‘Simulating Proactive Fault Detection in Distributed Systems.’

Proposal Draft for Capstone Project

Chair – Dr. Hans-Peter Bischof
Reader – Dr. Rajendra Raj

Namita Naik
nvn2271@cs.rit.edu

Date – 9/04/07

Table of Contents

Abstract

Fault Tolerance is an important issue considered when developing a reliable Distributed System. Reactive fault recovery is now provided in most of commercial, research and academically used computerized systems. These systems are designed to redistribute the current process on to other machines when failure occurs. In contrast to the conventional method of reactive recovery, an emerging concept in the field of fault tolerance is a proactive approach. This approach exploits pre fault symptoms and initiates fault recovery henceforth [1].

This project is to implement proactive fault prediction simulator for a distributed system, thus providing a base for further developing a real-time proactive fault tolerant distributed system. This will include developing a language for simulation, which allows the user to define a distributed system. The language is further used to develop an environment that utilizes two fault prediction algorithms, *DFT* (*Dispersion Frame Technique*) [1] used for MEAD (CMU) [1] and Wilcoxon's Rank-Sum described in detail in [8]. The project also includes providing a detailed comparison metrics for these prediction algorithms in terms of prediction precision and prediction accuracy.

1 – Introduction

With the increasing use of computer systems in everyday life, society is increasingly more dependent on computer systems. In recent years, the use of computers has covered varied fields including business and high-end technical applications such as air traffic control systems, automated engineering solutions, etc. This calls for a need to have highly reliable systems. *Fault tolerance* can be defined as "the ability of the system to perform in its predefined manner even in presence of internal faults" [2], thus making the system more reliable.

1.1 Classification of faults

Faults can be defined as unexpected behavior of the system that may or may not lead to an abnormal termination. Faults, being unpredictable in nature, make it difficult to identify them. Faults can be classified into various classes based on the duration of their prevalence, cause of their occurrence and behavior of the failed device [2].

1.1.1 Based on the duration of their prevalence

In this category faults can be classified mainly as transient and permanent. Transient faults are the temporary faults that may occur during the execution of the system, but will eventually disappear without any external intervention [2]. On the other hand permanent faults need external intervention [2]. Transient fault

has a variation know as intermittent fault that recurs often unpredictably being problematic to the normal execution of the system [2]. Studies show that though permanent faults may seem to be more severe, they are much easier to diagnose and repair [2].

1.1.2 Based on the cause of their occurrence

In this category faults can be classified into two classes design faults and operational faults [2]. Design faults are caused because of faults in the underlying design of the system. A faulty design of the system will lead to a faulty implementation, causing design faults during the running life of the system. Operational faults are faults occurring during the lifetime of the system due to failure of various physical parts of the system like processor failure or disk crash.

1.1.3 Based on the behavior of the failed component

In this category faults are divided into four different classes [2]. If the component completely stops working or will never return to a valid state, it is called a Crash fault [2]. Omission fault causes the component to fail completely during normal execution of the system [2]. If the component fails to complete its service on time, it is called a Time fault [2]. All faults of arbitrary nature are called Byzantine faults [2].

1.2 Fault tolerance mechanisms

There are mainly two types of fault tolerant mechanisms:
[1]- Reactive Fault Tolerance
[2]- Proactive/ Predictive Fault Tolerance

1.2.1 Reactive Fault Tolerance

There is widespread use of reactive fault tolerant mechanisms for fault detections. These systems detect fault after its occurrence in the system and starts the recovery henceforth. Such systems though provide good fault tolerant solution for general computing environment, cannot fulfill the time constrains set by real time computing systems. Real time computing systems like hospital- patient monitoring system, air traffic regulation system, etc. have critical running time constrains, which need to be fulfilled for its successful operation.

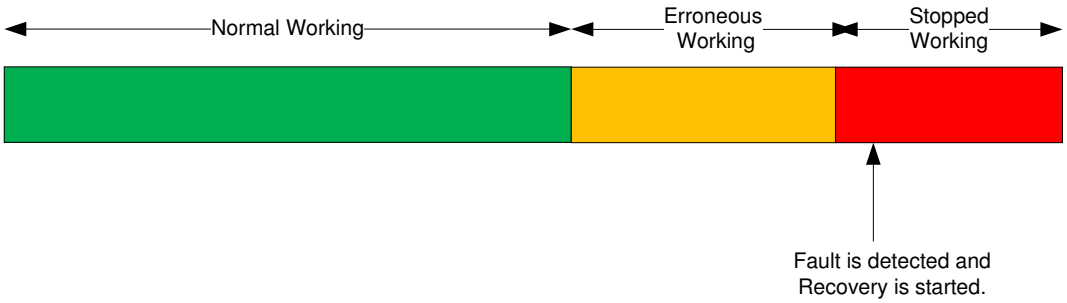

Fig. 1- Timeline for a Reactive fault detection system

Figure 1 shows the time line for working of a reactive fault tolerant system. Reactive fault tolerant system keeps a continuous check on the performance of various devices in the system. They poll the device for its failure status. If a failure is detected during the run of the system, it then informs the fault monitoring system and does the related recovery process.

In the working of a real time system, time constrains is an important factor to be considered. Reactive systems start a recovery procedure after detecting a failure. The time interval of starting a recovery after failure detection can be considerable; failing to meet the time constrains set by a real time system. Hence, an emerging concept in the field of fault detection is – 'predictive fault tolerance' mechanism.

1.2.2 Predictive fault tolerance

Predictive fault tolerant systems predict a fault before its occurrence, during the execution of the system, and hence are able to fulfill the time constrains set by real time systems.

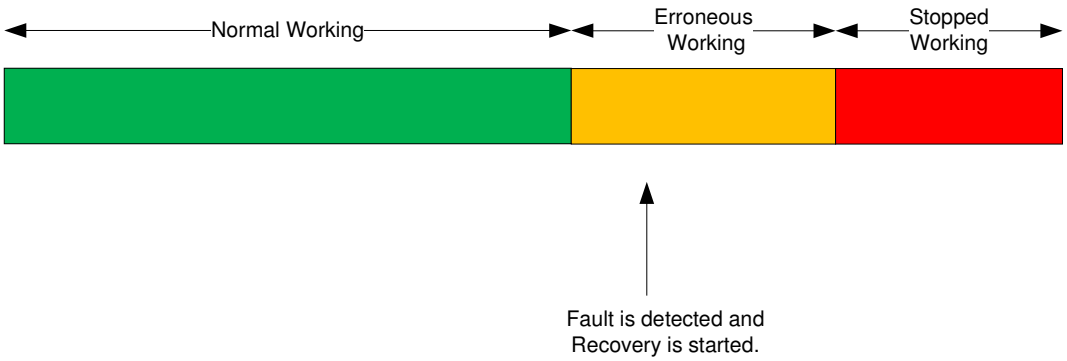

Fig. 2- Timeline for a Proactive fault tolerant system

Predictive systems assume that faults, which are inherently unpredictable, do show some disruptive effect on the performance of the system [3]. Research studies show that faults do show a pattern of occurrence. Predictive fault tolerant systems exploit these patterns to predict a fault.

Figure 2 shows the time line for the working of a proactive fault prediction system. As seen in fig.2, a fault prediction can be done before the actual failure of the device giving enough time to start the recovery and process migration before the complete failure of the system. Reactive systems on the other hand cannot detect a failure before its occurrence.

This project gives a detailed study and analytical comparison for two fault-predicting algorithms.

The DFT (Dispersion Frame Technique) developed at CMU for their MEAD architecture [1] predicts faults based on the error occurrence pattern in a given time frame. It tries to exploit the basic nature of 'decreasing time frames' for occurrence of errors. DFT is explained in detail in section-2.

The Wilcoxon's Rank Sum method, provided as a variation for SMART in [8], is basically tested for hard disk fault prediction. It predicts faults by comparing the behavior of the running device to a reference dataset. It assumes that faults do not have any pattern of occurrence and hence compares it with a reference set, generated from multiple device of the same type. Rank-Sum is explained in detail in Section-3.

2 - Dispersion Frame Technique

Dispersion Frame Technique is a prediction algorithm developed at CMU for their MEAD system. “DFT is implemented in a distributed online monitoring and predictive diagnostic system for their existing campus wide Andrew file system” [1]. Error logs for this predictive system were collected from 13 file servers over a period of 22 months [1]. [1] shows the descriptive study of the error log data analysis using joint probability statistics, statistical Weibull fit and DFT. The comparative results show that DFT had the highest success rate with least false alarms [1].

DFT uses intermittent and transient faults occurring during the run of the system to predict individual device failures. It determines the relationship between the error occurrences by examining the inter arrival time and the domain of their occurrence [1]. This technique uses *Dispersion Frames* (DF) defined as the inter arrival time between successive error events of the same type and *Error Dispersion Index* (EDI) defined as the number of error occurrences in half of the Dispersion Frame [1]. Knowledge gained in separating error logs into their constituent error sources and experiences of their hardware technicians were used for generating heuristics for DFT [1]. These heuristics combined with DF and EDI are used to predict failures for a system.

2.1 DFT Algorithm

DFT predicts faults by finding relationships between occurrences of fault events in terms of their closeness in time and the source of their occurrence. This algorithm is basically tested for hard drive fault predictions. Based on the statistical analysis done [1], a dispersion frame of 168 hours was used to activate the DFT algorithm. The DFT works as follows [1]:

Step 1: A time line of five most recent errors is taken into account for each device. DFT is activated when a frame size of less than 168 hours is encountered.

Step 2: Each error occurrence on the time line is centered on by the previous two time frames. i.e. Frame for event (i-3), the inter arrival time between events (i-4) and (i-3), is centered around error events (i-3) and (i-2).

Step 3: EDI is calculated as the number of errors from the center of each frame to its right most end.

Step 4: A failure warning is issued under the following conditions:

(a)- **3.3 rule**: If successive application of the same frame results in EDI of at least 3. In other words, it represents a scenario of steeply reducing dispersion frames based on single previous frame.

(b)- **2.2 rule**: If application of successive frames exhibits an EDI of at least 2. This as compared to '3.3 rule' differs as it exhibits uniform decreasing time frames based on two previous frames.

(c)- **2 in 1 rule**: A failure warning is generated if a dispersion frame is less than 1 hour in length.

(d)- **4 in 1 rule**: If four errors occur within a 24 hour.

(e)- **4 decreasing rule**: If four successive frames are decreasing in their inter arrival time and at least one frame is half the size of its previous frame.

Step5: Multiple iterations of step 2,3 and 4 are usually executed before issuing a failure warning. [1] describes in detail the working of DFT and the statistics about the heuristics used.

3 – Wilcoxon's Rank-Sum Algorithm

Failure rarity problem is a well-known issue in the field of failure prediction algorithms. The annual hard disk failure rate is approximately 1%. Because of this, about thousand of disk drives need to be tested for more than an year to get data for about 15 failed drives [8]. This increases the false alarm rate, as less knowledge is available about the behavior of fault disks. The SMART (Self Monitoring and Reporting Technology) uses around some 30 different internal drive measurements. It uses error threshold mechanism to predict fault. i.e. It uses a pre-declared value limit for the individual attributes and checks the reading for each attribute against it. It then does a 'OR' operation on the result of each attribute. This can cause a high false alarm rate [8].

[8] proposes the Wilcoxon's Rank-Sum algorithm to be used, replacing threshold tests for failure prediction. Rank Sum tests are recommended for rare failure events such as disk failures, rare disease, etc. where false alarm rates are high. They are particularly useful in cases where statistical distribution of the data is unknown and is suspected to be non-gaussian [8].

3.1 Rank-Sum Working

The Rank-Sum algorithm, as the name indicates, sorts a group of reference data points and warning data points and ranks them according to their sorted position. It then calculates the standard deviation of the set. Any warning point found to deviate higher than an acceptable value is detected as a fault behavior. This is done for each attribute in question and their results are then 'OR'ed.

Step 1: The Warning dataset for each drive is taken to be its last five sample reads for each attribute. The Reference dataset for each attribute is taken to be 50 random samples averaged over multiple good drives. The will-fail drives are not included in the reference set values. The optimum dataset size will vary depending on attribute read time intervals and certain other factors [8].

Step 2: Sort these combined set of values and rank them accordingly. An average rank is given to the items that have the same value. i.e. If three values in the set are 0 and they have the rank{1,2,3}, all three of them will be given the rank '2'.

Step 3: Statistics are calculated for this dataset. Statistics include mean, variance and standard deviation for the set. If the warning rank sum is exceeding the pre-calculated acceptable limit, a fault warning in the behavior of the drive is declared. This is done for each attribute and all individual results are then 'OR'ed to give a final result.

Rank-Sum compares the test data with the reference data for predicting faults, thus does not assume a particular behavior pattern for the drive.

4 - Limitations of a predictive system

In present scenario, a predictive fault tolerant system cannot replace a reactive system. This is because of the fact that not every fault has a pattern of occurrence or show pre-occurrence symptoms, making them impossible to predict. Also failure is a very rare event. Enough statistics are not available to know the behavior or pattern followed by bad working devices. i.e. More information about good working disk is available this makes prediction accuracy difficult to achieve. To nullify this effect, we may try to keep wider fault occurrence ranges, leading to high false alarm rate and making prediction much complicated process.

Predictive systems can be a considerable overhead on the performance of the system as they need the prediction mechanism to be activated every time the heuristics match, which may quite often reduce the system performance.

5 - Proposed Project

The project is to develop a simulation for proactive prediction in distributed systems. Initial phase of the project consisted of developing a language for simulation and writing a parser to generate the simulation environment. The simulation environment was then enhanced to support the prediction mechanism. This phase includes implementing the DFT algorithm and Rank-Sum algorithm in simulation and modifying the simulation environment to support this prediction technique. The project also consists of finding a valid dataset and modifying it for testing and collecting metrics.

5.1 Architecture Diagram

Fig.1 gives the architecture diagram for this project. It gives an overview of the working of the simulation system.

Fig.1 shows the parser reading the given input file containing information about the configuration of the distributed system. This file contains the simulation environment description in the language defined below. The parser generates a simulation environment setup along with a GUI interface for the user. Each lower level running component is assigned a behavior file. This file contains the working readings for that particular component.

A fault prediction algorithm is used to predict an upcoming fault or failure in the system. Any warning or error generated in the system is migrated onto the enclosing component. Further detailed explanation about the working of the prediction mechanism is given in the implementation diagram.

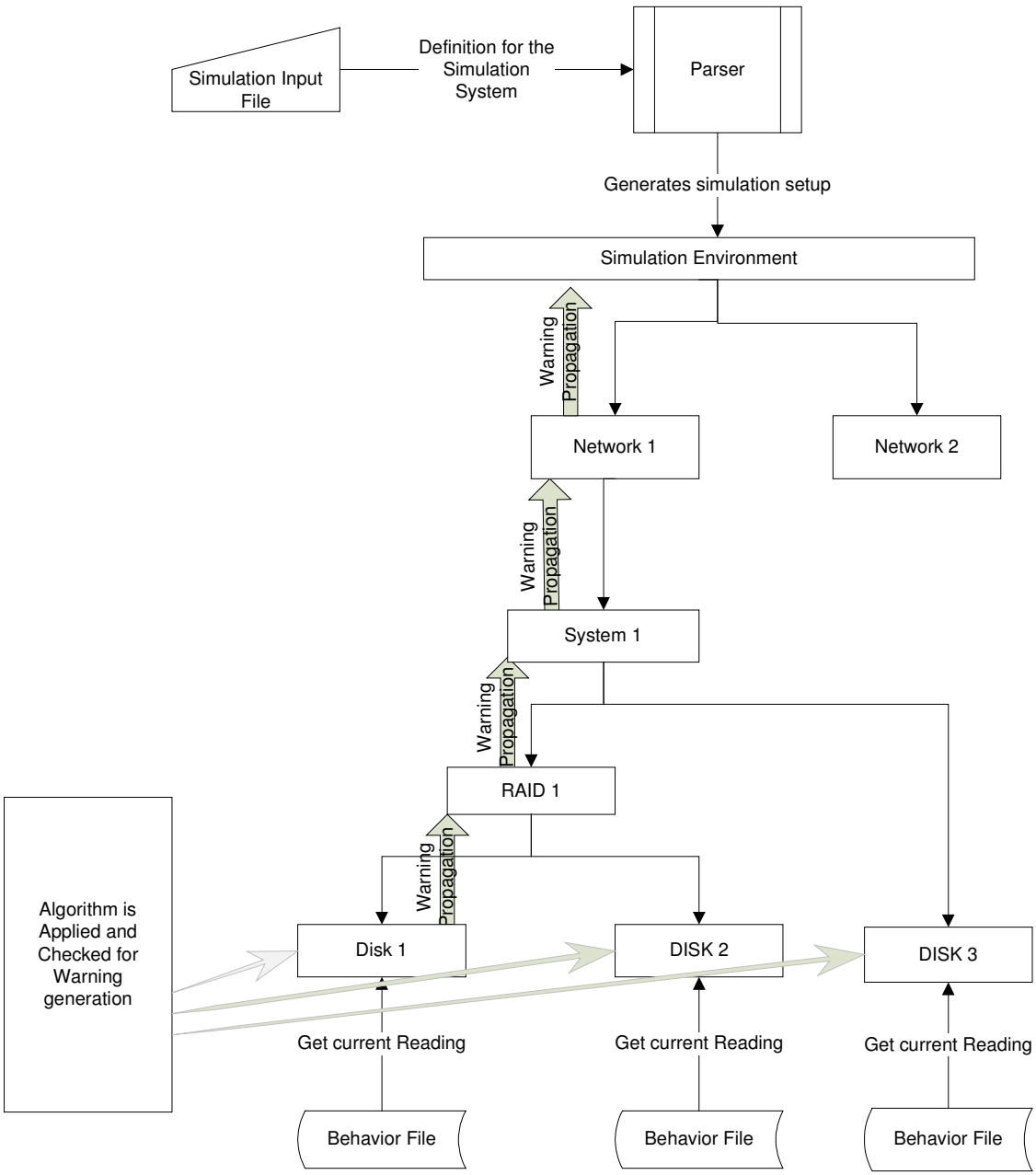

Fig.2- Architecture Diagram

A fault prediction algorithm is used to predict an upcoming fault or failure in the system. Any warning or error generated in the system is migrated onto the enclosing component. Further detailed explanation about the working of the prediction mechanism is given in the implementation diagram.

5.2 Implementation diagram

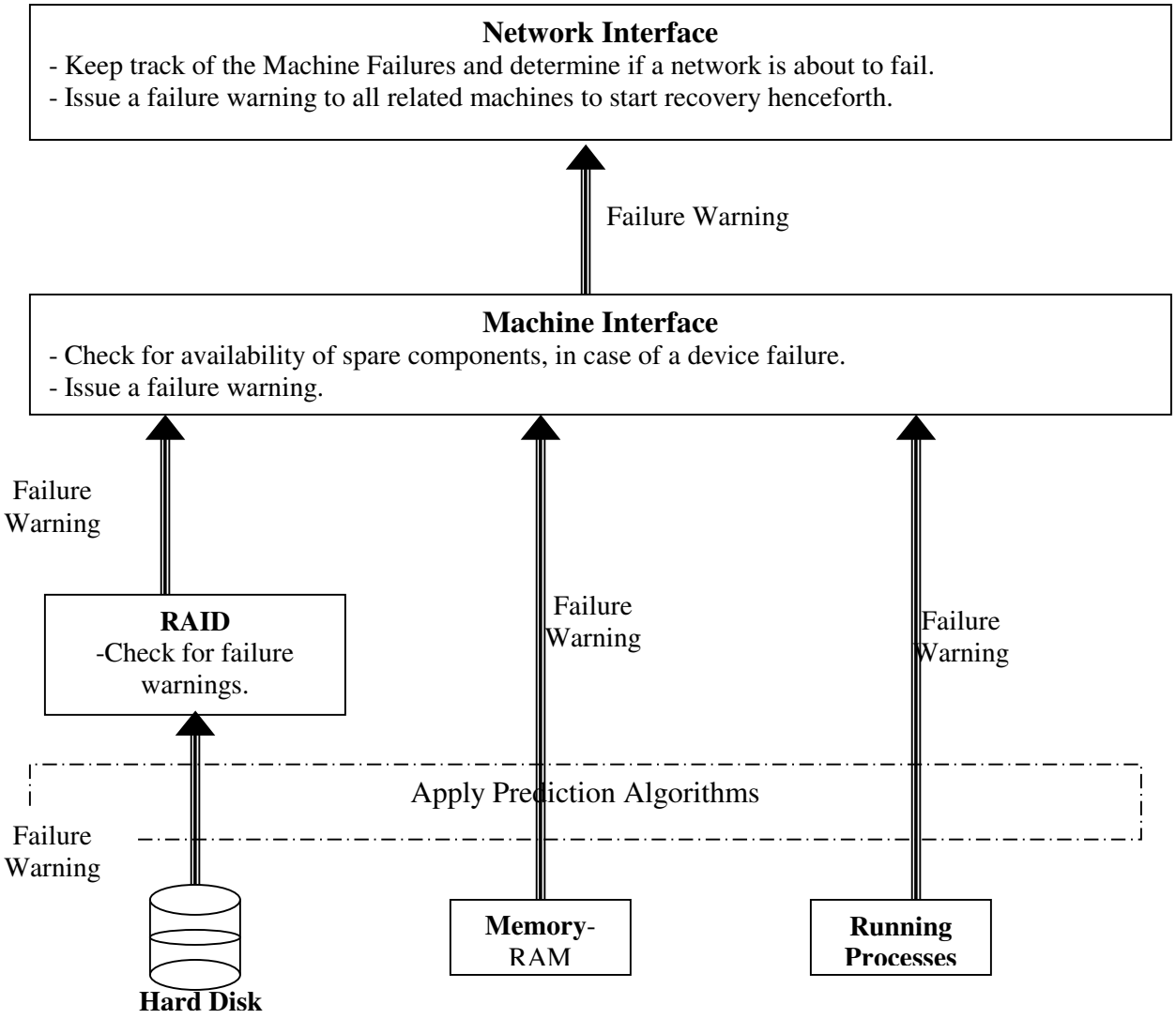

Fig. 2- Simulation Implementation Diagram

The given diagram shows the higher-level implementation for this project. The simulation works in the following steps:

1- Each individual component, i.e. disk, memory and process, read their respective behavior from a predefined file.
2- A failure prediction algorithm is invoked if necessary pre-failure conditions appear. Each individual component will check for its failure status continuously. If a pre failure condition is generated, each device will inform its parent/enclosing device/machine. E.g. If a RAID exist in the system, the individual disk failures will be reported to the RAID controller. Similarly, if any of the RAID, disk, memory or process encounters any pre-failure conditions,

they will report this to the machine controller.

3- The parent or the receiver device, usually a computer machine, then activates its failure prediction mechanism, which includes checking for possible failure on the basis of existing resources. Machine then issues a failure warning if its resources are about to fail.
4- Failure warning from the machine goes to the network controller, which in implementation is a general simulation controller. This controller has the knowledge about the devices in the network and their relations with each other. It then reports the devices that will be affected by this failure, thus warning the related devices about a predicted failure in their network and forcing them to save the current work and start the recovery henceforth.

The following section describes the language developed for defining the system in simulation. Section 6.3 gives an insight on the implementation of DFT in simulation. Section 6.4 describes the possible modification for the existing algorithm to allow precise prediction.

5.3 Language for Simulation

This project consisted of defining a simulation language which allows creation of a distributed environment for simulation purpose. Below defined language allows you to define an environment consisting of single or multiple networks. Each network can then be defined to hold single or multiple machines. Every machine can have one or more memory, disk or raid array. Also every machine can have single or multiple processes running on them. A RAID can be configured for the minimum number of disk failures to call it a RAID failure. Each of the lower end components (disk, memory and process) can be configured to use a file to read its running behavior. This language also facilitates to define the dependencies between systems. Given below is the grammar for the proposed language.

Simulation ::= (*Definition* | *RelationDef*)+ <EOL>

Definition ::= **DEFINE** (*EnvironmentDef* | *NetworkDef* | *SystemDef* | *MemDef* | *DiskDef* | *ProcDef* | *RaidDef*) **END;** <EOL>

EnvironmentDef ::= **ENVIRONMENT** *Id* **:** <EOL> *NetDeclare*+
NetworkDef ::= **NETWORK** *Id* **:** <EOL> (*SysDeclare* | *NetDeclare*)+
SystemDef ::= **SYSTEM** *Id* **:** <EOL> (*ProcDeclare* | *DiskDeclare* | *MemDeclare* | *RaidDeclare*)+
MemDef ::= **MEM** *Id* **;** <EOL> **RUNNING** *FileName* **;** <EOL>
DiskDef ::= **DISK** *Id* **;** <EOL> **RUNNING** *FileName* **;** <EOL>
ProcDef ::= **PROC** *Id* **;** <EOL> **RUNNING** *FileName* **;** <EOL>
RaidDef ::= **RAID** *Id* **;** <EOL> *DiskDeclare*+
RelationDef ::= **RELATION** *Id* **DEPENDS_ON** *Id* **;** <EOL>

NetDeclare ::= **NETWORK** *Id* **;** <EOL>
SysDeclare ::= **SYSTEM** *Id* **;** <EOL>
DiskDeclare ::= **DISK** *Id* **;** <EOL>
ProcDeclare ::= **PROC** *Id* **;** <EOL>
DiskDeclare ::= **DISK** *Id* **;** <EOL>
RaidDeclare ::= **RAID** *Id Number* **;** <EOL>

Id ::= [a-zA-Z][a-zA-Z_0-9]*
Number ::= [0-9]+
FileName ::= *Id* (**.** *Id*)?

<EOL> - defined as the newline (`\n`) character in a UNIX environment or carriage return (`\r\n`) in Windows.

Semantic Rules for the language:

1. Each Element needs to be declared before defining it. Here, each declaration is equivalent to creating an object and each definition is equivalent to configuring the component for its parameters
2. The Container needs to be declared and defined before the contained element. E.g. 'Environment' needs to be declared and defined before the 'Networks'.
3. The first component to be declared has to be an 'Environment'. Two environments can co-exist. But they cannot be related.
4. Each Component can declare multiple components in its definition. E.g. 'Environment' can contain multiple 'Networks' and a 'Network' can contain multiple 'Systems'.
5. Each declared component need not be defined.
6. Declaration of a 'Raid' takes the number of disks whose failure can be tolerated by the system as '*DiskDeclare*' parameter.
7. 'Relation' defines a dependency relationship between two components. Both the components involved in a 'Relation' need to be defined before defining a dependency between them.
8. *Id* can be any alphanumeric starting with an alphabet.
9. *FileName* can be any text file with a single '.' extension.

Following is an example that will clarify the grammar usage:

```
DEFINE ENVIRONMENT env1
NETWORK N1;
END;
```

Here an environment 'env1' is defined. It contains a declaration for a Network 'N1'. User can declare one or more networks in an environment.

```
DEFINE NETWORK N1:
SYSTEM SYS1;
SYSTEM SYS2;
END;
```

Similarly, the definition of each network can contain multiple systems.

```
DEFINE SYSTEM SYS1:
PROC PROC1;
DISK DISK1;
MEM MEM1;
RAID RAID1 1;
END;
```

Defining a system involves declaring single or multiple components in it. Note that the declaration of the RAID array is followed by a number indicating the number of disk failures that it can tolerate and run normally.

```
DEFINE RAID RAID1:
DISK DISK2;
DISK DISK3;
DISK DISK4;
END;
```

The definition of a raid array can contain multiple disks.

```
DEFINE DISK DISK2:
RUNNING simulation_testing_100002.csv;
END;
```

Each end level component like Memory, Disk and Procedure can be defined to have a file giving their readings for their run. If an end component does not specify a running file, it will not be monitored.

```
RELATION SYS2 DEPENDS_ON SYS1;
```

Users are allowed to define a relationship between two systems or networks, in terms of their dependence. Here the above declaration allows SYS2 to be defined as a dependent on SYS1. This declaration needs both the systems to be created before defining their relationship. Any fault or failure in SYS1 will result in a respective fault or failure in SYS2.

NOTE:

[1] - This language needs the components to be defined in their order of sub-grouping. i.e. the enclosing components in the architecture should be defined before the enclosed one (E.g. a machine should be defined before the disk in it.)

[2]- This language expects unique names of the components. i.e. no two devices can have the same name.

Developing a simulation includes building the specified environment from the given definition of the system. This includes writing a parser to read the configuration file and build an equivalent network configuration in simulation environment.

5.4 Implementing DFT and Rank Sum algorithms

This phase consists of implementing the DFT algorithm and the Rank Sum algorithm in simulation. The working details of these algorithms are explained in section 3.1 and 4.1 respectively. After successful integration of the fault prediction algorithms into the simulation environment, multiple experiments will be done to study the fault prediction pattern and prediction accuracy and precision of the algorithms. The dataset mentioned in [9] will be used in this project for testing the prediction algorithm. This dataset will also be used for collecting metrics for the project.

6 - Metrics

Metrics are a way of quantitative measurement usually within a certain domain [7].

The main emphasis of this project will be to understand the Output/Warning behavioral patterns followed by DFT and Rank-Sum for a given set of failure disks. Experiments will be conducted for collecting behavior statistics for both algorithms. Dataset to be used is given in [9]. The dataset will be studied and modified if needed, so as to meet the requirements of the project. Statistical analysis and data mining can be used for this purpose.

Other metrics that can be taken into consideration for this project are:

Prediction Accuracy: This is a measure of how exact is the prediction of the algorithm in question. In other words, this is the measure of the false alarms generated by the system. This is an important metric to be considered, as false alarms can incur high cost to vendor/ system owner. A false alarm may result in shutdown of a system. Hence it is very important to have minimum false alarms in any prediction.

Prediction Precision: Precision gives the measure of how precisely in time was the warning generated by the algorithms. This may allow us to distinguish the algorithms by when the warnings are generated in the time frame (early / late warnings). Certain applications may need an early warning whereas the others may need a just in time warning. This factor can help in choosing an algorithm based on the specified tradeoff.

Reference

[1] -Y. Lin and D. P. Siewioek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis", *IEEE Transaction on Reliability*, vol. 39, No. 4, 1990 October.

[2] - http://www-128.ibm.com/developerworks/rational/library/114.html

[3] - T. A. Dumitras and P. Narsimhan, "Fault-Tolerant Middleware and the Magical 1%", *ACM/IFIP/USENIX Conference on Middleware*, Grenhole, France, November-December 2005.

[4] - J. F. Murray, G. F. Hughes, K. Kreutz-Delgado, "Comparison of machine learning methods for predicting failures in hard drives", *Journal of Machine Learning Research*, volume 6, 2005.

[5]- Yawei Li and Zhiling Lan, "Exploit Failure Prediction for Adaptive Fault-Tolerance in Cluster Computing", *Proceeding of the sixth IEEE International Symposium on Cluster Computing and Grid*, 2006.
[6]- F. Pfisterer and L. Lehmann, " Processor System Modeling- A language and simulation system.

[7]- http://en.wikipedia.org/wiki/Metrics

[8]- G. F. Hughes, J. F. Murray, K. Kreutz-Delgoda and C.Elkan, "Improved Disk Drive Failure Warnings", *IEEE Transaction on reliability*, September 2002.

[9]- J. F. Murray, G. F. Hughes, K. Kreutz-Delgado, *Comparison of machine learning methods for predicting failures in hard drives, Journal of Machine Learning Research*, vol 6, 2005.