© 2007 JATIT. All rights reserved.

SIMULATION OF SYSTEM LEVEL DIAGNOSIS IN DISTRIBUTED ARBITRARY NETWORK

Lakshmi Prasad Saikia*, **Kattamanchi** Hemachandran**
*Senior Lecturer, CSE, Deptt of ETE, NIT Silchar, Assam, India. Email: lp_saikia@yahoo.co.in
**Professor,CS, AssamUniversity-Silchar, Assam,India. Email: khchandran@rediffmail.com

ABSTRACT

The problem of distributed diagnosis in arbitrary network failures and repairs is considered in this paper. The basic idea behind incorporating a fault tolerance capability to a distributed system is to provide the system with extra (redundant) resources. A number of investigations have been attempted to extend traditional notions of "fault-tolerant computing", to deal with the problem of failures, which affect the facilities of distributed systems and computer networks. as distinguish of diagnostic responsibility requires the flow of diagnostic information through the network, and the faulty facilities themselves may participate in this flow and may alter, destroy, or generate erroneous diagnostic information in the process, the whole diagnostic procedure itself becomes quite complex. The purpose of this study is to simulate a distributed system and carry out fault diagnosis under Arbitrary Network topologies. The distributed system level diagnosis algorithms discussed in this paper give a comprehensive idea about various issues one must keep track of while going to develop such fault tolerance algorithms. Since "system level diagnosis" is one of the steps in the process of building "distributed fault-tolerant systems", reliability of such a system depends heavily on proper functioning of the diagnosis algorithm.

The distributed system has been simulated in JAVA using the Console.Java program to create multiple windows, each one representing a different system node. Afterwards, distributed diagnosis algorithms have been simulated using this environment.

Key Words: Distributed diagnosis, System-Level diagnosis, Arbitrary Network, failures and repairs, fault tolerance, faulty, fault-free.

INTRODUCTION

When a group of people work together, there is a need to communicate with each other, to share data and expensive resources such as high quality printers, disk drives, etc. This requires interconnecting of computers and resources. Designing such systems became feasible with the availability of cheap and powerful microprocessors, and advances in communication technology.

The main advantage of distributed systems is that they have a decisive ***price*** and ***performance*** advantage over more traditional time-sharing system. Other significant advantages of distributed systems over traditional time-sharing systems are:-Resource sharing, Enhanced performance, Improved reliability and availability, and Modular expandability.

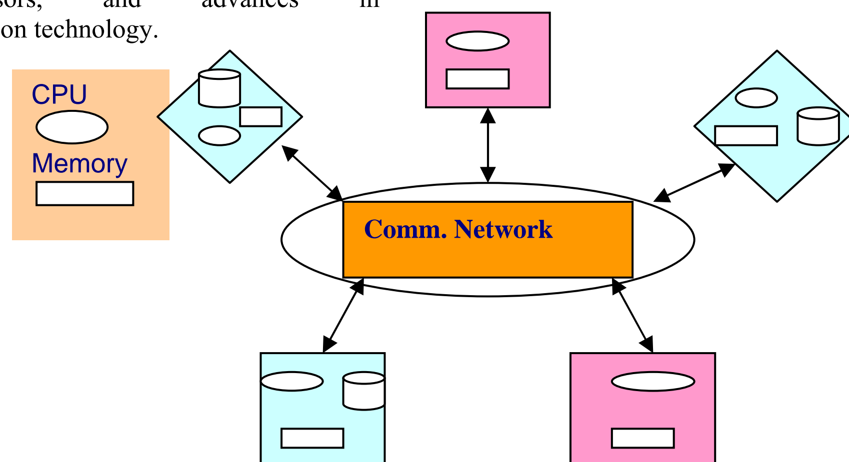

Figure 1: Architecture of a Distributed System

© 2007 JATIT. All rights reserved.

In distributed diagnosis, each working node must maintain correct information about the status (working or failed) of each component in the system. A node can be either fault-free or faulty. A fault-free node performs its specified system computation and commutation tasks correctly, and it has a local notion of time. A fault-free node is assumed to know which nodes are its physical neighbors in the network. This information can be ascertained by the node via external means or internally, through low-level hardware and software methods. Proper designing of Algorithms for System-Level Diagnosis is one of the most important phases in the process of building *distributed fault-tolerant system*s. 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 these units.

In a Distributed System-level Diagnosis algorithm for Arbitrary Network, fault-free processors perform simple periodic tests on one another; when a fault is detected or a newly–repaired processor joins the network, this new information is disseminated in parallel throughout the network.

RELATED WORKS

In 1991, Bagchi and Hakimi [1] presented an algorithm for diagnosing faulty processors in *arbitrary networks*. Initially, each fault-free processor knows only about itself and its physical neighbors. Fault-free processors start the algorithm by waking up (multiple processors are allowed to wake up simultaneously) and initiating the formation of a *tree-based* testing topology. Multiple trees that are being formed simultaneously are merged into one. Diagnosis information is sent along with the messages that form the tree. The authors show that the algorithm is optimal in that it requires at most 3n log(p)+O(n+pt) messages, where is the number of fault-free processors that start the tree and is the number of faulty processors. However, the algorithm requires that no processor becomes faulty and that no processor is repaired during the execution of the algorithm. Dissemination of the diagnosis information proceeds *sequentially* through the network due to the nature of the tree formation process. Thereafter, Stahl *et al.* and Bianchini *et al.* also addressed the problem of performing distributed system-level diagnosis in arbitrary networks [2], [3]. The strategies are event-driven, as in their earlier work; however, the response to an event differs. In the so-called *Adapt algorithm* [2], the underlying testing topology is strongly connected among all of the fault-free processors; when an event occurs, a search phase begins in which tests are added locally as the information about the event is disseminated to maintain the strong connectivity of the testing topology. This phase is performed in parallel. Afterwards, a destroy phase removes redundant tests so that a minimal strongly connected testing topology remains. This phase must be performed sequentially. In the *Adapt2* algorithm [3] for arbitrary networks, the underlying testing topology is a tree. Each processor is tested by its designated parent in the tree, except for the designated root node, which is tested by one of its children. When an event is detected, the processor that detects the event is the new designated root node. The new information is propagated through the network using a *depth-first search* (DFS) technique, which is inherently sequential. The path the message containing this information takes, in traversing the network, determines the new tree-based network topology.

Adaptive Distributed system-level diagnosis (ADSD) algorithm 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 [3] [4].

© 2007 JATIT. All rights reserved.

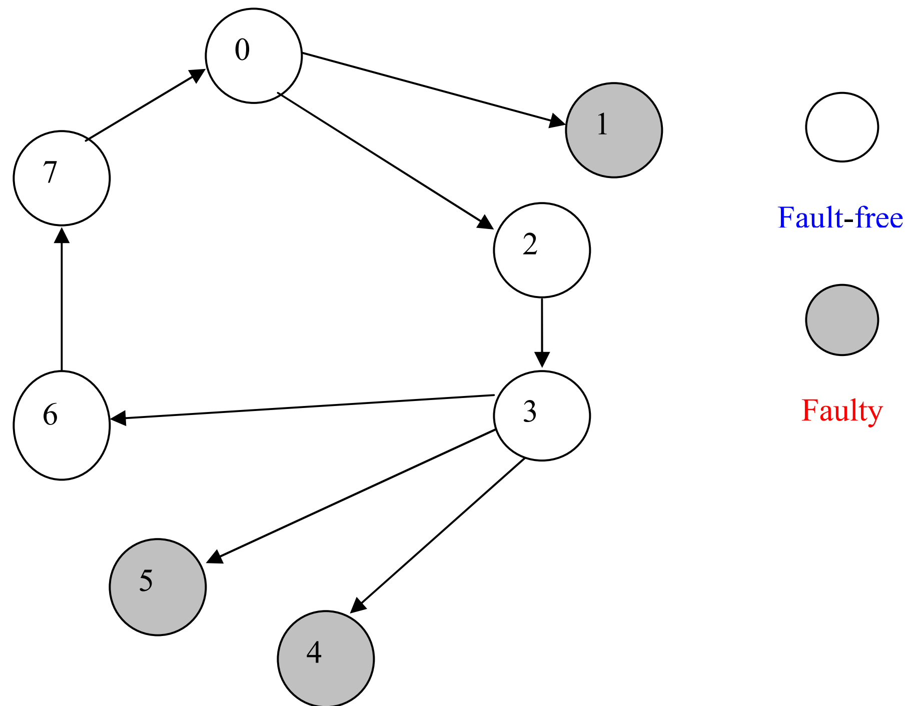

Figure 2: Example of test assignment in Adaptive-DSD

Node1, node4 and node5 are faulty, and rests are fault-free. Node0 tests node1 and finds it faulty, so it goes on and tests node2, which is fault-free, and then stops testing. Node2 then tests node3 as fault-free, and so on.

DISTRIBUTED SYSTEM-LEVEL DIAGNOSIS:

A distributed algorithm for system-level diagnosis in an arbitrary network is presented. In this algorithm, nodes *test,* or *monitor,* one another periodically such that each fault-free node is tested by exactly one other node. When a node detects that a fault-free node it is monitoring has become faulty or that a faulty neighbor has been repaired, it propagates this new information to its fault-free neighbors, which propagate the information to their neighbors, and so on. In this way, the system overhead due to monitoring is minimized during periods when the status of the nodes in the system does not change, and new information is disseminated as quickly as possible throughout the network. Each fault-free node is responsible for ensuring that exactly one of its fault-free neighbors (if one exists) is testing it. A procedure is described for guaranteeing that this property holds during the execution of the algorithm.

THE ADVANTAGES OF THIS ALGORITHM ARE AS FOLLOWS:

Nodes in the system can become faulty or can be repaired an arbitrary number of times during the execution of the algorithm, The detection of faulty nodes or newly-repaired nodes and the dissemination of this information are conceptually separate; in this way, the dissemination stage can be *parallelized* to reduce the time, henceforth called the *information latency*, required for all of the other fault-free nodes in the system to learn about the event once it is detected, Parallelizing the dissemination stage allows for nodes that are local to the event to, in general, learn about the event before more distant nodes, A newly repaired node can rejoin the system without relying on other nodes to first detect that it has been repaired; equivalently, faulty nodes do not have to be periodically tested, The concept of a *validating transaction* is introduced, in which the testing process is both strengthened and simplified without loss of information, There is no requirement for a global clock or *synchronized clocks* in the system.

© 2007 JATIT. All rights reserved.

This algorithm makes evident the tradeoff between information latency and message overhead. In networks consisting at least partially of point-to-point communication links (which is the case for the majority of arbitrary topologies), it is argued that the increased message overhead that leads to an optimal information latency is tolerable.

SYSTEM AND DIAGNOSIS MODEL:

A system is assumed in the most general case to consist of a collection of heterogeneous processing elements, or *nodes*, interconnected via point-to-point communication links, broadcast busses, or an arbitrary combination of each.

A node can be either *fault-free* or *faulty*. A fault-free node performs its specified system computational and communication tasks, and it has a local notion of time. A fault-free node is assumed to know which nodes are its physical neighbors in the network. This information can be ascertained by the node via external means or internally, through low-level hardware and software methods.

In addition, a fault-free node is assumed to be able to initiate a *test* of a neighboring node and to be able to respond to a test initiated by one of its neighbors. A fault-free node, by definition, responds correctly and within a specified time-out period to a test. Finally, a fault-free node is able to request that a neighboring node become its tester when dictated by the diagnosis algorithm.

In contrast, a faulty node is assumed to be unable to respond to a test, to a request, or to diagnosis information sent to it from a neighboring node. Also, a faulty node is assumed to be unable to format and forward diagnosis information messages and is assumed to be unable to generate spurious requests fro other nodes to test it. It is assumed that the node that issues a message to a neighboring node receives the corresponding reply (such as a test response or an acknowledgement) within a certain time period if and only if the neighbor is fault-free; otherwise, it times-out on the reply. In order to satisfy this requirement, it is necessary that the communication channels between the nodes have a bounded delay. Time-out periods are determined as a function of this delay. Also, it is assumed that message ordering is maintained within the communication channel; equivalently, an appropriate communication protocol can be used by the sender and receiver to guarantee that messages are received in the order they are delivered. A fault-free node can become faulty at any time. A faulty node can be repaired and reintegrated into the network at any time. When a faulty node is repaired, it is assumed that it regains all of the attributes of a faulty-free node, including ascertaining who its physical neighbors are. However, a newly repaired node has no knowledge of the fault-free or faulty status of the other nodes in the system.

This paper assumes no communication link faults. A test could fail due to a faulty node or due to a faulty communication link, without distinction. That means it treats link faults as node faults.

ALGORITHM DESCRIPTION:

In this diagnosis algorithm, nodes detect failures in neighboring nodes and then propagate this information to the other nodes in the network in two discrete steps: *detection* and *dissemination.* The failure information the nodes propagate consists of *failure events*, where a failure event is defined to be a transition of a node from either *fault-free to faulty* or *faulty to fault-free.*

The first step is to detect a failure event. This is accomplished by nodes testing each other periodically, such that each node in the network is tested by exactly one other fault-free node. When a failure event is detected by a node by timing-out on a response to a test on another node, it executes the dissemination step. In this step, the node that detects the failure event propagates this information to all of its neighbors in the network, which propagate the information to their neighbors, and so on.

VALIDATING TRANSACTIONS:

Information is propagated by means of a *validating transaction*. When a node i propagates information to another node j, it is required that i determine whether j is faulty or fault-free. In such a validating transaction, if i have information to be sent to j, it sends a message to j, which in turn processes this message and sends a *confirmation message* back to i. The confirmation message sent from j to i is a function of the contents of the message. That is, j if is faulty when it receives this message, even if it becomes fault-free before it sends a confirmation message back to i, i will either time out or i receive a wrong confirmation message; and this will let i know that j was faulty when it received the message from i.

When a fault-free node j receives a validating transaction message from another node i, it checks to see if the information contained in the message is the

© 2007 JATIT. All rights reserved.

same or *old* or *new* compared to its own information. If the information is the *same*, then j has the same information as i about the status of *all* the nodes in the network. If the information is *old*, then j has more recent information than i about at least one of the nodes and the same information about the rest of the nodes. If the information is *new*, than j has older information than i about the status of at least one of the nodes in the network and has the same or new information about the rest of the nodes. Note that incoming messages must be same, or old, or new relative to a node's existing information.

If the incoming information is the same (referred to as *sameinfo*), then this message is not propagated further by the receiving node. If the information is old (referred to as *oldinfo*), then the information contained in the message is updated and sent back to (only) the sender of that message. If the information is new (referred to as *newinfo*), then the local information as well as the information contained in the message (if needed) are updated and a validating transaction containing this message is sent to all the neighbors.

If a node j fails to send a correct confirmation message when it receives a message from i node, i will start disseminating information regarding this fault event (a transition of j from fault-free to faulty) assuming that j is faulty, even though j may have become fault-free but could not reply with the correct confirmation message because it was faulty when it received the message from i. Thus, other than direct tests, validating transactions also serve as tests on nodes. Given that lower level protocols that guarantee message delivery require acknowledgement by including a confirmation message as described above. Hence, it is quite natural to use validating transactions as tests.

FAULTY AND FAULT-FREE ORPHANS:

A node is called an *orphan* if no other node is testing it. A node can become either a *fault-free orphan* or a *faulty orphan*. If a fault-free node becomes faulty, its tester detects this failure event by timing out on a test response; it then disseminates the failure event information and stops actively testing the node any further. Therefore, all faulty nodes, once they are detected by their tester, become faulty orphans. When a faulty node is repaired, it requests one by one for its neighbors to test it until it finds a fault-free neighbor that acknowledges its request. Then it is no longer an orphan. The neighbor that has acknowledged to test the node starts disseminating information about the failure event (faulty orphan becoming fault-free) to the rest of the nodes in the network by sending a validating transaction to all of its neighbors. If a fault-free node's tester becomes faulty, then the node becomes a fault-free orphan because each node is tested by only one other node. In this case, the orphan will not necessarily realize immediately that no one is testing it. Eventually, one of its fault-free neighbors, if any, sends the orphan new information about the status of the nodes in the network. Once this happens, the orphan realizes that it is not being tested by anyone and requests one by one for its neighbors to test it until it finds a fault-free neighbor that acknowledges the request. A fault-free orphan can become faulty before information about its tester's failure reaches it. If this happens, the failure event (fault-free orphan becoming a faulty orphan) is not detected immediately. Eventually, the neighbor who sends information about the status of the network to the faulty orphan notices this fault event when it receives a wrong confirmation message from the faulty orphan or times out because it does not receive the confirmation message on time, and will then start disseminating information about the failure event.

DATA STRUCTURES:

Data needed for the execution of the algorithm is maintained in local data structures at the individual nodes and also in the data structures carried by the messages propagating through the network. From now on, we will refer to these as *local data* and *message data*, respectively.

Let N represent the total number of nodes (processors) in the network. Consider a node j. The local data stored at node j consists of the following:

1)An array of event-counters, denoted by $event_j[1\ldots N]$, where $event_j[i]$ contains a counter value for the most recent failure event that was detected at node I by the node that was testing node i at that time, as per the information that has reached node j so far. These events are measured in terms of the number of fault event changes. Thus, if a node k detects a fault event at node i that it tests, then it increments $event_k[i]$ by one. The event-counters are updated such that after the update, if event-counter $event_j[i]$ is even, then node i is considered fault-free by node j, and if is $event_j[i]$ odd then node i is

© 2007 JATIT. All rights reserved.

considered faulty by node j, as per the information that has reached node j so far.
2)A test/neighbor array denoted by $testnb_j[1...N]$ where $testnb_j[i]$ is 0,1,2,3, or 4, depending on whether node i (0) is *not* a neighbor of j, 1) is tested by j, 2) is testing j,
3) neither i nor j is testing each other but they are neighbors, or 4) i tests j and j tests i. Assuming that only neighbors can test one another, this means if $testnb_j[i]$ is 1,2,3,or 4, then i is a neighbor of j and if $testnb_j[i]$ is 0, then i is not a neighbor of j.
The message data consists of the following:
1)An array of event-counters *msg.event*[1…N], where *msg.event*[i] contains the counter value for the most recent failure event that was detected at node i by some other node in the network, as per the information that the message has gathered so far through its transit through the network. Similar to the event-counter in the local data, an even-counter value (after it has been updated as a result of the failure event) denotes a fault- free node, and an odd event-counter value denotes a faulty node.
2)An array of bits *intrapath*[1…N], where *intrapath*[i] is a 1 if the message has already visited node i.This is used to reduce the number of redundant messages that would otherwise be sent, as explained later. The *intrapath* array is initialized to 0 when a node forms a message with new information to be sent out.
In both the local data and the message data, it is redundant to carry information about the status (faulty/fault-free) of the nodes, since this can be deduced from the event-counter, as described earlier.

Specification of the Algorithm: A complete specification of the fault diagnosis algorithm for arbitrary network topologies has been given. The diagnosis algorithm is continually run as a background task in each of the nodes in the system.
A node can be in one of five states with respect to the execution of the diagnosis algorithm. In state 1 a node is *idle*, which means that it has not sent out a test to any neighbor. State 2 is the *wait* state, where the node has sent out a test to a neighbor and is waiting for a response to this test. In this model a node has at most one outstanding test request at a time, though this need not be the case in general. State 3 (*idle-orp*) and 4 (*wait-orp*) are similar to states 1 and 2, respectively, except in this case the node is a fault-free orphan. That is, the node is not being tested by anyone. State 5 (*fail*) denotes the state where the node has failed.

The messages that are sent/ received are as follows.

- *test* and *resp* refer to a test and a response to the test, respectively. If a response is not received by the node issuing the test, then it times out, denoted by the internal event *to(resp).*
- *sameinfo, oldinfo,* and *newinfo* refer to the status of the information that is being sent in a message that is part of a validating transaction. a*ckinfo* is the acknowledgment message that is sent out by a node when it receives information through a validating transaction. If this message is not received within a specified time by the sender of the validating transaction then it times out, denoted by the internal event *to(ackinfo).*
- *testme* is a message that is sent by a fault-free orphan to one of its neighbors asking that neighbor to test it.
- *tmj* stands for "testme-join" and is a message sent by a newly repaired node requesting one of its neighbors to test it as it joins the system. Note that both *testme* and *tmj* are used by nodes to request that they be tested; however, *tmj* also notifies that the sender of the message has just been repaired so that the corresponding failure event (faulty to fault-free) can be detected by the receiver of this message, which disseminates the new information.
- *join* is a message sent when a node joins the system to all of its neighbors except the one to which it sent a *tmj.* This message, in addition to informing the neighbors that the node has been repaired, serves to remove any old test links that may exist from a neighbor to the node, so that each node is tested by no more than one other node. In this case, the only tester of the node is the neighbor that receives (and acknowledges) a *tmj.*
- *acktmj* and *acktm* are acknowledgment messages that are sent if a node that receives a *tmj* or *testme*, respectively, agrees to test the sender. If these messages are not received by the node requesting a tester, then it times out, denoted by the internal events *to*(*acktmj*) and *to*(*acktm*), respectively. When such time-outs occur, the node that wishes to be tested sends a *tmj* or *testme* to another neighbor. This process is repeated until a neighbor that is willing to conduct the test is found.

© 2007 JATIT. All rights reserved.

```
1)        Every node i should proceed as follows:
          Process REPAIR/INITIALIZATION ( i )
          1.1 Process_MSG_READER ( i )    /* msg_reader receives most
                                             messages and starts appropriate processes */
          1.2   j := neighbor
          1.3   Process_ORPHAN( i, j )          /* find parent to test i            */

2)   Process_MSG_READER( i )
          2.1 REPEAT                            /* do until i is faulty             */
          2.2 wait_MESSAGE( j )        /* wait for message from node j      */
          2.3  If j is fault-free
          2.4        switch (type of message)

          2.5           case TEST-message:   /* respond to test message   */
          2.6                 send RESP-message( j )
          2.7            case INFO-message:  /* Disseminate INFO-message        */
          2.8               Process_INFO( i, j )      /* Acknowledge INFO-message       */
          2.9               send  ACKINFO-message( j )

          2.10       case TESTME-message:
          2.11              set to test_j
          2.12              Process_TEST( i, j )      /*        start testing node j              */
          2.13              send ACKTM-message( j )  /*  Acknowledgement testme      */
         2.14          case TESTMEJOIN-message:
          2.15              remove test-by_j  /* remove j's test on i if any              */
          2.16              set to test_j              /*start testing node j                    */
          2.17              Process_TEST( i, j )
          2.18              send ACKMJ-message( j ) /* Acknowledge TMJ-message    */
          2.19              event_counter_i[j]++               /* increment EC to next even */

          2.20              for k := first to last neighbor do /*  Disseminate INFO-msg    */
          2.21                      Procedure_SEND_INFO( i, k )
          2.22              If ORPHAN                      /* if i is ORPHAN then find node*/
          2.23                    Process_ORPHAN( i, 1 ) /* to test i ( 1 <>j<>k )         */

          2.24      case JOINNOTEST-message:
          2.25              remove test-by_j             /* remove j's test on i if any  */
          2.26              remove test_j                        /* and i's test on j if any    */
          2.27              If  ORPHAN                      /* if i is ORPHAN then find node*/
          2.28                      Process_ORPHAN( i, j ) /* to test i, first try j           */
          2.29      FOREVER

3)    Process_ORPHAN( i, j )
          3.1       If event-counter_i [j] even /* if j is thought to the fault-free ask j */
          3.2               send TESTME-message( j ) /*        to test node i                    */
          3.3               timed-wait_ACKTM-message( j )
          3.4               If to( ACKTM )              /* since time-out, j is faulty now       */
          3.5                       event_counter_i [ j ] ++  /* increment EC to next odd       */
          3.6               for  k := first to last neighbor do /* disseminate INFO to all nbs     */
          3.7               Procedure_SEND_INFO( k )
          3.8          Process_ORPHAN( i, mod_n(j + 1)) /*try to find another one to test I    */
          3.9       Else add test-by_j                   /* update testing array              */
          3.10  Else Process_ORPHAN( i, mod_n(j + 1))/*try to find another one to test i     */

   4)    Procedure_SEND_INFO( i, j )
```

© 2007 JATIT. All rights reserved.

```
4.1      send INFO-message( j )            /* disseminate INFO to j               */
4.2      If event-counter_i [j ] even       /* if j is thought to be fault-free   */
4.3            timed-wait_ACKINFO-message( j )        /* Timed wait for Ack        */
4.4            If to(ACKINFO)              /* if node j is faulty                */
4.5            event-counter_i [ j ] ++    /* increment EC to next odd               */
4.6            for  k := first to last neighbor do /* disseminate INFO to all nbs    */
4.7                  Procedure_SEND_INFO( k )
4.8                  If  i is an ORPHAN
4.9                   Process_ORPHAN( i, k ) /* find a node k <> j to test i   */

5)  Process_INFO( i, j)

5.1  If  "sameinfo" exit( )  /* if msg has sameinfo as i's, then exit process     */
5.2  If  "oldinfo"            /* if only oldinfo, send INFO-message                 */
5.3      Process_SEND_INFO( i, j )   /* to node j only                                */
5.4  If ("newinfo") or ("newinfo" and "oldinfo") /* if info is new or mixed         */
5.5      update event-counter to "newest" info     /* update EC to maximum       */
5.6      for k := first to last neighbor  /* values and send to all neighbors              */
5.7            Procedure_SEND_INFO( i, k )

6)     Process_TEST( i, j )         /* procedure for node i to test node j. Do until node i      */

6.1  REPEAT                  /* receives JOIN from node j                                   */
6.2      If (i tests j)

6.3      If ( j is fault-free )
6.4          send TEST-message( j )
6.5          timed-wait_RESP-message( j )  /* wait a "timeout" period          */
6.6          If to(RESP)               /* timeout indicates failure of node j              */
6.7            event-counter_i [ j ] ++  /* increment  EC to next odd value        */

6.8            for k := first to last neighbor do /*send INFO to all neighbors      */
6.9            Procedure_SEND_INFO(i, k )

6.10           If ORPHAN process_ORPHAN(i , 1)  /* if ORPHAN find new par*/
6.11         sleep( periodic test delay )
6.12     Else sleep( short time delay )  /* node j is faulty, just wait a short time   */
6.13   Else stop periodically testing j   /* if i not to test j then end process                    */
```

The diagnosis algorithm uses two fault detection mechanisms. One mechanism uses a test sent from one node to another to detect failures; the other mechanism uses time-out on an acknowledgement that is expected when a message is sent from one node to another (as part of a validating transaction). It should be clear that if the last failure event is the failure of a node that is being tested by one of the nodes in the fault-free connected component, then this will be detected. If the last event is the simultaneous failure of multiple nodes (all neighbors of the given fault-free connected component), then all of these failures will be detected if at least one of these nodes is tested by a node in the connected component. This is because of the fact that

the node that is tested from within the fault-free connected component will be detected by the *test* and the other failures, which are neighbors of the connected component, will be detected by the time-out on the acknowledgement to the *newinfo* that is sent to all of these nodes by nodes within the fault-free connected component. There is the possibility of the last failure event being a set of multiple failures leading to a *jellyfish* faulty node configuration.

ALGORITHM EXECUTION STEPS AT A NOD*E:*

Consider a fault-free node j in the system. Node j will periodically test a neighbor i if $testnb_j[i]$ = 1 or 4. In

© 2007 JATIT. All rights reserved.

the algorithm, only fault-free nodes are tested, and so node i, in node j's view, have to be fault-free. If node j detects a failure event (in this case, node i goes from fault-free to faulty), then node j will do the following: 1.increment $event_j[i]$ by one.2.set $testnb_j[i]$ to 3, which means that node j will not actively test node i any more. That is, node i becomes a faulty orphan.

Node j will then start the disseminate step. In the disseminate step, a validating transaction is created with a message. The message is updated with the event-counter information as msg .event [i] = $event_j[i]$, for all i, $1 \leq i \leq N$ and *intrapath* information as *intrapath*[j] = 1. The rest of the entries in the *intrapath*[ ] array are initialized to zero. This means that this message has visited only node j, the initiator of this message, so far. This transaction is sent to all the fault-free (as per the information that j currently has) neighbors of j. This is done by executing a *send-transaction* procedure, invoked as

{ $\forall_k$ | $event_j[k]$ is even and $testnb_j[k]$ = 1, 2, 3, or 4: *send-transaction(k)* }.

When a message was sent by a fault-free node k, then node j uses the contents of the message to produce the confirmation message and sends it to k. It then checks to see if it is a fault-free orphan (if it is not already one). Each node is tested by exactly one node. So, node i for which $testnb_j[i]$ = 2 or $testnb_j[i]$ = 4 is the node that tests j. Node j will check to see if msg.*event*[i] is even. If it is odd, j determines that it is a fault-free orphan as its tester has become faulty. Also, it checks to see if according to the information that j had, k is fault-free; that is, if $event_j[k]$ is even. If not, it increments $event_j[k]$ by 1 (deem k fault-free). Then, j checks to see if the message carries the same, or old, or new information. If the information is the same, the message is discarded. If it is old, then the message information is updated using the local information and this message is sent out to (only) node k. The update is made as $\forall$i , if, $event_j[i]$ > msg.*event*[i], msg.*event*[i] = $event_j[i]$. If the message is new, then the appropriate message data and local data updates are made. That is, $\forall_i$, if msg.*event*[i] > $event_j[i]$, $event_j[i]$ is updated to msg.*event*[i] and if, $event_j[i]$ > msg.*event*[i] , msg.*event*[i] is updated to $event_j[i]$. Then, a message is created and forwarded to all the neighbors. If it finds that it is a fault-free orphan, it repeatedly sends a message to one of the neighbors requesting it to become its tester until it finds a willing neighbor, as described earlier.

When a faulty node j receives an information message from one of its neighbors (say k), it cannot produce a correct confirmation message. Even if node j becomes fault-free before sending the confirmation message, it can only send a wrong confirmation message back to k. In either case, node k will realize that node j was faulty when it received the transaction from k. The local information at node k shows that node j was fault-free (because k would have sent this transaction to j only if it had information that said that j was fault-free). In this case, it is possible that the failure of node j was not noticed because it was a fault-free orphan and then failed. So, node k updates this information in its local data by incrementing $event_k[j]$ to odd, configures a new message with this information, and forwards it to all of its neighbors.

When a faulty orphan j becomes fault-free, it has no one testing it. It requests (by sending a *tmj*; it sends a *join* to the other neighbors), one by one, for its neighbors to test it until it finds a willing fault-free neighbor, say node k, which acknowledges this request by issuing an *acktmj*. Node k updates $testnb_k[j]$ to 1 and then checks to see if node k has local information that node j was faulty ($event_k[j]$ is odd). If so, node k increments $event_k[j]$ by 1. If not, it increments $event_k[j]$ by 2. It then starts disseminating information about this fault event by forwarding a transaction to all its neighbors. Note that if node k has information that node j is fault-free when the testme-join request came in, that means information about j's failure has not reached k yet. Therefore, k increments $event_k[j]$ by 2 to record that j is still fault-free but has gone through a transition from fault-free to faulty that was not known to k. If node j had failed and been repaired a multiple number of times, all of this information will be in transit and will eventually reach k. In this case, node k updates its event-counter to the appropriate even value once all of the messages arrive at k.

SIMULATION:

In simulation of the network, the following programs and classes are used.

1. systemgraph.class: It has *event_counter* and *testnb* arrays (the purpose of these arrays was explained in the previous section). It also contains the following methods which manipulate its arrays:

© 2007 JATIT. All rights reserved.

1.1 public void proc_ORPHAN( ); //implements listing (3) of Pseudo-Code.

1.2 public void proc_SEND_INFO(info_msg img,int m_r_node); //implements listing (4) of Pseudo-Code.

1.3 public void proc_INFO(info_msg img); // implements listing (5).

2. The nodei.java (for any i=1, 2…, N): It runs 2 infinite threads, server thread (which waits for messages from other nodes), and Decisionmaking thread (which periodically diagnoses the status of other nodes in the network based on the information stored in *event_counter* array).

3. The tester .class is a thread that tests a node to get status of that and broadcasts the test results to its neighbors.

4. The server.class is a thread that is waiting at each node i for messages from other nodes in the system. For each such message it is receiving, it runs a w_r_thread thread that processes that message. The w_r_thread.class is a thread for processing the messages received by node i from other nodes in the system.

5. The Decisionmaking.class is an independent thread that periodically diagnoses the status of other nodes in the network based on the information stored in *even_counter* array (i.e. For any node k <> i, if even_counter[k] is even, then k is fault-free else faulty).

6. The t_msg.class is used to compose request/acknowledgement messages.

7. The info_msg.class is used to compose messages based on test results for broadcasting purpose. These messages are composed from the data stored in local data structures. It has *event* and *intrapath* arrays for that.

SIMULATION OUTPUT:

As per implementation of the algorithm for arbitrary networks concerned, 9 nodes have been considered as 9 consoles. It is interesting to see visually each of the nodes running on a respective Console. In each of the Console, multiple threads are running by which execution of the respective algorithm's output can be viewed. As there is scrolling facility in each of the Console, we can view each of the steps running in a particular node.

As node. java program is concerned, no need to run simultaneously each of the nodes; rather it automatically runs specific number of nodes simultaneously as specified in that program. For each of the node, respective Console is executed.

In figure 3, system level diagnosis algorithm for arbitrary network is running on multiple windows shown execution of the event counter list and diagnosis information of each of the node.

© 2007 JATIT. All rights reserved.

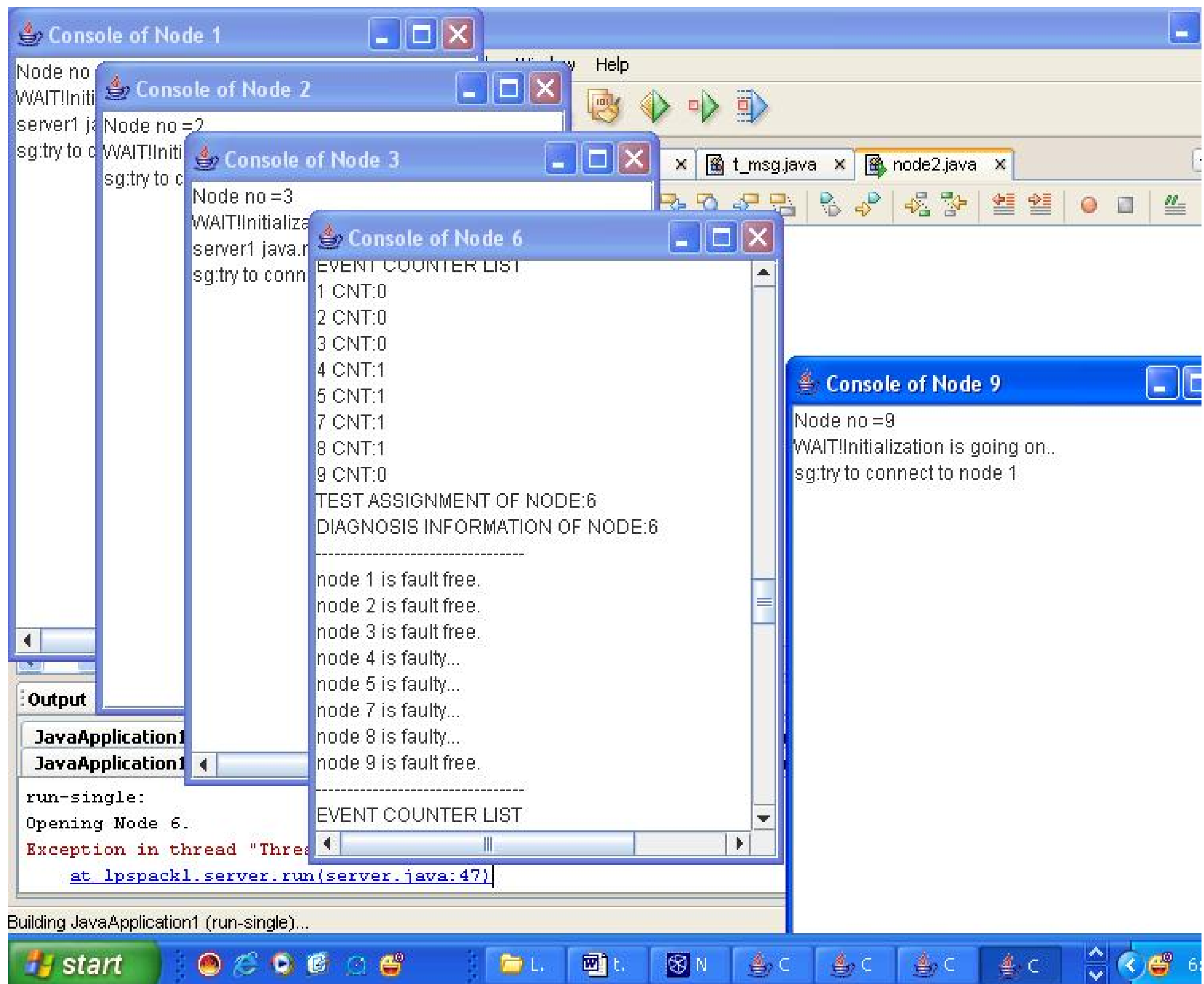

Figure 3. Here Consoles of node 1, 2, 3, 6 and 9 are running on its own window. Each of the console, there are different threads are running. Console for node 6 shows the diagnosis information as running nodes are fault-free where node 4, 5, 7 and 8 are shown faulty ; also even counter list (counter value against node).

CONCLUSION:

The distributed system level diagnosis algorithm discussed give a comprehensive idea about various issues one must keep track of while going to develop such fault tolerance algorithms. Since "system level diagnosis" is one of the steps in the process of building "distributed fault-tolerant systems", reliability of such a system depends heavily on proper functioning of the diagnosis algorithm.

In this paper distributed algorithm for fault diagnosis that uses parallel dissemination of fault event information to minimize the information latency in the network is presented. Even though it works for arbitrary networks, it assumes no link failures. A newly repaired node can rejoin the system without relying on other nodes to first detect that it has been repaired; equivalently, faulty nodes do not have to be periodically tested.

REFERENCES

1. A.D. Singh, Sigaravel Murugesan, "Fault Tolerant Systems", IEEE Computer, July 1990.
2. A. Bagchi and S.L. Hakimi, "An optimal algorithm for distributed system level diagnosis ", in proc. 21st Int. Symp. FT computing, June 1991.

© 2007 JATIT. All rights reserved.

3. P. Bianchini and R. Buskens, "An adaptive distributed system level diagnosis algorithm and its implementation," Proc. FTCS-21, pp222-229, 1991.
4. M.Stahl, R.Buskens, and R. Bianchini, "The Adap2 on line diagnosis algorithms for general topology networks", in proc. Globecom, pp610-614, 1992.
5. M.Stahl, R.Buskens, and R. Bianchini,"On line diagnosis in general topology networks", in proc. Workshop Fault-Tolerant Parallel and Distributed systems, July 1992.
6. P. Bianchini and R. Buskens,"Implementation of on-line distributed system-level diagnosis theory,"IEEE Trans. Computers, vol. 41, pp.616-626, 1992.
7. S. Rangarajan, A. T. Dahbura, and E. A. Ziegler, "A distributed system-level diagnosis algorithm for arbitrary network topologies", IEEE Trans. Computers, vol.44, No.2, pp. 312-333, Feb. 1995.
8. A.K Somani and N H Vaidya, "Understanding Fault-Tolerance & Reliability", IEEE Computer ,Vol. 30,No. 4, pp 45-50,April 1997.
9. E.P. Duarte Jr. and T.Nanya, "A Hierarchical adaptive distributed system-level diagnosis algorithm," IEEE Trans. Computers, vol.47, No.1, pp.34-45, 1998.
10. A. Subbiah, D.M. Blough, "Distributed Diagnosis in Dynamic Fault Environments", IEEE trans. on Parallel and Distributed Systems, Vol. 15, No.5, May 2004.
11. Saikia L. P., "Java based simulation on distributed diagnosis", 15th IASTED proc. on Applied Simulation and Modeling 2006, pp. 406-411, June 2006.