A Distributed Algorithm of Fault Recovery For Stateful Failover

Indranil Saha[1] and Debapriyay Mukhopadhyay[2] *

[1] Honeywell Technology Solutions Lab (HTSL),
151/1, Doraisanipalya, Bannerghatta Road, Bangalore 560 076, India
Email: indranil.saha@honeywell.com
[2] Ixia Technologies,
Infinity Tower II, 8th Floor, Block GP, Sector V, Salt Lake, Kolkata 700091, India
Email:dmukhopadhyay@ixiacom.com

Abstract. In [8], a high availability framework based on Harary graph as network topology has been proposed for stateful failover. Framework proposed therein exhibits an interesting property that an uniform load can be given to each non-faulty node while maintaining fault tolerance. A challenging problem in this context, which has not been addressed in [8] is to be able to come up with a distributed algorithm of automated fault recovery which can exploit the properties exhibited by the framework. In this work, we propose a distributed algorithm with low message and round complexity for automated fault recovery in case of stateful failover. We then prove the correctness of the algorithm using techniques from formal verification. The safety, liveness and the timeliness properties of the algorithm have been verified by the model checker SPIN.

Keywords: distributed algorithm, stateful failover, verification of programs, SPIN model checker.

1 Introduction

Critical business processes and mission critical systems should provide a high degree of availability and reliability to the end users and redundancy techniques are mostly used to achieve this. For distributed systems, redundancy can be achieved by using extra copies of its components which include hardware, software and network components. These systems are often called *highly available systems* or *fault tolerant systems* and they are capable of tolerating against certain kind of failures which can be either software or hardware component failures.

From the availability perspective, systems can be broadly classified into two heads depending on their intended application. In one hand, we have applications in which availability is one of the requirements, but occasional loss of application state information or data is tolerable. On the other hand, there are

* When this work was carried out, the author was a member of Honeywell Technology Solution Lab, Bangalore, India.

mission critical systems with stringent fault tolerance requirements, such as aircraft flight control systems, where restoration of the state or data pertaining to the application is required for highly accurate recovery. Thus, we have two types of failovers - stateless and stateful. In stateless failover of an application, the system is simply restarted without any state or data restoration.

A number of fault tolerant designs for specific multi-processor architectures have been proposed in the past [2–7]. Graph theoretic models have been extensively used to represent processor-to-processor interconnection structure is represented as a graph. As for example, minimum k-Hamilton graphs are widely used to meet reliability considerations for loop type communication networks [5–7]. Fault tolerant networks based on de Bruijn graph are proposed in [4] and can tolerate up to $k-2$ node faults, where the graph is regular of degree k and have k^n number of vertices for some n. But, none of these works discriminates between stateless and stateful failover. In case of stateless failover, any live node in the network is a promising candidate to take over the processes of any failed node, which in case of stateful failover is not. So the most pertinent questions that need to be addressed for stateful failover are - i) how to distribute the state information of a node across the network, i.e., how to decide who are the nodes to receive the state information of which nodes; ii) how to checkpoint the state information and how often it is required to be done.

A framework of high service availability for stateful failover, which is tolerant up to a maximum of k-processor faults in a network is proposed in [8] based on Harary graph. In this work the answer of the first question has been provided with sufficient theoretical justifications, but the second problem has not been dealt with. It has been assumed that the network is consisted of a set of multi-processors and they are connected via bi-directional links. In addition, this work also relies on the assumption that the processors are equipped with enough computing power and thus can execute more than one processes.

High availability framework proposed there [8] not only enables the network to tolerate a maximum of k processor faults, but also guarantees that in the event of k processor failures, the load can be uniformly distributed across the rest of the network. Design always ensures connectedness among non-faulty nodes in the network, exploiting which it is possible to develop a distributed solution of automated recovery. But, the problem of finding a distributed solution of automated fault recovery has not been considered in that work. In this work, we aim to solve this problem and have come up with a distributed algorithm of automated fault recovery for stateful failover based on the framework proposed in [8] utilizing all the properties of the framework. Analysis shows that message and the round complexity of the algorithm are considerably low. Distributed algorithms are very hard to design and more so is to prove that they are correct. So, formal verification can be applied to prove the correctness of the distributed algorithms. Model checking is one such technique where a formal model of the system is constructed against which the properties of the system are verified. To prove the correctness of our algorithm we have used the model checker SPIN,

which is extensively used to formally verify distributed algorithms. Relevant properties of our algorithm, we have verified here using SPIN.

We organize our work as follows. Section 2 outlines the network and fault model and also reviews the framework for k-fault tolerance proposed in [8]. Distributed algorithm ensuring automated fault recovery is described in section 3 along with its analysis. In section 4, we describe the formal verification experiment on the proposed distributed algorithm. Section 5, finally, concludes our paper.

2 High Availability Framework for k-fault Tolerance

2.1 Network and Fault Model

In this work we consider the same network and the fault model under which the framework proposed in [8] has been shown to function properly. The network is consisted of the set of nodes N with $|N| = n$ and each node is labeled with a unique id from 0 to $n-1$. We assume that each node is handling one process initially, and is capable of executing at most m processes simultaneously. We denote by p_i, the process which node $i \in N$ starts executing initially when the network becomes functional. We consider in this work failstop kind of failures, i.e., the nodes in the network can stop operating at any point of time due to a crash and k node faults are allowed in the network. With a processor failed, all the links incident on that node also becomes non-functional.

Given the set of nodes N, each node $i \in N, (0 \leq i \leq n-1)$, in the network is connected to the set of nodes $P_i \subseteq N$, such that $|P_i| = l = k + x$, where $k + x (\leq n-1)$ is even, and

$$P_i = \{j \in N : j = (i+p)(mod\ n), where - l/2 \leq p \leq l/2, p \neq 0\} \qquad (1)$$

Thus, the underlying undirected graph modeling the network can be written as (N, E) where $E = \cup_{i=0}^{n-1}\{(i,j) : j \in P_i\}$. Since, we are talking about undirected graphs, so (i,j) and (j,i) represents the same edge (link) connecting node i with j. Its easy to see here that the graph (N, E) represents a regular network, for, the degree of each node is l.

Thus, for any n and k, the graph (N, E) corresponds to the Harary Graph [1] $H_{l,n}$, where

$$l = k + x \geq \begin{cases} k+2, & \text{for } k \text{ even,} \\ k+1, & \text{for } k \text{ odd,} \end{cases}$$

and hence is l-connected with $\chi(G) \geq l (> k)$, $\chi(G)$ denotes the connectivity of G. This implies that there exists no node cutset of size k.

2.2 Framework

Given the value of k, the amount of fault tolerance that we require, the state information of processor i, $i \in N$, is periodically forwarded to all the nodes in

the set $F_i \subseteq N$ such that $|F_i| = k$ and

$$F_i = \{j \in S : j = (i+p)(mod\ n), where - \lfloor k/2 \rfloor \leq p \leq \lceil k/2 \rceil, p \neq 0\}. \quad (2)$$

Following Theorems have been theoretically proved in [8] for the framework. Here we will simply state the theorems for the sake of completeness as the distributed algorithm that we propose here utilizes those properties.

Theorem 1. A. *Forwarding state information of each process to k other nodes in the network following (2) ensures k-fault tolerance.*
B. *A sufficient condition to ensure k-fault tolerance is to forward the state information by each node to at least k other nodes in the network.*

Theorem 2. *As long as $k \leq \lfloor \frac{m-1}{m}.n \rfloor$, no live node has to execute more than m processes including one of its own and an algorithm to attain the same under the proposed framework can also be found.*

Theorem 3. *Minimum number of nodes with which any node in a network with $n > 2k$ (or $n = 2k$) is required to be connected directly is $2k$ (or $2k-1$) to ensure that all the eligible nodes corresponding to a process can be updated about its state information all the time in one hop.*

Note that, $F_i \subseteq P_i$ and only the nodes in F_i are updated about the state information of process p_i by node i in the event of no failure. Lets now consider the case that a set T of nodes has failed with $|T| \leq k$. Then, for each node $i \in T$, one of the node, say $j \in F_i - T$, starts executing p_i apart from running its own process p_j. For each $i \in T$, existence of such a $j \in F_i - T$ is guaranteed by Theorem 1. Node j then starts forwarding the state information of process p_i to the set of nodes in $F_i - T - \{j\}$ (Theorem 3 assures that node j is directly connected to all the nodes in $F_i - T - \{j\}$) and also continues sending the state information of its own process p_j to the set of nodes in $F_j - T$.

Given a set of k faulty nodes, Theorem 2 can be applied to find out a fault tolerant solution which will map each faulty node to a live node eligible to take up the process of the faulty node. This only applies when all the k nodes have either failed at the same time or they have failed at different times but fault tolerance is sought only after k nodes have failed. A more realistic consideration is to be able to apply fault tolerant solution when faults occur, i.e., being able to take into account the order in which faults have happened. To be specific, when node i has failed, the node that will fail next is unknown - it could even be the node who has taken up the process of node i. Designing a fault tolerant solution considering this realistic scenario is a challenging problem and in Section 3, we have provided a distributed solution for that.

3 Automated Fault Recovery

3.1 Distributed Algorithm

In this section, we present a distributed algorithm of automated fault recovery up to a maximum of k faults, the design of which has been done following a round

based model. In the first round when the network starts up, each and every node i sends an $INFO$ message to all the nodes in the set F_i. It also receives $INFO$ messages from all the nodes j in the network such that $i \in F_j$. $INFO$ message is consisted of the tuple (j, F_j) and thus helps the recipient (node i) of the message to know about the set of nodes eligible to take up the process p_j when node j fails. Every node i stores this (j, F_j) pair on its local memory on receipt of an $INFO$ message.

Node i then evaluates $(i-j)$ mod n and this value is assigned to the variable Id_diff_j. Node i also finds out its position in the set F_j and assigns it to the variable $pref_j^i$. Nodes that are ahead of i in their position in the set F_j will be considered before i when the need of executing process p_j of j arises. But, still they may not be in a position to take over process p_j because of the fact that either they are overloaded (i.e., executing m processes) or they have failed, in which case responsibility goes to node i. Node i assigns values to the variables Id_diff_j and $pref_j^i$ for each node j from which it receives an $INFO$ message. $pref_j^i$ is assigned either of the values $\frac{k}{2} - Id\_diff_j + 1$ or $\frac{k}{2} + (n - Id\_diff_j)$ depending on whether Id_diff_j is less than equal to or greater than $\frac{n}{2}$. The rational behind this assignment is to ensure that eligible nodes in the right of j are given higher preferences than nodes in the left of j while it comes to take over p_j. If $i \in F_j$ is the rightmost node from j then $pref_j^i = 1$, meaning that node i will be given the chance first to take over the process p_j of j.

Preference Finding Algorithm

[Code for node i, $0 \leq i \leq n-1$]

$node\_status = NOT\_OVERLOADED$;
$no\_of\_processes = 1$;
$T = \phi$;
$Other\_Processes = \phi$;

forall *nodes j such that $i \in F_j$* **do**
  $fault\_flag_j = FALSE$;
end

Send $INFO$ message to all the nodes in F_i;

Receive $INFO$ messages from all the nodes j such that $i \in F_j$;

forall *such node j* **do**
  Save the tuple (j, F_j) in local memory;
  $Id\_diff_j = (i - j)$ mod n;
  if $Id\_diff_j \leq \frac{n}{2}$ **then**
    $pref_j^i = \frac{k}{2} - Id\_diff_j + 1$;
  else
    $pref_j^i = \frac{k}{2} + (n - Id\_diff_j)$;
  end
end

Algorithm 3.1: Preference Finding Algorithm for Node i

Starting from the second round in each successive rounds, every live node i sends $STATUS$ message for every process p_j (including its own process p_i) that is running on it to all the live nodes in the set F_j. $STATUS$ message for a process p_j is consisted of the tuple (p_j, S_{p_j}), where S_{p_j} denotes the state information of the corresponding process, which is required for any node in the set F_j to initiate the process. If in a round, node i does not receive the $STATUS$ message from a live node j in the network such that $i \in F_j$, then it assumes that node j has failed. If a node fails, then for each process running on it there can be at most k nodes to take up the load of this process. To decide which node will take up the load of which process being executed by this failed node, we adopt the following scheme.

When node i detects that node j such that $i \in F_j$ has failed, then it first checks whether it is already overloaded (i.e., executing m processes) or not. if it is not already overloaded, then node i waits up to $pref_j^i$ rounds to start the process p_j of the faulty node j by using the state information obtained from the last $STATUS$ message from j. After initiating process p_j of j, node i sends $RESOLVED_j$ message to all other nodes in the set F_j. But, if node i receives the $RESOLVED_j$ message within $pref_j^i$ rounds, then that means some other node in F_j has already taken up the process p_j and in which case node i does not require to start p_j. In case node i is already overloaded, it then expects to receive $RESOLVED_j$ message within $k + pref_j^i$ rounds. But if it does not receive the $RESOLVED_j$ message, then in the $k + pref_j^i$-th round it constructs a set A consisting of those processes $p_l, (l \neq i)$ such that $|F_j - T| \leq |F_l - T|$. If A is not an empty set (According to Theorem 1, the set A can never be empty, unless the number of faulty nodes is more than k), then node i finds out a process $p_l \in A$ for which $|F_l - T|$ is maximum and then stops this process and starts executing p_j. The process p_k that gets stopped by node i will be then taken up by some other node in F_k and this is possible by following the same algorithm since nodes in F_k without having received the $STATUS$ message corresponding to p_k detects that process k has failed.

We have two implicit requirements in order for this distributed algorithm to work. They are as follows. All the nodes in the network are required to be time synchronized; and the time required for a message to reach from one node to another should be less than the span of each round. We will now illustrate our algorithm through an example. Lets consider that network is consisted of 10 nodes and the value of k is equal to 4 and as such it is only required on behalf of each node to execute at most 2 processes (i.e., $m = 2$). Following the framework described in Section 2, the underlying graph modeling the network will be a regular graph with degree of each node equals to 8 and also $H_{4,10}$ comes as a subgraph of this network. Preferences of each node i, can then be found by following the *Preference Finding Algorithm* and they are as shown in Table 1. If we now consider that nodes $9, 2, 8$, and 0 will fail in this order, then Table 2 illustrates how the distributed *Fault Recovery Algorithm* works.

Fault Recovery Algorithm

[Code for node i, $0 \leq i \leq n-1$]

forall *process* p_k *executing in node* i **do**
  Send $STATUS$ message to all the nodes in F_k;
end

Receive $STATUS$ message from all the processes p_j such that $i \in F_j$;

if $STATUS$ *message is not received from* p_j *such that* $i \in F_j$ **then**
  $fault\_flag_j = TRUE$;
  $round_j = 0$;
  $T = T \cup \{j\}$;
end

forall *all the processes* p_j *such that* $i \in F_j$ **do**
  if $node\_status == NOT\_OVERLOADED$ **then**
    if $fault\_flag_j$ **then**
      $round_j = round_j + 1$;
      if $RESOLVED_j$ *is not yet received* **then**
        if $round_j == pref_j^i$ **then**
          Start process p_j;
          Send $RESOLVED_j$ message to all the other nodes in F_j;
          $Other\_Processes = Other\_Processes \cup \{p_j\}$;
          $fault\_flag_j = FALSE$;
          $no\_of\_processes = no\_of\_processes + 1$;
          if $no\_of\_processes == m$ **then**
            $node\_status = OVERLOADED$;
          end
        end
      else
        $fault\_flag_j = FALSE$;
      end
    end
  else
    if $fault\_flag_j$ **then**
      $round_j = round_j + 1$;
      if $RESOLVED_j$ *is not yet received* **then**
        if $round_j == k + pref_j^i$ **then**
          Find $A = \{p_l \in Other\_Processes : |F_j - T| \leq |F_l - T|\}$;
          if $A \neq \phi$ **then**
            Stop process $p_l \in A$ for which $|F_l - T|$ is maximum;
            Start process p_j;
            Send $RESOLVED_j$ message to all the other nodes in F_j;
            $Other\_Processes = (Other\_Processes - \{p_l\}) \cup \{p_j\}$;
            $fault\_flag_j = FALSE$;
          end
        end
      else
        $fault\_flag_j = FALSE$;
      end
    end
  end
end

Algorithm 3.2: Fault Recovery Algorithm for Node i

i	$pref^i_{i-2}$	$pref^i_{i-1}$	$pref^i_{i+1}$	$pref^i_{i+1}$
0	$pref^0_8 = 1$	$pref^0_9 = 2$	$pref^0_1 = 3$	$pref^0_2 = 4$
1	$pref^1_9 = 1$	$pref^1_0 = 2$	$pref^1_2 = 3$	$pref^1_3 = 4$
2	$pref^2_0 = 1$	$pref^2_1 = 2$	$pref^2_3 = 3$	$pref^2_4 = 4$
3	$pref^3_1 = 1$	$pref^3_2 = 2$	$pref^3_4 = 3$	$pref^3_5 = 4$
4	$pref^4_2 = 1$	$pref^4_3 = 2$	$pref^4_5 = 3$	$pref^4_6 = 4$
5	$pref^5_3 = 1$	$pref^5_4 = 2$	$pref^5_6 = 3$	$pref^5_7 = 4$
6	$pref^6_4 = 1$	$pref^6_5 = 2$	$pref^6_7 = 3$	$pref^6_8 = 4$
7	$pref^7_5 = 1$	$pref^7_6 = 2$	$pref^7_8 = 3$	$pref^7_9 = 4$
8	$pref^8_6 = 1$	$pref^8_7 = 2$	$pref^8_9 = 3$	$pref^8_0 = 4$
9	$pref^9_7 = 1$	$pref^9_8 = 2$	$pref^9_0 = 3$	$pref^9_1 = 4$

Table 1. Preference Table

Failed Node Id	*Assigned To*	*No. of Rounds*	*Explanation*
9	1	1	$pref^1_9 = 1$
2	4	1	$pref^4_2 = 1$
8	0	1	$pref^0_8 = 1$
0	1	6	$pref^1_0 = 2$ (Stops p_9)
8	7	3	$pref^7_8 = 3$
9	7	8	$pref^7_9 = 4$ (Stops p_8)
8	6	4	$pref^6_8 = 4$

Table 2. Illustration of the Algorithm

3.2 Analysis of the Algorithm

Theorem 4. *At most $2k$ rounds are required to resolve a single fault.*

Proof. For any process p_i corresponding to the node i, we have $|F_i| = k$. Now, when it is detected that node i has failed, the *Fault Recovery Algorithm* in each round examines a particular node in the set F_i to check whether it is in the $NOT\_OVERLOADED$ condition. That requires k rounds. Even in these k rounds it may not be possible for any of the nodes in F_i to start process p_i because some of them may have failed and some of them are overloaded. But, according to Theorem 1, it is not possible that all the nodes in F_i are faulty and as such there must be one live node eligible to take up the process p_i. If it happens to be the case that the only live node eligible to take up process p_i is having the highest $pref$ value, then k more rounds are still required for this node to start process p_i by forcefully stopping another process that it was executing previously. This is the worst case that we can think of fault to get resolved as is also evident from the example that we have explained earlier. Hence, at most $2k$ rounds are required to resolve a single fault. □

Theorem 5. *To resolve a single fault, the maximum number of $RESOLVED$ messages that is required to be sent across the network is $(k-2)m+1$, where m is the maximum number of processes that a node is capable of executing.*

Proof. Let us first consider that node i is executing m processes, viz, p_i (of its own), and $m-1$ other processes $p_{j_1}, p_{j_2}, \ldots, p_{j_{m-1}}$. This essentially means that nodes $j_1, j_2, \ldots, j_{m-1}$ have already failed. Then all these m processes are taken up by appropriate eligible processors. For each of these $m-1$ other processes there can be at most $k-2$ live nodes and it is required to send the $RESOLVED$ message to each of these live nodes. For p_i, the number of live nodes can be at most $k-1$ and it is also required to send the $RESOLVED$ message of p_i to each of them. Hence, the upper bound of the number of $RESOLVED$ messages that are required to be sent across the network is $(k-2)(m-1)+k-1 = (k-2)m+1$. □

4 Formal Verification of the Distributed Algorithm

To prove the correctness of the proposed algorithm, we use model checker SPIN [9] to formally verify the desired properties of the algorithm. SPIN is a tool for model checking distributed systems automatically and verifies properties of distributed algorithms modeled in the Promela language, by exploring their state space. Promela is a non-deterministic guarded-command language for modeling systems of concurrent processes that can interact via shared variables and message channels. Given a concurrent system modeled by a Promela program, SPIN can check for violations of user specified assertions, and temporal properties expressed by LTL formulas. When a violation of a property is detected, SPIN reports a scenario i.e. an execution sequence where the property is not valid.

We have modeled the processor as a parameterized process *processor*. The number of nodes, maximum fault-tolerance, maximum number of loads that can be taken by a processor are defined at the start of the model, so that they can be changed very easily to create different instances. The nodes have been defined as a structure which is as given below.

```
typedef Node {
        byte id;
        byte status;    /* 0 - Faulty, 1- Not Overloaded, 2 - Overloaded */
        byte noofprocesses;
        Neighbour neighbours[L];    /* L = 2k */
        Process processes[P];       /* P = m + 1 */
}
```

The neighbors and the processes follow the following data types

```
typedef Neighbour {
    byte id;
    byte status;        /* 0 - Faulty, 1- Not Overloaded, 2 - Overloaded */
}
typedef Process {
    byte procid;
    byte nodeid;
    byte faultflag;     /* 0 - Not Faulty,  1- Faulty,  2 - Unknown */
    byte round;
}
```

The *neighbors* of a node are those nodes to whom the $STATUS$ message of the node is required to be sent. The *processes* for a node represents the processes that the node may have to take at any point of time in future. The *procid* of a process is the ID of the node where the process was running for the first time. The *nodeid* field of a process indicates on which node the process is currently running. The *faultflag* variable is used to keep track of the status of the node where the process was running last. When a neighbor of a node detects that the node has become faulty it makes the flag corresponding to that process 1. Whenever it takes up that process or receives a $RESOLVED$ message corresponding to that process, it makes the faultflag 0. This is done to check the liveness property that will be discussed later. The variable *round* indicates that how many rounds have been passed after the node detected that the node where the process was running is faulty.

To introduce faults in any order, a separate process *endround* has been introduced. After the end of a round, zero, one or more number of processors are randomly made faulty by this process until the total number of faulty nodes are less than equal to k.

In order to prove the correctness, we have considered for verification safety property, liveness property, and timeliness property. Liveness property has been expressed as a LTL formula, and other properties are inserted as simple assertions at proper places in the Promela model. The properties are described below.

Safety 1 Whenever a node becomes faulty, at least one of its neighboring nodes is non-faulty. This property has been checked as an assertion in the *endround* process after making a node faulty.
assert(node[node[j].neighbours[0].id].status != 0 ||
node[node[j].neighbours[1].id].status != 0);

Safety 2 No node has to take more than M processes at any point of time.
In the process corresponding to a processor, when a node take up a new process we increase its *noofprocesses* field by 1 and this is followed by the following assertion
$assert(node[nodeid].noofprocesses <= M)$,
where the meaning of M has been discussed earlier.

Liveness Whenever a node becomes faulty, its process is eventually taken up by some other live nodes.
The property has been expressed by the following LTL formula
$[](s- >< > t)$,
$s \cong$ node[0].processes[1].faultflag==1, $t \cong$ node[0].processes[1].faultflag==0

Timeliness Every fault is recovered in no more than 2K rounds.
When a processor detects that the node where one of the processes that it is supposed to take was running is faulty, it starts counting the round. When this round is equal to its preference value for that process, then if that process has not yet been taken up by some other process, it takes up that process. So, in any case, the value of the round should not be more than 2K according to Theorem 3. We check this by the following assertion
$assert(node[nodeid].processes[j].round <= 2 * K)$

We have been able to verify our model for N=8, K=3 and M=2 and all lower instances. Due to the state-space explosion problem inherent in model checker SPIN, we could not verity our algorithm for more than 8 processors.

5 Conclusion

In this paper we have presented a distributed algorithm of automated fault recovery for stateful failover in a network. In whatever way the fault may arise the algorithm can handle that fault and in at most $2k$ rounds the processes of the faulty processor are taken up by a(some) eligible live node(nodes) in the network. The message complexity of our algorithm is linear with the number of nodes. The correctness of the algorithm has been proved by modeling the algorithm in SPIN and verifying its desired properties.

References

1. F. Harary, "The Maximum Connectivity of a Graph", *Proc. Nat. Acad. Sci.*, U.S.A., 48, pages 1142-1146, 1962.
2. J. G. Kuhl and S. M. Reddy, "Distributed Fault Tolerance for Large Multiprocessor Systems", *Computer Architecture News*, 8, page 23-30, *7th Intl. Symposium on Computer Architectures*, 1980.
3. C.L. Yang and G. M. Massona, "Distributed Algorithm for Fault Diagnosis in Systems with Soft Failures", *IEEE Transaction on Computers*, Vol. 37, No. 11, November 1988.
4. M. A. Sridhar, C. S. Raghavendra, "Fault-Tolerant Network Based on De Bruijn Graph", *IEEE Transaction On Computers*, Vol. 40, No. 10, October 1991.
5. K.Mukhopadhyay, B.P. Sinha, "Hamiltonian Graphs with Minimum Number of Edges For Fault-Tolerant Topologies", *Information Processing Letters*, 44, pages 95-99, November, 1992.
6. T. Sung, T. HO, C. Chang AND L. Hsu, "Optimal k-Fault-Tolerant Networks for Token Rings", *Journal of Information Science and Engineering*, 16, pages 381-390, 2000.
7. C. Hung, L. Hsu, T. Sung, "On the construction of combined k-fault-tolerant Hamiltonian graphs", *Networks*, 37(3), pages 165-170, 2001.
8. I. Saha, D. Mukhopadhyay, S. Banerjee, "Designing Reliable Architecture for Stateful Fault Tolerance, In Proceedings of *Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06)*, pages 545-551, 2006.
9. G. J. Holtzman. The SPIN Model Checker, Primer and Reference Manual, Addison-Wesley, 2003.

6 Appendix

6.1 Harary Graph

An undirected graph G can be represented as (N, E), where N denotes a set of nodes and E denotes a set of edges which can be defined as an unordered pair of distinct nodes. A *node cut* of G is a subset N' of N such that $G - N'$ is disconnected, where $G - N'$ is obtained by removing all the nodes in N' and also by removing all the edges incident on the nodes in N'. A *k-node cut* is a node cut of cardinality k. The minimum cardinality of a node cut is called *connectivity* $\chi(G)$ of G and the graph G is called *k-connected* if $\chi(G) \geq k$, i.e., there exists no node cut of size $k-1$. Its a known result that the least number of edges that a k-connected graph on $n(n > k)$ vertices can have

is greater or equal to $\frac{nk}{2}$. Harary graph [1] $H_{k,n}$, which we will be describing next, is a k-connected graph on n vertices with exactly $\frac{nk}{2}$ edges.

Structure of a Harary graph $H_{k,n}$ is defined for three different cases.

Case 1: k even. Then $H_{k,n} = (N, E)$ is defined as follows. The nodes in N are labeled as $0, 1, 2, \ldots, n-1$ and

$$E = \cup_{i=0}^{n-1}\{(i,j) : j = (i+p)(mod\ n), where - k/2 \leq p \leq k/2 \text{ and } p \neq 0\}. \quad (3)$$

(i, j) and (j, i) both denotes the same edge, since we are talking about undirected graphs.

Case 2: k odd, n even. $H_{k,n}$ is then constructed by first drawing $H_{k-1,n}$ following case 1 and then by adding edges joining node i to node $(i + \frac{n}{2})(mod\ n)$ for $1 \leq i \leq \frac{n}{2}$.

Case 3: k odd, n odd. $H_{k,n}$ is then constructed by first drawing $H_{k-1,n}$ and then joining node 0 to nodes $\frac{n-1}{2}$ and $\frac{n+1}{2}$ and node i to node $(i + \frac{n+1}{2})(mod\ n)$ for $1 \leq i \leq \frac{n-1}{2}$.

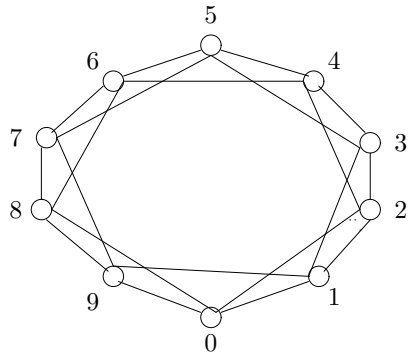

Fig. 1. Harary Graph $H_{4,10}$

The fact that the graph $H_{k,n}$ is k-connected with minimum number of edges is known from Harary, 1962.