. The parameters chosen were the following : $\alpha = 1$ second and $\Omega = 2$.

The DIP provides a failure detection capability to be incorporated in a recovery mechanism, based on a distributed checkpointing algorithm presented elsewhere [28].

8. Conclusions and Future Work

We have presented a protocol for checking the availability of sites in local area networks. During steady-state operation it uses the minimum number of messages, while in case of failure it assures the lowest diagnosis latency. Its testing graph is adaptive which provides a complete diagnosability. It is also robust against false alarms, out-of-order messages and network partitions. The only shortcoming is its reduced failure model. If anyone wants to extend the model should try one of the techniques presented by Neiger and Toueg [29] to automatically translate this algorithm, that only tolerates crash failures, into another one that tolerates more "severe" failures (i.e. arbitrary or byzantine). As future work we are planning to implement the protocol on a network of transputers.

Acknowledgments

L.M.Silva was supported by JNICT under the Programa Ciência (BM-38-90-IA). This work was partially supported by INIC and by JNICT under contract PMCT/C/TIT/451/90.

References

1. F.P.Preparata,G.Metze and R.T.Chien . " On the connection assignment problem of diagnosable systems" , IEEE Transactions on Electronic Computing, 16, Dec.1967, pp.848-854
2. S.E.Kreutzer and S.L.Hakimi. "System - Level Fault Diagnosis : A Survey", Euromicro Journal , May 1987, pp.323-330
3. J.G.Kuhl and S.M.Reddy. "Distributed Fault-Tolerance for Large Multi-Processor Systems", Proc. 7th Annual Symposium on Computer Architectures, May 1980,pp. 23-30
4. J.G.Kuhl and S.M.Reddy. "Fault-Diagnosis in Fully Distributed Systems", Proc. FTCS-11, 1981 pp. 100-105
5. S.H.Hosseini, J.G.Kuhl and S.M.Reddy. "On Self-Diagnosis of the Distributed Systems", Proc. FTCS-15, 1985, pp. 30-35
6. S.H.Hosseini, J.G.Kuhl and S.M.Reddy. "A Diagnosis Algorithm for Distributed Computing Systems with Dynamic Failure and Repair", IEEE Trans.on Computers,March 1984,pp.223-233
7. S.E.Kreutzer and S.L.Hakimi. "Distributed Diagnosis and the System User", IEEE Trans. on Computers, January 1988, pp. 71-78
8. N.H. Vaidja and D.K. Pradhan. "System Level Diagnosis : Combining Detection and Location", Proc. FTCS-21, 1991 , pp.488-495
9. A.Baghi and S.L.Hakimi. "An Optimal Algorithm for Distributed System Level Diagnosis", Proc. FTCS-21 , 1991, pp. 214-221
10. R.Bianchini , K.Goodwin and D.S.Nydick. "Practical Application and Implementation of Distributed System-Level Diagnosis Theory " Proc. FTCS-20 , 1990 , pp. 332-339
11. R.Bianchini and R.Buskens. " An Adaptive Distributed System-Level Diagnosis Algorithm and Its Implementation", Proc. FTCS-21, 1991, pp. 222-229
12. L.Lamport, R.Shostak and M.Pease. "The Byzantine Generals Problem", ACM Trans. on Programming Languages and Systems, V. 4 1982, pp.382-401
13. K.V.S.Ramarao and J.C.Adams. "On the Diagnosis of Byzantine Faults", Proc. of the 7th Symposium on Reliable Distributed Systems , 1988, pp.144-153
14. K.C.Shin and P.Ramanathan. "Diagnosis of Processors with Byzantine Faults in a Distributed Computing System", Proc. FTCS-17, 1987, pp.55-60
15. J.C.Adams and K.V.S. Ramarao. "Distributed Diagnosis of Byzantine Processors and Links", Proc. 9th Int. Conf. on Distributed Computing Systems, 1989 , pp.562-569
16. R.A. Maxion and F.E. Feather . "A Case Study of Ethernet Anomalies in a Distributed Computing Environment ", IEEE Transactions on Reliability , Vol 39, October 1990 , pp.433-443
17. J.F. Schock et al. " Evolution of the Ethernet Local Computer Network", IEEE Computer , August 1982 , Vol. 5 (8) , pp. 10-27
18. M.L.Powell and D.L.Presoto. "Publishing : A Reliable Broadcast Communication Mechanism", Proc. 9th Symposium on Operating Systems Principles, 1983 , pp.100-109
19. T. Becker. "Keeping Processes Under Surveillance ", Proc. 10th Symposium on Reliable Distributed Systems,1991,pp.198-205
20. C.I. Dimmer. "The Tandem Non-Stop System", in *Resilient Computing Systems* , Ed. by T. Anderson,1985, pp. 178-196
21. S.Bruso. "A Failure Detection and Notification Protocol for Distributed Computing Systems", Proc. 5th Int. Conf. on Distributed Computing Systems, 1985, pp.116-123
22. B.Walter. "A Robust and Efficient Protocol for Checking the Availability of Remote Sites", Computer Networks, Vol. 6, 1982 ,pp. 173-188
23. M.Hammer and D.Shipman . "Reliability Mechanisms for SDD-1 : A System for Distributed Databases", ACM Transactions on Database Systems, Vol.5 (4) , Dec. 1980, pp. 431-466
24. F.Cristian."Agreeing on Who Is Present and Who Is Absent In a Synchronous Distributed System" Proc. FTCS-18, 1988, pp. 206-211
25. H.Kopetz,G.Grunsteidl and J.Reisinger. "Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System", in *Dependable Computing for Critical Applications*, Springer Verlag, 1991
26. B.Walter. "Network Partitioning and Symmetric Surveillance Protocols", Proc. 5th Int. Conf. on Distributed Computing Systems,85,pp.124 -129
27. A.V.Ma and G.G.Belford."A Failure and Recovery Detection Protocol for Optimistic Partitioned Operation in Distributed Database Systems", Proc. 6th Int. Conf. on Distributed Computing Systems,1986, pp.540-547
28. L.M.Silva , J.G. Silva. "Global Checkpointing for Distributed Programs", Proc. 11th Symp. on Reliable Distributed Systems, pp. 155-162, 1992, Houston (USA)
29. G.Neiger and S.Toueg . " Automatically Increasing the Fault-Tolerance of Distributed Systems", Proc. 7th Symp. on Principles of Distributed Computing, 1988, pp.248-262