

DESIGN AND IMPLEMENTATION OF A MONITORING FRAMEWORK

A THESIS SUBMITTED TO THE GRADUATE SCHOOL OF NATURAL AND APPLIED
SCIENCES
OF MIDDLE EAST TECHNICAL UNIVERSITY

BY

KADİR KUZ

IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER
OF SCIENCE IN
COMPUTER ENGINEERING

MAY 2009

Approval of the thesis:

DESIGN AND IMPLEMENTATION OF A MONITORING FRAMEWORK

submitted by **Kadir Kuz** in partial fulfillment of the requirements for the degree of
**Master of Science in Computer Engineering Department, Middle East Technical
University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslüm Bozyiğit
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Ali Doğru
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Computer Engineering Dept., METU

Assoc. Prof. Dr. Ali Doğru
Computer Engineering Dept., METU

Assist. Prof. Dr. Tolga Can
Computer Engineering Dept., METU

Assist. Prof. Dr. Pınar Şenkul
Computer Engineering Dept., METU

Onur Aktuğ, M.Sc.
Lead Design Engineer, ASELSAN

Date: 12.05.2009

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Kadir KUZ

Signature : _____

ABSTRACT

DESIGN AND IMPLEMENTATION OF A MONITORING FRAMEWORK

Kuz, Kadir

M. S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali Dođru

May 2009, 61 pages

In this thesis work, the symptoms in Windows XP operating system for fault monitoring are investigated and a fault monitoring library is developed. A test GUI is implemented to examine this library. Performance tests including memory and CPU usage are done to see its overhead to the system and platform tests on the current version of Windows operating system series (Windows Vista) are done to see for compatibility.

In this thesis, fault monitor-fault detector interface is also defined and implemented. To monitor a symptom that is not implemented in the monitoring library, projects can implement their own monitors. A monitoring framework is designed to control and coordinate these monitors with the main one. To create monitors for Java projects easily, a monitor creator library is developed.

Keywords: Fault Monitoring, Fault Detection, Windows XP.

ÖZ

HATA İZLEME ALTYAPISI TASARIM VE GERÇEKLEMESİ

Kuz, Kadir

Yüksek Lisans , Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Ali Doğru

Mayıs 2009, 61 Sayfa

Bu tez çalışmasında Windows XP işletim sisteminde hata olarak neler izlenebileceği araştırılmış ve bir hata izleme kütüphanesi oluşturulmuştur. Bu kütüphanenin testi için kullanıcı ara yüzü geliştirilmiştir. Sisteme getirdiği yükü ölçmek için hafıza ve işlemci kullanımını içeren performans testi ile Windows işletim sisteminin şu andaki sürümü (Windows Vista) üzerinde platform uyumluluk testi yapılmıştır.

Bu tezde hata izleme-hata tespit ara yüzü de tanımlanmış ve gerçekleştirilmesi yapılmıştır. Hata izleme kütüphanesinde gerçekleştirilmiş bir belirtiyi izlemek için projeler kendi izleyicilerini oluşturabilirler. Bu izleyicileri kontrol etmek ve ana izleyici ile bir arada koordine bir şekilde çalıştırmak için izleme altyapısı oluşturulmuştur. Java projeleri için kolay bir şekilde izleyici oluşturmaya yardımcı olacak izleyici oluşturma kütüphanesi geliştirilmiştir.

Anahtar Kelimeler: Hata İzleme, Hata Tespit, Windows XP.

To My Family

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Assoc. Prof. Dr. Ali Doğru for his keen interest, guidance, encouragement, endless patience and insight throughout the completion of the thesis.

I am indebted to all of my friends and colleagues for their support and encouragements. I am also grateful to ASELSAN Inc. for the facilities that made my work easier.

I also thank to TÜBİTAK (The Scientific and Technological Research Council of Turkey) for their support through these hard days.

I would also like to thank Metin Tekkalmaz for his suggestions and comments.

The last but not least I would like to express my thanks to my family for the life they provide to me.

TABLE OF CONTENTS

ABSTRACT	iv
ÖZ	v
DEDICATION	vii
LIST OF TABLES	x
LIST OF FIGURES.....	xi
LIST OF ABBREVIATIONS	xiii
CHAPTERS	
1. INTRODUCTION.....	1
2. BACKGROUND INFORMATION.....	5
2.1 FAULT MANAGEMENT	5
2.1.1 Basic Definitions	5
2.1.2 Fault Management Stages.....	7
2.1.2.1 Monitoring	8
2.1.2.2 Detection.....	8
2.1.2.3 Prediction	11
2.1.2.4 Diagnosis	11
2.1.2.5 Isolation	12
2.1.2.6 Recovery	14
2.1.2.7 Repair.....	15
2.1.2.8 Reporting	16
2.1.3 Fault Management in Aselsan Inc.....	16
2.2 JAVA NATIVE INTERFACE (JNI).....	18
2.3 REMOTE METHOD INVOCATION (RMI)	19

3. DESIGN AND IMPLEMENTATION	22
3.1 FAULT MONITORING LIBRARY	22
3.1.1 Symptoms in Windows XP	22
3.1.2 Monitoring Library	34
3.1.3 Test GUI for Monitoring Library.....	35
3.2 MONITORING FRAMEWORK	40
3.2.1 Monitor – Detector Interface	40
3.2.2 Monitoring Framework.....	41
4. EVALUATION AND TEST RESULTS.....	46
4.1 PERFORMANCE TESTS	46
4.1.1 Testing Environment	46
4.1.2 Memory Tests	48
4.1.3 CPU Tests	50
4.2 PLATFORM TESTS	55
5. CONCLUSION	57
REFERENCES.....	59

LIST OF TABLES

TABLES

Table 1-1 Classes of High Availability Systems [1]	1
Table 4-1 CPU Usage with respect to Common Applications.....	55

LIST OF FIGURES

FIGURES

Figure 1-1 Fault Management Flow Diagram [1]	3
Figure 2-1 Fault, error, failure [4]	6
Figure 2-2 Fault Management Activity Diagram [5]	17
Figure 2-3 Java Native Interface Architecture [7]	18
Figure 2-4 Writing Native Methods [7]	20
Figure 2-5 RMI architecture [8]	21
Figure 3-1 Fault Symptoms Related With Memory Usage.....	23
Figure 3-2 CPU Usage	26
Figure 3-3 Disc Usage.....	26
Figure 3-4 Process I/O Status.....	27
Figure 3-5 Process Times.....	28
Figure 3-6 Version Information	29
Figure 3-7 Process Information.....	30
Figure 3-8 System Information	31
Figure 3-9 Application Information	31
Figure 3-10 Power Status	32
Figure 3-11 Network Information.....	33
Figure 3-12 JNI Framework in Fault Monitoring	34
Figure 3-13 Applications Tab	35
Figure 3-14 Processes Tab	36
Figure 3-15 Process Table Column Chooser	37
Figure 3-16 Process Information.....	38
Figure 3-17 Performance Tab	38
Figure 3-18 System Information Tab.....	39
Figure 3-19 Monitor – Detector Interface Sequence Diagram.....	42
Figure 3-20 Monitoring Framework	43

Figure 3-21 Monitor Interface Definition	44
Figure 3-22 Sequence Diagram between Monitor Controller and Detector	45
Figure 4-1 Working Set Size for Test GUI of Monitoring Library	48
Figure 4-2 Working Set Size for Process Explorer Tool	49
Figure 4-3 Heap Usage for Test GUI of Monitoring Library	49
Figure 4-4 Heap Usage for Process Explorer Tool	50
Figure 4-5 CPU Usage for Test GUI of Monitoring Library	51
Figure 4-6 CPU Usage for Test GUI of Monitoring Library under Heavy Load	51
Figure 4-7 CPU Usage for Task Manager Processes Tab	52
Figure 4-8 CPU Usage for Task Manager Performance Tab	52
Figure 4-9 CPU Usage for Task Manager Applications Tab	53
Figure 4-10 CPU Usage for Task Manager Network Connections Tab	53
Figure 4-11 CPU Usage for Process Explorer	54

LIST OF ABBREVIATIONS

API: Application Programming Interface

CPU: Central Processing Unit

DLL: Dynamic-Link Library

DOT: Distributed Object Technology

FM: Fault Management

GUI: Graphical User Interface

HA: High Availability

JNI: Java Native Interface

JVM: Java Virtual Machine

MMU: Memory Management Unit

RMI: Remote Method Invocation

XML: Extensible Markup Language

CHAPTER 1

INTRODUCTION

High availability (HA) is the term associated with computer systems which are able to provide service most of the time. The demand for these systems has grown over the past two decades. When everything is going well, the systems do not catch our attention! However, if they start to fail to perform their expected behavior, they got our immediate attention [1].

High available systems are those that are able to tolerate both expected and unexpected faults [2]. They exhibit almost no downtime. HA is often quantified in terms of the number of “9s” [1]. Table 1-1 shows the annual downtime of typical applications:

Table 1-1 Classes of High Availability Systems [1]

Number of 9s	Downtime per Year	Typical Application
3 Nines (99.9%)	9 hours	Typical Desktop or Server
4 Nines (99.99%)	1 hour	Enterprise Server
5 Nines (99.999%)	5 minutes	Carrier Class Server
6 Nines (99.9999%)	31 seconds	Carrier Switch Equipment

The bigger the number of 9s, the lesser downtime per year, means the higher uptime per year.

In software case, to show that an application does not include any errors is very difficult and it is known that every software have some defects. These defects cause failures. While some of these do not prevent the application performing its main task, some of them may be fatal [5, 25].

In order to make a system highly available, fully automated, intelligent and well-rounded fault management is needed to avoid unavoidable resource failures. Fault management (FM) is the process of detecting and identifying system faults and then taking corrective steps to get rid of them and making the system provide service again [3]. The stages of fault management are [1, 5]:

- Monitoring: The system resources are tracked.
- Detection: The fault is found
- Prediction: It is predicted that a fault will occur.
- Diagnosis: The cause and location of the fault is determined.
- Isolation: The faulty part is isolated form the rest of the system.
- Recovery: The faulty part is adjusted or re-started to function properly
- Repair: The faulty part is replaced.
- Reporting: Used in every stage of FM to notify the user about the goings on FM.

There are fine lines between these stages. The first three steps detect or predict that a fault has occurred. The fourth step examines the fault characteristics and the next three steps take corrective actions. Flow diagram of FM can be seen in Figure 1-1.

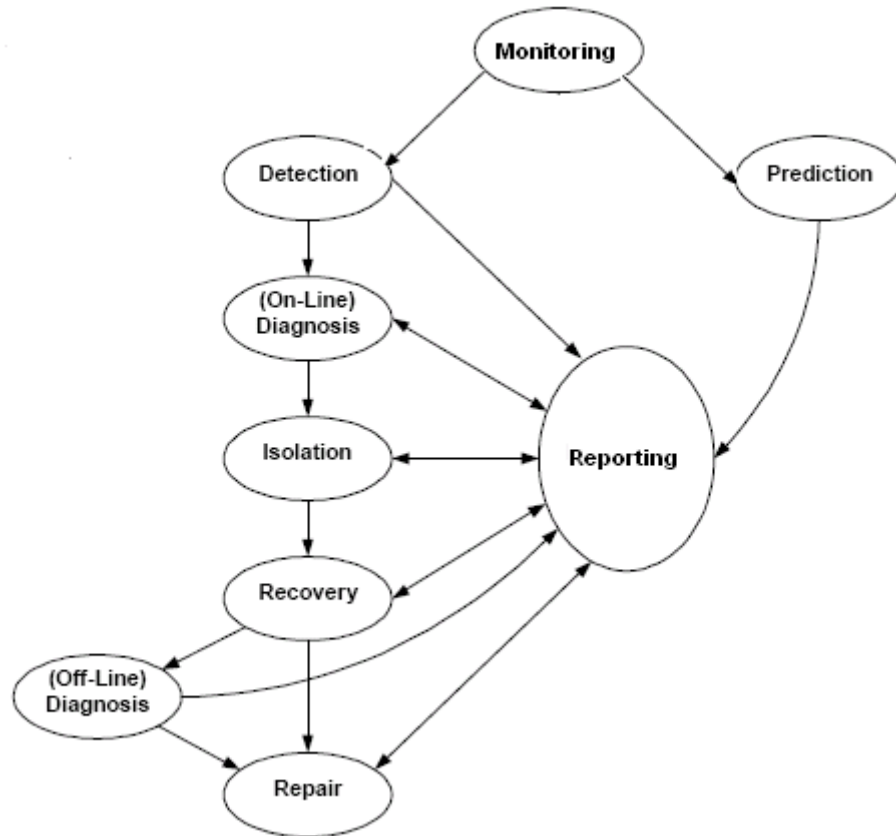


Figure 1-1 Fault Management Flow Diagram [1]

Motivation

In Aselsan Inc., fault management is being developed for REHIS Mission software department to make the systems more reliable and highly available. All of the fault management stages mentioned above are being carried out for embedded software which runs on VxWorks operating system. From now on, FM is planned to be applied to software running on different platforms like Windows, Solaris, etc. For

that reason, within the scope of this work, in this thesis, a fault monitoring library is developed for Windows XP operating system, since the software for user interface in our systems runs on Windows XP. First of all, the symptoms that can be monitored on this platform are investigated and the ones that are most useful for our fault detection and prediction mechanisms are considered together and combined in a library. The library is tested to see its behavior under heavy load. However, our aim is not to conduct performance tests on the library, only to see whether the overhead of the library to the system is acceptable or not. Moreover, it is also tested on Windows Vista to see for compatibility on the current version of Microsoft's Windows operating systems series.

A project may want to monitor project specific things or symptoms that are not considered in this library and may want to write its own monitor. For this purpose, a monitoring framework is designed for coordinating multiple monitors in the system and a helper library is developed to create monitors for Java projects easily.

Thesis Organization

This thesis is organized as follows: In Chapter 2, background information on Fault Management is given besides JNI and RMI. The implemented fault monitoring library and monitoring framework are detailed in Chapter 3. After detailed explanation, performance and platform test results are presented in Chapter 4. Finally, Chapter 5 presents our conclusions.

CHAPTER 2

BACKGROUND INFORMATION

In this chapter, Fault Management and the technologies that are used in the implementation of the fault monitoring library and monitoring framework are reviewed. After presenting the definition and steps of fault management, a brief explanation will be given for Java Native Interface (JNI) and Remote Method Invocation (RMI).

2.1 FAULT MANAGEMENT

This section briefly defines the concepts, principles and steps of fault management.

2.1.1 Basic Definitions

“High Availability”, (HA), is the term associated with computer systems that exhibit almost no downtime. High available systems provide service most of the time and this is quantified in terms of “number of 9s”. The higher the number of 9s means the system has lower downtime [1]. While 3 nines availability (9 hours downtime per year) is acceptable for a home user, at least 6 nines is required (30 seconds downtime per year) for carrier switching applications [9].

All systems are designed to have “Service Availability” most of the time. A service is basically the thing or behavior that user wants from a system. These systems are also called “Reliable” systems. The probability that a system will continue its

function properly without failure shows how reliable that system is. Besides, in [1] “Availability” is defined as the readiness of the system. The availability is expressed by:

$$\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$

MTTF: Mean Time to Failure.

MTTR: Mean Time to Repair.

Three terms are confused much; fault, error and failure. In Figure 2-1, they are explained briefly.

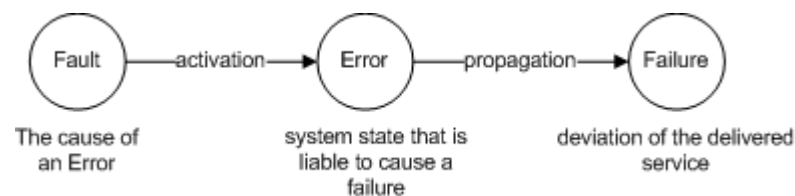


Figure 2-1 Fault, error, failure [4]

For example, throwing a telephone away is a fault. If it is broken, it is an error and this may result speaking failure. As it is seen, a fault may not cause an error and in the same way an error may not trigger a failure.

Fault types are [4]:

- Transient: Appears, and then disappears.
- Permanent: Remains until repaired.
- Intermittent: Occurs from time to time.

There are basically four main approaches to deal with these faults [2, 24]:

1. Fault avoidance
2. Fault removal
3. Fault masking
4. Fault forecasting

From these, last two techniques are implemented by using redundant or diverse implementation of a system to avoid the effects of faults whereas the first two are implemented by software and hardware.

To avoid or remove faults occurred in the system, system resources must be monitored like process memory usage, process thread count, CPU usage of the system, etc. In this thesis work, the symptoms that can be monitored in the system are researched and a fault monitoring library is developed for Windows XP.

2.1.2 Fault Management Stages

Fault management is considered as an eight-stage process in this research:

1. Monitoring
2. Detection
3. Prediction
4. Diagnosis
5. Isolation
6. Recovery
7. Repair

8. Reporting

Hereon, these stages will be explained briefly [1], [5].

2.1.2.1 Monitoring

Monitoring is the process of observing the state of the system resources such as CPU usage of a process, memory usage in system or process, etc. The states of the resources are sent to the detection or prediction stages or both of them for analysis.

To provide high levels of service availability, it is desirable that each component, process or system should be monitored. To say more, each component may report its current status to the main fault detector of the whole system. In this thesis, we provide a monitoring framework to achieve this desire. One main tracker contacts other component or process trackers and gets relevant information from them to detect faults and isolate or recover later.

Monitoring can be done synchronously or asynchronously. Fault detector may want the monitoring results at the time of the function call or tracker may send the results to the detector in every predefined time interval. However, there is a balance between monitoring and performance. Higher monitoring frequency brings more degraded system performance.

2.1.2.2 Detection

Detection is the process of identifying undesirable conditions which may lead to loss of service. It may be conducted as direct observation or by inference (observing other behavior of the system).

The main objective of fault detection is to detect if a fault occurs and provide information about a fault such as the location and type of the fault, the time of occurrence and may be the most likely next affected component.

There are two types of faults for a detector:

- **Active faults:** These are the faults that have been detected but not isolated or recovered from. The system may or may not continue its function when an active fault occurs. The system can deliver an acceptable level of service. For instance, when one of the CPUs of a dual core system fails, the system may continue its services if there is not much workload on it.
- **Latent faults:** A latent fault is a fault that escapes from the detection schema and remains undetected for a period of time. Uncovered fault scenarios are the main source of these types of faults.

A fault detector is a hardware or software component that detects if a fault occurs. If a fault detector finds an error in the output of a component, it is said that component has faulted. This is an example of direct detection. However, in indirect detection, the faulty component may be found by looking at more system-centric errors, such as high temperature, excessive memory or CPU usage of a process or system.

Faults can be detected through various ways within a system. For example, a fault may be detected at the source of the fault itself. Lots of components are designed so that the component can report error or undesirable conditions. An intelligent power supply is an instance that may send its state like working, degraded, etc. to its observers.

Faults may also be detected outside the faulty component. For example, a non-intelligent fan may not say to its observers that it is no longer working. But, if we

monitor the system temperature periodically, we may indicate that there is a problem in the cooling system.

Some of the techniques to detect faults are:

- **Value Range Checking:** It is expected that most of the operations must fall within a certain range. Tests can be done to check whether the result satisfies boundary conditions or not. For example, if the temperature of the system is above some threshold, it indicates an error.
- **Data Integrity Checking:** The data may be transferred from one component to another during the operation and it may get corrupted. This is true especially when the data is being transferred between hardware components. It may be useful to check for integrity at multiple positions. This may be achieved by inserting some information to the passed data to understand whether it is deformed or not.
- **Comparison Testing:** When redundant components are used within a system, it is possible to make calculations in parallel and compare the results. If they are not the same, there is an error.
- **Time Testing:** This is the simplest form of fault detection techniques. If an event is expected within a certain time period and if it has not happened, a fault is detected.

In fault detection, it is very important that if a component fails, it should fail with notification. Meaning, it can write its status to a log or notifies its listeners that it is failing.

2.1.2.3 Prediction

Prediction is the process of calculating the probability that a fault is about to occur in the system. Prediction gets the monitoring results and runs prediction rule set on these results and decides about the probability of a fault.

Prediction may decide also by referring to the error patterns. For example, in the past, usually an error happened when event A has occurred after events B and C. In runtime, prediction may estimate that there is an error if the same scenario occurs. Then, prediction sends its guess to the stage that takes corrective actions.

2.1.2.4 Diagnosis

After detection operation, a fault must be diagnosed to determine appropriate isolation and recovery techniques. The main objective of diagnosis operation is to identify the nature and location of the fault. To do so, diagnosis may check one or more system events or symptoms.

This step can be automatic or can be invoked by the user depending on the specific type of a component that is being diagnosed (hardware, operating system, application, etc.) or the type of the component which is responsible for diagnosis operation. For instance, a power supply may diagnose its problems while a fan may be diagnosed elsewhere in the system.

Diagnosis can be conducted while the system is continuing its normal operations (online). This implies that the fault is not fatal to the system. It is also done when the system is not running (offline). In offline diagnosis, there is no need to close all system applications every time. Only the part that will be conducted offline diagnosis may be restricted from being available to running applications. Thus, the system continues to supply services up to some level.

If the diagnosis is done within a system resource, it is local diagnosis. There is only one failure mode and diagnosis is implicit. Implicit diagnosis is the simplest form of diagnosis. The component reporting itself as failing is the faulty component. For example, low disk drive speed is determined with implicit diagnosis. However, if it is performed for the whole system by another resource in the system, it is global diagnosis. There are multiple failure modes and some additional work and query are required to find the location and nature of the fault. For instance, high CPU temperature may be caused by low fan speed or blocked air vents, etc.

Systems are composed of large number of resources including hardware and software that may fail anytime. In order to provide service availability most of the time, a quick response should be given in case of a failure. For that reason, the primary goal of the diagnosis is to find the faulty component to be replaced or repaired.

Diagnosis largely depends the following:

- The systems fault detection ability.
- Having information about the dependencies of the system components.
- Having information about the fault detectors in the system.

2.1.2.5 Isolation

The objective of isolation is to separate the faulty component from the service so as to keep the fault from scattering to other components in the system and cause secondary failures. To isolate, all the interactions with other components is removed.

There are two types of fault isolation:

- **Physical Isolation:** Physical isolation prevents the component from interacting with the system. It can be done, in hardware, by disconnecting the component from the bus or powering off the component. In software case, it can be accomplished through the Memory Management Unit (MMU) by preventing read, write on a page or memory location. This is also called “Data Isolation”.
- **Logical Isolation:** Logical isolation in hardware is to remove the device entries from the I/O subsystem so that no further interactions with the device are possible. This can be done either by preventing interactions with the device driver or by removal of the device driver. In software case, logical isolation can be done by the removal of the software like killing a process, unloading a library, etc.

Both of the techniques can also be used for isolation. For example, if a process in a card is failing, the process may be killed (logical isolation) and the card may be removed from the system (physical isolation).

In a system that is designed to achieve high availability, the cause and effect of a component removal must be clearly understood. Otherwise, fault management damages service availability.

Fault isolation includes actions that prevent faults from propagating in the system. Their aim is not to make system function properly. This is the job of the recovery step.

Some isolation techniques are:

- **Component Isolation:** If a fault on a component has a fatal effect, that component can be isolated physically or logically.

- **Quiescing Components:** Although the fault occurred on a component has detrimental effects on the component, it may be desirable that the component should complete its operations before isolation. To do so, input isolation is done first and then waited for the output in operation. After getting all of the output, component is isolated.
- **Safe Value Output:** If it is expected that a component should give results within a certain range and fails to do so, removing the component is one solution. However, this may cause undesirable conditions. So, it is designed that the component should give a safe value in case of failure until a recovery is made. This value can be the last valid value as well as the pre-programmed table value. However, the components interacting with the failing one should be notified about the error condition.
- **Routing Change:** If there are multiple components that can do the same job as the faulty one, one of them can be assigned to take the faulty components responsibilities.

2.1.2.6 Recovery

Recovery is the process of restoring the system in a state that it is again operable even in a degraded mode. Recovery includes the components that are affected by the failing component as well as failing one. It aims to provide service availability back to some level.

When redundant components are used for recovery, one is active while the others are in standby. In case of failure, one of the standby units become active. In order to do this, all of the standby components must know what is the last input and in which state the active component is. The technique that a standby component knows where and how it should start processing is “checkpointing”. In [2, 26], it is

stated that checkpointing includes taking a snapshot of the component's situation just as it is about to begin the first step of the next transaction. The snapshot is taken only if the previous transaction was completed successfully.

Another technique for recovery is "Software Rejuvenation". This is reloading an image, restarting an application, dynamic library or initializing the resources that the service uses. In some cases, doing the things above may not be sufficient. Restarting the operating system may be needed [26].

Recovery depends on redundant elements and the techniques that can be applied which are stated on policies while the system is being configured. A policy initiates actions according to the information that comes from the detector. It is invoked when a detector fires. This makes a policy to respond several detectors in the same way.

2.1.2.7 Repair

The aim of the repair is to return the system to its original state including all the redundancies. This operation particularly depends on the type of the fault. If the faulty component is a physical hardware unit, then a hot replacement is done. Before removal of the component, diagnostic tests can be run in order to detect the component that causes the failure. These tests are also done after replacement of the new component to verify that it is working correctly.

In case of software, the repair operation may be conducted as downloading another copy of it. However, sometimes this may not be sufficient. A system patch may be needed. Patches are small changes in the code and can be applied while the system is performing its operation. Another method is upgrading the software. The software version that has been loaded to the system may include lots of bugs and in

the new release these bugs may have been covered. So, upgrading the software to the new version may also repair software [27].

2.1.2.8 Reporting

Reporting is the process of notifying the user by the fault management process. This includes logging to a file, printing a message, giving alarms, displaying graphs, etc.

2.1.3 Fault Management in Aselsan Inc.

Figure 2-2 shows the activity diagram of fault management procedure that is developed in Aselsan Inc.

In [5], it is stated that monitoring is done for the symptoms that may lead to failures in the system such as CPU Usage, memory usage, process status, etc and the results are sent to the detection, prediction or both of them. In case of prediction, the probability that a fault occurs is calculated according to the prediction rules. In case of detection, it is determined whether a fault is occurred and sent to the diagnosis. In diagnosis, the type, location and nature of the fault is determined by using rules, algorithms and previous fault decisions. In the correction step, the fault either came from diagnosis or prediction. Corrective action may isolate the faulty component from the system in order not to cause secondary failures, switch on a backup component in place of the faulty one, or restart the faulty component. Correction step understands whether the action is successful or not by the information that comes from diagnosis. If not, it can retry or make a comprehensive corrective action. Other than these actions, repair is made to the faulty component offline. Repair includes debugging and recompiling for software or changing the defective part of hardware.

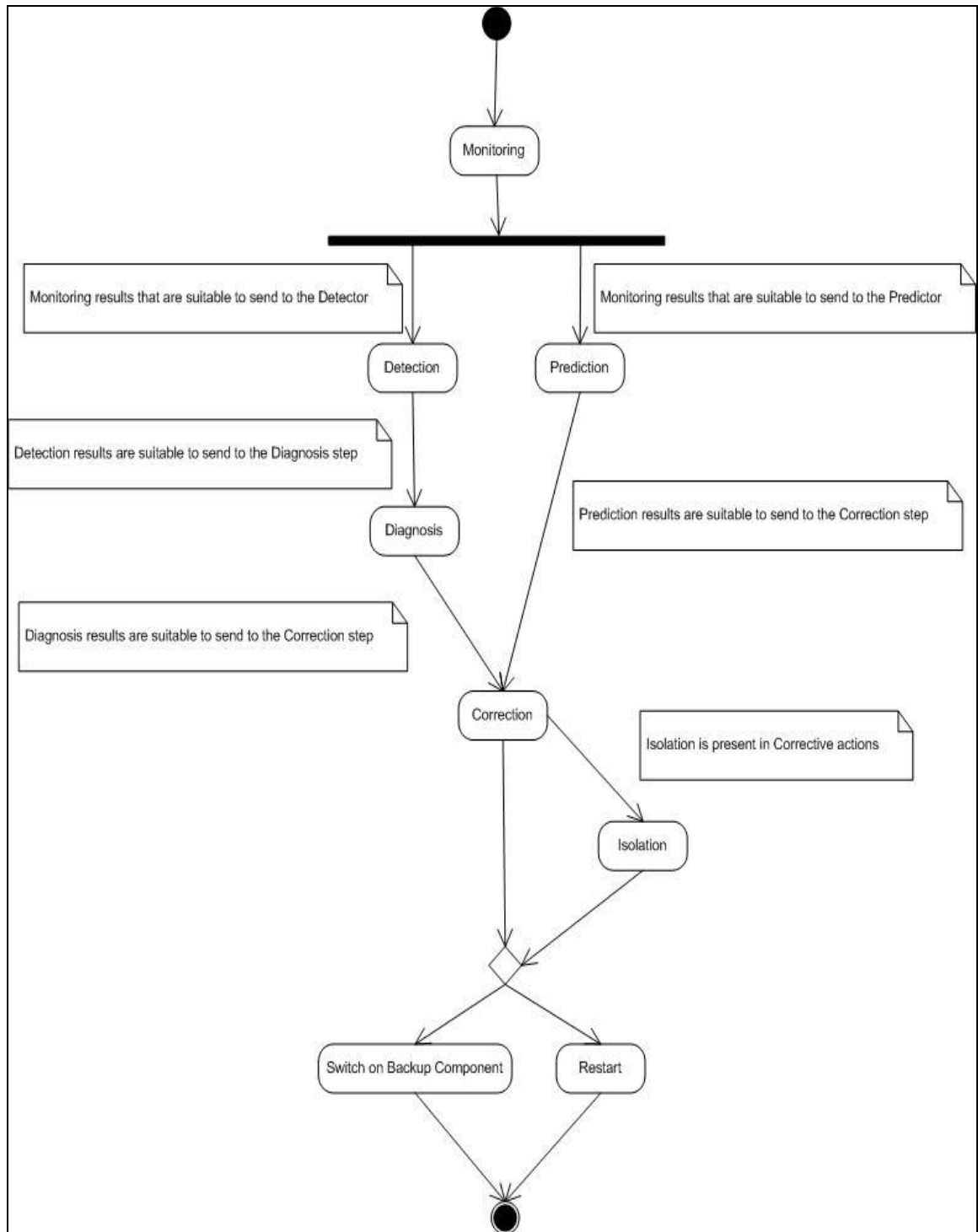


Figure 2-2 Fault Management Activity Diagram [5]

2.2 JAVA NATIVE INTERFACE (JNI)

Java Native Interface (JNI) is a programming framework which allows Java code running in a Java Virtual Machine (JVM) to call and be called by native applications and libraries written in other languages, such as C, C++, etc. [6].

In [7], it is stated that JNI is generally used in the following situations:

- ✓ When program needs platform-dependent features that standard Java class library does not support.
- ✓ When libraries and applications already written in other programming languages like C are wanted to be accessible in Java applications.
- ✓ In time-critical functions, Java may not be a good choice. For that reason, it may be preferred to write the critical part of the application in low-level programming languages such as assembly and to be called within Java.

JNI can be thought as a glue between Java and native applications. Figure 3 shows how JNI ties the C side of the program to the Java side [7].

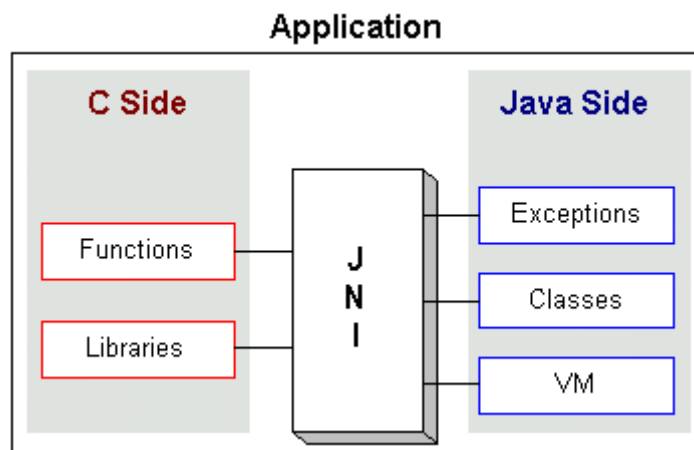


Figure 2-3 Java Native Interface Architecture [7]

The steps to write native methods for Java are [7]:

1. A Java class that declares native methods is created.
2. The Java class that includes the native methods is compiled.
3. A header file which contains the signature of native methods is generated by using `javah` with `-jni` flag.
4. Native method's implementation is written in the language that is preferred (C, C++, or Assembly).
5. A shared library is created by compiling the header and the implementation file.
6. Java program is ready to run.

These steps are illustrated in Figure 2-4. In the figure, the `HelloWorld` class calls the native method `displayHelloWorld` and loads the shared library `hello`. `HelloWorldImp` class implements the native method and is compiled with the `HelloWorld` header file to generate the `hello` library.

2.3 REMOTE METHOD INVOCATION (RMI)

Java Remote Method Invocation (RMI) allows a Java Virtual Machine (JVM) to call a method of an object in another Java Virtual Machine. This JVM may be on the same machine or on a different one.

Figure 2-5 shows the RMI architecture.

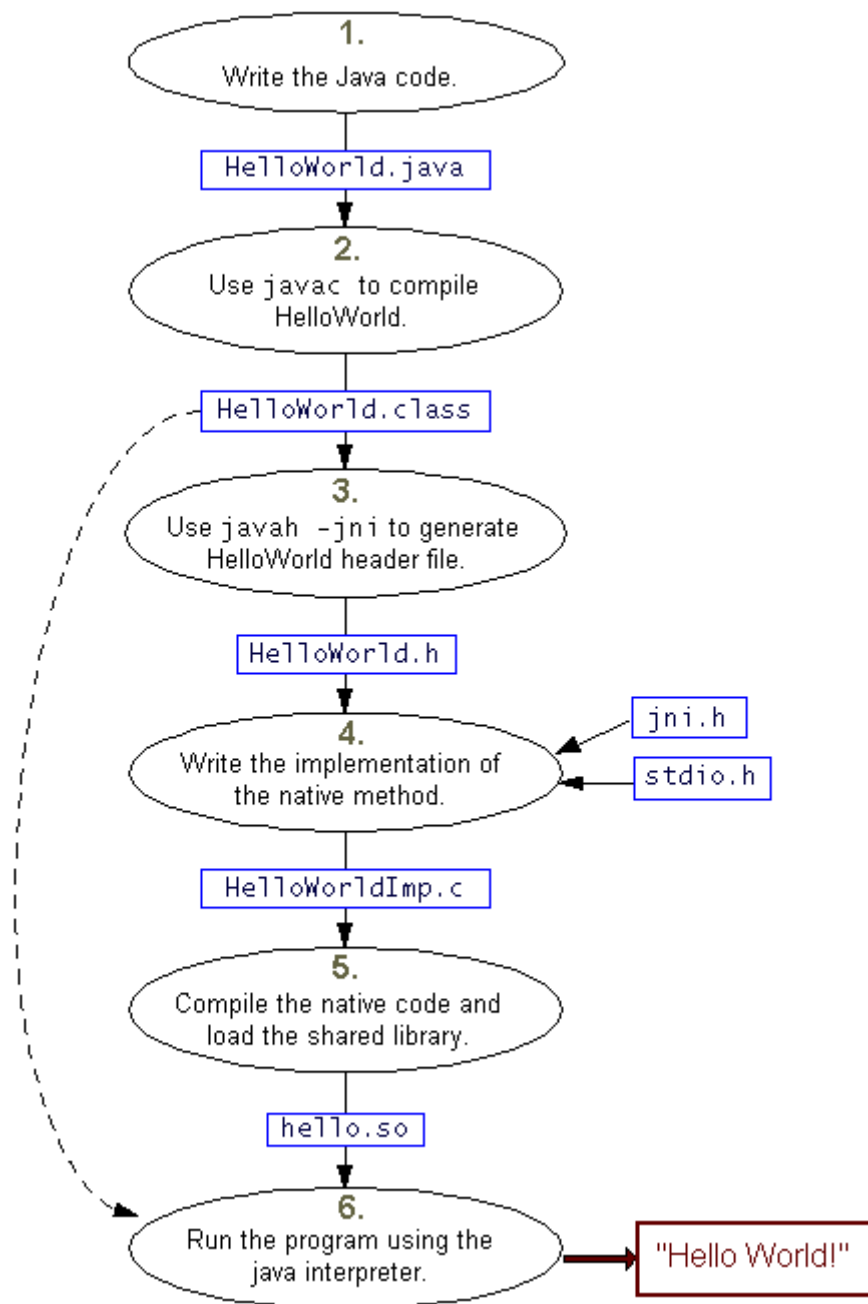


Figure 2-4 Writing Native Methods [7]

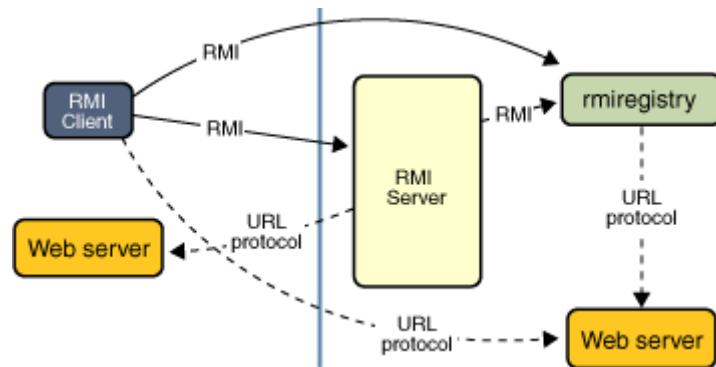


Figure 2-5 RMI architecture [8]

In RMI, there are often two programs; a client and a server. Server creates remote objects and puts references of these objects to the RMI registry. The client program looks up this registry for the reference of the object to be invoked. Once obtaining the reference, the client can call methods of the remote object. This makes two JVM communicate with each other.

Dynamic code downloading is one of the main features of RMI. If server object's class is not defined in the receivers' side, RMI downloads the class from the server JVM so that all the types and behavior of the object is transmitted to the receivers JVM [8].

CHAPTER 3

DESIGN AND IMPLEMENTATION

In this chapter, fault monitoring library is discussed first. Then, a GUI is presented that is for testing whether the library works correctly or not. Finally, design and implementation of the monitoring framework is explained in a detailed way.

3.1 FAULT MONITORING LIBRARY

The first step of fault management is monitoring the system resources for fault detection and prediction. There are lots of symptom types in an operating system that can be used for fault monitoring. However, monitoring all of them brings performance bottleneck. For this reason, the most useful ones should be selected.

In this section, the symptoms in the library will be presented and the test GUI developed to test this library will be shown.

3.1.1 Symptoms in Windows XP

In this subsection, symptoms that can be monitored in Windows XP operating system will be explained. Here are the most useful symptoms for fault management [28]:

- **Memory Usage:** Figure 3-1 shows all the symptoms that can be monitored for memory usage in the system.

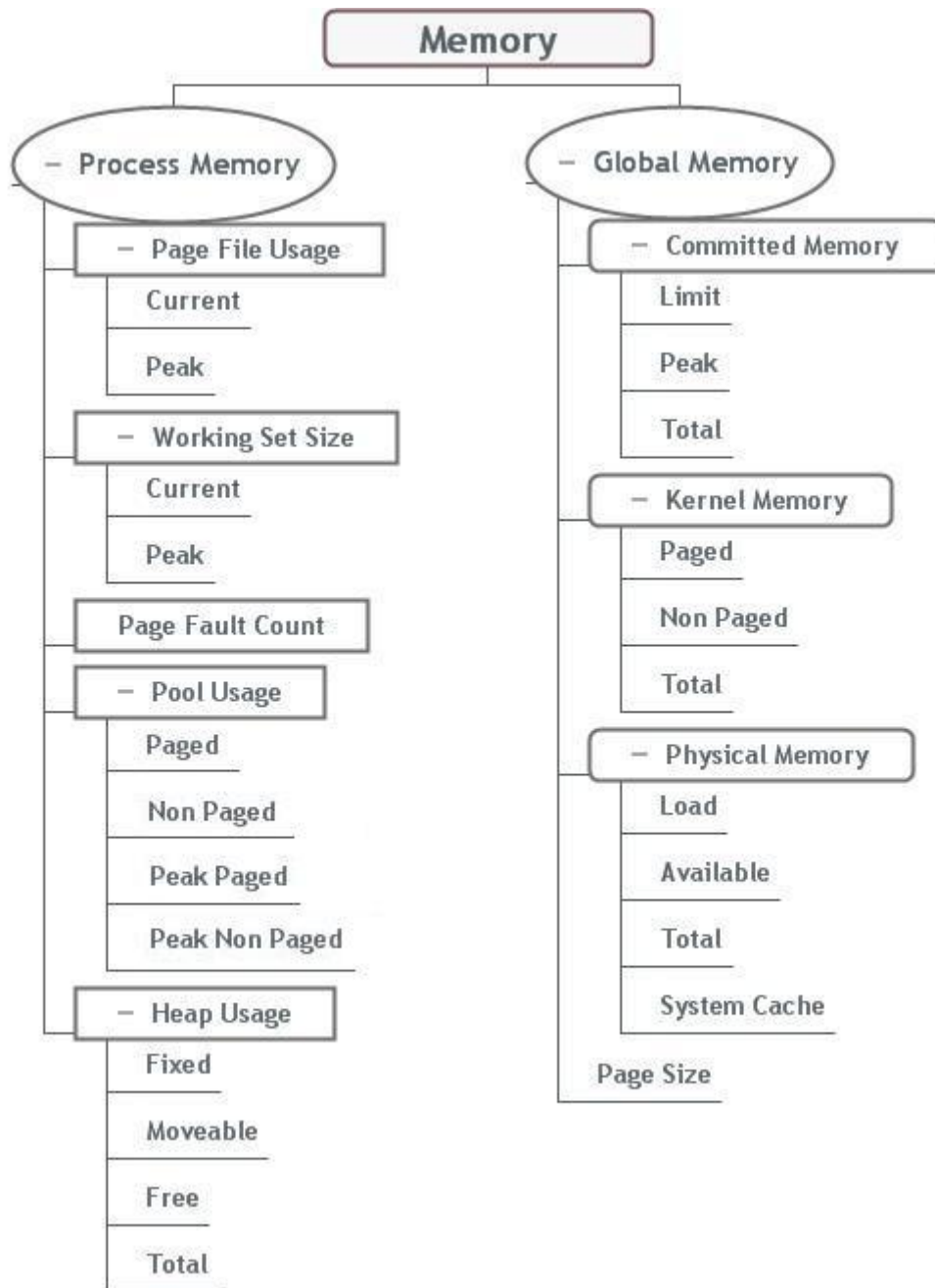


Figure 3-1 Fault Symptoms Related With Memory Usage

All application programs and many system processes except core parts of the operating system use virtual memory addresses which are then translated

to the RAM addresses by the hardware. A 32 bit Windows O.S. has approximately 4 GB virtual memory addresses no matter how much physical memory exists in the system. 2 GB of it is used for each process' private usage and the other 2 GB is shared between all processes and the operating system. When the memory used by all processes exceeds the amount of memory available (RAM), operating system moves some pages to the hard disk to free some memory. Thus, the increase in page file usage results in performance decrease [10].

In [11], it is defined that working set of a process is the set of pages in its virtual address space which are available in RAM. When a process wants to access a page that is not included in the working set, a "page fault" occurs. If the page fault handler resolves the fault, the page is added to the working set. The more page fault the more degraded performance.

A memory leak occurs when a process allocates memory for use and does not free when it finishes its job. These leaks cause Windows to slow down. Leaks generally occur in small amounts and may take time to detect. Memory manager creates paged and nonpaged pools to allocate memory. The nonpaged pool contains virtual memory addresses that are guaranteed to reside in RAM whereas the paged pool consists of virtual memory that can be paged in or out [12, 13].

When an application is started, the operating system creates a virtual address space that contains stack, application code, application data, heap, etc. Heap is the place where dynamic memory allocation occurs. The heap memory block can have a fixed location or the location can be moveable. If a memory allocated on the heap is not freed after its job is finished, the heap may be bloated.

Every address can be thought of as one of these; free, reserved or committed at any given time. At the beginning all of the addresses of a process are free. Then, an address may be reserved or committed. An address is reserved to protect it from allocation requests of other processes. This type is usually needed in operations such as loading a DLL or resource that could occupy specific addresses. To use reserved addresses, memory must be committed to the addresses. When memory is committed, pages of it always exists as either physical pages of memory or as pages that have been paged to the pagefile on disk [15]. The committed memory limit shows the maximum size that virtual memory can be without making any changes to its configuration. Peak committed memory is the maximum amount of memory used since tracking begun. Total committed memory shows the virtual memory size in use [16].

Kernel is the core part of the operating system that does most of the background work. Paged kernel memory shows the amount of virtual memory set aside for the kernel while nonpaged kernel memory shows the amount of RAM dedicated to the kernel [16].

Physical memory is the total amount of memory installed in the computer. The physical memory is not used fully. If it is becoming full some of the memory is written to the disk. System cache shows the amount of memory which is used for quick access of the recently used data.

- **CPU Usage:** CPU (Central Processing Unit) is the main and perhaps the most important component of a computer. It does the execution of the commands and data processing. High CPU Usage means lots of processing is being done in the CPU that results in slowing of programs. In order to run all the programs concurrently, the CPU will execute commands of each program in some order. If there are lots of programs that uses CPU much,

the time given to each program will decrease and the program will do its job in a bigger time interval. As it is seen from Figure 3-2, two kinds of CPU usage is being monitored; system and process. “System CPU Usage” shows the total CPU usage of the computer while “Process CPU Usage” shows the CPU usage of the specified process. If a process or system highly utilizes the CPU for a long time, there may be a problem with the system or process. Moreover, in systems that have multiple CPUs, each CPUs burden can be retrieved and some arrangements can be done to decrease the burden of heavy ones. Hence CPU usage is usually the most important symptom for monitoring.

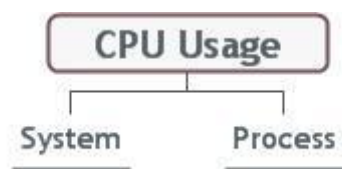


Figure 3-2 CPU Usage

- **Disc Usage:** Nearly all of the application programs write something to disc. Log information, output of a program such as screen shots or backup data for later usage, and temporary files that are created by the program are some examples of disc usage.



Figure 3-3 Disc Usage

In Windows platforms, disks are usually partitioned into disc drives. Each drive has free and used space. In the monitoring library, free and total space of each drive is being monitored.

- **Process Input/Output Status:** Each process performs I/O operations. The monitoring library contains APIs for process I/O accounting information such as read, write and other operations (Figure 3-4) [17].

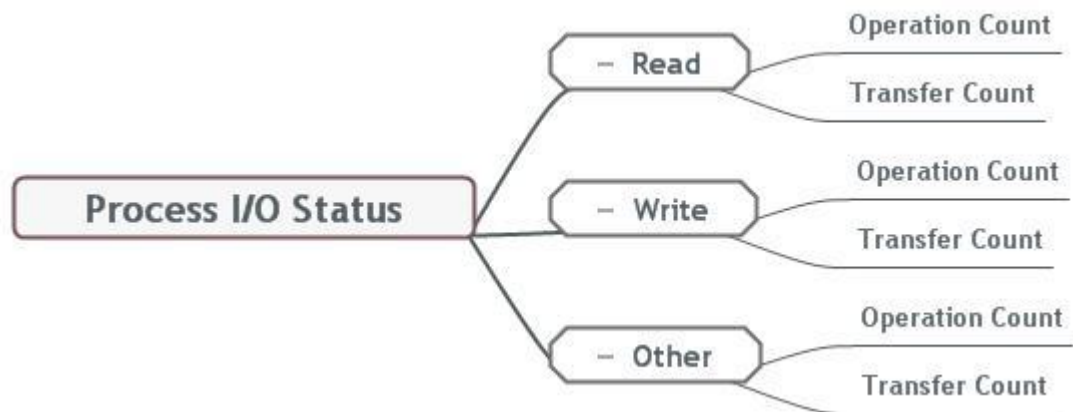


Figure 3-4 Process I/O Status

- *Read Operation Count*: number of read operations performed.
 - *Read Transfer Count*: number of bytes read.
 - *Write Operation Count*: number of write operations performed.
 - *Write Transfer Count*: number of bytes written.
 - *Other Operation Count*: number of I/O operations (other than read and write operations) performed.
 - *Other Transfer Count*: number of bytes transferred during I/O operations other than read and write.
- **Process Times:** This symptom shows timing information about each process [18].



Figure 3-5 Process Times

- *Start Time*: shows the creation time of the process.
- *Exit Time*: shows the exit time of the process. If the process is not exited, it is undefined.
- *Kernel Time*: shows the amount of time that process has executed in kernel mode.
- *User Timer*: shows the amount of time that process has executed in user mode.

In kernel mode, the application has unrestricted access to the underlying hardware. It can execute any instruction and reference any memory address. Kernel mode is generally used by the most trusted operating system functions. In user mode, the application can not directly access the hardware or reference any memory location. To access, it uses system APIs [19]. Every process except “System” is started in user mode in Windows. When a program wants to call a Windows API, it triggers an interrupt and CPU switches to the kernel mode and begins executing the API. After finishing its job, CPU switches back to the user mode again and continues execution of the program [20]. For that reason, total CPU time for a process is the sum of the kernel time and the user time.

- **Version Information:** Version information is useful for applications to install the files properly. For example, an application may need the Adobe

Reader version 5.0 in order to run. If not present, it may decide to install that version. For another example, a program may run efficiently on versions of the Windows operating system that are above some version number.

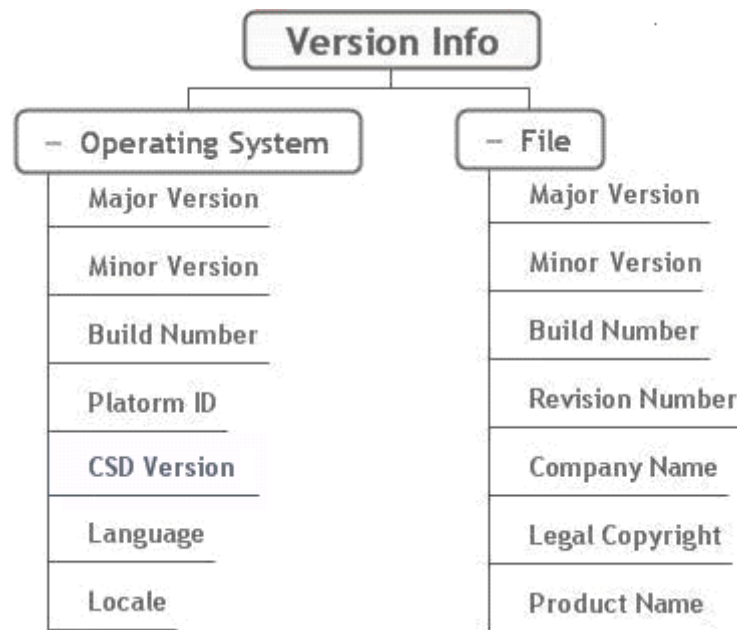


Figure 3-6 Version Information

Figure 3-6 shows the types of version information that monitoring library tracks:

- *Operating System*: shows the major and minor versions, build number and platform ID. “CSD Version” shows the latest Service Pack installed such as “Service Pack 3”. While “Language” shows the operating systems language like “Turkish”, “Locale” represents a specific geographical region.
- *File*: shows the major and minor versions, build, and revision numbers. Besides these, name of the product, company produced the file, and copyright applied to the file is given.

- **Process Information:** Figure 3-7 shows some other process information that is monitored.

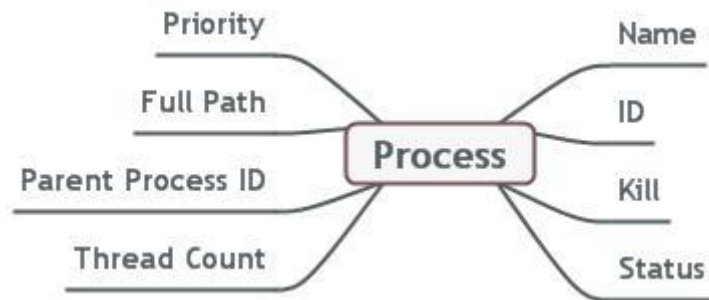


Figure 3-7 Process Information

- *Priority*: shows the priority value of the process like Normal, High, and Idle. The operating system uses this information to determine which process (thread) gets the next slice of CPU time.
- *Status*: shows whether the process is running or not. In windows, information about status of a process like running, pending, stable, killed etc is not given with an API therefore; only the status types of “available” or “not available” are being monitored.
- *Thread Count*: shows the total number of threads of a process.

Moreover, name, ID, parent process ID and full path of a process can be retrieved from the operating system. To kill a process, an API is written which will be useful especially in the isolation step of fault management.

- **System Information:** Total thread count, handle count and process count in the system is monitored (Figure 3-8).



Figure 3-8 System Information

- **Application Information:** A process may have one or more applications in the system. For example, Firefox process [23] can open three web sites as three different applications. Killing the Firefox process kills all the applications; however, killing one of the windows (web site) does not kill others. In the monitoring library, required APIs are written to monitor all of the applications in the system.

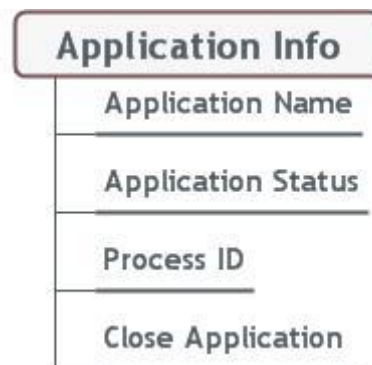


Figure 3-9 Application Information

- *Application Name*: shows the title of the application.
- *Application Status*: shows the status of the application. It can be "Running" or "Not Responding".
- *Process ID*: shows the ID of the process that created the application.

Moreover, an API is written to close an application which may be used in the isolation step of the fault management mechanism.

- **Power Status:** Power status of the system is one of the most important symptoms. It indicates whether the system is running on AC or DC power and whether the remaining battery life is sufficient or not [21].

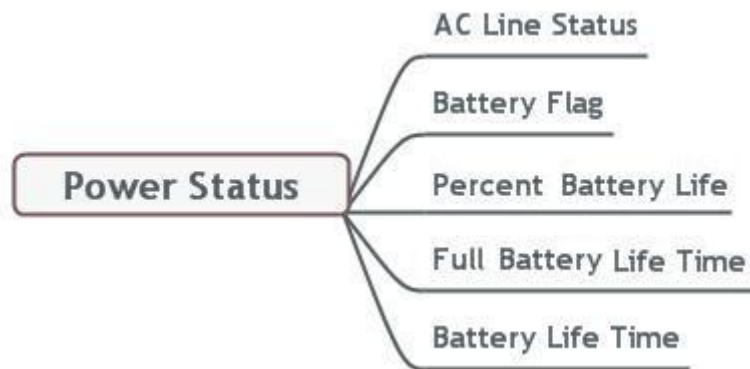


Figure 3-10 Power Status

- *AC Line Status*: indicates AC power status: online, offline.
- *Battery Flag*: shows charge status: High, low, Critical, Charging, No System Battery.
- *Percent Battery Life*: shows the percentage of the remaining battery charge.
- *Full Battery Life Time*: shows the number of seconds of battery life at full charge.
- *Battery Life Time*: shows the number of seconds of battery life remaining.

- **Network Information:** Applications may send and receive information to/from a specific address over network. For that reason, information about the quality and status of a network is very important.



Figure 3-11 Network Information

- *Network Status:* shows the status of the network adapter; connected or not.
- *Network Address Reachability:* shows whether the specified network address is reachable or not.
- *Band Width:* shows the band width of the network adapter.
- *Sent Packets:* shows the number of packets sent over the network adapter.
- *Received Packets:* shows the number of packets received over the network adapter.
- *Sent Bytes:* shows the number of bytes sent over the network adapter.
- *Received Bytes:* shows the number of bytes received over the network adapter.

- *Network Usage*: shows the percentage of the band width being used.

3.1.2 Monitoring Library

The symptoms that are detailed in section 3.1.1 are implemented and a shared monitoring library is developed. The library is written in C++ to access operating system resources easily and is developed in .NET 2005 environment.

In Aselsan Inc, in Mission Software department, user interface software is being developed in Java programming language and run on the Windows XP operating system. As it is discussed in Section 3.2, monitoring framework is developed in Java to add Java Project trackers to the framework easily. This framework includes the main tracker that uses fault monitoring library. For that reason, the library is compiled in a way that is compatible with Java Native Interface (JNI) framework to be accessed within Java.

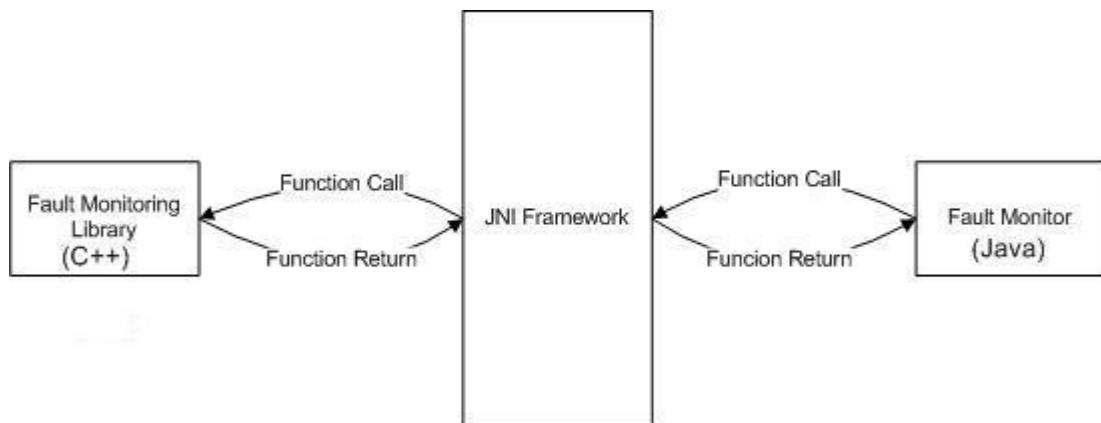


Figure 3-12 JNI Framework in Fault Monitoring

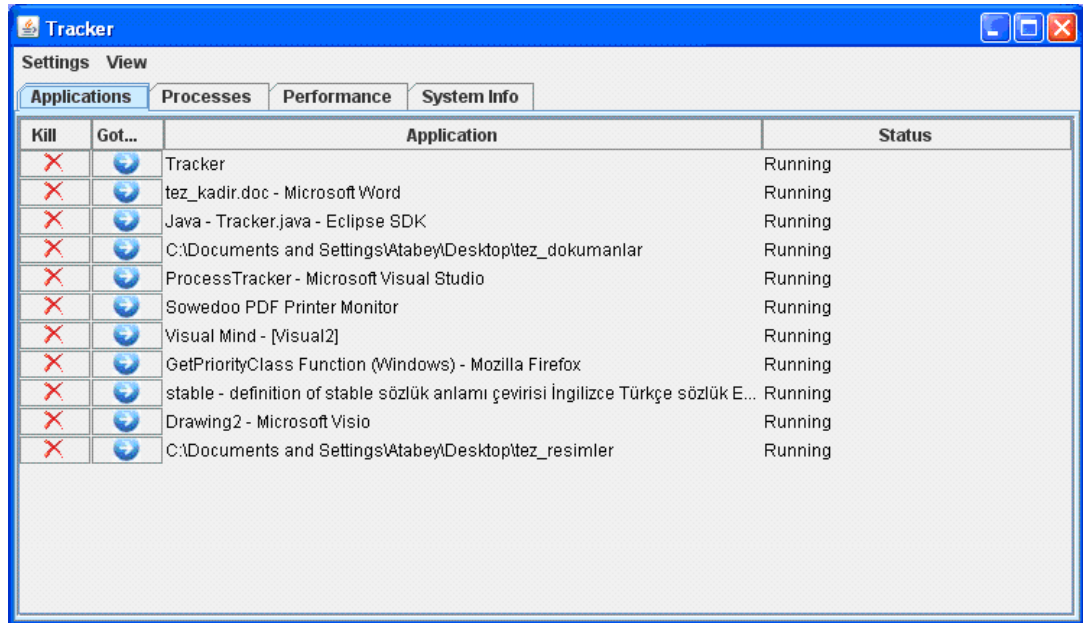
As it is seen in Figure 3-12, fault monitor that is written in Java makes function calls to the fault monitoring library written in C++ like its own function via JNI framework. JNI does all the communication work between. If the library is wanted

to be called from a C or C++ project, compiling the library with suitable headers is enough.

3.1.3 Test GUI for Monitoring Library

The monitoring library is tested for all symptoms by implementing a graphical user interface (GUI).

The “Applications” tab, Figure 3-13, shows all the applications in the system. In the table, application name and status are displayed. The status of the application is “Running” if the window of it responds to the sent message within specified time. If it does not respond, status becomes “Not Responding”. While “Kill” button closes the application, “Go to” button opens the “Processes” tab and selects the process of the application. The table grows as new applications opened and shrinks if they are closed.



Kill	Got...	Application	Status
X	→	Tracker	Running
X	→	tez_kadir.doc - Microsoft Word	Running
X	→	Java - Tracker.java - Eclipse SDK	Running
X	→	C:\Documents and Settings\Atabey\Desktop\tez_dokumanlar	Running
X	→	ProcessTracker - Microsoft Visual Studio	Running
X	→	Sowedoo PDF Printer Monitor	Running
X	→	Visual Mind - [Visual2]	Running
X	→	GetPriorityClass Function (Windows) - Mozilla Firefox	Running
X	→	stable - definition of stable sözlük anlamı çevirisi İngilizce Türkçe sözlük E...	Running
X	→	Drawing2 - Microsoft Visio	Running
X	→	C:\Documents and Settings\Atabey\Desktop\tez_resimler	Running

Figure 3-13 Applications Tab

Figure 3-14 shows the “Processes” tab. All of the processes in the system are listed in this tab. ID, name, CPU usage, priority, full path and thread count of a process are presented in each row. “Kill” button, at the beginning of each row, kills the selected process.

Kill	ID	Name	CPU	Priority	Full Path	Thread Count
X	0	System Idle Process	854	Absent	Unknown	1
X	2852	VISUAL-1.EXE	57	Normal	C:\PROGRAM-1\WINDTE-1\VISUAL-1\VISUAL-1.EXE	6
X	3452	sunum.TrackerMain	38	Normal	C:\Program Files\Java\jre6\bin\javaw.exe	18
X	4	System	19	Normal	Unknown	55
X	3420	firefox.exe	9	Normal	C:\Program Files\Mozilla Firefox\firefox.exe	14
X	388	mpd.exe	9	Normal	C:\Program Files\MPICH\mpd\bin\mpd.exe	6
X	516	sqlservr.exe	0	Normal	C:\Program Files\Microsoft SQL Server\MSSQL\1\MSSQL\Binn...	20
X	1952	PadExe.exe	0	Normal	C:\Program Files\TOSHIBA\Touch and Launch\PadExe.exe	6
X	1040	svchost.exe	0	Normal	C:\WINDOWS\system32\svchost.exe	76
X	576	svchost.exe	0	Normal	C:\Program Files\Java\jre6\bin\svchost.exe	1
X	652	csrss.exe	0	Normal	C:\WINDOWS\system32\csrss.exe	12
X	1588	C:\Documents	0	Normal	C:\WINDOWS\system32\javaw.exe	24
X	544	ctfmon.exe	0	Normal	C:\WINDOWS\system32\ctfmon.exe	1
X	3928	WINWORD.EXE	0	Normal	C:\Program Files\Microsoft Office\OFFICE11\WINWORD.EXE	9
X	1968	alg.exe	0	Normal	C:\WINDOWS\system32\alg.exe	6
X	1448	explorer.exe	0	Normal	C:\WINDOWS\explorer.exe	15
X	676	winlogon.exe	0	High	C:\WINDOWS\system32\winlogon.exe	16
X	260	avgemc.exe	0	Normal	C:\PROGRAM-1\Grisoft\AVG7\avgemc.exe	10
X	608	avgcc.exe	0	Normal	C:\PROGRAM-1\Grisoft\AVG7\avgcc.exe	9
X	152	avgamsvr.exe	0	Normal	C:\PROGRAM-1\Grisoft\AVG7\avgamsvr.exe	8
X	2620	tbnote.exe	0	Normal	C:\Program Files\TurboNote\tbnote.exe	3
X	272	CFSvcs.exe	0	Normal	C:\Program Files\TOSHIBA\ConfigFree\CFSvcs.exe	3
X	632	SNM.exe	0	Normal	C:\Program Files\SpyNoMore\SNM.exe	4
X	540	qttask.exe	0	Normal	C:\Program Files\QuickTime\qttask.exe	2
X	3698	usnsvc.exe	0	Normal	C:\Program Files\MSN Messenger\usnsvc.exe	3
X	1280	msnappau.exe	0	Normal	C:\Program Files\MSN Apps\Updater\01.02.3000.1001\tr-trim...	2
X	732	lsass.exe	0	Normal	C:\WINDOWS\system32\lsass.exe	19
X	2740	wmiapsrv.exe	0	Normal	C:\WINDOWS\system32\wbem\wmiapsrv.exe	3
X	236	avgupsvc.exe	0	Normal	C:\PROGRAM-1\Grisoft\AVG7\avgupsvc.exe	3
X	4060	devenv.exe	0	Normal	C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\de...	11
X	3200	VISIO.EXE	0	Normal	C:\Program Files\Microsoft Office\Office12\VISIO.EXE	4
X	884	svchost.exe	0	Normal	C:\WINDOWS\system32\svchost.exe	17
X	1184	jucheck.exe	0	Normal	C:\Program Files\Java\jre6\bin\jucheck.exe	5
X	328	jqs.exe	0	Idle	C:\Program Files\Java\jre6\bin\jqs.exe	10
X	1752	spoolsv.exe	0	Normal	C:\WINDOWS\system32\spoolsv.exe	10
X	940	ULCDRsvr.exe	0	Normal	C:\Program Files\Common Files\Ulead Systems\DVD\ULCD...	2
X	560	SwFmMon.exe	0	Normal	C:\Program Files\Common Files\Sowedoo Shared\Sowedoo...	4
X	312	Acrotray.exe	0	Normal	C:\Program Files\Adobe\Acrobat 7.0\Distillr\Acrotray.exe	2
X	2840	Acrobat.exe	0	Normal	C:\Program Files\Adobe\Acrobat 7.0\Acrobat\Acrobat.exe	3
X	1920	eclipse.exe	0	Normal	C:\Documents and Settings\Atabey\Desktop\eclipse-SDK-3.2...	1
X	1680	GoogleToolbarNotifier.exe	0	Normal	C:\Program Files\Google\GoogleToolbarNotifier\GoogleTool...	4
X	588	smss.exe	0	Normal	C:\WINDOWS\system32\smss.exe	3
X	1464	svchost.exe	0	Normal	C:\WINDOWS\system32\svchost.exe	15
X	1160	svchost.exe	0	Normal	C:\WINDOWS\system32\svchost.exe	4

Figure 3-14 Processes Tab

The table can be sorted according to the specified column like CPU usage, Thread count so as to see the dominant processes in that column type. The table is refreshed in each predefined time interval. As defined in section 3.1.1, there are much more symptoms about a process. The user can add these symptoms to the table by using the “Select Column” menu item under “View” menu. In the opened window (Figure 3-15) the user may add new columns to the table or remove the existing ones.

Furthermore, version information, product name, legal copyright of a process can be viewed by double-clicking on a process row in the table. The opened window can be seen in Figure 3-16.

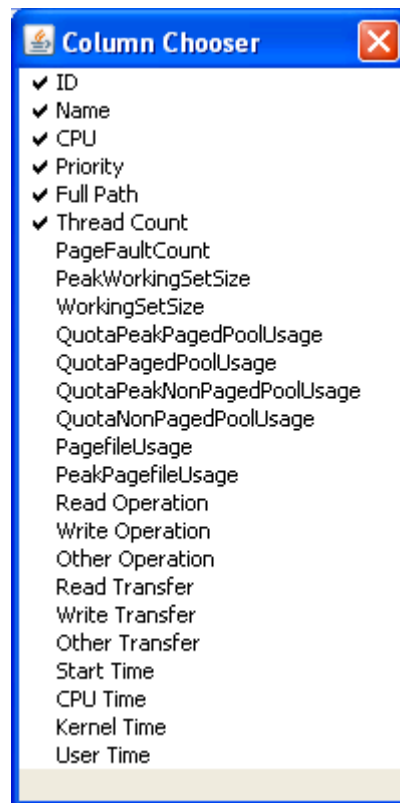


Figure 3-15 Process Table Column Chooser

The “Performance” tab, as it is seen in Figure 3-17, shows the general performance information about the system. Total or available physical memory; paged or nonpaged kernel memory; total or peak committed memory in the system are presented numerically and graphically. Moreover, total process, handle and thread count in the system are shown. These values are refreshed in every predefined time interval.



Figure 3-16 Process Information

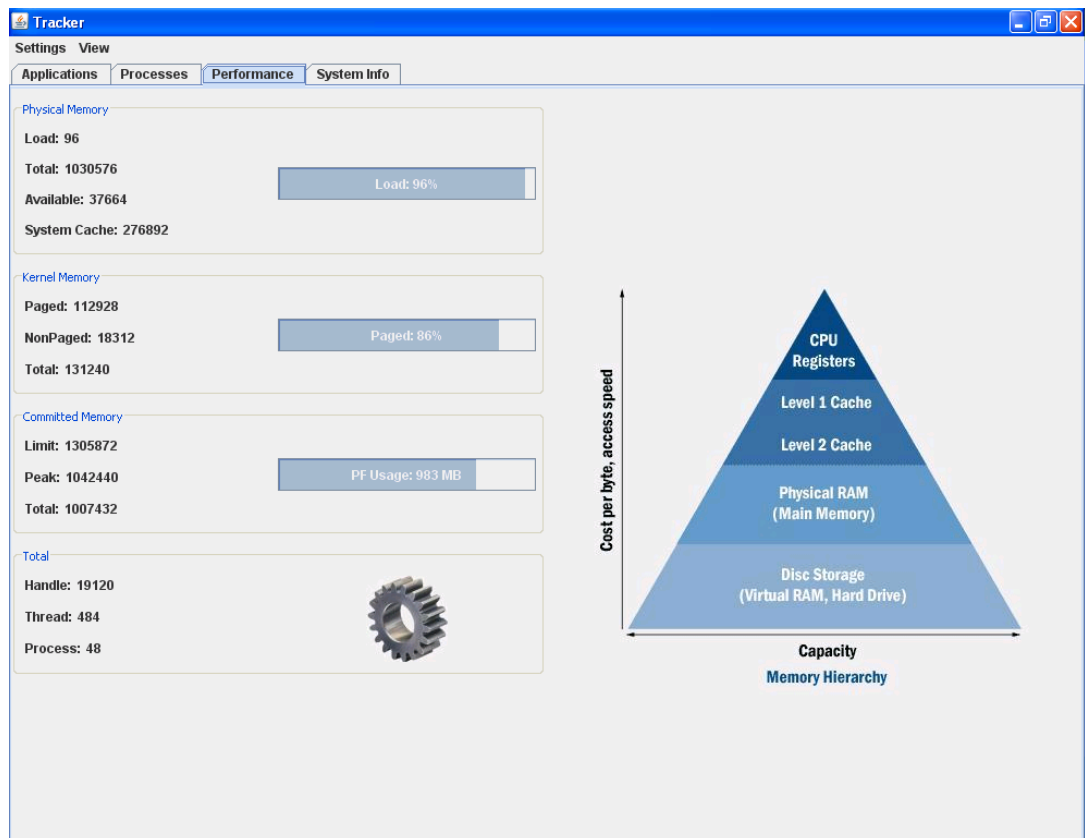


Figure 3-17 Performance Tab

The last tab “System Info” shows some general information about the system such as operating system information, system power status information and free disk spaces in each drive and network usage in the system (see Figure 3-18).

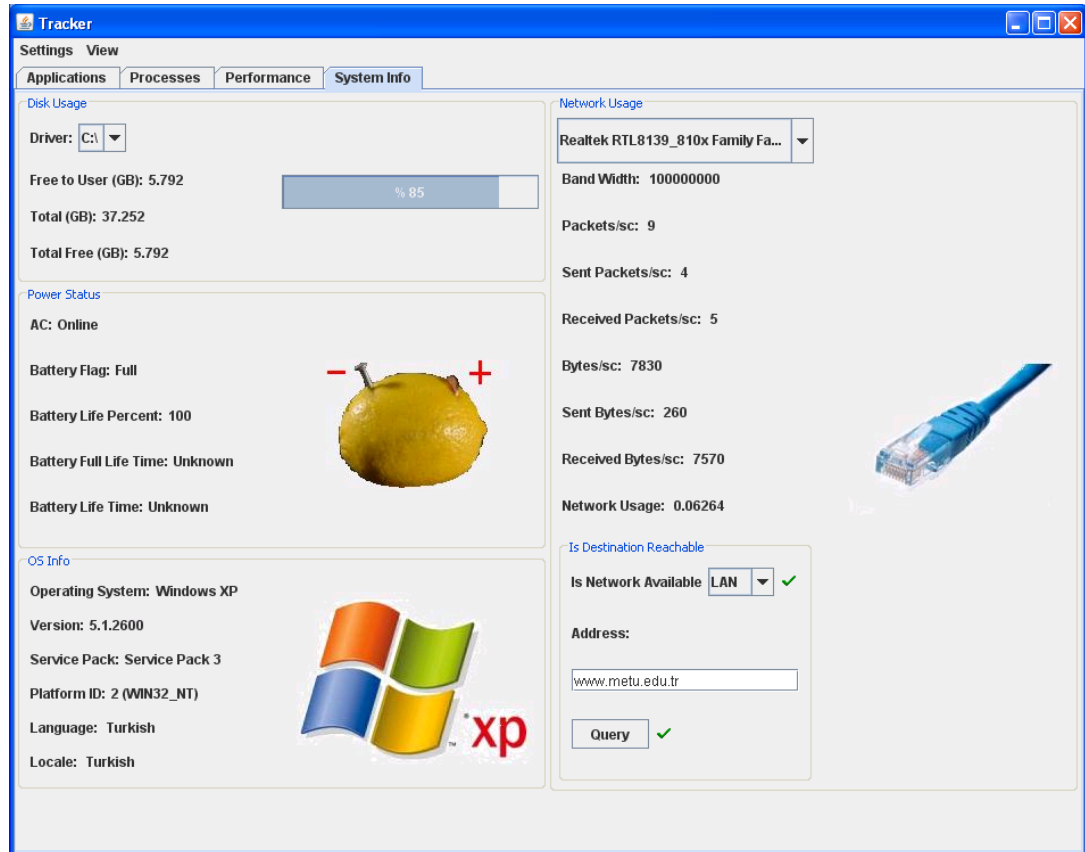


Figure 3-18 System Information Tab

All of the information in the test GUI is updated within some predefined time. These update times can be changed by using the configuration file.

3.2 MONITORING FRAMEWORK

The monitoring library includes the most useful symptoms that can be tracked in Windows XP. However, there may be some that the library does not cover or projects may want to write their own monitors. To control and coordinate all of these monitors, a monitoring framework is designed and implemented.

3.2.1 Monitor – Detector Interface

In fault management, a fault detector communicates with fault monitors for the symptoms that are being tracked in the system. Since there may be lots of monitors in the system, an interface must be defined that must be implemented by all monitors. The monitor – detector interface used in this thesis is as follows:

```
void startMonitoring();  
void stopMonitoring();  
Vector<SymptomInterface> getMonitoredSymptoms();  
boolean setThreshold(SymptomInterface symp, Object min, Object max);  
boolean assignPeriod(SymptomInterface symp, int time);  
boolean assignPeriod (int time);  
SymptomInterface monitorSymptom(SymptomInterface symp);  
Vector< SymptomInterface > monitorAllSymptoms();  
void attachObserver(MonitorObserver observer);  
void detachObserver(MonitorObserver observer);
```

According to the monitor-detector interface, the detector can start and stop the monitoring process. The detector firstly gets all the symptoms that monitor tracks. Then it can set threshold values or assign a period to each symptom or the general monitoring process. The detector may want the result of a specific symptom or the

result of all symptoms. Monitor tracks each symptom in each period time interval. If the tracked value is within threshold values, it sends the result to the log and its observers. If a detector wants to get these periodic results, it must be attached to the monitor and implement the “MonitorObserver” interface below (Figure 3-19):

```
void sendMonitoredSymptom(SymptomInterface symp);  
void sendMonitoredSymptoms(Vector< SymptomInterface > sympList);
```

The “SymptomInterface” that all symptom classes should implement is as follows:

```
int getSymptomType();  
Object getSymptomValue();  
Class<?> getSymptomValueType();  
Source getSource();  
void setThresholdValue(Object min, Object max);
```

According to the interface, all symptoms must have a type which shows what it is tracking. Each symptom has a value and value type since they may return different types of values such as “String”, “Integer”, etc. While source of the symptom shows the place of the tracking point, threshold values indicate where tracking value is worth-attending.

3.2.2 Monitoring Framework

The monitor-detector interface presented in Section 3.2.1 presents a single-monitor case. However, what about if there are two or three? The detector may connect to each separately and start monitoring all of them. In the beginning, this solution seems fine. However, if the detector is designed to detect the faults in a system

which consists of multiple subsystems and each subsystem has more than one monitor inside, then things are getting complicated [29].

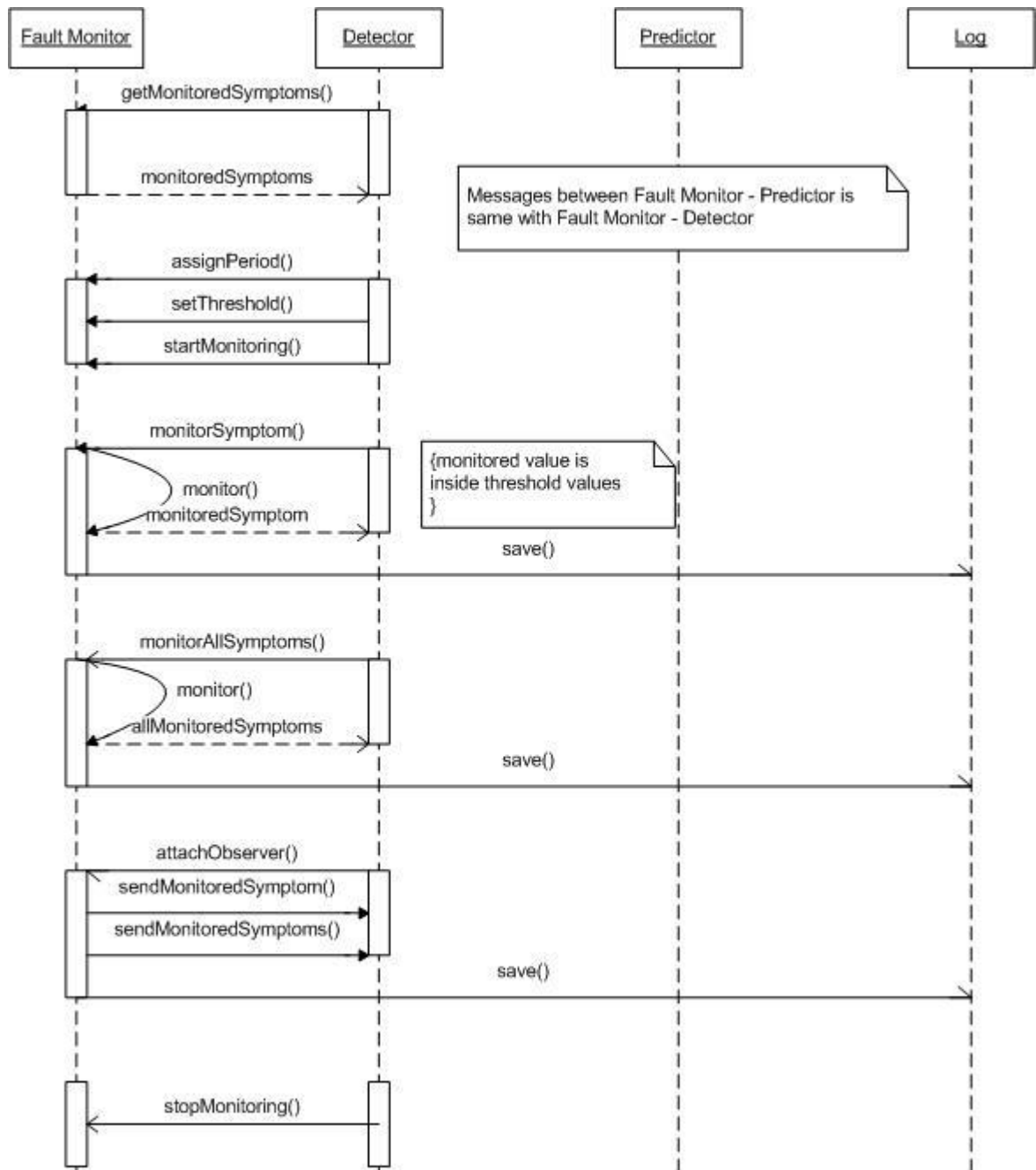


Figure 3-19 Monitor – Detector Interface Sequence Diagram

As it is frequently mentioned, the monitoring library may not be covering some symptoms that may be useful for some projects. Hence, each project may decide to use its own monitor besides the main one. To coordinate these monitors, a monitoring framework is designed and implemented (Figure 3-20).

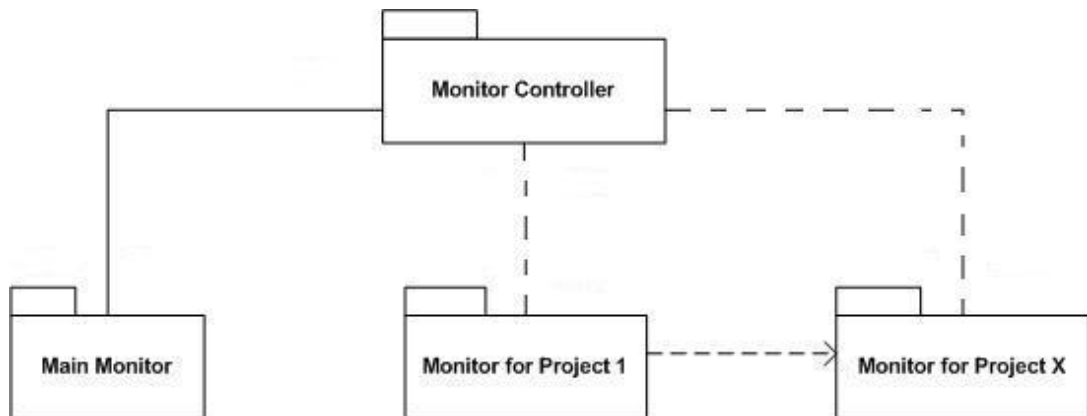


Figure 3-20 Monitoring Framework

As it is seen in Figure 3-20, the monitor controller does the main job in the framework. The primary job of it is making it easy to add new monitors to the framework and as its name implies controlling all of them.

Monitor controller implements the monitor-detector interface and behaves like a monitor that consists of sub-monitors. The messages coming from the detector or the predictor are interpreted here and the correct action is done. When the detector wants to get all the symptoms that are being monitored, controller gets all symptoms from its sub-monitors and sends the result to the detector.

Likewise, when the detector sends “Start Monitoring” message, controller sends this message to all its sub-monitors and monitoring starts in all of them. The controller knows which symptom is monitored by which monitor so that it directs the “Monitor Symptom” message to the correct monitor and gets the result. Each

monitor tracks its symptoms in every period and the results are sent to the observers. Monitor controller is one of the observers in each monitor and the results that come here are sent to the fault detector or the predictor (see Figure 3-22).

Monitoring framework uses Remote Method Invocation (RMI) to communicate with each monitor. RMI has two main advantages:

- Calling methods of the remote object that may be on the same computer or on another computer like a local function call (for distributed monitors).
- Dynamic code downloading that enables the receiver side to use the classes of the server side.

Each monitor in the framework implements the monitor interface that is extended from the “Remote” object (Figure 3-21). This interface is almost the same with the monitor-detector interface. The monitor puts the remote object that implements this interface to the RMI registry so that the monitor controller can call its functions like local functions. Likewise, the monitor controller creates a remote object and puts it to the RMI registry by which each monitor sends its monitored symptoms.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Vector;

public interface MonitorInterface extends Remote{

    void startMonitoring() throws RemoteException;
    void stopMonitoring() throws RemoteException;
    Vector<SymptomInterface> getMonitoredSymptoms() throws RemoteException;
    boolean setThreshold(SymptomInterface symp, int min, int max) throws RemoteException;
    boolean assignPeriod(SymptomInterface symp, int time) throws RemoteException;
    void assignPeriod(int time) throws RemoteException;
    SymptomInterface monitorSymptom(SymptomInterface belirti) throws RemoteException;
    Vector<SymptomInterface> monitorAllSymptoms() throws RemoteException;
    void sendMessage(String msg) throws RemoteException;
}
```

Figure 3-21 Monitor Interface Definition

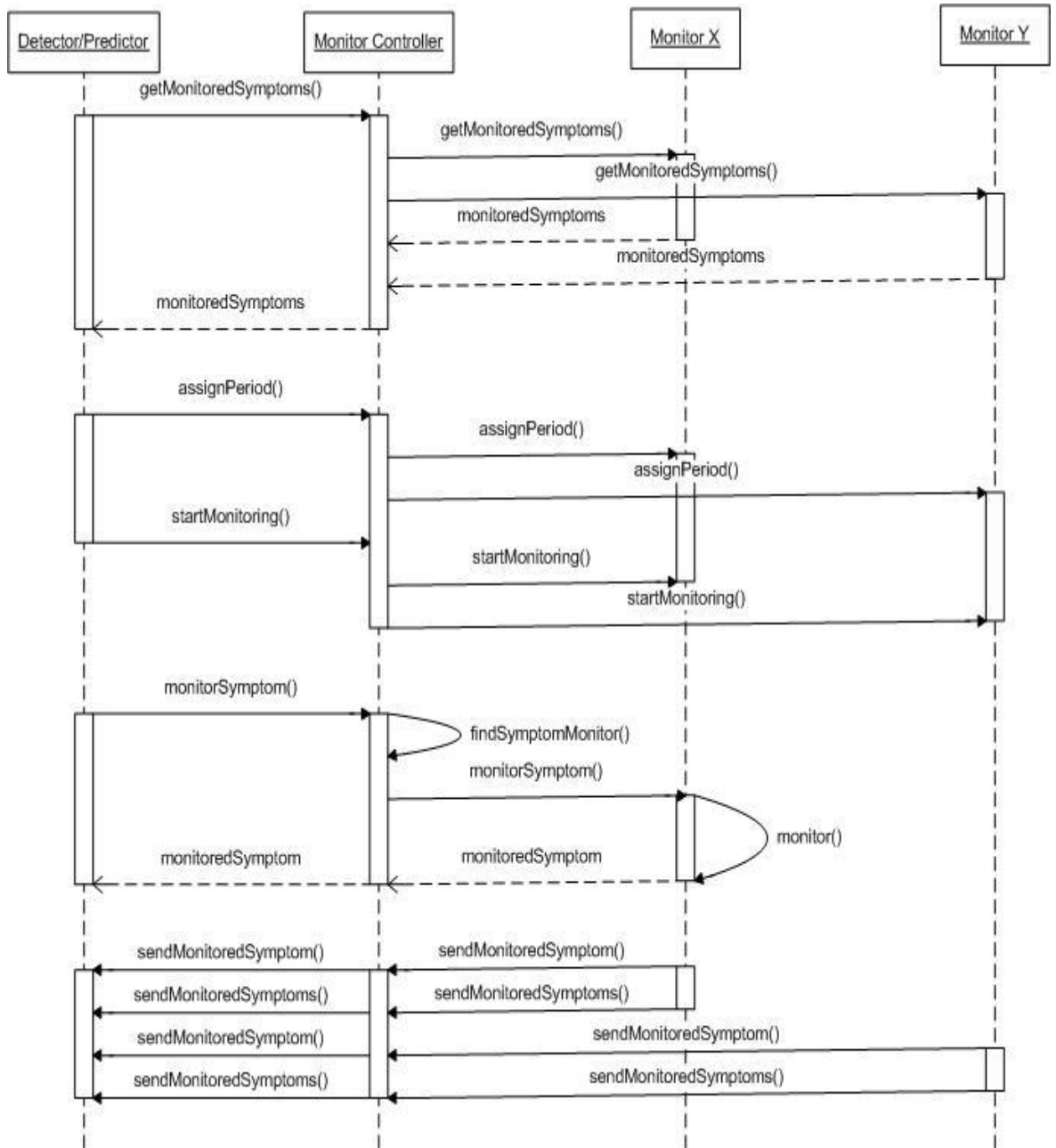


Figure 3-22 Sequence Diagram between Monitor Controller and Detector

CHAPTER 4

EVALUATION AND TEST RESULTS

Three monitors have been written and run on three different systems to test the monitoring framework. According to the test results, it is seen that the framework runs smoothly and supports working with multiple monitors that are distributed on different environments.

In this chapter, performance and platform tests that were conducted on the monitoring library are explained. The results show that the library's overhead to the system is acceptable and the overhead of the library does not change when the total load in the system increases.

4.1 PERFORMANCE TESTS

Performance tests done on the developed monitoring library will be introduced in this section. After presenting the testing environment, memory and CPU test results of the library will be presented and comparison of the results with similar programs in the market will be discussed.

4.1.1 Testing Environment

Intel Pentium personal computer which is containing a single 1.70 GHz processor, 1 GB RAM, running on Windows XP and Service Pack 3 is used for the tests which

are detailed below. Moreover, there were 53 processes in the system during these tests.

Test GUI that is explained in section 3.1.3 is used for monitoring library tests. The refresh time for the information in tabs of the GUI is like below:

- Applications tab: 2000 milliseconds.
- Processes tab: 2000 milliseconds.
- Performance tab: 1000 milliseconds.
- System Information tab: 1000 milliseconds.

All of the tabs refresh their information during the running time of the test GUI to see its performance under heavy load. However, similar programs in the market such as “Task Manager” and “Process Explorer”, rightly, do not follow this strategy to consume less system resources. They update the information only if it is seen by the user. Only the selected tab runs. Moreover, only the visible part of the table is updated during run time, for efficiency. For that reason, the comparison will be based on some estimation.

The refresh time for the “Process Explorer” tool is 2000 milliseconds and the refresh times for the “Task Manager” tool are:

- Applications tab: 2000 milliseconds.
- Processes tab: 2000 milliseconds.
- Performance tab: 1000 milliseconds.
- Network Communication tab: 1000 milliseconds.

The results in these tests are recorded by the “Performance Monitor” program of Windows XP except for the Heap Memory Usage tests which are recorded by a small program that uses the monitoring library.

4.1.2 Memory Tests

Working set size and heap memory usage is determined for the test GUI of the monitoring library and the “Process Explorer” tool. Memory usage depends on many parameters besides the program such as the middleware used (JVM) or figures and pictures in the program, etc. For that reason, comparison of the test GUI regarding memory usage with other tools does not give us much information. In Figures 4-1 to 4-11, memory consumption is in bytes and number of recorded samples is shown. Record interval is 1 seconds for each test.

While Figure 4-1 and 4-2 show the working set size, Figure 4-3 and 4-4 show the heap usages for the test GUI and the Process Explorer tool.

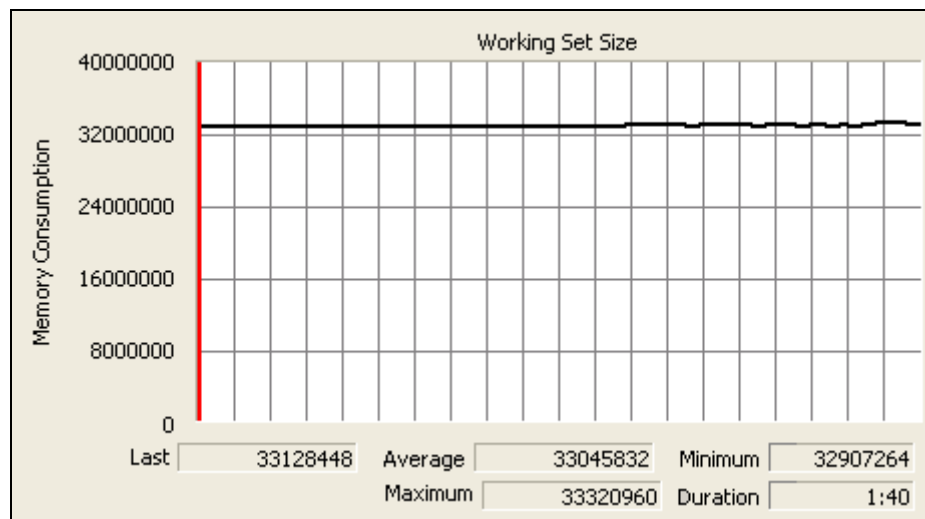


Figure 4-1 Working Set Size for Test GUI of Monitoring Library

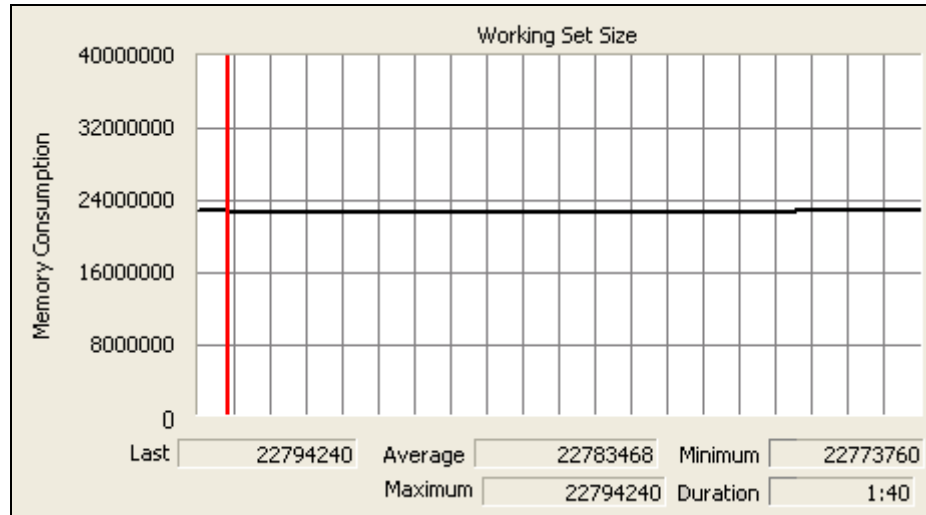


Figure 4-2 Working Set Size for Process Explorer Tool

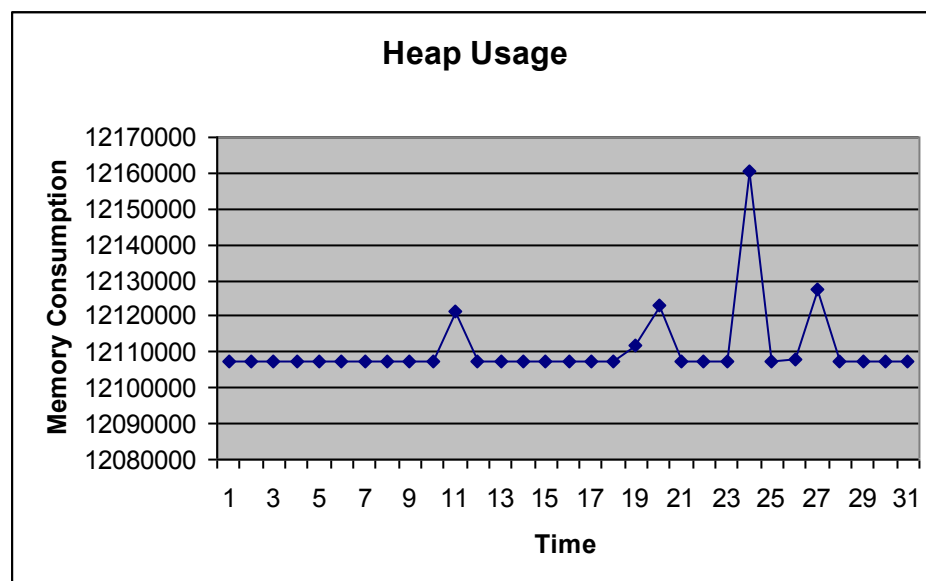


Figure 4-3 Heap Usage for Test GUI of Monitoring Library

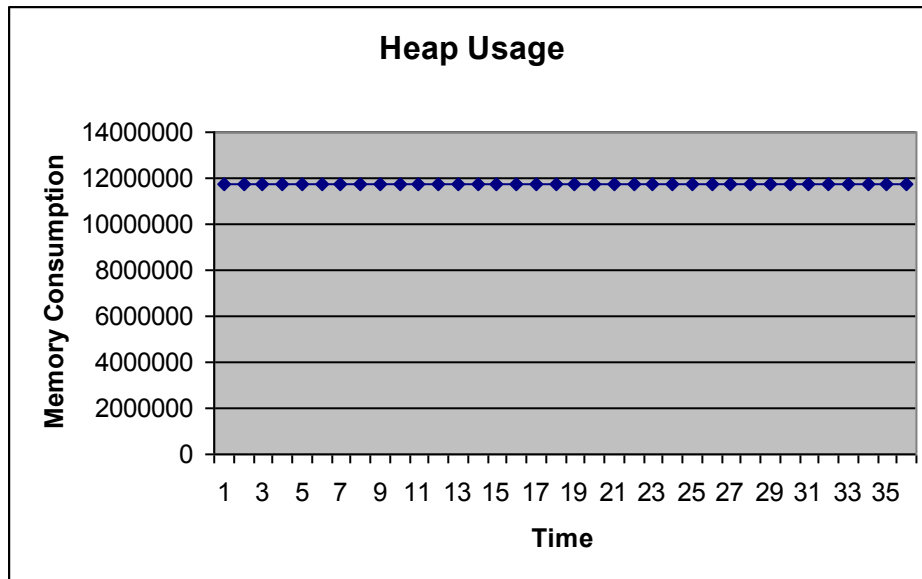


Figure 4-4 Heap Usage for Process Explorer Tool

The fluctuation in the test GUI is more than the other for both working set size and heap usage because of the garbage collector in Java. At the time of the decrease, the garbage collector releases unused allocated memory.

Comparison of memory usage does not guide us much in the case of small memory usage values. Figure 4-1 shows us that test GUI uses approximately 32 MB of memory which is appropriate.

4.1.3 CPU Tests

CPU usage of the test GUI, task manager and process explorer tool are tested and the results are shown in Figures 4-5 to 4-11.

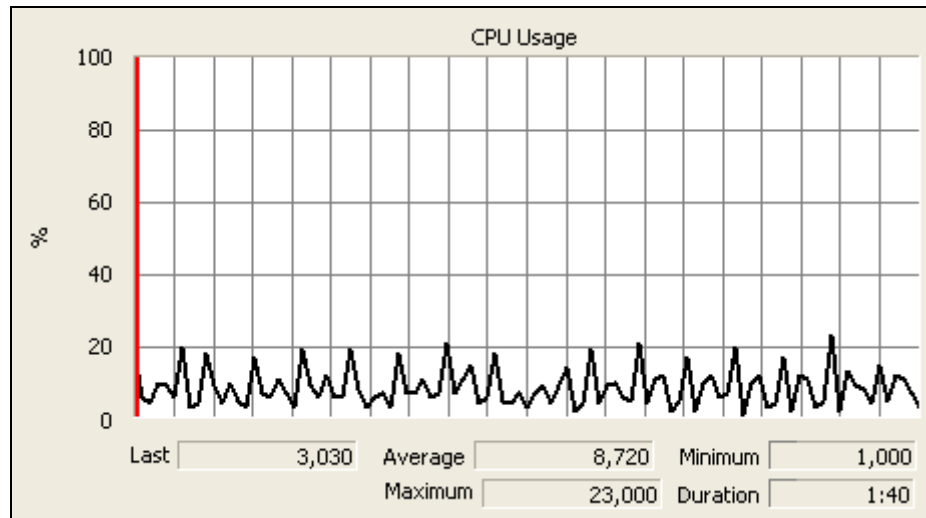


Figure 4-5 CPU Usage for Test GUI of Monitoring Library

As can be seen in Figure 4-5, CPU usage of the test GUI is 8.7 on the average. Figure 4-6 shows the behavior of the CPU usage of the test GUI under heavy load. According to the figure, CPU Usage of the test GUI (thicker curve) does not change when the load in the system increases.

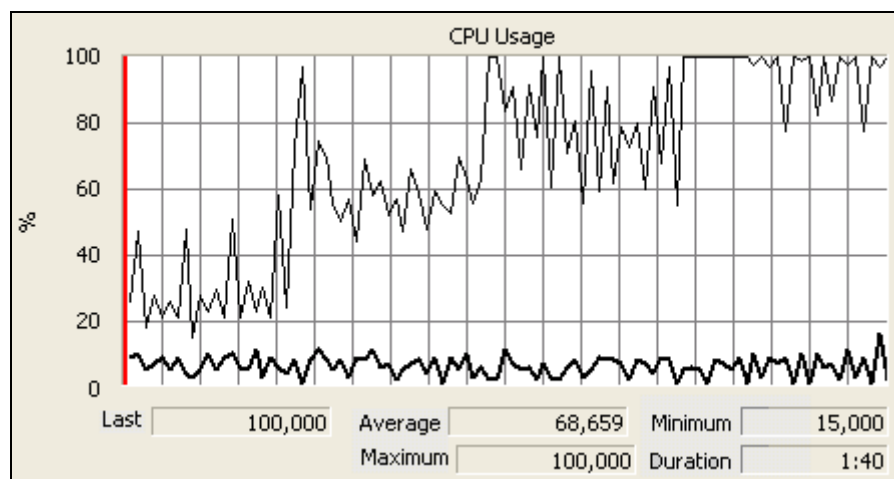


Figure 4-6 CPU Usage for Test GUI of Monitoring Library under Heavy Load

Since the task manager runs only one of its tabs actively, each tab's CPU usage is calculated separately (Figures 4-7, 4-8, 4-9 and 4-10).

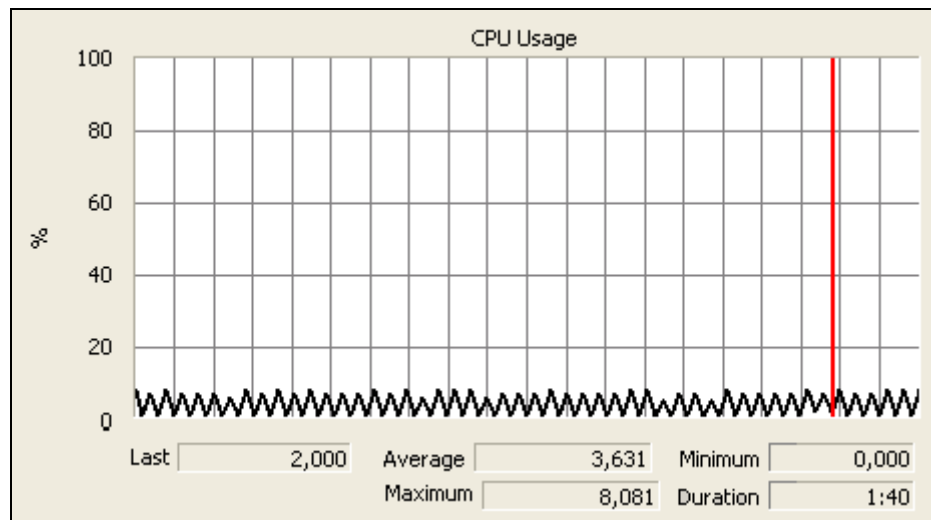


Figure 4-7 CPU Usage for Task Manager Processes Tab

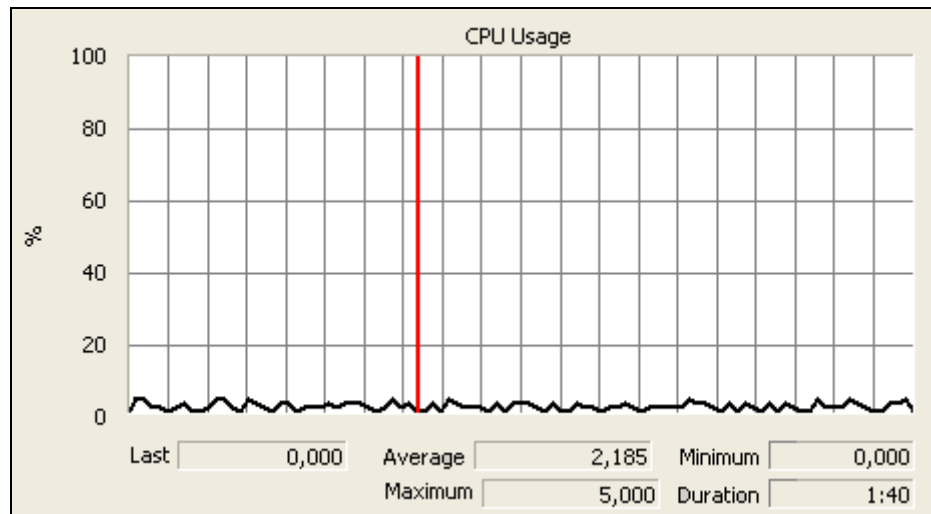


Figure 4-8 CPU Usage for Task Manager Performance Tab

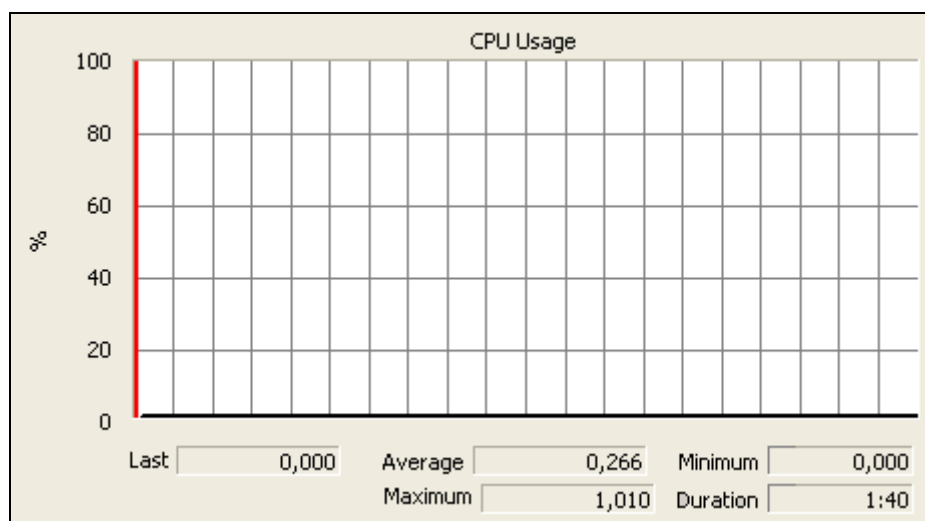


Figure 4-9 CPU Usage for Task Manager Applications Tab

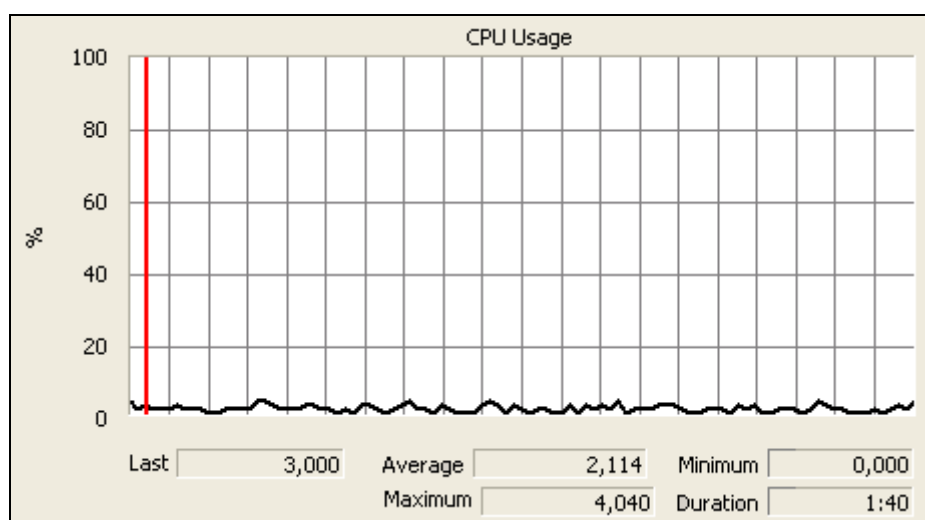


Figure 4-10 CPU Usage for Task Manager Network Connections Tab

According to Figures 4-7 to 4-10, processes tab uses 3.63%, performance tab uses 2.18%, applications tab uses 0.26% and network communication tab uses 2.11% CPU usage, on the average. Totally, task manager uses 8.18% CPU usage. When

compared with monitoring library, it is a bit less. However, monitoring library offers more information than task manager. Moreover it uses JNI, a middleware, to access system resources. So, the difference is acceptable.

Figure 4-11 shows the CPU usage of the Process Explorer tool which is 7.2% on the average. Although it shows only process related information, it consumes CPU a similar time as the test GUI does for the monitoring library.

Of course, performance test results shown above may change according to the system status at the specific time. However, these results, surely, indicate that monitoring library runs within appropriate values as compared to others.

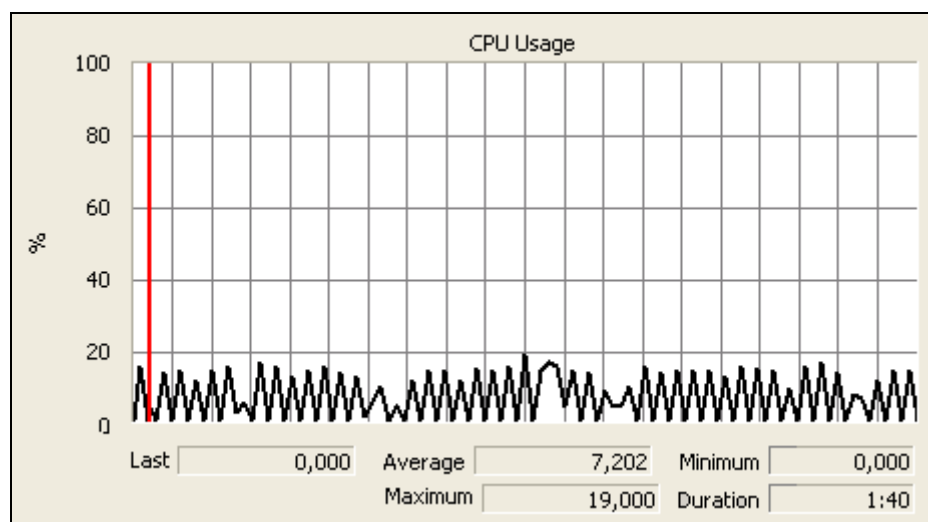


Figure 4-11 CPU Usage for Process Explorer

Table 4-1 summarizes the CPU Usage for all these programs.

Table 4-1 CPU Usage with respect to Common Applications

	Min	Max	Average
Test GUI of Monitoring Library	1	23	8.7
Task Manager	0	18	8.2
Process Explorer	0	19	7.2

4.2 PLATFORM TESTS

In Aselsan Inc, XP is used as the Windows operating system and currently there is no plan to use new versions of Windows. However, this is not the final decision. Therefore, the monitoring library is tested on Windows Vista, the current version of Microsoft's Windows operating system series.

Different from XP, Vista includes an updated graphical user interface, improved searching features, new multimedia creation tools, etc. However, the primary objective of Microsoft to release Vista is improving the state of security in the Windows operating system. One of the main criticism about XP and its predecessors is the security vulnerabilities to viruses, malware and buffer overflows [22].

The monitoring library is tested on Windows Vista by the main monitor that is designed for the monitoring framework. Each function in the library is tested one by one. All of the functions passed the test, except one. Getting the process name from Vista has some problems. The name comes in a corrupted way. However, this is not a big problem and it can be solved easily.

The main problem with Vista is its security settings. In Microsoft operating systems, although user can see all of the process IDs in the system, he/she can not see all of the information about some processes because of the privileges he/she has. Privileges determine the operation types that user is allowed. In XP, the monitoring library grants access to debug programs and see all of the process related information. However, in Vista, because of its advanced security mechanisms, the monitoring library could not get all process information such as name, full path, etc. for system processes like “svchost.exe”.

Other than two problems mentioned above which may be solved easily, the monitoring library runs also on Windows Vista. This is achieved mostly by Microsoft’s backward compatibility in system APIs.

CHAPTER 5

CONCLUSION

In this thesis, a fault monitoring library for Windows XP operating system is developed and a monitoring framework is designed and implemented for integration of multiple trackers that run in the system.

In the concept of this work, symptoms in Windows XP operating system are investigated and most useful ones for monitoring system resources are extracted. A dynamic-link library (DLL) is developed which includes all monitoring Application Programming Interfaces (API). The library is compiled in a way that is compatible with Java Native Interface (JNI) framework to be called in Java.

In order to test the monitoring library, a graphical user interface (GUI) is implemented in Java. It is seen that all symptoms are monitored correctly by the library. A series of performance tests are done both to the test GUI of the library and similar programs in the market. According to the test results, the test GUI is using a reasonable amount of memory and CPU as compared with other programs. The overhead of the library is acceptable and does not change when the load in the system increases. The platform test that was conducted on Windows Vista showed that nearly all of the API services in the library are also running on the current version of Microsoft's Windows operating system series.

Monitoring framework which is responsible for communication with fault detector or predictor and coordinating multiple trackers in the system is designed and

implemented. However, it is not yet integrated to the fault management framework in Aselsan Inc. To test this framework, three trackers are implemented:

- Main Tracker: monitors all system resources by using the monitoring library.
- Two Project Trackers: monitor project specific symptoms.

It is seen that these trackers are integrated to the framework easily and communication between these trackers and fault detector is established successfully. The trackers run on different systems to show the framework supports monitors on distributed environments. A tracker library is written in Java to make Java projects implement their trackers easily.

As future work, the following can be done:

- Monitoring library can be revised for other Windows platforms like Vista.
- Tracking framework can be supported to work with all distributed object technologies such as CORBA, sockets. By this, trackers written in other languages such as C can be easily added to the framework.
- The integration of the monitoring framework with the fault management project in Aselsan is not carried out and planned yet due to lack of time. This work can be done as a future work.
- New monitoring libraries can be written for other platforms like Solaris, Unix, etc.

REFERENCES

- [1] HA Forum, "Providing Open Architecture High Availability Solutions", Revision 1.0, February, 2001.
- [2] <http://www.kalinskyassociates.com/Wpaper6.html>, Last access date: February 2009.
- [3] GoAhead Software, FAQ: Fault Management How Does Fault Management Work?, 2000.
- [4] Kalinsky D, "Principles of High Availability Embedded Systems Design", Embedded World Conference, Nürnberg 2006.
- [5] Tekkalmaz M., Kaya Ö., Dursun M., Tekkalmaz T. S., Doğru A., "Görev Kritik ve Gömülü Sistemler için Hata Yönetimi Klavuz Mimarisi: T5D", 2. National Software Architecture Conference, İzmir, September 2008.
- [6] Wikipedia, http://en.wikipedia.org/wiki/Java_Native_Interface, Last access date: February 2009.
- [7] Sun Microsystems, The Java Tutorial, version 1.5
- [8] Sun Microsystems, <http://java.sun.com/docs/books/tutorial/rmi/TOC.html>, Last access date: February 2009.
- [9] GoAhead Software, High Availability: A Design Primer, 2004
- [10] <http://support.microsoft.com/kb/555223/en-us>, Last access date: February 2009.

- [11] [http://msdn.microsoft.com/en-us/library/cc441804\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/cc441804(VS.85).aspx), Last access date: February 2009.
- [12] <http://msdn.microsoft.com/en-us/library/cc267832.aspx>, Last access date: February 2009.
- [13] [http://msdn.microsoft.com/en-us/library/aa965226\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa965226(VS.85).aspx), Last access date: February 2009.
- [14] <http://developers.sun.com/solaris/articles/multiproc/multiproc.html>, Last access date: February 2009.
- [15] <http://msdn.microsoft.com/en-us/library/ms810627.aspx>, Last access date: February 2009.
- [16] <http://technet.microsoft.com/en-us/library/bb742598.aspx>, Last access date: February 2009.
- [17] [http://msdn.microsoft.com/en-us/library/ms684125\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms684125(VS.85).aspx), Last access date: February 2009.
- [18] [http://msdn.microsoft.com/en-us/library/ms683223\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms683223(VS.85).aspx), Last access date: February 2009.
- [19] <http://www.codinghorror.com/blog/archives/001029.html>, Last access date: February 2009.
- [20] http://en.wikibooks.org/wiki/Windows_Programming/User_vs_Kernel, Last access date: February 2009.
- [21] [http://msdn.microsoft.com/en-us/library/aa372693\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa372693(VS.85).aspx), Last access date: February 2009.

- [22] http://en.wikipedia.org/wiki/Windows_Vista, Last access date: February 2009.
- [23] <http://www.mozilla.com/en-US/firefox/personal.html>, Last access date: February 2009.
- [24] Laura L. Pullum, Software fault tolerance techniques and implementation, Artech House, Inc., Norwood, MA, 2001.
- [25] Martin L. Shooman, Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design, John Wiley & Sons, Inc., New York, NY, 2002 .
- [26] A Middleware for Dependable Distributed Real-Time systems: Tom Bracewell, Priya Narasimhan, Raytheon, CMU, 2003.
- [27] Designing Systems for High Availability, Shiv Nagarajan (QNX Software systems), Embedded System Conference, San Jose, April 2006.
- [28] Kuz K., Kaya Ö., Alkışlar L., Tekkalmaz M., Doğru A. H., “A DSK Based Reference Architecture for Fault Management in Mission Critical Systems”, Submitted to SDRS conference, Niagara Falls, New York, September 27-30, 2009.
- [29] Pohl K., Böckle G., ve v. d. Linden F., “Software Product Line Engineering: Foundations, Principles, and Techniques”, Springer, Berlin Heidelberg New York, 2005.