DEVELOPMENT OF A MODIFIED REAL-TIME FAULT-TOLERANT TASK ALLOCATION SCHEME FOR WIRELESS SENSOR NETWORKS

By

MARSHALL Francis Franklin

Department of Computer Engineering

Faculty of Engineering

Ahmadu Bello University Zaria, Nigeria

July, 2018

DEVELOPMENT OF A MODIFIED REAL-TIME FAULT-TOLERANT TASK ALLOCATION SCHEME FOR WIRELESS SENSOR NETWORKS

By

MARSHALL Francis Franklin. BSc (2010) ABU, PGDCP (2015) ABU

P15EGCP8001

frankdidam14@yahoo.com

SUPERVISORY COMMITTEE:

Prof. M. B. Mu'azu	**Chairman**
Dr. I. J. Umoh	**Member**

A Dissertation Submitted to the Department of Computer Engineering, Ahmadu Bello University Zaria, in Partial Fulfilment of the Requirements for the Award of Master of Science (M.Sc.) Degree in Computer Engineering.

July, 2018

DECLARATION

I MARSHALL Francis Franklin, hereby declare that the work in this dissertation entitled "Development of a Modified Real-Time Fault-Tolerant Task Allocation Scheme for Wireless Sensor Networks" has been carried out by me in the Department of Computer Engineering. The information obtained from literature has been duly acknowledged in the text and a list of references presented. No part of this dissertation was previously presented for another degree or diploma at this or any other institution.

MARSHALL Francis Franklin	…………………………..	……………………..
	Signature	Date

CERTIFICATION

This Dissertation titled DEVELOPMENT OF A MODIFIED REAL-TIME FAULT-TOLERANT TASK ALLOCATION SCHEME FOR WIRELESS SENSOR NETWORKS by MARSHALL Francis Franklin meets the regulations governing the award of the degree of Master of Science (MSc) in Computer Engineering of the Ahmadu Bello University, and is approved for its contribution to knowledge and literary presentation.

Prof. M. B. Mu'azu ………………………….. …………………
Chairman, Supervisory Committee Signature Date

Dr. I. J. Umoh ………………………….. …………………
Member, Supervisory Committee Signature Date

Prof. M. B. Mu'azu …………………………… …………………
Head of Department Signature Date

Prof. S. Z. Abubakar ………………………….. ………………..
Dean. School of Postgraduate Studies Signature Date

DEDICATION

This dissertation is dedicated to God Almighty, my family, friends and the special memories of my late mother and late grandmother.

ACKNOWLEDGEMENT

In the name of God, the Father, God the Son and God the Holy Spirit, Amen. The most gracious and merciful God, all glory and praises are unto the King of glory and Lord of lords, the Alpha and the Omega, the beginning and the end, the creator of heaven and the earth, the Master of all universes. In spite of several efforts put into this dissertation, it wouldn't have been possible without the support and guidance of the Almighty God, the spring of all wisdom, knowledge and understanding. I am most grateful to you, oh Lord my Saviour.

I wish to express my deepest gratitude to my supervisor and role model Prof. M. B. Mu'azu, for his tireless efforts, valuable guidance and constant supervision towards the success of this dissertation. The completion of this work could not have been possible without your constant participation and assistance. Thank you very much Prof. My thanks also go to my co-supervisor Dr. I. J. Umoh for his encouragement and expertise. My deep appreciation goes to the Computer & Control Research Group for their keen interest, suggestions and constructive criticisms during all those stages of my presentations.

I acknowledge with thanks the lecturers of Computer Engineering, Ahmadu Bello University, namely: Dr. E. A. Adedokun, Dr. A. Sha'aban, Dr. Abdulrazak, Dr. I. A. Bello, Dr. Bashir Sadiq, Dr. Salawudeen A. Tijjani, Engr. Yusuf, Engr. Shehu and others whose names are not listed.

My appreciations to my aunt, Pharm. L Balat, my Daddy thank you all for both your moral and financial support.

MARSHALL Francis Franklin
July 2018

ABSTRACT

This research aimed at the development of a modified real-time fault-tolerant task allocation scheme (mRFTAS) for wireless sensor networks (WSNs), using active replication backup techniques. In WSNs, the sensor nodes are at risk of failure and malicious attacks (selective forwarding) and this can have a profoundly negative effect on real-time WSNs. The real-time fault-tolerant task allocation scheme (RFTAS) was developed to address these issues, however, it has the problem of processing time delay. This is attributed to the characteristic of the passive backup copy technique adopted for the RFTAS in which the backup copy is only activated when the primary copy has failed. The delay in the activation of the backup copies of the primary tasks in tasks allocation execution processes as a result of a failure of sensor nodes or the primary tasks, will cause disastrous consequences if the systems are safety-critical, e.g. aircraft, nuclear power plant, forest fire detection, battlefield monitoring. The mRFTAS was therefore developed using the active replication backup technique where both the primary and backup copies of tasks are executed concurrently. The performance of the mRFTAS and RFTAS was compared using total execution time of the task, energy consumption, reliability cost and network lifetime. A graphical user interface (GUI) based system for simulation of sensor nodes in WSNs using RFTAS and mRFTAS called the task allocation scheme simulator (TASS) was developed in order to carry out the performance evaluation. The performance of mRFTAS showed an improvement over RFTAS in terms of minimizing task execution time (28.65%) and reliability cost (7.29%) while maximizing network lifetime (22.26%) but with a trade-off in energy consumption (-17.32%).

TABLE OF CONTENTS

TITLE PAGE **I**

DECLARATION **III**

CERTIFICATION **IV**

DEDICATION **V**

ACKNOWLEDGEMENT **VI**

ABSTRACT **VII**

TABLE OF CONTENTS **VIII**

LIST OF FIGURES **XII**

LIST OF TABLES **XIII**

LIST OF ABBREVIATIONS **XIV**

CHAPTER ONE : INTRODUCTION

1.1 Background 1

1.2 Significance of Research 3

1.3 Statement of Problem 4

1.4 Aim and Objectives 4

1.5 Methodology 5

1.6 Dissertation Outline 6

CHAPTER TWO : LITERATURE REVIEW

2.1 Introduction 7

2.2 Review of Fundamental Concepts 7

2.2.1 Wireless sensor networks (WSNs) 7

2.2.2 Task allocation in WSNs 9

2.2.3 Fault-tolerance in wireless sensor networks 11

2.2.4 Real-time WSNs system 13

2.2.5 The real-time fault tolerant task allocation schemes 14

2.2.6 Performance metrics. 16

2.3 Review of Similar Works 19

CHAPTER THREE : MATERIALS AND METHODS

3.1 Introduction 27

3.2 Materials 27

3.3 Analyses of the Real-time Fault-tolerant Task Allocation Scheme 27

3.3.1 Task allocation mechanism based on dPSO 27

3.3.2 The encoding of particle 30

3.3.3 Fitness function 31

3.3.4 dPSO-Based task allocation algorithm description 31

3.3.5 Calculation process of start execution time of the task's primary and backup copies 33

3.3.6 Calculation process of the earliest start time of the task's primary copy 33

3.3.7 Calculation process of the latest start time of the task's backup copy 33

3.3.8 Allocation process of the task's primary copy 34

3.3.9 Allocation process of the task's backup copy 36

3.4 The Modification of the Real-time Fault-tolerant Task Allocation Scheme 38

3.4.1 Allocation process of the task's backup copy 38

3.4.2 The flowchart of the modified real-time fault tolerant task allocation scheme 40

3.5 Development of the GUI for the Wireless Sensor Network Simulations 41

3.5.1 Wireless sensor networks simulator 41

3.5.2 Programming 44

3.5.3 Wireless sensor network simulation 44

3.6 Comparisons of the Performance of mRFTAS AND RFTAS 45

CHAPTER FOUR : RESULTS AND DISCUSSIONS

4.1 Introduction 46

4.2 Task Allocation Scheme Simulator 46

4.3 Result of the Analyses of the Real-time Fault-tolerant Task Allocation Scheme 47

4.4 Result of the Analyses of the Modified Real-time Fault-tolerant Task Allocation Scheme50

4.6 Result Performance Percentage Improvement of the mRFTAS over RFTAS 54

CHAPTER FIVE CONCLUSION AND RECOMMENDATIONS

5.1 Summary 57

5.2 Conclusion 57

5.3 Significant Contributions 58

5.4 Recommendations for Further Work 58

REFERENCE 59

APPENDIX 62

Appendix A: WSNs Simulator/GUI code 62

Appendix B: Codes for graphs of RFTAS 89

Appendix C: Codes for graphs of mRFTAS 92

Appendix D: Codes for graphs of both mRFTAS and RFTAS 95

Appendix E: simulation results table for 10 runs of sensor nodes energy consumption simulations 98

Appendix F: simulation results table for 10 runs of sensor nodes task execution time simulations 100

Appendix G: simulation results for 10 runs of sensor nodes reliability cost simulations 102

Appendix H: simulation results for 10 runs of sensor nodes network lifetime simulations 104

LIST OF FIGURES

Fig. 2.1: Wireless Sensor Networks with Allocated Tasks 12

Fig. 2.2: Flowchart of RFTAS 18

Fig. 3.1: Pseudo code of the DPSO algorithms base on task assignment scheme 32

Fig. 3.2: Flow Chart of mRFTAS 40

Fig 3.3: Code for deployment of nodes 42

Fig. 4.1: Picture of nodes deployment of mRFTAS and RFTAS 46

Fig. 4.2: Picture of the dual simulation of mRFTAS and RFTAS 47

Fig. 4.3: Plot of Energy consumption for RFTAS 48

Fig. 4.4: Plot of Task Execution Time for RFTAS 48

Fig. 4.5: Plot of Reliability cost for RFTAS 49

Fig. 4.6: Plot of Network lifetime for RFTAS 49

Fig. 4.7: Plot of Energy consumption for mRFTAS 50

Fig. 4.8: Plot of Task Execution Time for mRFTAS 50

Fig. 4.9: Plot of Reliability cost for mRFTAS 51

Fig. 4.10: Plot of Network lifetime for mRFTAS 51

Fig. 4.11: Plot Energy Consumption for mRFTAS and RFTAS 52

Fig. 4.12: Plot of Task Execution Time for both mRFTAS and RFTAS 53

Fig. 4.13: Plot of Reliability Cost for mRFTAS and RFTAS 53

Fig. 4.14: Plot of Network lifetime for both mRFTAS and RFTAS 54

LIST OF TABLES

Table 1.1: Fault-tolerant innovation in different layers of WSNs abstraction 2

Table 4.1 Performance Comparison for Energy Consumption 55

Table 4.2 Performance Comparison for Task Execution Time 55

Table 4.3 Performance Comparison for Reliability Cost 56

Table 4.4 Performance Comparison for Network Lifetime 56

LIST OF ABBREVIATIONS

Acronyms	Definition
bPSO	Binary particles swarm optimization
CBS	Condition-based support
CR	Coverage rate
DOOTA	Distributed optimal on-line task allocation
dPSO	Discrete particle swarm optimization
dPSO-DA	Dynamic alliance discrete particles swarm optimization
DTAS	Dynamic task allocation scheme
DYFARS	Dynamic and reliability-driven real-time fault-tolerant scheduling algorithms
EMM	Extended Min-Min
FTAS	Fault-tolerant task allocation scheme
GA	Genetic Algorithms
GUI	Graphical User Interface
IoT	Internet of Things
LANs	Local area networks
MCT	Minimum Complete Time
MOP	Multi-objective optimization problem
mRFTAS	Modified real-time fault-tolerant task allocation schemes
NOQAFT	Non-overlapping QoS-aware fault-tolerant scheduling Algorithms
NS2	Networks simulator 2
OS	Operating system
P/B	Primary/Backup
PC	Personal computer
PSO	Particle swarm optimization
QAFT	Quality of Service Aware fault-tolerant
QoS	Quality of Service
RAM	Random access memory
RC	Reliability Cost
RFTAS	Real-time fault-tolerant task allocation schemes
SR	Sleeping rate
TAA-GT	Dynamic task allocation scheme base on game theory
TAA-PAGA	Parallel alliance with greedy algorithm-based dynamic task allocation scheme
TAA-PARA	Parallel alliance with random algorithm-based dynamic task allocation scheme
TAGA	Fault-tolerant on genetic algorithm task allocation
TASS	Task allocation scheme simulator
TASSIM	Task allocation scheme based on score incentive method
TCL	Tool Command Language
VLSI	Very-large-scale integration
WSNs	Wireless sensor networks

CHAPTER ONE

INTRODUCTION

1.1 Background to the Study

Wireless sensor networks (WSNs) comprise of a number of sensor nodes, which are mostly used in obtaining important information in some target areas (Chen *et al*., 2012). WSNs have multiple areas of applications, such as battlefield (Gangadharaiah *et al*, 2014), military intelligence sensing and tracking, environmental tracking (Mei *et al*., 2010), emergency response and disaster management (Guo *et al*, 2014), bio-complexity mapping of the environment, flood detection, precision agriculture, medical telemonitoring, chemical and structural monitoring (Shi *et al*., 2012). WSNs are also invaluable in fields such as the Internet of Things (IoT), cyber-physical systems, intelligent vehicle systems and smart cities (Zhang & Long, 2017). WSNs are known to have a major constraint, which is the low power consumption requirement of sensor nodes (Guo *et al*, 2015b). Parallel processing between sensor nodes is an innovative technology which supports the essential computation capacity in WSNs (Guo *et al*, 2014, 2015b*)* while ensuring low power consumption by the sensor nodes.

Task allocation plays a significant role in parallel processing. Assigning a task to the suitable sensor nodes and simultaneously balancing the network load in context of the uncertain and dynamic network environments are essential to parallel processing (Guo *et al*, 2014). However, studies carried out on the problem of task allocation in distributed systems indicate that the drawbacks encountered in task allocation in WSNs are different when compared with those of conventional distributed systems (Guo *et al*, 2014, 2015b). In WSNs, the issue of task allocation involves assigning tasks logically within sensor nodes so as to reduce the general power utilization while still guaranteeing that the tasks are completed before the set deadlines, thus prolonging the lifetime of the sensor network (Guo *et al*, 2011, 2015b). Load balancing is an essential factor for prolonging the network lifetime (Suganya & Jayanthi, 2016); Guo *et al*, 2015b). In the absence of proper task allocation techniques, every sensor node will work

independent of others (Guo *et al*, 2015b). WSNs have challenges such as instability of wireless communication links and dynamically changing topologies. As a result of which, there are potentially additional uncertainties and vulnerabilities for real-time applications (Suganya & Jayanthi, 2016; Guo *et al*, 2015b). A sensor node failure should not necessarily affect the entire tasks processing of the sensor network especially for safety and security critical applications. In order to sustain the performance of the sensor networks without interruption due to failures of the sensor node, fault tolerance turns out to be an invaluable initiative (Suganya & Jayanthi, 2016; Guo *et al*, 2015b). For instance, if sensor nodes are being deployed in a battlefield or military-camp for surveillance and detection, the fault tolerance has to be high because the sensed data are critical for security and safety reasons (Priyanka *et al*, 2016; Guo *et al*, 2015b). Hence, a fault-tolerant mechanism is mandatory for such safety and time-critical applications (Guo *et al*, 2015b; Zhu *et al*, 2011). Fault-tolerant innovation in different layers of WSNs abstraction is summarized in Table 1.1.

Table 1.1: Fault-Tolerant Innovation in Different Layers of WSNs Abstraction (Guo *et al*, 2015b)

Abstraction Structure	Goals	Methods used
Application layer	Obtaining proper reliability by reducing the data redundancy	backup/primary copy or data aggregation
Transport layer	improving monitoring quality by the discovery of faulty nodes	Fault detection and isolation
Network layer	providing better resilience and controlling traffic congestion	Fault-tolerant routing
Data link layer	enhancing the coverage level with a robust link	Fault-tolerant coverage and topology control
Physical layer	building reliability by exploring the natural information redundancy	Hardware redundancy, multiple sensors exploration

The issue of task allocation is handled at the application layer as can be seen in Table 1.1, with focus on reducing data redundancy while ensuring enhanced reliability (Guo *et al*, 2015b).

(Priyanka *et al.*, 2016) used information aggregation as a mean of reducing data redundancy and securing precise data. The replication technique which involves the use of primary/backup (P/B) system is the most widely used technique for fault-tolerant task allocation as it tolerates copies of a task to be processed on separate sensor nodes (Guo *et al*, 2015b). There are two types of the backup scheme namely, passive and active (Priyanka *et al*, 2016). The active is the process in which both the primary and the backup copies are executed concurrently while the process in which the backup copies are activated only after there is an incorrect result obtained from the primary copies is known as the passive (Guo *et al*, 2015b).

The real-time fault-tolerant task allocation scheme is the kind of scheme that is used to prevent system failure or system breakdown and has mostly employed the passive replication backup technology. However, the passive replication scheme has the problem of delay in task processing time (Han, 2015). Delay in task processing time can be disastrous to real-time systems that are critical in term of safety.

1.2 Significance of Research

Task allocation and scheduling are essential for WSNs in order to maximize the lifetime of the networks by either reducing the energy consumption or the processing time. Recently, task allocation researches are being carried out with the view of providing systems that are fault tolerant with emphasis on energy minimization and less attention is given to the total time of tasks execution.

There is need for a real-time WSN that is fault-tolerant and safety critical, thus the need for the development of a modified real-time fault-tolerant task allocation scheme, such as to avoid the breakdown of a system as a result of the failure of some sensor nodes. The modified real-time fault tolerant task allocation scheme will help a system to keep operating even in the presence

of some failures, inevitably reducing the total time of an entire task allocation process, thus prolonging the system lifetime.

1.3 Statement of Problem

A WSN comprises of numerous resource-constrained sensor nodes, which are frequently deployed in unattended environments. Consequently, the sensor nodes are at risk of failure and/or malicious attacks and failed nodes can have a profoundly negative effect on real-time WSNs. Therefore, a key issue to be considered in real-time systems is the time delay in task processing (which can arise from issues associated with nodes failures). Accordingly, it is crucial that network failure is identified early and properly taken care of in order to ensure network reliability and lifetime.

The Real-time Fault-tolerant Task Allocation Scheme (RFTAS) was designed to prevent system breakdown due to node failures/fault, however, it had the issue of delay in task processing time. The developed modified Real-time Fault-tolerant Task Allocation Scheme (mRFTAS) is to address the issue of delay in task processing time associated with the RFTAS.

1.4 Aim and Objectives

The aim of the research is the development of a modified real-time fault-tolerant task allocation scheme (mRFTAS) for WSNs.

The following are the objectives of this research work:

1. Development and implementation of the mRFTAS using active replication backup techniques.
2. Development of a graphical user interface (GUI) for simulation of sensor nodes in WSNs using real-time fault-tolerant task allocation scheme (RFTAS) and mRFTAS respectively called the task allocation scheme simulator (TASS).

3. Comparison of the performance of mRFTAS and RFTAS using task execution time, energy consumption, reliability cost and network lifetime as the performance metrics.

1.5 Methodology

The steps of the methodology adopted for this work are as follows:

1. Development of modified real-time fault-tolerant task allocation scheme (mRFTAS).
   i. Adoption of Real-time fault-tolerant task allocation scheme by (Guo *et al,* 2015).
   ii. Remove the laxity time included in the real-time fault-tolerant task allocation scheme (RFTAS).
2. Development of a GUI (graphic user interface).
   i. Download of Visual Studio 2015 Professional.
   ii. Instillation of Visual Studio 2015 in windows 10.
   iii. Setting the Network Configuration
      a. Set the size of the network
      b. Set the radius of the sensor
      c. Set the sensor time
      d. Set the sensor cost
      e. Set the communication range
      f. Set the transmitter time
      g. Set the transmitting cost
      h. Set the received cost
3. Implementing the real-time fault-tolerant task allocation scheme in wireless sensor networks using C# in the developed graphical users interface (GUI).
4. Implementing the modified real-time fault-tolerant task allocation scheme in wireless sensor networks using C# in the developed graphical users interface (GUI).

5. Simulation of the sensor nodes with *RFTAS* and *mRFTAS.*
6. Comparison of the performance of *RFTAS* and *mRFTAS* using task execution time, energy consumption, reliability cost and network lifetime as the performance metrics.

1.6 Dissertation Outline

The outline of this dissertation report is as follows, Chapter One entails the general background of the research work; Chapter Two presents the review of the fundamental concepts essential to the research work and also detailed review of similar works; Chapter Three comprises of detailed explanation of materials and methods used for the research; Chapter Four presented results and discussions; Chapter Five concludes dissertation and provides a number of recommendations for future work.

CHAPTER TWO
LITERATURE REVIEW

2.1 Introduction

The chapter encompasses the review of essential concepts fundamental to the research and a review of similar and related works in the area of fault-tolerant task allocation scheme for WSNs.

2.2 Review of Fundamental Concepts

Fundamental concepts such as wireless sensor networks (WSNs), fault tolerance and task allocation schemes, are discussed in this sub section in addition to the basics of the performance metrics considered in this study.

2.2.1 Wireless sensor networks (WSNs)

A WSN is a network created by a sizeable number of sensor nodes where respectively nodes are equipped with a sensor to sense physical phenomena such as light, pressure and heat(Bröring *et al.*, 2011). WSNs are considered as a revolutionary data gathering scheme to build the communication and information system which will significantly enhance the reliability and effectiveness of infrastructure schemes. Associated along with the wired solution, WSNs attribute is easier deployment and enhanced the flexibility of devices. Along with the fast technological growth of sensors, WSNs will turn out to be the central technology for (Internet of things) IoT (Matin & Islam, 2012).

WSNs can largely be termed as networks of nodes that jointly sense and could control the environment, consequently enabling communication between computers or human beings and the immediate location (Presser *et al.*, 2009). In reality, the act of sensing, processing and communicating with a reduced quantity of energy, ignites a cross-layer design style typically necessitating the collaborative consideration (Matin & Islam, 2012) of distributed signal/data processing, medium access control and communication protocols (Bröring *et al.*, 2011).

In WSNs, the usage of resources is usually highly related to the execution of tasks which consume a certain amount of computing and communication bandwidth (Guo *et al*, 2011; Guo *et al*, 2014). Parallel processing among sensor nodes is a promising innovation to provide the demanded computation capacity in WSNs, task allocation plays an essential role in parallel processing (Guo *et al*, 2014).

2.2.1.1 Applications of WSNs

WSNs have acquired significant popularity owing to their flexibility in resolving problems (Matin & Islam, 2012) in diverse application areas and have the capability to revolutionize our lives in various ways. WSNs have been effectively harnessed in diverse application fields such as (Matin & Islam, 2012):

1. **Military applications:** Wireless sensor networks are an integral component of military command, computing, control, intelligence, communications, battlefield surveillance, inspection and targeting system.
2. **Area monitoring:** The sensor nodes are deployed over an area where some events are being monitored. Once the sensors detect an event being monitored (temperature, humidity), that event is conveyed to one of the base stations and the appropriate actions are taken immediately.
3. **Transportation:** Real-time traffic data are being collected by the WSNs to feed into the transportation models and alert drivers of congestion and traffic issues.
4. **Health applications:** The health applications for sensor networks are the supporting interfaces for the disabled, integrated patient monitoring, diagnostics, and drug administration in hospitals, tele-monitoring of human physiological data and tracking and monitoring doctors or patients inside a hospital.

5. **Environmental sensing:** Environmental sensor networks have been built to cover lots of applications of WSNs for earth science research. They include sensing volcanoes, oceans, glaciers and forests fire. Some other key fields are itemized as follows:
   A) Air pollution monitoring
   B) Landslide detection
   C) Forest fires detection
   D) Greenhouse monitoring
6. **Structural monitoring:** Wireless sensors can be employed to monitor the movement in buildings and infrastructure such as flyovers, bridges, tunnels and thus empowering engineering practices to monitor assets remotely with no need for expensive site visits.
7. **Industrial monitoring:** Wireless sensor networks have been industrialized for machinery condition-based maintenance (CBM) as they offer substantial cost savings and permit new functionalities. In wired schemes, the installation of adequate sensors is often limited by the cost of wiring.

2.2.2 Task allocation in WSNs

Task allocation in WSNs is the process that gives rise to a particular number of sensor nodes being involved in certain tasks, for the purpose of carrying out or executing such given tasks. Due to the several issues present in the WSNs, the process of task allocation is highly essential in order to maintain the failure-free operation of the WSN (Gangadharaiah *et al*, 2014). Task allocation and scheduling play an essential role in parallel processing (Suganya & Jayanthi, 2016). The way to assign a task to the most appropriate sensor node and concurrently balance the network load in view of the uncertain and dynamic network conditions are essential in WSNs. Task allocation and scheduling have been widely studied in different areas of applications, such as multiprocessor system, grid computing, social networks, multi-hop wireless networks (Suganya & Jayanthi, 2016).

Task allocation issues have been addressed to a larger extent in distributed systems but the problem for WSNs is different from traditional distributed systems. In WSNs, the challenge of task allocation is distributing sensing tasks rationally among sensor nodes to reduce overall power consumption while ensuring these tasks being finished before deadlines and prolonging network lifetime (Suganya & Jayanthi, 2016). Load balancing is a major issue for prolonging the lifetime of the network. An inferior task allocation mechanism will lead to an overload of nodes which might be harmful to the networks (Suganya & Jayanthi, 2016). Due to the challenging features and limitations of WSNs, such as environmental restraints, dynamic topology, and unsteadiness of wireless link, there exist more vulnerabilities and reservations for real-time applications in WSNs.

The sensor networks with allocated tasks are shown in Fig. 2.1 (Suganya & Jayanthi, 2016; Guo *et al*, 2014). The purpose of task allocation can be basically expressed as follows. Generally, let m tasks be allocated satisfactorily to n sensors in order to minimize task execution time, save the energy consumption of nodes, balance the network load, prolong the network lifetime, certify that the task would not fail by unexpected failure of nodes and improve the dependability of task supervision (Guo *et al*, 2014).

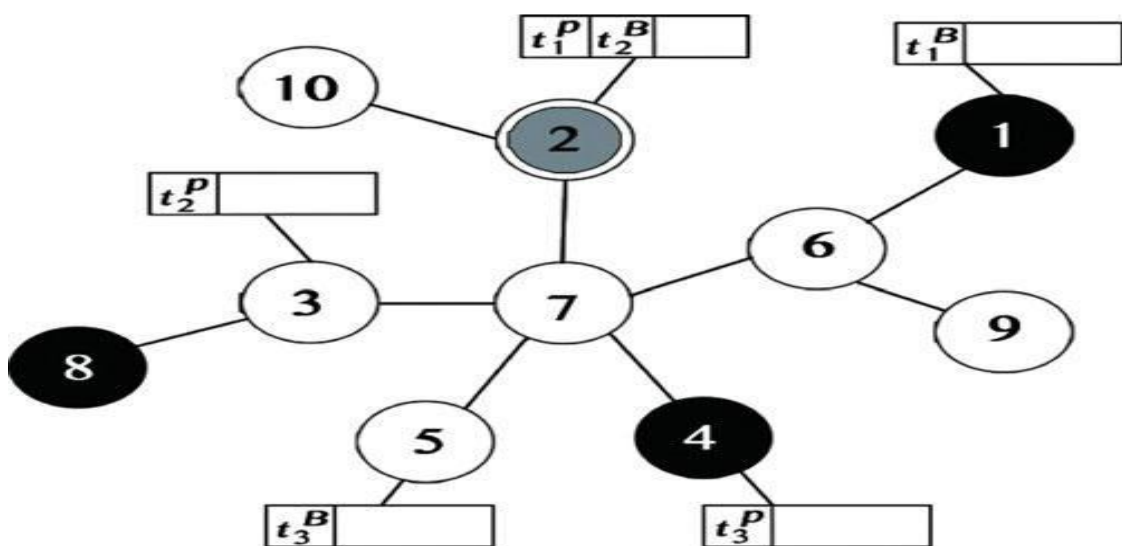

Fig. 2.1: Wireless Sensor Networks with Allocated Tasks (Suganya & Jayanthi, 2016; Guo *et al*, 2015b)

2.2.3 Fault-tolerance in wireless sensor networks

Fault tolerant schemes are required to aid WSNs to continue their operations even after the failure of several nodes (Chouikhi *et al*, 2014). There are certain protective techniques that can be used such as data aggregation, node sleeping and multi-channel communication (Chouikhi *et al*, 2014). The reactive practices consist of the use of backup paths, node redundancy, and nodes reorganization.

Faults in real-time systems which are not tackled perfectly in a well-timed manner will lead to a violation of timing limitations which can cause devastating consequences if the schemes are time and/or safety-critical, e.g. aircraft, nuclear power plant (Han, 2015). Consequently, delivering fault-tolerant attributes to attain high reliability is specifically sought after in such systems (Han, 2015). Fault tolerance can be defined as a system's ability to keep on operating as scheduled notwithstanding the presence of faults. There are diverse approaches to achieve fault tolerance (Arar & Khireddine, 2016). Conventional fault-tolerance procedures to deal with faults have two components, i.e. fault detection and fault recovery (Han, 2015). Fault detection is the process of recognizing that a fault has occurred whereas fault recovery is the process of regaining operational status via reconfiguration even in the presence of faults. Fault detection is often required before any recovery procedure can be initiated.

In order to tolerate or recover from faults, various hardware/software replication techniques have been developed. For instance, two task replication schemes (active and passive replication schemes) have been proposed to support fault tolerance in multi-core systems (Han, 2015).

1. **Active replication:** This technique is based on space redundancy and does not necessitate fault detection (Balasangameshwara, 2015). In active replication backup, one or more independent active copies of a task run concurrently on separate sensor nodes (Han, 2015).

2. **Passive replication:** One or more backups of a task are assigned either to the same node or to a backup node (Han, 2015). A backup copy of a job is activated only if a fault occurs while executing its primary copy (Balasangameshwara, 2015). The techniques applied while scheduling primary and backup copies of a job are (Balasangameshwara, 2015);

A) Backup overloading: This consists of scheduling backups for multiple primary jobs during the same time slot.

B) De-allocation of resources reserved for backup jobs when the corresponding primaries complete successfully.

Active replication schemes typically have need of extra system resources, such as processing cores. However, they can tolerate run-time faults timely and promptly (Han, 2015). In contrast, passive replications are only invoked in the event of run-time failure(s) and therefore, do not exhaust system properties when no faults transpire (Balasangameshwara, 2015). In spite of this, passive replications take longer to recover from faults and place the system at risk when timing constraints are very stringent (Han, 2015). The choice of the replication schemes for diverse hard real-time systems is a strategic decision drawback and requires precise investigations (Balasangameshwara, 2015). Established techniques for safeguarding the timing constraints for real-time systems with no explicit fault-tolerance requirements are becoming ineffective (Han,2015; Balasangameshwara, 2015). It is essential to explore advanced approaches to safeguard the timing constraints in the presence of faults for real-time systems. Besides, equally fault tolerance and energy reduction are essentially reached by utilizing system slack time, as a result they are two conflicting objectives in nature (Balasangameshwara, 2015). It is appropriate that some constraints, such as timing, power, reliability and their relationships be considered in a broad and logical way to achieve a choice of design objectives for real-time systems (Han, 2015).

2.2.3.1 Task allocation with fault-tolerant WSNs

Task allocation by means of fault tolerance is an effectual way of enriching WSNs reliability (Guo *et al*., 2015a). Allocating a number of copies of tasks on diverse sensor nodes is a simple and effectual way of actualizing a fault-tolerant system. In this research, the primary/backup (P/B) model is put to use. The P/B model has two copies of a lone task that are allocated on two individual sensor nodes (Chen *et al*, 2012; Guo *et al*., 2015a). For a lone task, its primary copy should be scheduled in conjunction with its backup copy and both are executing concurrently (Guo *et al*., 2015b). Several backup copies of tasks whose primary copies are allocated on independent sensor nodes may overlap in the same time slot on the same sensor node (Chen *et al*, 2012). For the system to proffer high reliability, the research considers the active backup copies so that for a particular task with limited laxity (Guo *et al*., 2015b), its two copies are concurrently executed in the same time slot. This mechanism can reduce unnecessary energy consumption for task execution in WSNs (Chen *et al,* 2012).

2.2.4 Real-time WSNs system

Most of the tasks in WSN are requested to run in a real-time way (Harkut *et al*, 2014). Real-time systems refer to computing schemes that are subject to "real-time" limitations where the precision of an output hangs on not only its logic accuracy but when the output is delivered (Han, 2015). Applications in conjunction with timings constraints are called "real-time". As soon as an event occurs, the application has to deal with it within a recognized and confined delay. An application taking more time can have a possibly negative impact (Augé-Blum *et al*, 2012).

Mostly real-time systems are divided into two main categories: hard real-time and soft real-time system (Harkut *et al*, 2014). If a system is hard real-time then all tasks must be completed before its deadline so that the system will never face any catastrophic situation whereas in soft real-time system, if tasks miss their deadlines then no such disastrous situation arises (Harkut

et al, 2014). Examples of hard real-time systems comprise of forest fire detection, volcanic eruption monitoring and emergency disaster management (Augé-Blum *et al*, 2012). Examples of soft real-time systems include multimedia applications. These applications can be classified into four classes, according to how data exchange is initiated (Han, 2015). In event-driven applications, communication starts when some event happens whereas, in the query-driven applications, communication only starts when a specific node (like the sink node in the case of WSNs) sends a query (Han, 2015). In time-triggered applications, communication happens at the predefined instance of time (Augé-Blum *et al*, 2012). Finally, some applications, such as live video transmission, require data to continuously flow through the network. Each one of these application categories has specific time-related constraints (Harkut *et al*, 2014).

For an application to be real-time, the underlying network must be able to deal with real-time constraints (Augé-Blum *et al*, 2012). So far, only local area networks (LANs) are considered, as they are deployed only over small areas (Augé-Blum *et al.*, 2011). The size of such an area is typically a building (Han, 2015) (e.g. nuclear power plant surveillance network) or smaller (network embedded in planes, cars or trains)(Augé-Blum *et al.*, 2011).

2.2.5 The real-time fault tolerant task allocation schemes

A system is said to be a real-time fault tolerant task allocation schemes; if it generates a result within a specified time so that the job will complete its execution before the deadline and system will never fail if all jobs will complete its task before the deadline (Harkut *et al*, 2014). The RFTAS and mRFTAS employed a variant of the Particle Swarm Optimization (PSO) called the discrete Particle Swarm Optimization (dPSO) for the generation of particle position and velocity, also optimizes the parameter metrics.

Particle Swarm Optimization (PSO) is patterned on the social behaviour of a flock of birds (Vaishnavi & Sukumar, 2015). It comprises of a swarm of *s* candidate solutions called particles, which search an n-dimensional hyperspace in the exploration of the global solution

(n represents the number of optimal parameters to be determined). A particle i take up position X_{id} and velocity V_{id} in the dth dimension of the hyperspace, $1 \cdot i \cdot s$ and $1 \cdot d \cdot n$. Every single particle is estimated through an objective function $f(x_1; x_2; \ldots; x_n)$, where $f: R^n \rightarrow R$ (Guo *et al*, 2015b). The cost (fitness) of a particle near to the global solution is lower (higher) than that of a particle that is farther. PSO thrives to minimize (maximize) the cost (fitness) function (Vaishnavi & Sukumar 2015; Guo *et al*, 2015b). In the global-best version of PSO, the position where the particle i has its lowest cost is stored as $(pbest_{id})$. Besides, $gbest_d$, the position of the best particle. In each iteration k, velocity V and position X are updated using (Vaishnavi & Sukumar 2015; Guo *et al*, 2015b):

$$V_{id}(k+1) = w\,V_{id}(k) + \varphi_1 r_1(k)(pbest_{id} - X_{id}) + \varphi_2\, r_2(k)(gbest_d - X_{id}) \quad (2.1)$$

$$X_{id}(k+1) = X_{id}(k) + V_{id}(k+1) \quad (2.2)$$

Here, φ_1 and φ_2 are constants, and $r_1(k)$ and $r_2(k)$ are random numbers uniformly distributed in [0,1].

The update process is iteratively repeated until either an acceptable *gbest* is achieved or a fixed number of iterations k_{max} is reached (Vaishnavi & Sukumar 2015; Guo *et al*, 2015b).

The advantages of PSO over many other optimization algorithms are its ease of implementation and ability to converge to a reasonably good solution quickly (Vaishnavi & Sukumar, 2015). PSO is also more efficient in preserving population diversity to avoid premature convergence issue.

In (Guo *et al*, 2015b), PSO method is also employed in RFTAS scheme in a wireless sensor network environment. The flow chart of the algorithm is shown in Fig. 2.2. It can be seen from Fig. 2.2 that the sink node collects tasks firstly, then it generates position and velocity in parallel for one generation of dPSO. Next, it begins a series of operations. After that, if the termination condition is not met, the next iteration begins (Guo *et al*, 2015b). Otherwise, it publishes tasks and waits. The right part of Fig. 2.2 is the execution process of tasks. If the primary version is

finished successfully, no matter the mode of corresponding backup version is active or not, it does not have to be executed (Guo *et al*, 2015b). If the mode of backup copy is passive, it is executed only when the primary copy fails.

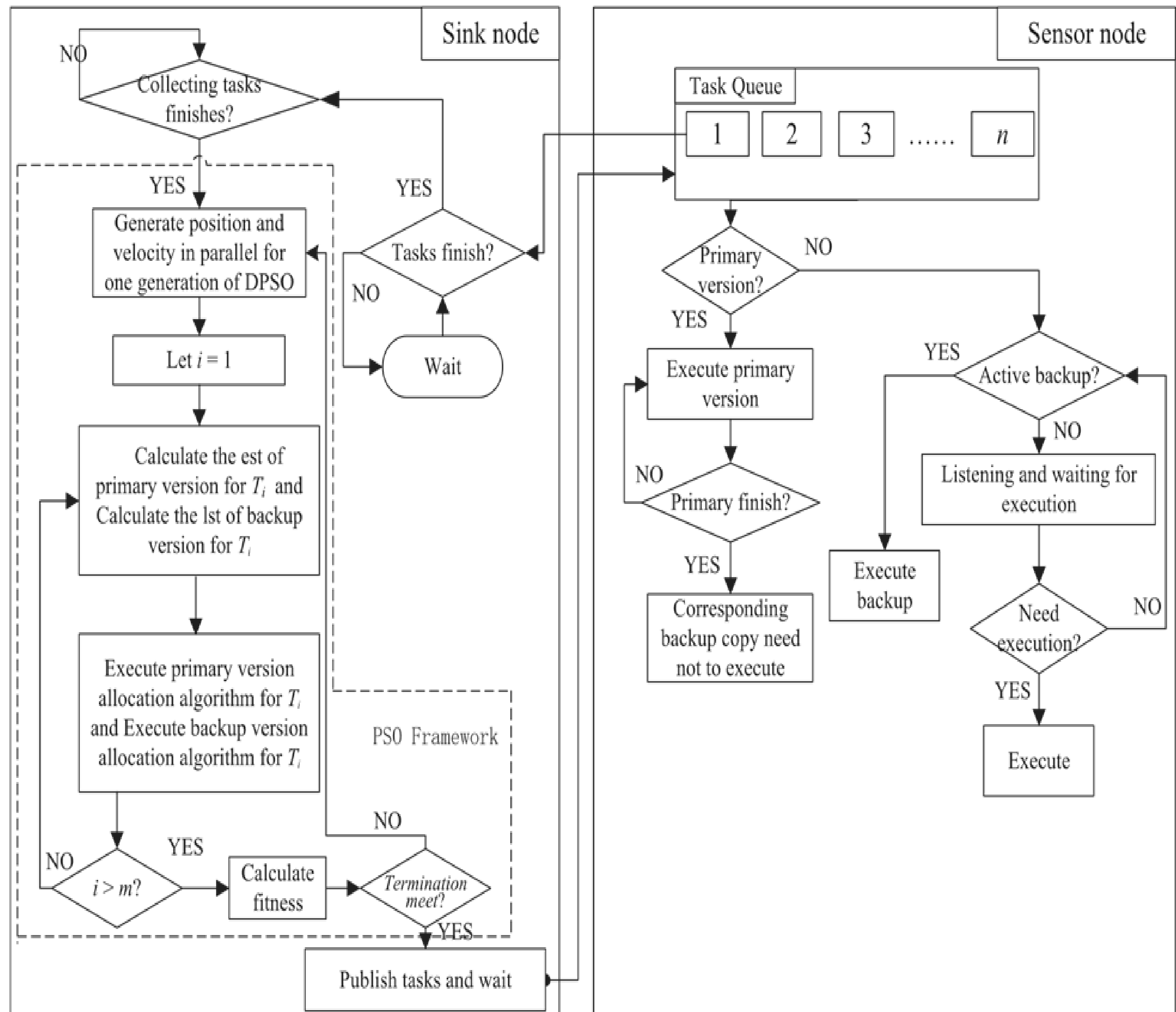

Fig. 2.2: RFTAS Flowchart (Guo *et al*., 2015b).

2.2.6 Performance metrics.

The performance metrics used for this research are briefly discussed as follows and they include total time of task execution, energy consumption of task, reliability cost and network lifetime.

2.2.6.1 Total time of task execution

The total execution time to finish all tasks is composed of the execution time of the primary copies and execution time of the active backup copies for *m* tasks and can be defined as in (Guo *et al*, 2015b):

$$Time = \sum_{i=1}^{m} et_{i.P\left(t_i^P\right)} + \sum_{i=1}^{m} S\left(t_i^B\right) \times \left(f_i^P - lst_{i.P\left(t_i^B\right)} \right) \tag{2.3}$$

where m is number of tasks, *et* is early start time, i & j are allocated tasks, *lst* is latest start time of backup task, t_i^B denotes backup task copy, t_i^P denotes primary task copy, f_i^P denotes finish time of primary copy of task, d_i denote deadline of t_i, $P\left(t_i^B\right)$ and $P\left(t_i^P\right)$ denotes the nodes t_i^B and t_i^P are allocated, $S(t_i^B)$ represents the mode of t_i^B, l_i is laxity of t_i^P, ($l_i = d_i - f_i^P$).

If l_i is not less than $et_{i.p(t_i^B)}$, t_i^B need not to be executed before f_i^P for fault-tolerant. Because even $n_{P(t_i^P)}$ fails before the time f_i^P, t_i^B also can be finished before d_i. Thus, it uses passive mode. Otherwise, t_i^B needs to be executed before f_i^P for fault-tolerant. Thus, it employs active mode. Let $S(t_i^B)$ is equal to 1, which means t_i^B uses active mode. Otherwise, t_i^B uses passive mode. $S(t_i^B)$ can be expressed as in (Guo *et al*, 2015b):

$$S\left(t_i^B\right) = \begin{cases} 1, \ if \left(l_i < et_{i.P\left(t_i^B\right)} \right) \\ 0, \ else \end{cases} \left(1 \le i \le m\right) \tag{2.4}$$

Generally, when the primary copy is executed successfully, the corresponding backup copy needs not to be executed any more.

2.2.6.2 Energy consumption of task execution

The energy consumption of task execution, contains the energy of the primary copy and the active backup copy consumed for *m* tasks and can be expressed as (Guo *et al*, 2015b):

$$Energy = \sum_{i=1}^{m} ene_{i.P\left(t_i^P\right)} + \sum_{i=1}^{m} S\left(t_i^B\right) \times \mathrm{ene}_{i.P\left(t_i^B\right)} \times per_{i.P\left(t_i^B\right)} \tag{2.5}$$

where $per_{ij} = \frac{f_i^P - st_{ij}}{et_{ij}} \times 100\%$ denotes the percentage of execution time that active mode t_i^B executed on node n_j to the total time that the task executed on node n_j, m is number of tasks, et is early start time, ene is energy, i & j are allocated tasks, lst is latest start time of backup task, t_i^B denotes backup task copy, t_i^P denotes primary task copy, f_i^P denotes finish time of primary copy of task, d_i denote deadline of t_i , $\mathrm{P}\left(t_i^B\right)$ and $\mathrm{P}\left(t_i^P\right)$ denotes the nodes t_i^B and t_i^P are allocated, $S(t_i^B)$ represents the mode of t_i^B, l_i is laxity of t_i^P, ($l_i = d_i - f_i^P$).

2.2.6.3 Reliability

Reliability is defined as the probability that none of real-time tasks fail even in the presence of failures. Therefore, the reliability model is given as follows (Guo *et al*, 2015b):

$$RC = \left(RC\left(zb\right) + RC\left(zp\right)\right) \times 100 \tag{2.6}$$

$$RC\left(zp\right) = \sum_{i=1}^{m} \lambda_{P\left(t_i^P\right)} \times et_{i.P\left(t_i^P\right)} \tag{2.7}$$

$$RC(zb) = \sum_{i=1}^{m} S\left(t_i^B\right) \times \lambda_{P\left(t_i^B\right)} \times \left(\mathrm{f}_i^P - lst_{i.P\left(t_i^B\right)}\right) \tag{2.8}$$

$$\mathrm{Re}\,l = \exp\left(-RC\right) \tag{2.9}$$

where m is number of tasks, et is early start time, i & j are allocated tasks, lst is latest start time of backup task, t_i^B denotes backup task copy, t_i^P denotes primary task copy, f_i^P denotes

finish time of primary copy of task, d_i denote deadline of t_i , $P(t_i^B)$ and $P(t_i^P)$ denotes the nodes t_i^B and t_i^P are allocated, $S(t_i^B)$ represents the mode of t_i^B , l_i is laxity of t_i^P , ($l_i = d_i - f_i^P$). *Rel* denotes the reliability of the network, RC denotes the reliability cost of the network, *RC(zp)* and *RC(zb)* are the total reliability costs of primary copy and a backup copy of *m* tasks, and λ_i denotes the failure probability of n_i. Take Fig. 2.1 as an example, if the failure probability of n_2 is relatively large and it also needs to work a long time, it is obvious that the reliability of the system must be declined. Thus, the reliability of networks decreases with the increase of the reliability cost as shown in equation (2.9).

2.2.6.4 Network lifetime

Task allocation is an optimization problem that can be formalized as follows (Guo *et al*, 2015b):

$$Maximize(lt)\ S.T \begin{cases} \sum_{i=1}^{m} y_i \le num \times 20\% \\ \forall t_i \in T, n_{p(t_i^P)} \ne n_{p(t_i^B)} \end{cases} \tag{2.10}$$

where *lt* denotes the lifetime of WSNs and *num* denotes the total number of the sensor nodes. $y_i = 1$ means that node n_i is failure and $y_i = 0$ means that node n_i is normal. Assume that WSNs are paralyzed when the number of failure nodes exceeds 20% of the total number. For example, if n_1, n_4 and n_8 break down in Fig. 2.1, then $\sum_{i=1}^{10} y_i = 3$ is larger than $num \times 20\% = 2$, therefore the network is paralyzed.

2.3 Review of Similar Works

This section presents an overview of works similar and relevant to the proposed research.

Zhu *et al.*, (2010) developed a QoS-aware fault-tolerant (QAFT) scheduling algorithm that could tolerate one processor's permanent failure at a one-time instance for real-time tasks with QoS needs in heterogeneous systems. The fault-tolerant capability was fused into the algorithm by using a primary/backup (P/B) model. QAFT employed the backup copy overlapping

technology to improve the scalability and enhance the real-time tasks' QoS levels. In addition, QAFT adaptively adjusted the QoS levels of incoming real-time tasks based on the system load, which enhanced the system flexibility. The system reliability was further enhanced by assigning tasks to processors that are able to yield high reliability. Simulation studies were carried out to compare it with two other relevant scheduling algorithms, non-overlapping QoS-aware fault-tolerance scheduling algorithm (NOQAFT) and dynamic and reliability-driven real-time fault-tolerant scheduling algorithm (DYFARS). The experimental results indicated that QAFT outperformed the others for real-time tasks with QoS needs in heterogeneous systems. However, the work employed the passive replication techniques which resulted in increased task waiting time (time delay), thus prolonging the processing task execution time.

Guo *et al.*, (2011) developed a multi-agent architecture based self-adaptive task scheduling strategy in WSNs. The dPSO algorithm for dynamic alliance (dPSO-DA) with a well-designed particle position code and fitness function was developed with a mutation operator which effectively improved the algorithm's ability of global search and population diversity was also introduced in the strategy. The simulated results showed that the developed solution achieved significantly better performance than those that used other algorithms such as, Max-min (Max-min completion Time), (MCT) Minimum Complete Time, GA (Genetic Algorithm). However, the system could not be said to be reliable as system fault tolerance was not the focus of the work, processing time delay was not also taken into consideration and this is a problem for safety-critical applications.

Zhu *et al.*, (2011) developed an efficient QAFT that scheduled independent real-time tasks that tolerated hardware failures in a heterogeneous cluster. Fault-tolerance capability was incorporated into the QAFT algorithm by organizing a primary backup model. To essentially utilize system resources, QAFT employed the backup copy overlapping technology, desiring

to forward the start time of primary copies and delay the start time of backup copies within a time constraint. In order to evaluate the performance of QAFT, the research conducted an extensive simulation to compare QAFT with NOQAFT and DYFARS. The experimental results showed that QAFT significantly improved the scalability and QoS levels of real-time tasks on heterogeneous clusters as compared to NOQAFT and DYFARS. However, the work employed the passive replication techniques which resulted in increased task waiting time (time delay).

Chen *et al.*, (2012) developed a new task allocation algorithm based on dynamic coalition in WSNs and the dPSO framework. The algorithm could tolerate one sensor node failure in a heterogeneous WSNs by the use of primary/backup copy technology and the overlapping backup copy technology which reduced unnecessary energy consumption. The algorithm developed adopted the node with minimum energy consumption to execute tasks. The experimental simulation result showed that the developed algorithm in WSNs could effectively accept more tasks scheduled, save energy consumption, balance the load of the network, thus prolonging the network lifetime. However, the issue of the waiting time for processing tasks was not addressed which is vital for real-time system processes.

Chouikhi *et al.*, (2014) developed preventive/reactive fault tolerant multi-channel allocation scheme dedicated to managing the energy consumption and reconnection of the WSN in case of delivery node failure. Precisely, the work developed two heuristics for channel allocation/reallocation and WSN reorganization after a network failure. The work developed a fault tolerant multi-channel scheme aimed at avoiding premature failure and ensuring recovery from the network partition. The research used a multi-channel communication to reduce the interferences and a WSN reorganization technique to restore the network connectivity after failure. However, the choice of different parameters is still far from the optimal. The

reorganization and the retransmission of tasks will also prolong the task execution time, which is not suitable for safety critical systems.

Gangadharaiah *et al.*, (2014) developed an algorithm based on soft real-time auction approach for task allocation in WSNs. The winner of the auction process was not decided as soon as auctioneers got the bid value higher than the threshold value since the auctioneer had a particular time slot, which was based on application requirements. The algorithm gave a fair chance to all sensor nodes which had sufficient energy to participate in the auction, thereby maintaining the energy balance in the network. Also, an improved real-time approach by using wait time at the bidder side was incorporated. The wait time method tried to get the best bid value by allowing the node with the highest bid value to bid earlier than other nodes. This gave a much better bid value in less time in real time application. However, since the scheme was not based on fault tolerance as the auction bids do not have backup copies, a failure or fault on any sensor node can affect the determination of the node with the highest bidding value.

Guo *et al.*, (2014) developed a dynamic task scheduling scheme with the application of the game theory in WSNs. An effective parallel alliance generating algorithm was developed in order to process tasks in a multi-task environment. The game theory was used to allocate the tasks as well as eliminate the resource conflict during the allocation as an added value. The work compared game theory-based dynamic task allocation algorithm (TAA-GT) with two algorithms respectively named parallel alliance and greed algorithm-based dynamic task allocation algorithm (TAA-PAGA) and parallel alliance and random algorithm-based dynamic task allocation algorithm (TAA-PARA). Experimental results showed that the proposed strategy indeed obtained a better performance of the network relative to TAA-PAGA and TAA-PARA algorithms. However, the focus of the study was not on fault tolerance. As such, a failure

or fault on any sensor node will affect the execution length or time of the task allocation processes.

Guo *et al.*, (2015a) developed a trust dynamic task allocation algorithm with Nash equilibrium to support heterogeneous WSNs. In order to guarantee that the sensor nodes in the same coalition are adjacent in geographical position, a dPSO was designed to divide the nodes distributed randomly into several coalitions. The research work invoked the game theory in WSNs and also designed the strategies for each task and payoff function. The dPSO with the redesigned fitness function was also used to obtain the Nash equilibrium point for the purpose of minimizing task execution time, saving node energy cost, balancing network load and prolonging network lifetime. However, the research was not focussed on developing a fault tolerant system.

Guo *et al.*, (2015b) developed a PSO soft real-time fault-tolerant task allocation algorithm (FTAOA) for WSNs using primary/backup (P/B) technique to support fault tolerance mechanism. The work employed P/B technology to tolerate permanent node failures and passive backup overlapping technology to reduce redundancy. The research conducted extensive experiments to compare the FTAOA with the QAFT scheduling algorithm, fault-tolerance task allocation algorithm without overlapping (FTAA), fault-tolerant task allocation with genetic algorithm (TAGA) and extended Min-Min (EMM) algorithm. Results showed that the proposed scheme outperformed all the others. However, the strategy incurred time delay in the activation of the backup copies of the primary tasks which is not suitable for systems that are safety critical.

Subalakshmi & Helen, (2015) developed an information recuperation system by assigning task in WSN. The binary particle swarm optimization (bPSO) was employed for assigned tasks in WSNs. This work developed task assignment for information recuperation and topology

permutation for improving the permanency of the sensor network. The task assignment problem for data recuperation was implemented in TCL language on NS2 running under Windows environment. In the developed system, the calculated execution time subtracted the execution time of the vulnerable sensor node. The task execution time, energy utilization and sensor network lifetime were considered in the fitness function to make a legitimate trade-off among distinctive metrics in order to obtain the best overall execution time. However, trying to retrieve information lost as a result of failure of sensor nodes leads to more energy consumption, degradation of reliability and lifetime of the WSNs.

Wang *et al.*, (2015) developed a task allocation algorithm based on score incentive mechanism (TASIM) for complex task execution in WSNs. A sensor node which successfully completed tasks could be rewarded with some scores, while a different sensor node which was unable to finish tasks was punished by deducting some scores. Based on the scores, the uncompleted tasks on failure nodes are migrated to other sensor nodes for further execution in a timely manner, which ensured that the complex tasks were finished. Simulated results demonstrated that the performance of the TASIM was clearly better than those of conventional task allocation algorithms such as EcoMapS and DTAS, in terms of both network load balance and energy consumption. However, transferring information or tasks to new nodes for execution as a result of failure of sensor nodes will also cause a rise in energy consumption, degrade the reliability and lead to prolonged task execution time.

Priyanka *et al.*, (2016) developed a PSO-optimized Real-time fault tolerant algorithm (FTAOA) for WSNs proposed in VLSI to conquer the prevailing problems. A P/B system was implemented to conquer the faults in the sensor network and allocate the given task logically. FTAOA employed the P/B scheme which used passive backup copies overlapping approach to monitor the mode of backup copies adaptively through planning primary copies early and

backup copies delayed. Modelsim was used for simulation and XilinxISE language for coding the FTAOA algorithm. An extensive simulation was carried to test the performance of the proposed scheme as compared to the existing schemes (such as GA, Min-Min (EMM) algorithm) and results showed that it out performed them. However, the work incurred time delay in the activation of the backup copies of the primary tasks which is inimical to systems that are safety critical.

Tien *et al.*, (2017) proposed a novel path redundancy-based algorithm termed dual separate paths (DSP), which supported fault-tolerant transmission with the enhancement of the network traffic operation for WSNs application, such as fault-tolerant CPSs. The proposed DSP algorithm determined two independent paths between a source and a destination in a network based on the network topology communication. The analyzed and simulated results showed that the DSP-based methodology did not only offer fault-tolerant transmission, but also advanced network traffic operation. The DSP algorithm was utilized to high-availability seamless redundancy (HSR) networks and it minimized the network traffic by 80% to 88% compared to the standard HSR protocol, thus enhancing network traffic operation.

Yu *et al.*, (2017) proposed a distributed optimal on-line task allocation (DOOTA) algorithm, by studying the energy cost of transmitting, sensing and sleeping events to optimally balance the load capacity allocation between the sensor nodes. The work proved that the optimal partition result for individual nodes comprises of at most 2 division cuts with the equivalent weights. The simulation results showed the proposed algorithm enhanced the network lifetime by 12.26 times compared with the approach of no scheduling, which is 2.22 times more than preceding off-line task allocation techniques. However, trying to retrieve information lost as a result of failure of sensor nodes leads to more energy consumption, degradation of reliability and lifetime of the WSNs.

Duan *et al.*, (2017) developed a fault tolerant scheduling algorithm with a deferred active backup copy scheduling algorithm in distributed sensor network such as to ensure that the entire distributed sensor network utilized a reduced amount of energy though implementation of fault tolerance at equal times. Experimental simulations were carried out in a 300×300-unit area and the sensor nodes were placed randomly with a 40-unit radius. Results showed improvements with respect to the sleeping rate (SR) and coverage rate (CR) compared to other strategies. However, the work incurred time delay in the activation of the backup copies of the primary tasks which is not required in real-time systems that are safety critical.

It is clear from literature reviewed that several research works have been carried out on the area of task allocation/scheduling schemes and fault tolerant task allocation scheme in WSNs. Successful execution of task allocation in real-time systems with little or no delay in task processing time, reduced task waiting time is still an open-ended research area for real-time WSNs. In order to tackle certain issues related with most of the reviewed literature, this research seeks to develop a modified real-time fault tolerant task allocation scheme (mRFTAS) using the active replication backup scheme. The mRFTAS is expected to minimize fault and processing time delay in WSNs real-time task allocation system.

CHAPTER THREE
MATERIALS AND METHODS

3.1 Introduction

In this chapter, the materials, the methods used for the implementation, development and simulation of the RFTAS and mRFTAS are described in details based on the outline developed in section 1.6.

3.2 Materials

The following are the materials used in this research:

1) Window with .NET Framework 4.0 SDK
2) Microsoft Windows 10
3) Microsoft Visual Studio 2015
4) Matlab 2017a
5) Microsoft word and Visio
6) Language : C# Programming Language
7) Processor : 1.0 GHz Processor
8) Memory : 2GB of RAM recommended
9) Hard Disk : 80GB of hard disk space required
10) Display : 1024x768 or higher-resolution display with 16 bits colors

3.3 Analyses of the Real-time Fault-tolerant Task Allocation Scheme

The processes involved in the replication of the real-time fault tolerant task allocation scheme for WSNs are discussed in details in the following sub-section

3.3.1 Task allocation mechanism based on dPSO

PSO was initially employed in continuous space optimization issue. In any case, allocation of the task is a discrete optimization issue, a dPSO model ought to be developed by growing the fundamental PSO in a binary space. Propelled by the possibility of PSO technique, we can find

that task allotment issue can be portrayed as a binary encoding in a matrix and the relating fitness function can be defined with task execution time, energy utilization, the balance of system energy dispersion and dependability cost as optimization objects to guide the evolution for the optimal solution.

3.3.2 The encoding of particle

The position and speed of particles are characterized as two $m\times n$ dimensional frameworks X and V, individually, and the particle position encoding is presented as (Guo *et al*, 2015b):

$$x_{ij} = \begin{cases} 0, & else \\ 1, & if\ \ jth\ node\ take\ part\ in\ ith\ task \end{cases} \tag{3.1}$$

where $1 \le i \le m; 1 \le j \le n$

At each progression, particles are controlled by the accompanying equations (Guo *et al*, 2015b):

$$V^{t+1}(i) = wV^{t}(i) + c_1 r_1 \left(pBest(i) - X^{t}(i)\right) + c_2 r_2 \left(gBest - X^{t}(i)\right) \tag{3.2}$$

$$X^{t+1}(i) = \begin{cases} 0, & else \\ 1, & if\ \ rand() < sigmoid\left(V^{t+1}(i)\right), \end{cases} \tag{3.3}$$

where t is the present iteration times, $X(i)$ and $V(i)$ indicate the position and speed of the *ith* particle, $pBest(i)$ is the best position of particle i, $gBest$ is the global best position, w is inertia weight, c_1 and c_2 signify acceleration factors, r_1 and r_2 are two variable numbers in the scope

of [0,1], *rand ()* is the random function for producing random number in the scope of [0,1], and $sigmoid(V) = 1/(1+\exp(-V))$

As an essential parameter, a suitable inertia weight *w* can acquire a balance amongst global and local search. In order to enhance the global search execution of PSO, *w* updates as per the traditional linear descending strategy (Guo *et al*, 2015b):

$$w = w_{max} - c_iter \times \frac{w_{max} - w_{min}}{m_iter} \tag{3.4}$$

where c_iter signifies the present iteration times, m_iter means the maximum iteration times, w_{max} and w_{min} are the inertia and terminal inertia weights, respectively.

3.3.3 Fitness function

Task allotment in WSNs is a multi-objective optimization problem (MOP). Here the work changed the issue into a single-objective optimization issue by means of weighted *sum* system. The fitness function is represented by equation 3.5 and α, β, γ are weighting factors, FR is the failure ratio and RC denote reliability cost (Guo *et al*, 2015b):

$$Fitness\ function = \alpha \times Time \times (1 + FR) + \beta \times Energy + \gamma \times RC \tag{3.5}$$

3.3.4 dPSO-Based task allocation algorithm description

Algorithm 1 in Fig. 3.1 demonstrates the pseudo code of dPSO-based tasks distribution methodology. *i* indicates the serial number of the current particle. *j* and *m* mean the serial number of task and the aggregate number of tasks, separately. $cur_particle_fit$ stands for the fitness estimation of the current particle. The smaller the value, the better the position of the current molecule (giving Equation. 3.5). In general, the condition for termination is the most

maximum iteration times or a sufficient solution. In the event that the condition for termination is met, the result (solution) will be acquired and the run is ended.

1. Replication and Implementation of the RFTAS**.**

At the Sensor Nodes

A) Initialize the tasks or particles of both primary and backup copies.

B) Determine if the task is a primary or backup copy.

C) Execute if the task copy is primary task copy, wait and listens if the task copy is a backup copy.

D) Execution of primary task finish? if yes, No need for the execution of backup copy of the task.

E) If the primary task fails, execute the backup copy of the task to finish.

At the Sink Node

F) Collection of the finished task either primary or backup task copy.

G) Produce velocity and position in parallel for one initiation of DPSO.

H) Check if $i=1$.

I) Compute the *est* of the primary copy for T_i and compute the *lst* of the backup copy for T_i.

J) Implement primary copy of the task allocation scheme for T_i and perform backup copy of the task allocation algorithm for T_i.

K) Is $i > m$*?* if yes estimate the fitness value, if No go back to step I.

L) Termination condition met? if yes publish tasks and wait, if No, go back to step G.

2. Development and Implementation of a modified fault tolerant task allocation scheme *(mRFTAS),* using active replication backup scheme.

At the Sensor Nodes

A) Initialize the tasks or particles of both primary and backup copies.

B) Determine if the task is a primary or backup copy.

C) Execute both the primary and the backup copies of the tasks concurrently on independent sensor nodes.

D) Execution of both primary and backup tasks finish? If No go back to step C.

E) If either of the task copy finished successful execution, no need to execute the corresponding task copy.

At the Sink Node

F) Repeat steps F) to L) in methodology 3.

Algorithm 1. DPSO-Based Task Assignment Scheme

```
% Input:Tasks;Particles
% Output:gBest;
for each particle Pi do
   pBesti= Generate_initial_position(Pi);
end for
   for the pBest of each particle Pi do
      gBest = Max(pBest1,pBest2...);
     end for
      while the termination condition is not met do
        Let j=1;
        while j≤ m do
            Select the task tj;
            calcuate_est(t_j^P);
            calculate_lst(t_j^B);
            allocate_ptask(t_j^P);
            allocate_btask(t_j^B);
            j++;
         end while
           for each particle Pi do
              Calculate cur_particle_fit;
              if cur_particle_fit<pBesti_fit then
                 update(pBesti);
              end if
               if cur_particle_fit<gBesti_fit then
                  update(gBest);
                  end if
                end for
                 if the termination condition is met then
                    output gBest;Break;
                  else
                    for each particle Pi do
                        update(Pi_position);
                        update(Pi_velocity);
                      end for
                    end if
                  end while
```

Fig 3.1; Pseudo-Code of dPSO Algorithms Based on Task Assignment Scheme (Guo *et al*, 2015b)

3.3.5 Calculation process of start execution time of the task's primary and backup copies

In (Guo *et al*,2015b) the model, start time of primary copy ought to be early schedule as could reasonably be expected while the start time of backup copy ought to be as late as viable, with the goal that it can give enough laxity for the corresponding backup task copy to receive passive (uninvolved) mode, primary copy and active copy of backup to overlap as slightly as possible, and along these lines enhance the usage of system resources. In this way, it is important to compute the earliest start time of tasks primary copy and the most recent start time of a tasks reinforcement copy.

3.3.6 Calculation process of the earliest start time of the task's primary copy

Let's assume there is a task t_j, the primary start time of t_j^P which is designated to node n_i can be calculated, we examine every node's idle time space $\left[0,S_1^{'}\right],\left[f_1^{'},S_2^{'}\right],\left[f_2^{'},S_3^{'}\right],...,\left[f_n^{'},+\infty\right]$ from left to right, where $S_i^{'}$ and $f_i^{'}$ stands for the start time and finish time of the *ith* task in the task queue of n_i, respectively. For predictable expression, let $f_0^{'}=0$. In the event that the first idle time space [$f_k^{'}$, $S_{k+1}^{'}$], which can meet $\max\left(a_j, f_k^{'}\right)+et_{j.i}\le d_j$, is discovered, the earliest start time of t_j^P on node n_i would be noted as $est_{ji}=f_k^{'}$, generally est_{ji} would be noted as $+\infty$.

3.3.7 Calculation process of the latest start time of the task's backup copy

Let us assume there is a task t_j, the primary start time of t_j^B which is apportioned to node n_i can be determined as follows:

Step 1. Initialize time $space=\left[d_j-et_{ji},d_j\right]$, for comfort, the start time and finish time of space can be set apart as s_start and s_finish, i.e. $space=\left[s_begin,s_find\right]$;

Step 2. Scan the task queue of n_i. If *slot* is situated in its idle times, let lst_{ji} be s_start and after that the procedure is finished. Else, it implies that space overlaps with other task in the line, hence, stamp the overlapped task as t_x for further consideration. If t_x is a passive reinforcement copy, let lst_{ji} likewise be s_start and after that the procedure is finished. In any case, if t_x is a primary copy or an active reinforcement copy, update space with $s_finish =$ the start time of t_x and $s_start = s_finish - et_{ji}$. And after that if $s_finish < 0$, let lst_{ji} be $+\infty$ and the procedure is finished, else goes to Step 2 once more.

3.3.8 Allocation process of the task's primary copy

In this area, the work plans a task designation process for the primary task's copy. Accepting that a task primary duplicate to be assigned is t_j^P, the procedure considers task deadline limitation, and in addition every node's load, energy utilization and the failure proportion. Those elements are quantified and weighting collected by function of information standardization. Then plan a utility function $U^P(i,j)$ to quantify the comprehensive execution of every node processing t_j^P and allot t_j^P to a superior node (Guo *et al*, 2015b):

$$U^P(i,j) = wt_1 \times UB(i,j) + wt_2 \times UE(i,j) + wt_3 \times UR(i,j), \tag{3.6}$$

where wt_1, wt_2 and wt_3 are weight coefficient, $U^P(i,j)$ is the utility function of primary duplicate, the slighter the estimation of $U^P(i,j)$ is, the better n_i execute t_j^P completely. $UB(i,j)$, $UE(i,j)$ and $UR(i,j)$ signify the load degree, energy utilization degree and failure proportion level of n_i which executing t_j^P contrasted and different nodes which participate in t_j The present load b_i, energy utilization $ene_{j,i}$ and failure proportion λ_i of n_i are mapped in the scope of 0 and 0.5 by utilizing a comparable *sigmoid* function as information standardization

function to achieve $UB(i,j)$, $UE(i,j)$ and $UR(i,j)$. The specific estimation equations are presented by the following equations (Guo *et al*, 2015b):

$$UB(i,j) = \begin{cases} 0, & if \ \left(b_{\max} - b_{\min}\right) = 0 \\ \dfrac{1}{e^{\frac{x_{ji} \times b_i - b_{\min}}{b_{\max} - b_{\min} + 1}}} - 0.5 & else, \end{cases} \tag{3.7}$$

$$UE(i,j) = \begin{cases} 0, & if \ \left(ene_{\max} - ene_{\min}\right) = 0 \\ \dfrac{1}{e^{\frac{x_{ji} \times ene_{ji} - ene_{\min}}{ene_{\max} - ene_{\min} + 1}}} - 0.5 & else, \end{cases} \tag{3.8}$$

$$UR(i,j) = \begin{cases} 0, & if \ \left(\lambda_{\max} - \lambda_{\min}\right) = 0 \\ \dfrac{1}{e^{\frac{x_{ji} \times \lambda_i - \lambda_{\min}}{\lambda_{\max} - \lambda_{\min} + 1}}} - 0.5 & else, \end{cases} \tag{3.9}$$

where $b_{\max}$ and $b_{\min}$ indicate the high load and the lightest load of nodes which take an interest in t_j, separately, the smaller the $UB(i,j)$ is, the lighter the load of n_i is when executing t_j^P. $ene_{\max}$ and $ene_{\min}$ signify the biggest and the slightest energy utilization of nodes which take an interest in t_j, separately. The smaller the $UE(i,j)$ is, the less the energy utilization of n_i is

when executing t_j^P . $\lambda_{\max}$ and $\lambda_{\min}$ mean the biggest and the minimum failure proportion of nodes which take an interest in t_j, separately. The specific allotment procedure of a task's primary copy can be presented by the two steps:

Step 1. For every node n_i which takes part in t_j, calculate the total of est_{ji} and et_{ji}, foresee the time required of t_j^P to be finished on node n_i. In the event that the deadline requirement of t_j is met, $U^P(i,j)$ will be calculated by equation (3.6). Generally, $U^P(i,j)$ will be noted $+\infty$ until each one of those nodes have been taken into consideration.

Stage 2. Select a node with the less $U^P(i,j)$, and afterward allot t_j^P to the chose node and update its comparing $U^P(i,j)$ and $UB(i,j)$ for next estimation convenience.

3.3.9 Allocation process of the task's backup copy

Usually, the passive reinforcement copy is not executed, accordingly, just failure proportion and load ought to be considered to quantify nodes utility for the passive reinforcement duplicate, while failure proportion, load, and energy utilization ought to be considered for the dynamic/active reinforcement copy. In this manner, the research designs another utility function, which can treat passive and dynamic reinforcement copy without contrast. By utilizing information standardization function, the execution can likewise be quantified and weighting collected to determine extensive execution of every node and allot the reinforcement copy to a superior node.

Step 1. There exists a reinforcement copy t_j^B . For every node n_i which takes part in t_j, compute the total of lst_{ji} and et_{ji} to determinate the perform method of reinforcement copy and forecast the completion time of t_j^B on node n_i. In the event that the deadline requirement

of t_j^B can be met, $U^B(i,j)$ will be computed using equation (3.11), else, $U^B(i,j)$ will be observed as $+\infty$ pending when all nodes are taken into account.

Step 2. Choose a node with the least value of $U^B(i,j)$, assign t_j^B to the chosen node and update the relating $U^B(i,j)$ and $UB(i,j)$ (Guo *et al*, 2015b):

$$UE'(i,j) = \begin{cases} 0, \quad if \quad \left(ene'_{max} - ene'_{min}\right) = 0 \\ \dfrac{1}{e^{\frac{x_{ji} \times S\left(t_j^B\right) \times ene'_{ji} - ene'_{min}}{ene'_{max} - ene'_{min} + 1}}} - 0.5 \quad else, \end{cases} \tag{3.10}$$

$$U^B(i,j) = wt_1 \times UB(i,j) + wt_2 \times UE'(i,j) + wt_3 \times UR(i,j), \tag{3.11}$$

where wt_1 , wt_2 and wt_3 are weight coefficients, $U^B(i,j)$ is the utility function of reinforcement copy. The slighter the quantity of $U^B(i,j)$ is, the better n_i execute t_j^B widely. ene'_{min} and ene'_{max} represent the minimum and the maximum energy utilization of the nodes which take part in t_j, correspondingly. ene'_{min} and ene'_{max} can be estimated by the following equation (Guo *et al*, 2015b):

$$ene'_{min} = \min_{i=1}^{n} \left(x_{ji} \times S\left(t_j^B\right) \times ene_{ji} \times per_{ji} \right) \tag{3.12}$$

$$ene'_{max} = \max_{i=1}^{n} \left(x_{ji} \times S\left(t_j^B\right) \times ene_{ji} \times per_{ji} \right) \tag{3.13}$$

3.4 The Modification of the Real-time Fault-tolerant Task Allocation Scheme

The modification of the real-time fault tolerant task allocation scheme is carried out by utilizing the active replication techniques for the backup systems. In the active replication backup scheme both the primary and the backup copies of tasks are processed concurrently, this takes care of the time for waiting and listening for the failure of the primary task copy. The waiting and the listening time of the backup copy is the time of laxity. The modification is removing the laxity of the backup task copy.

3.4.1 Allocation process of the task's backup copy

The research designs a utility function, which can treat active reinforcement copy without contrast. By utilizing information standardization function, the execution can likewise be quantified and weighting collected to determine extensive execution of every node and allot the reinforcement copy to a superior node.

Step 1. There exists a reinforcement duplicate t_j^B . For every node n_i which takes part in t_j, compute the total of lst_{ji} and et_{ji} to determinate the perform method of reinforcement copy and forecast the completion time of t_j^B on node n_i. In the event that the deadline requirement of t_j^B can be met, $U^B(i, j)$ will be computed using equation (3.15), else, $U^B(i, j)$ will be observed as $+\infty$ pending when all nodes are taken into account.

Step 2. Choose a node with the least value of $U^B(i, j)$, assign t_j^B to the chosen node and update the relating $U^B(i, j)$ and $UB(i, j)$ (Guo *et al*, 2015b):

$$UE^{'}\left(i,j\right)=\begin{cases}0, & if \quad \left(ene^{'}_{\max}-ene^{'}_{\min}\right)=0\\ \dfrac{1}{e^{\frac{x_{ji}\times S\left(t_j^B\right)\times ene^{'}_{ji}-ene^{'}_{\min}}{ene^{'}_{\max}-ene^{'}_{\min}+1}}}-0.5 & else,\end{cases} \tag{3.14}$$

$$U^{B}\left(i,j\right)=wt_1\times UB\left(i,j\right)+wt_2\times UE^{'}\left(i,j\right)+wt_3\times UR\left(i,j\right) \tag{3.15}$$

where wt_1 , wt_2 and wt_3 are weight coefficients, $U^B\left(i,j\right)$ is the utility function of reinforcement copy. The slighter the quantity of $U^B\left(i,j\right)$ is, the better n_i execute t_j^B widely. $ene^{'}_{\min}$ and $ene^{'}_{\max}$ represent the minimum and the maximum energy utilization of the nodes which take part in t_j, correspondingly. $ene^{'}_{\min}$ and $ene^{'}_{\max}$ can be estimated by the following equation (Guo *et al*, 2015b):

$$ene^{'}_{\min}=\min_{i=1}^{n}\left(x_{ji}\times S\left(t_j^B\right)\times ene_{ji}\times per_{ji}\right) \tag{3.16}$$

$$ene^{'}_{\max}=\max_{i=1}^{n}\left(x_{ji}\times S\left(t_j^B\right)\times ene_{ji}\times per_{ji}\right) \tag{3.17}$$

where $S\left(t_j^B\right)=1$ not as in the case of the passive replication techniques where it is equal to zero

3.4.2 The flowchart of the modified real-time fault tolerant task allocation scheme

In this research, dPSO method is also employed in a mRFTAS scheme in a WSNs framework. The scheme flow chart is represented in Fig. 3.2. The Fig. 3.2 shows that the sink node at first collects the processed tasks, it then generates the velocity, the position in parallel for a single generation of the dPSO. After which, series of operations begins. The terminal conditions are checked to know if the conditions are met, then it begins with the next iteration or else, tasks are published and waits Also Fig. 3.2 right part is the process of execution of the task, where both primary and backup task are runs concurrently on the separate sensor node. In Fig. 3.2 there is no waiting time for the primary task to fail first before running the backup copy of the task.

Also, when one of the task is completed, either the primary or the backup copy a query is sent to stop the corresponding copy from executing.

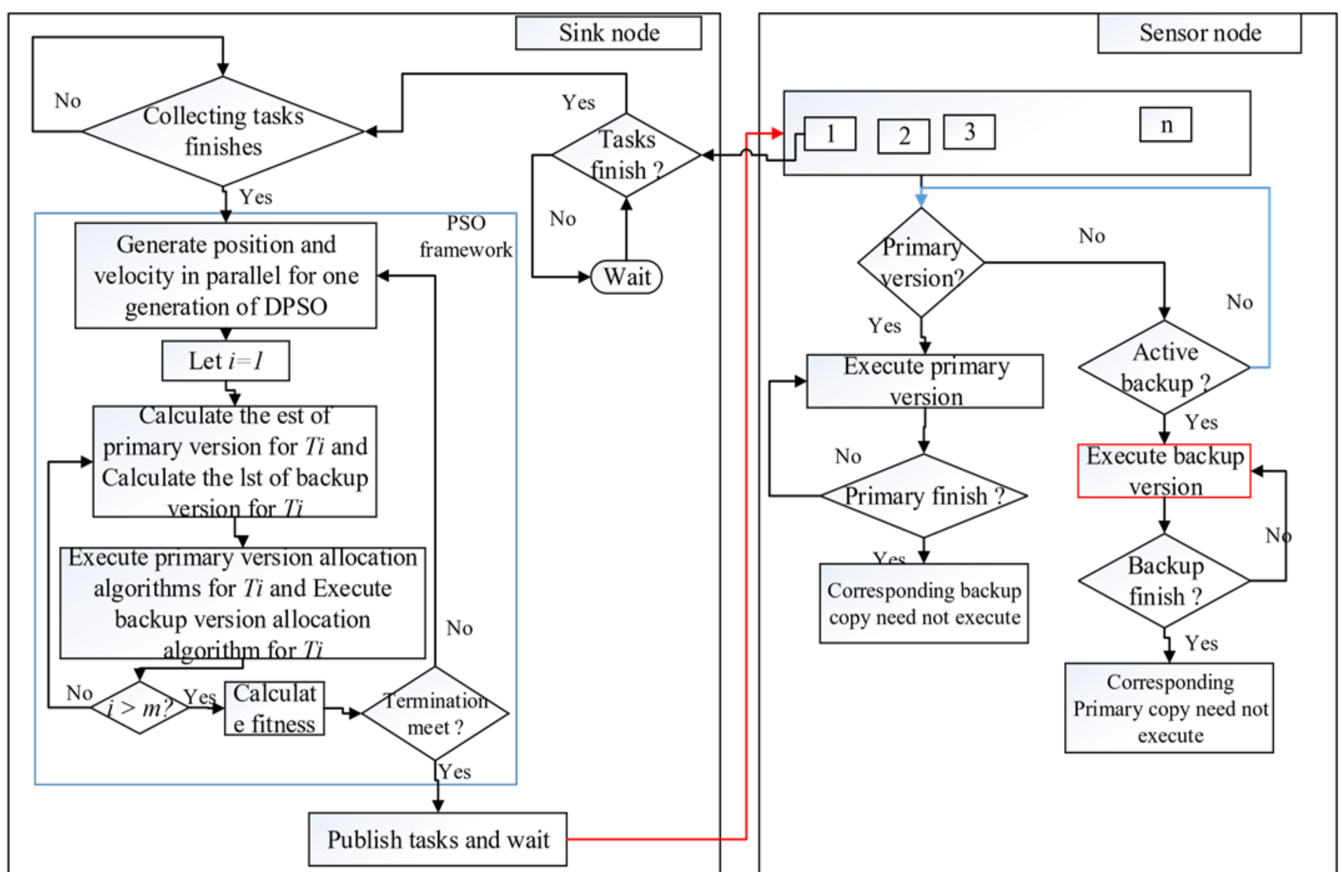

Fig. 3.2: Flow Chart of mRFTAS

3.5 Development of the GUI for the Wireless Sensor Network Simulations

The developed GUI for the Wireless Sensor Network simulation is known as Tasks allocation schemes simulator.

3.5.1 Wireless sensor networks simulator

The research built up a simulating system to address the issue real-time fault tolerant task assignment. It incorporates node deployment, node discovery, energy management mechanism, and the routing protocol.

The research simulator was designed and implemented using C# programming language to simulate the sensor node network which comprised of interconnected sensor nodes exchanging data over the wireless communication channel. The sensor (*n*) nodes were placed random but uniformly in a rectangular region of 100 by 100 meters.

The system designed using a specified number of nodes in the network, the initial energy, every node arraigned as:

1. X dimension of the map (for randomly placing nodes)
2. Y dimension of the map (for randomly placing nodes)

Sensor node were, added randomly to the network (keep sorted by x, and make sure no other nodes with this x/y coordinate are in the list) this provides an unambiguous metric for establishing routing connections (from nodes earlier in the list to nodes further down the list)

1. **Deployment of Node:** this is carried out in a random manner. To cover a vast area, instead of deployment of a nodes at a given moment, an irregular organization technique is thus represented by:

```
//
// This module contains all of the code for the simulated wireless network.

#region Using declarations

using System;
using System.Collections;
using System.Threading;

#endregion

namespace Wireless_Sensor_Network_Simulator {

        public class WirelessSensorNetwork {
                // This class represents the entire wireless sensor network.

                #region Variables and initialization code

                // network data structures
                public ArrayList aSensors;
                                                                            // array of sensors (network
nodes)
                public ArrayList aRadar;
                                                                                    // array of packets that
have reached the radar
                public VectorList vectors;
                                                                            // list of vectors
                // network characteristics
                public int iNetworkSize =10;
                                                                    // number of nodes in the network
                public static int iMaxEnergy = 1000;
                                        // the initial energy of every node
                public int iMaxX = 460;
                                                                                    // max X dimension of
the map (for randomly placing nodes)
                public int iMaxY = 300;
                                                                                    // max Y dimension of
the map (for randomly placing nodes)
```

Fig 3.3: Code for Deployment of Nodes

2. **Discovery of Node:** This protocol communicates by a broadcast "hello" message, if a node is inside the transmitting/broadcast limit, it would accept the message and send a feedback reaction. The node that started the message will then add the recognizing node to a rundown list of neighbors. At the point when a node completes a broadcast and gets no affirmation, the node will expand the broadcasting radius until negligible numbers of neighbors are found. The nodes utilize a von Neumann Neighborhood to check out the deployment framework. To guarantee that a point in the network is in the range, the distance equation is given by an algorithm. The distance equation utilizes the Pythagorean Theorem to decide the separation of two distinct nodes. The algorithm utilizes the x-y directions of the broadcasting node and a neighboring node to decide the sides of the triangle. The hypotenuse is then calculated. In the event that the hypotenuse does not surpass the nodes broadcasting radius and the node is right now

wakeful then the node is a substantial neighbor. That node is then added to the rundown list of neighbors of the broadcasting node. This procedure is repeated for all the nodes not in sleep mode, in the simulation.

3. **Utilization and Power Production:** The fundamental cause of energy loss by a node is because of sending and accepting messages. Loss of energy because of the number of messages being sent can be simulated utilizing Equation 3.18, gotten from the nodes broadcasting radius and the energy loss per message sent.(Marsal *et al.*, 2014)

$$\text{Energy} = \text{Energy} - \frac{\text{BroadcastingRange} \times \text{Energy_Per_Msg}}{100} \tag{3.18}$$

Loss of energy is more when delivering a package, however accepting a message makes energy to be utilized. Equation 3.19 exhibits a formula to usage in the energy loss because of a message being accepted. In the equation, just a little fraction of the measure of the energy per message is being lost. Different types of energy loss come as computation and node overhead. To represent this kind of loss, another equation is utilized. Equation 3.20 indicates the amount of energy being lost because of overhead per time period. (Marsal *et al.*, 2014)

$$\text{Energy} = \text{Energy} - \frac{\text{Energy_Per_Msg}}{100} \tag{3.19}$$

$$\text{Energy} = \text{Energy} - \text{Energy_Per_Step} \tag{3.20}$$

At any point the energy level of a node drops past a specific threshold, the node will enter a rest mode. The node will stay in the rest mode till it recovers enough energy to start functioning again. Energy in a node is delivered essentially using solar panels. The measure of energy that is being delivered is constrained and is reliant on the node's environment. By

utilizing a normal node energy creation formula in equation 3.21 one can evaluate the measure of energy delivered at each time allotment. (Marsal *et al.*, 2014)

$$\text{Energy} = \text{Energy} + \text{Energy_Produced} \tag{3.21}$$

At each time allotment, the node will lose energy from messages being sent and received, then in addition to node overhead. While the measure of energy that the node is delivering won't be sufficient to allow the node to run continually, appropriate energy management allows the node to extend its life, thus enabling the network to keep operating.

3.5.2 Programming

The simulator was composed in C# utilizing Microsoft Visual Studio.NET 2015 professional. The application is in two sections: one module for the WSN, the other is the simulator that has the WSN objects. The classes that involve the WSN are Wireless Sensor, iSensor Radius, iSensor Delay, aPackets, aConnections, Connection Current, iResidual Energy, Wireless Sensor Connection, sSender and sReceiver, Packet, Wireless Sensor Network, VectorList and Vector. This simulator supports a tremendous networks 400x400 nodes. Simulating transmissions over these connections, and then drawing the whole network several times per second, needs a non-trivial deal of processing. The CPU of a standard 3.2GHz machine isn't sufficient for this task with the message pump that controls window occurrences; henceforth, packing the two functions into one thread brings about a totally, unresponsive simulator window. Rather, the simulation thread is begun in a different thread, which runs consistently until the point that the client stops it (clicking Stop Button).

3.5.3 Wireless sensor network simulation

This application is the simulation of the WSN. The network is deployed in light of parameters: Size of network (number of nodes), Transmitting/communication range, energy costs for transmitting and accepting messages. The Network was then be utilized to simulate the

discovery of vectors traversing the sensor network field. In the simulation, when a vector trips the sensor of a system node, the node generates an information packet and sends it to a downstream system node. The packets are routed suitably until the point that they reach a sensor in the "uplink zone". Every node additionally simulates an energy store, which is exhausted by sending and accepting packets, by recognizing vectors. Since the nodes have limited energy, they will at long-run drop out of the communications network because of shut down, causing system failure.

The simulation comprises of two phases: network deployment and running simulations. Prior to network deployment, the properties of the system ought to be set by utilizing the configuration sliders. The network setup properties are assembled into two classifications.

1. **The Configuration of Network:** these components decide the hardware properties of the system. The accompanying factors can be configured as, Network Size, Sensor Radius, Sensor Period, Sensor Cost, Transmission Radius, Transmitter Period, Transmit Cost, Receive Cost.
2. **Routing Parameters:** These variables decide the software properties of the system: basically, the packet-routing strategy to be utilized. If routing is set to "Arbitrary," every node chooses a downstream connection arbitrarily for every packet.

3.6 Comparisons of the Performance of mRFTAS AND RFTAS

A comparison in term of the performance of the two schemes (i.e mRFTAS and RFTAS) was carried out, this was done such as to discover the performance of each scheme. The percentage improvement from the performance of mRFTAS over that of RFTAS was obtained by using:

$$\text{Performace Improvement} = \frac{\text{Parameter_mRFTAS - } \textit{Parameter}\text{_RFTAS}}{\text{Parameter_RFTAS}} \times 100\% \qquad (3.22)$$

The following are the parameters considered; network lifetime, reliability cost, energy consumption and execution time.

CHAPTER FOUR
RESULTS AND DISCUSSIONS

4.1 Introduction

In this section, the performance of the real-time fault-tolerant task allocation scheme and that of the modified real-time fault-tolerant task allocation scheme of the WSNs, with processing time delay and fault-tolerant problem are discussed and appropriate results reported.

4.2 Task Allocation Scheme Simulator

The interface of the developed TASS, which is used for all the simulations, is as shown in Fig. 4.1. A typical simulation scenario is as shown in Fig. 4.2.

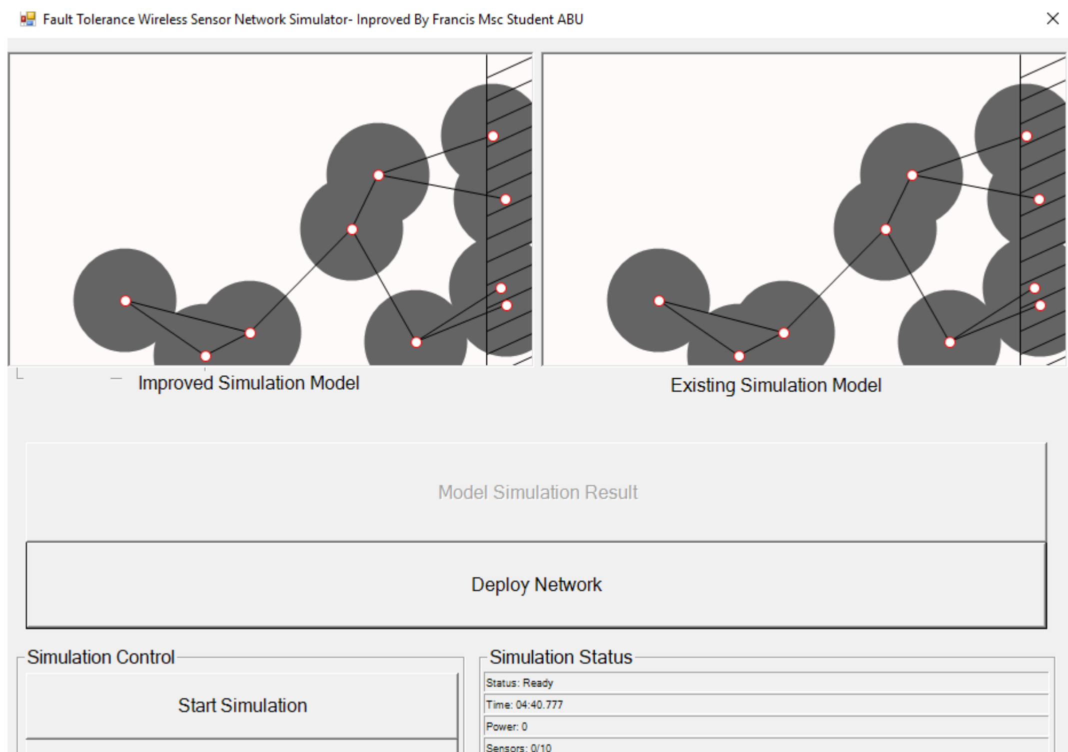

Fig. 4.1: Nodes Deployment of mRFTAS and RFTAS using TASS

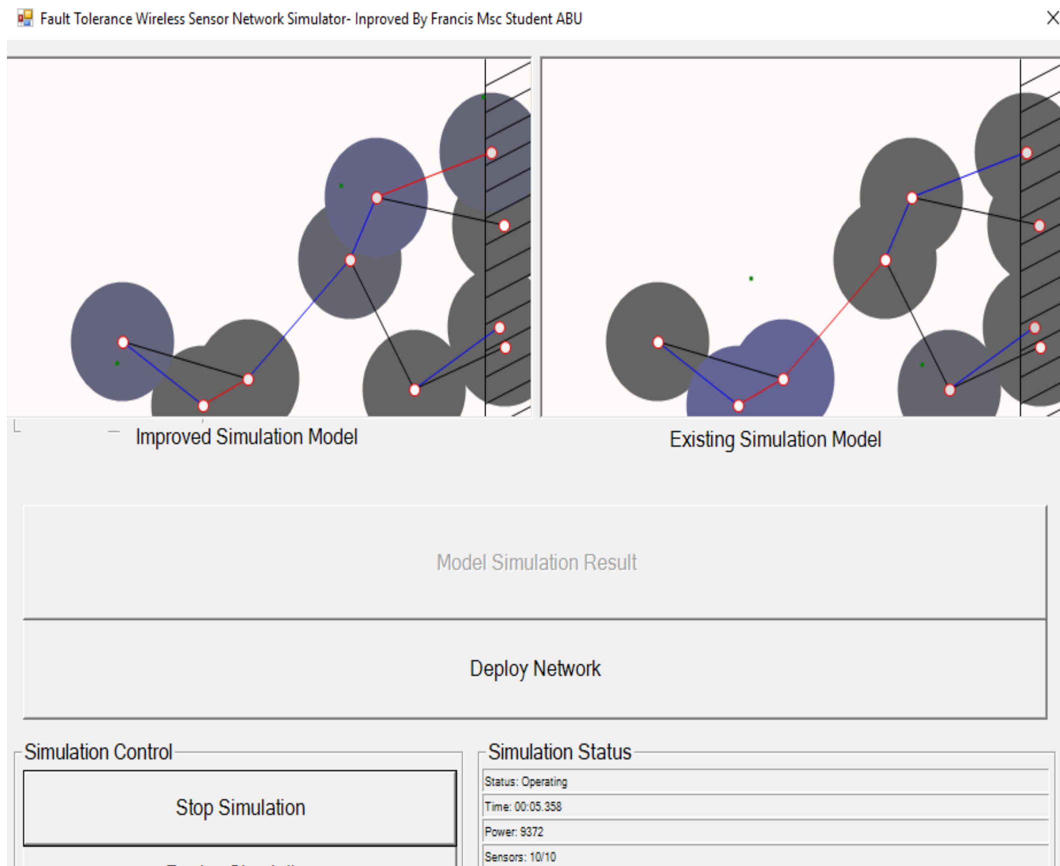

Fig. 4.2: Simulation Scenario using TASS

4.3 Result of the Analyses of the Real-time Fault-tolerant Task Allocation Scheme

The plots obtained from the simulation of the RFTAS for WSNs using the passive replication technique as the backup systems are as shown in Figs 4.3 to 4.6. The RFTAS was implemented on the developed GUI simulator.

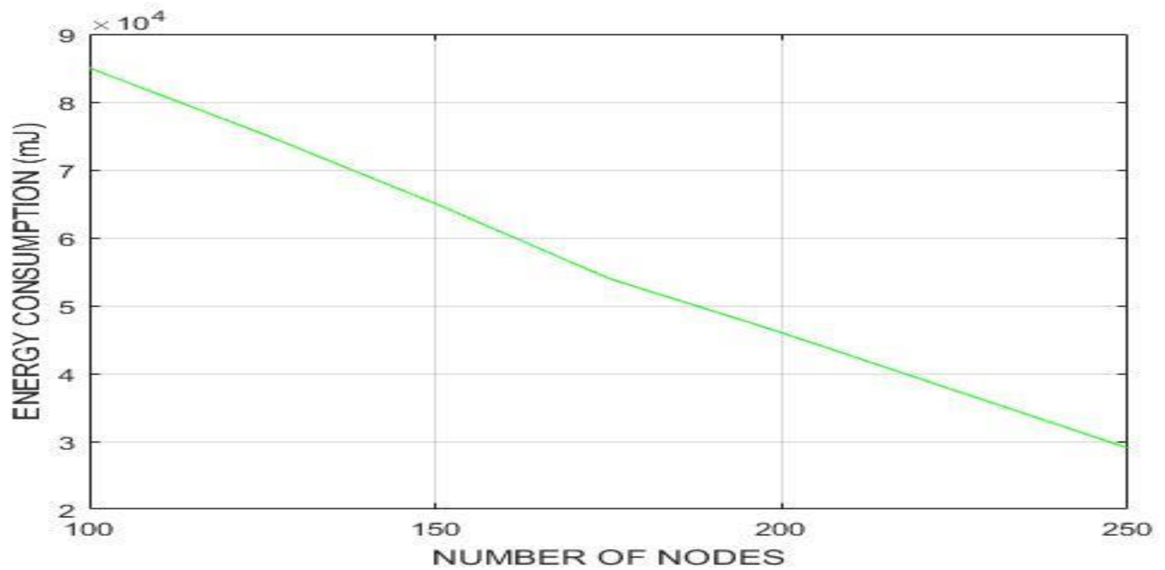

Fig. 4.3: Plot of Energy Consumption for RFTAS

Fig. 4.3 is the plot of energy consumption against the number of nodes. It can be seen that the more the number of sensor nodes employed for the execution of the 400 tasks, the lesser the energy consumed by the nodes.

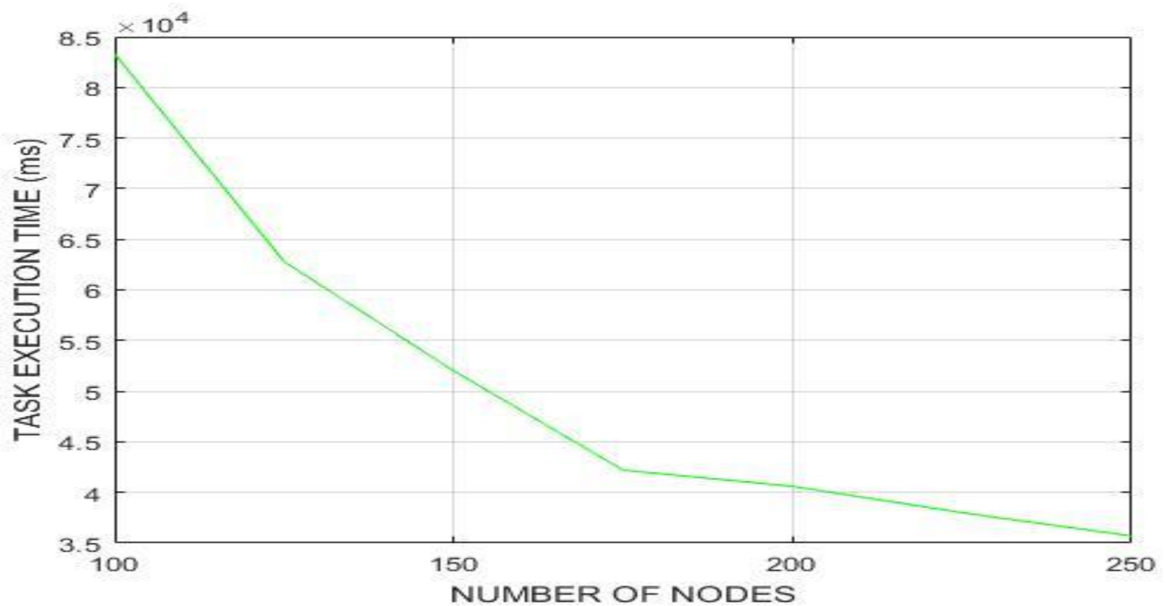

Fig. 4.4: Plot of Task Execution Time for RFTAS

Fig. 4.4 is the plot of execution time against the number of nodes. It can be seen that the more the number of sensor nodes employed for the execution of the 400 tasks, the lesser the time required by the nodes.

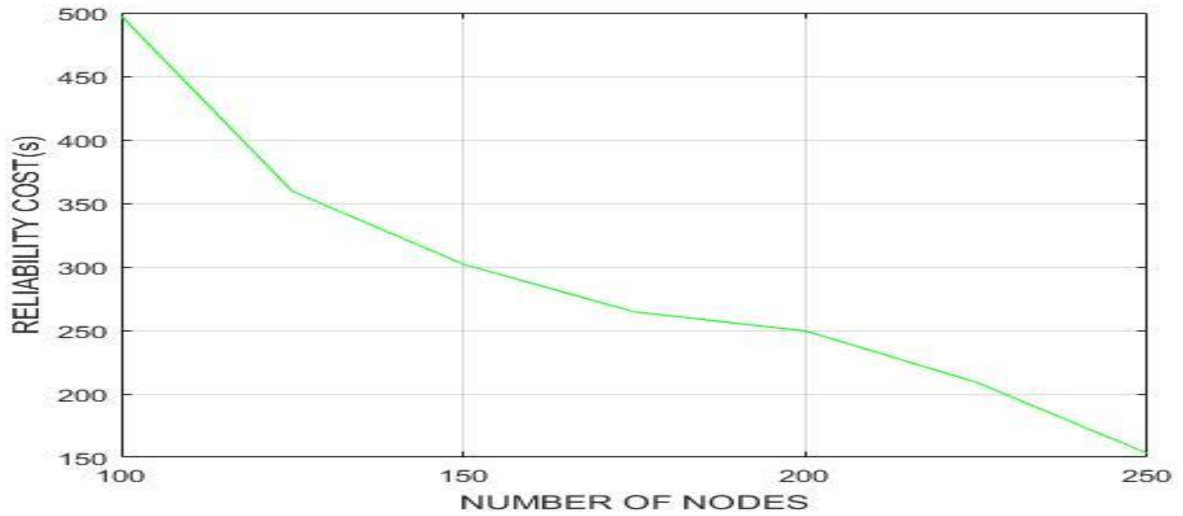

Fig. 4.5: Plot of Reliability Cost for RFTAS

Fig. 4.5 is the plot of reliability cost against the number of nodes. It can be seen that the more the number of sensor nodes employed for the execution of the 400 tasks, the lesser the reliability cost associated with the nodes.

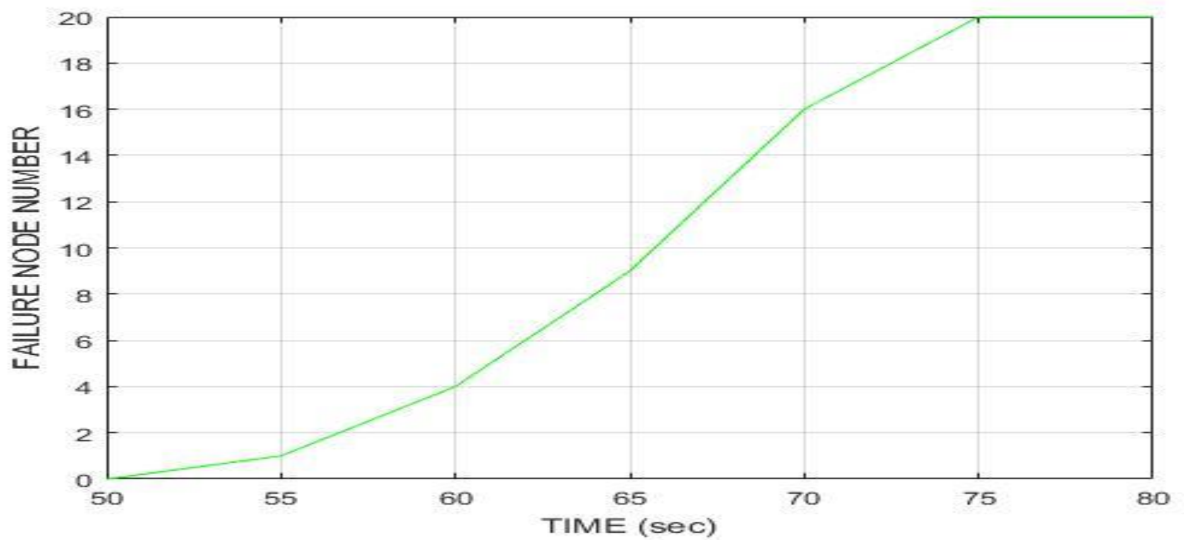

Fig 4.6: Plot of Network lifetime for RFTAS.

Fig. 4.6 is the plot of number of failed nodes against the execution time (s). It can be seen that the more the time taken for the execution of the 400 tasks, the more the number of failed nodes. This affects performance and inevitably network lifetime.

4.4 Result of the Analyses of the Modified Real-time Fault-tolerant Task Allocation Scheme

The plots obtained from the simulation of the mRFTAS for WSNs using the active replication technique as the backup systems are as shown in Figs 4.7 to 4.10. The mRFTAS was implemented on the developed GUI simulator.

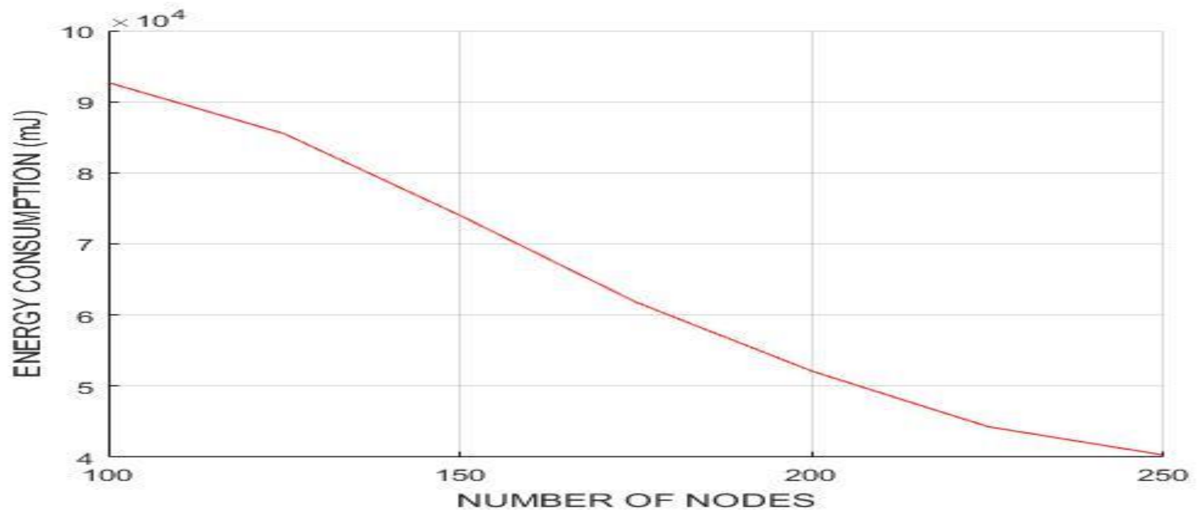

Fig. 4.7: Plots of Energy Consumption for mRFTAS

Fig.4.7 is the plot of energy consumption against the number of nodes. It can be seen that the more the number of sensor nodes employed for the execution of the 400 tasks, the lesser the average energy consumed by the nodes.

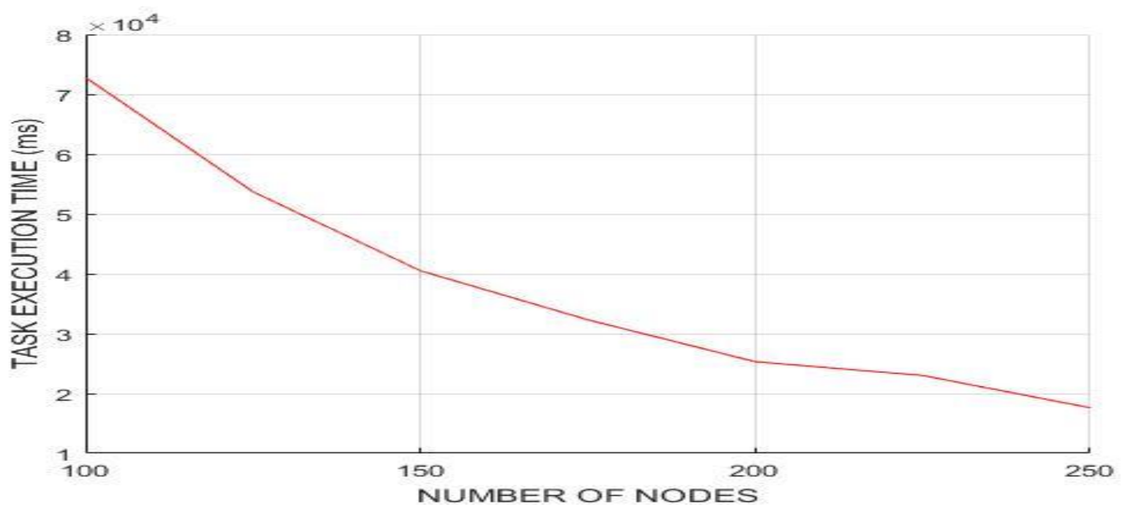

Fig. 4.8: Task Execution Time for mRFTAS.

Fig.4.8 is the plot of execution time against the number of nodes. It can be seen that the more the number of sensor nodes employed for the execution of the 400 tasks, the lesser the average time required by the nodes.

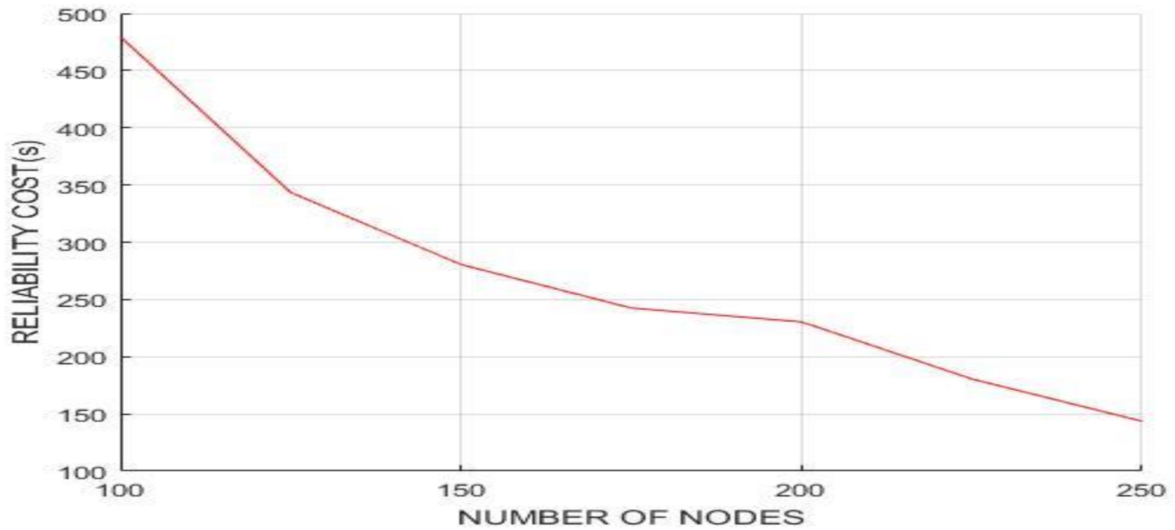

Fig. 4.9: Plot of Reliability Cost for mRFTAS

Fig.4.9 is the plot of reliability cost against the number of nodes. It can be seen that the more the number of sensor nodes employed for the execution of the 400 tasks, the lesser the reliability cost associated with the nodes.

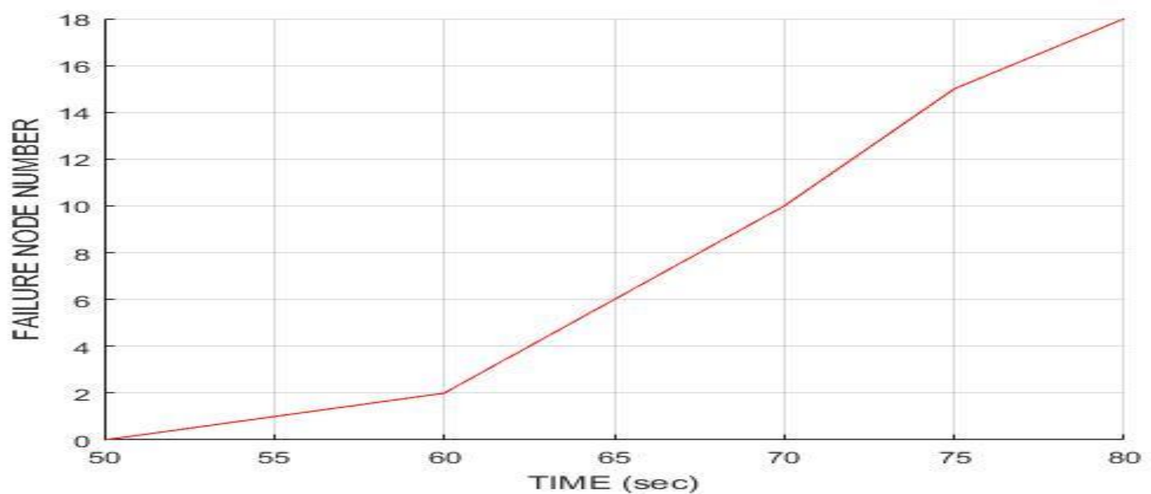

Fig. 4.10: Plot of Network lifetime for mRFTAS.

Fig. 4.10; is the plot of the number of failed nodes against the execution time (s). It can be seen that the more the time taken for the execution of the 400 tasks, the more the number of failed nodes. This affects performance and inevitably network lifetime.

4.5 Result of the Comparative Analyses of the Performance of both the mRFTAS and RFTAS

The plots obtained from the simulation of the RFTAS and mRFTAS for WSNs are compared as shown in Figs 4.11 to 4.14.

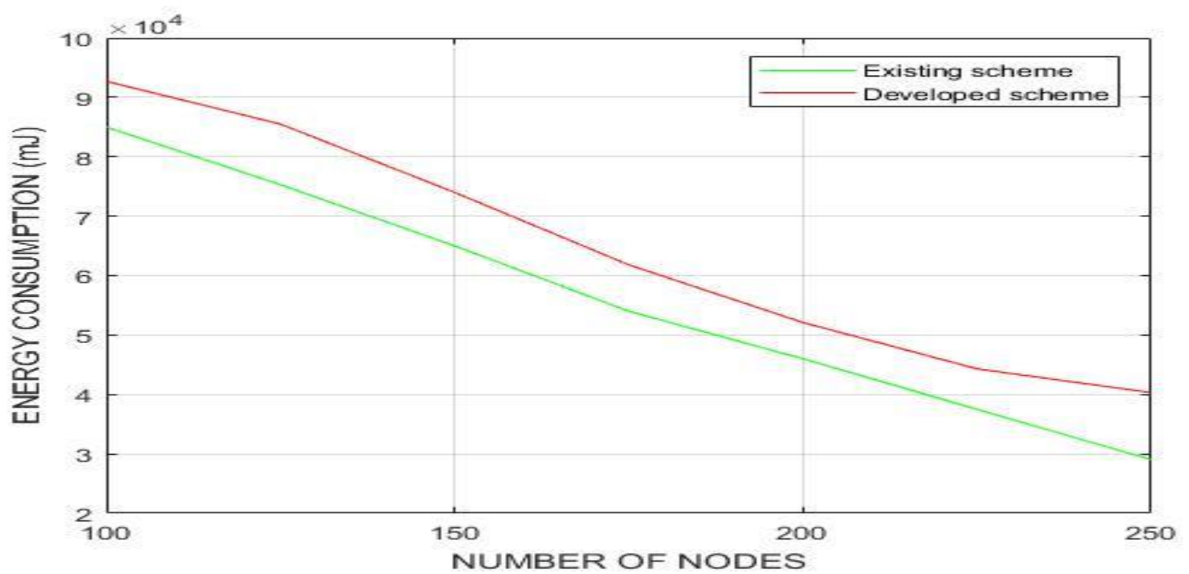

Fig. 4.11: Plot Energy Consumption for mRFTAS and RFTAS

Fig. 4.11 is the comparative plot of energy consumption against the number of nodes for the RFTAS and mRFTAS. It can be seen that the more the number of sensor nodes employed for the execution of the 400 tasks, the lesser the average energy consumed by the nodes. It is also evident that the energy consumed by the RFTAS is less than that by mRFTAS. The energy consumed by mRFTAS for the task allocation was more than the energy consumed by RFTAS, reason being mRFTAS task execution is carried out using double copies of task whereas RFTAS only executes one copy of task.

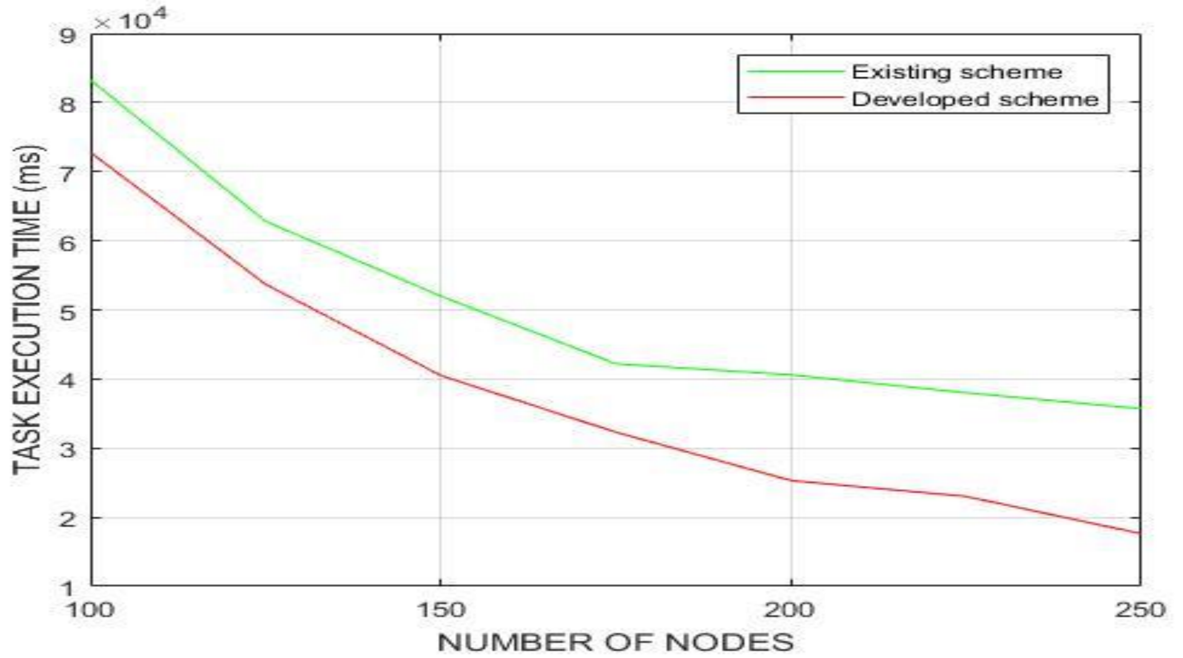

Fig. 4.12: Plot of Task Execution Time for mRFTAS and RFTAS.

Fig. 4.12 is the comparative plot of execution time against the number of nodes for the RFTAS and mRFTAS. It can be seen that the more the number of sensor nodes employed for the execution of 400 tasks, the lesser the average time required by the nodes. It is also evident that the execution time required by the mRFTAS is less than that required by the RFTAS

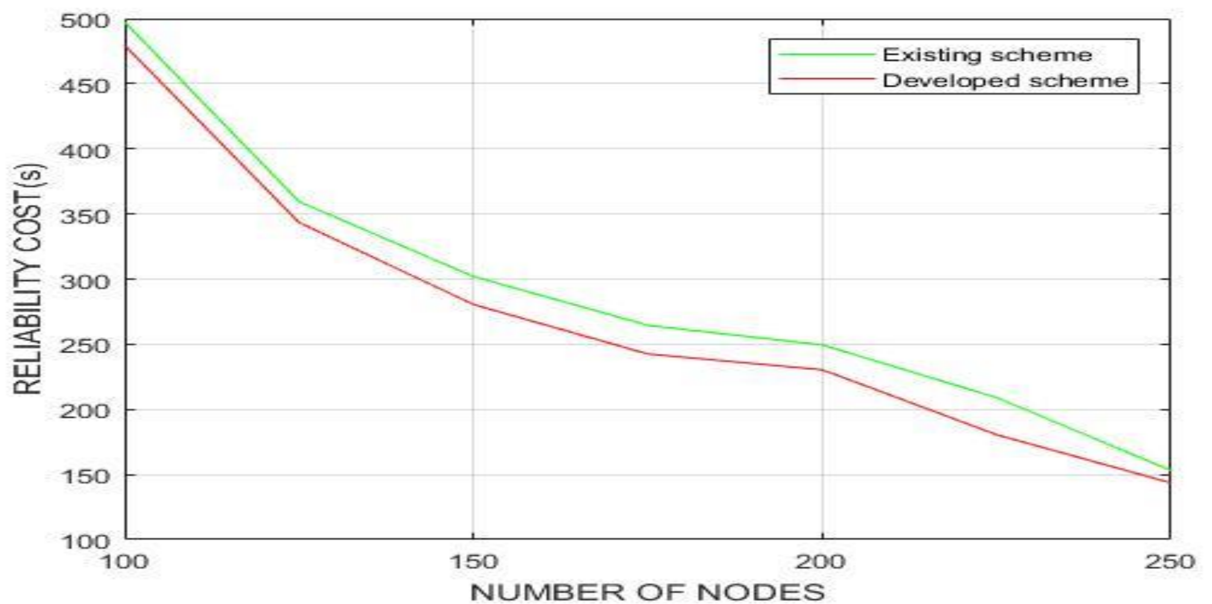

Fig. 4.13: Plot of Reliability Cost for mRFTAS and RFTAS

Fig. 4.13 is the comparative plot of reliability cost against the number of nodes for the RFTAS and mRFTAS. It can be seen that the more the number of sensor nodes employed for the execution of 400 tasks, the lesser the reliability cost associated with the nodes. It is also evident that the reliability cost associated with the mRFTAS is less than that required by the RFTAS.

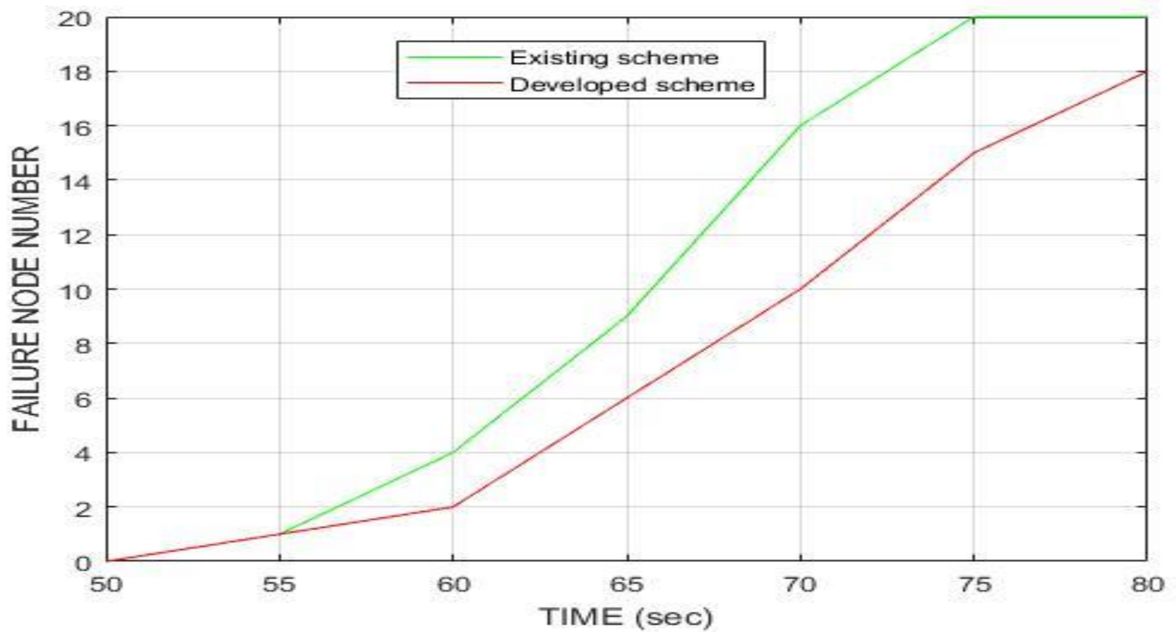

Figure 4.14: Plot of Network Lifetime for mRFTAS and RFTAS.

Fig. 4.14 is the comparative plot of the number of failed nodes against the execution time (s) for the RFTAS and mRFTAS. It can be seen that the longer it takes to execute the 400 tasks, the more the number of failed nodes. This affects performance and inevitably network lifetime. It is also evident that the mRFTAS had lesser number of failed nodes recorded and invariably a longer network lifetime as compared to the RFTAS.

4.6 Result Performance Percentage Improvement of the mRFTAS over RFTAS

The Tables 4.1 to 4.4, showed results obtained for performance between RFTAS and mRFTAS. The performance improvements are evaluated using equation (3.22).The tabulated results is the average value obtained from 10 runs of simulations for each range of sensor nodes number and for the metrics considered

Table 4.1 Performance Comparison for Energy Consumption

Nodes Number	RFTAS (mJ)	mRFTAS (mJ)	% improvement
100	8.50×10^4	9.27×10^4	-9.06
125	7.53×10^4	8.55×10^4	-13.55
150	6.50×10^4	7.40×10^4	-13.85
175	5.40×10^4	6.18×10^4	-14.44
200	4.60×10^4	5.21×10^4	-13.26
225	3.75×10^4	4.43×10^4	-18.13
250	2.90×10^4	4.03×10^4	-38.97

The performance improvement for energy consumption obtained is -17.32% since the RFTAS outperformed mRFTAS.

Table 4.2: Performance Comparison for Task Execution Time

Nodes Number	RFTAS(ms)	mRFTAS (ms)	% improvement
100	8.33×10^4	7.28×10^4	12.61
125	6.28×10^4	5.37×10^4	14.49
150	5.20×10^4	4.05×10^4	22.12
175	4.22×10^4	3.23×10^4	23.46
200	4.06×10^4	2.53×10^4	37.69
225	3.80×10^4	2.30×10^4	39.47
250	3.57×10^4	1.76×10^4	50.70

The performance improvement for the execution time obtained is 28.65% since mRFTAS outperformed RFTAS.

Table 4.3: Performance Comparison for Reliability Cost

Nodes Number	RFTAS (s)	mRFTAS (s)	% improvement
100	497.50	479.50	3.62
125	359.50	343.50	4.45
150	302.00	280.50	7.12
175	264.50	242.50	8.32
200	249.50	230.40	7.66
225	209.00	180.50	13.64
250	153.00	143.50	6.21

The performance improvement for reliability cost obtained is 7.29% since the mRFTAS outperformed the RFTAS.

Table 4.4: Performance Comparison for Network Lifetime

Time (sec)	RFTAS (number of failed nodes)	mRFTAS (number of failed nodes)	% improvement
50	0	0	0.00
55	1	1	0.00
60	4	2	50.00
65	9	6	33.33
70	16	10	37.50
75	20	15	25.00
80	20	18	10.00

The performance improvement for network lifetime obtained is 22.26% since mRFTAS outperformed RFTAS.

CHAPTER FIVE
CONCLUSION AND RECOMMENDATIONS

5.1 Summary

RFTAS is a task allocation scheme that was modelled based on the principle of the fault tolerance ability of a system (such as the WSN). This is to ensure that the system keeps processing even when there is sudden failure of a component (e.g. node of the WSN) of the system. The RFTAS was developed based on the issues associated with real-time task allocation in WSNs using the backup copies techniques. The passive replication backup technique, which is the most commonly used technique for the RFTAS, has the problem of processing time delay. This is as a result of the delay arising from the backup task copy activation. Processing time delay in systems that are safety- or security-critical can be inimical to lives and properties. In order to overcome the problems associated with the RFTAS in terms of fault tolerance and processing time delay, the mRFTAS using active replication backup copy technique was developed.

A GUI-based simulator called the TASS was developed using Visual Studio 2015 and simulations involving the RFTAS and mRFTAS were carried on task allocation involving 400 tasks amongst a different number of nodes in a WSN. Analyses were carried out on the performance of the RFTAS and the mRFTAS using the following metrics: energy consumption, task execution time, reliability cost and network lifetime. The results indicated that mRFTAS performed better in terms of reducing the task execution time and reliability cost while increasing the network lifetime by 28.65%, 7.29% and 22.26% respectively when compared with the RFTAS but with a trade-off in energy consumption (with -17.32% performance measure).

5.2 Conclusion

This research developed the mRFTAS using the active replication backup technique for addressing the real-time fault tolerance and processing time delay in WSNs by reducing the

task execution time and reliability cost and increasing the network lifetime but at the expense of energy consumption,

A GUI was developed for simulation of real-time task allocation using mRFTAS and RFTAS for WSNs called the TASS.

The comparisons of the performance of mRFTAS and RFTAS were carried out for energy consumption, task execution time, reliability cost and network lifetime. The mRFTAS outperformed the RFTAS in all the metrics except for energy consumption.

5.3 Significant Contributions

The significant contributions of this research work are as follows:

a) Development of a C# based GUI model for real-time fault-tolerant task allocation scheme (mRFTAS) for WSNs using active replication backup technology.

b) The mRFTAS produced a performance improvement of 28.65% over RFTAS for task execution time, 7.29% improvement in reliability cost and 22.26% improvement in network lifetime. The mRFTAS failed to perform better than the RFTAS in term of energy reduction, its performance was -17.32%

5.4 Recommendations for Further Work

The following possible areas of further work are recommended for consideration for future research:

a) The research conducted did not consider the presence of malicious nodes, thus further work can be carried over by implementing security.

b) The modified Real-time Fault-tolerant task allocation scheme can be further modified for minimization of energy consumption and task execution time in WSNs.

REFERENCE

Arar, C., & Khireddine, M. S. (2016). An Efficient Fault-Tolerant Multi-Bus Data Scheduling Algorithm Based on Replication and Deallocation. *Cybernetics and Information Technologies, Bulgarian Academy of Sciences (DOI: 10.1515/cait-2016-0021), Bulgarian. 16*(2), 69-84.

Augé-Blum, I., Yang, F., & Watteyne, T. (2011). Real-Time Communications in Wireless Sensor Networks *Next Generation Mobile Networks and Ubiquitous Computing; DOI: 10.4018/978-1-61350-101-6.ch107 ,IGI Global, France. 69-78.*

Balasangameshwara, J. (2015). Improving Fault-Tolerant Load Balancing Algorithms in Computational Grids. *International Journal of Information Engineering and Electronic Business, © 2015 MECS, India, 7*(6), 53-62.

Bröring, A., Echterhoff, J., Jirka, S., Simonis, I., Everding, T., Stasch, C., . . . Lemmens, R. (2011). New generation sensor web enablement. *Sensors, 11*(3), 2652-2699.

Chen, C., Guo, W., & Chen, G. (2012). *A new task allocation algorithm based on dynamic coalition in WSNs. Paper presented on Parallel and Distributed Processinga at the 26th International IEEE Symposium Workshops & PhD Forum (IPDPSW), © 2012 IEEE ,China.1243-1248*

Chouikhi, S., El Korbi, I., Ghamri-Doudane, Y., & Saidane, L. A. (2014). *Fault tolerant multichannel allocation scheme for wireless sensor networks.* Paper presented at the Wireless Communications and Networking Conference (WCNC), IEEE WCNC'14, Paris-France, 2014 (3), 2438-2443.

Duan, H., Zhou, Y., & Liu, M. (2017). Fault tolerant scheduling algorithm in distributed sensor networks. *Journal of Information Hiding and Multimedia SignalProcessing,Ubiquitous International, China. 8*(1), 127-137.

Gangadharaiah, S., Hallur, U. M., & Jamadar, S. S. (2014). Soft real time auction scheme for task allocation in wireless sensor networks. *International Journal of Research in Engineering and Technology (IJRET)*, India. 3(4), 274-280.

Guo, W., Xiong, N., Chao, H.-C., Hussain, S., & Chen, G. (2011). Design and analysis of selfadapted task scheduling strategies in wireless sensor networks. *Sensors, www.mdpi.com/journal/sensors, China. 11*(7), 6533-6554.

Guo, W., Chen, Y., & Chen, G. (2014). Dynamic Task Scheduling Strategy with Game Theory in Wireless Sensor Networks. *New Mathematics and Natural Computation,* @World Scientific Publishing Company DOI: 10.1142/S1793005714500124, Fujian 350108China, *10*(03), 211-224.

Guo, W. Z., Chen, J. Y., Chen, G. L., & Zheng, H. F. (2015a). Trust dynamic task allocation algorithm with Nash equilibrium for heterogeneous wireless sensor network. *Security and Communication Networks, © 2014 John Wiley & Sons, Ltd, Fuzhou-China. 8(10), 1865-1877.*

Guo, W., Li, J., Chen, G., Niu, Y., & Chen, C. (2015b). A PSO-optimized real-time faulttolerant task allocation algorithm in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems, 2014 IEEE-China. 26*(12), 3236-3249.

Han, Q. (2015). Energy-aware Fault-tolerant Scheduling for Hard Real-time Systems. FIU Electronic Theses and Dissertations. 2222. Florida International University, @ FIU Digital Commons, (Miami, Florida) USA, 2222, 1-163

Harkut, M., & Lohiya, P. (2014). Real-time scheduling algorithms for wireless sensor network. *Circuits and Systems: An International Journal (CSIJ)*, 1(1) 11-18.

Marsal, K. A., Abdullah, I., Ismai, W., & Rahim, K. A. (2014). Controlling Algorithm for Energy-Consumption, Radio Bandwidth and Signal Strength Deploying Single Fitness Function to Solve Coverage Area Problems. *Advanced Science Letters © 2011 American Scientific Publishers, United States of America. 20*(10-11), 2147-2151.

Matin, M., & Islam, M. (2012). Overview of wireless sensor network, *Wireless Sensor Networks-Technology and Protocols*: *InTech. http://dx.doi.org/10.5772/49376, Bangladesh. 2-24.*

Mei, L., Dao-ping, H., & Xiao-ling, X. (2010). Node task allocation based on pso in wsn multitarget tracking. *dvances in Information Sciences and Service Sciences, 2*(2.2), 13-18.

Presser, M., Barnaghi, P. M., Eurich, M., & Villalonga, C. (2009). The SENSEI project: Integrating the physical world with the digital world of the network of the future. *IEEE Communications Magazine, 47*(4), 1-4.

Priyanka, M., Anisha, S., & Sakthi Prabha, R. (2016). Vlsi Design for A Pso-Optimize Real-Time Fault-Tolerant Task Allocation Algorithm In Wireless Sensor Network; © Asian Research Publishing Network (ARPN), Chennai-India, 11,(13), 8226 – 8230.

Shi, H.-Y., Wang, W.-L., Kwok, N.-M., & Chen, S.-Y. (2012). Game theory for wireless sensor networks: a survey. *Sensors, 12*(7), 9055-9097.

Subalakshmi, M., & Helen, W. (2015). Information Recuperation by Assigning Task in Wireless Sensor Network. *Indian Journal of Science and Technology (www.indjst.org), India. 8*(13), 0974-5645

Suganya P., & N., J. (2016). Sensor Task Reassignment In Wireless Sensor Networks Using Bio-Inspired Algorithm, *International, Journal of Future Innovative Science and Engineering Research (IJFISER) Tamil Nadu, India, 2*(1), 217-226.

Tien, N. X., Kim, S., Rhee, J. M., & Park, S. Y. (2017). A novel dual separate paths (DSP) algorithm providing fault-tolerant communication for wireless sensor networks. *Sensors, 17*(8), 1699.

Vaishnavi, R. S., & Sukumar, M. (2015). Survey on Task Allocation using Particle Swarm Optimization in Wireless Sensor Network, *International Journal of Innovative Research in Computer and Communication Engineering,* IJIRCCE., India. *3*(11), 10705-10709.

Wang, F., Han, G., Jiang, J., Li, W., & Shu, L. (2015). A task allocation algorithm based on score incentive mechanism for wireless sensor networks. *International Journal of Distributed Sensor Networks, Hindawi Pubishing Corperation, China 2015(286589), 1-12.*

Yu, W., Huang, Y., & Garcia-Ortiz, A. (2017). Distributed Optimal On-line Task Allocation Algorithm for Wireless Sensor Networks. *IEEE Sensors Journal, DOI:10.1109/JSEN.2017.2768659, Germany. 18(1), 446-458.*

Zhang, J., & Long, J. (2017). An Energy-Aware Hybrid ARQ Scheme with Multi-ACKs for Data Sensing Wireless Sensor Networks. *Sensors, www.mdpi.com/journal/sensors, China 17*(6), 1366.

Zhu, X., Zhu, J., Ma, M., & Qiu, D. (2010). *QAFT: A QoS-Aware Fault-Tolerant Scheduling Algorithm for Real-Time Tasks in Heterogeneous Systems.* Paper presented on

Computational Science and Engineering (CSE), at the 13th International Conference of IEEE, © 2010 IEEE, *China.2010(20) 80-87*.

Zhu, X., Qin, X., & Qiu, M. (2011). QoS-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters. *IEEE transactions on Computers,* 2011 IEEE Computer Society,*China.* *60*(6), 800-812.

APPENDIX

Appendix A: WSNs Simulator/GUI code

```
#region Using declarations

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Threading;

#endregion

namespace Wireless_Sensor_Network_Simulator {

        public class WirelessSensorNetworkSimulator : System.Windows.Forms.Form {

                #region Variables, window management code, and Main

                private System.Windows.Forms.GroupBox groupBox1;
                private System.Windows.Forms.GroupBox groupBox2;
                private System.Windows.Forms.Button btnDeploy;
                private System.Windows.Forms.Button btnStart;
                private System.Windows.Forms.Label label3;
                private System.Windows.Forms.Label label4;
                private System.Windows.Forms.Label label6;
                private System.Windows.Forms.PictureBox picNetwork;
                private System.Windows.Forms.TrackBar trackReceiveCost;
                private System.Windows.Forms.TrackBar trackTransmissionCost;
                private System.Windows.Forms.TrackBar trackTransmissionRadius;
                private System.Windows.Forms.TrackBar trackNetworkSize;
                private System.Windows.Forms.TrackBar trackSensorRadius;
                private System.Windows.Forms.TrackBar trackResidualEnergy;
                private System.Windows.Forms.TrackBar trackInitialEnergy;
                private System.Windows.Forms.TrackBar trackEnergyCost;
                private System.ComponentModel.IContainer components;
                private WirelessSensorNetwork network = null;
      private WirelessSensorNetwork network1 = null;
      private System.Timers.Timer timerUpdate;
                private System.Windows.Forms.Button btnReplay;
                private System.Windows.Forms.TrackBar trackUpdateFrequency;
                private System.Windows.Forms.Label label10;
                private System.Windows.Forms.Label lblTime;
                private System.Windows.Forms.Label lblPower;
                private System.Windows.Forms.Label lblStatus;
                private System.Windows.Forms.TrackBar trackSensorDelay;
                private System.Windows.Forms.TrackBar trackTransmitterDelay;
```

```
		private System.Windows.Forms.TrackBar trackSensorCost;
		private System.Windows.Forms.GroupBox groupBox3;
		private System.Windows.Forms.GroupBox groupBox5;
		private System.Windows.Forms.Label lblSensors;
		private System.Windows.Forms.ToolTip toolTip;
		private System.Windows.Forms.RadioButton radioRandom;
		private System.Windows.Forms.RadioButton radioDirected;
		private System.Windows.Forms.Label label13;
		private System.Windows.Forms.Label label14;
		private System.Windows.Forms.Label lblLivePackets;
private System.Windows.Forms.Label lblRecdPackets;
private PictureBox picNetwork1;
private Button button1;
private Label label16;
private Label label15;
private TextBox textBox2;
private TextBox textBox1;
private Label label18;
private Label label17;
private Label label23;
private Label label22;
private Label label21;
private Label label20;
private Label label9;
private Label label8;
private Label label11;
private Label label19;
private int iSetupDisplay = -1;

		public WirelessSensorNetworkSimulator() {
			InitializeComponent();
		}

		#region Uninteresting Windows Form Designer generated code
		private void InitializeComponent() {
  this.components = new System.ComponentModel.Container();
  this.picNetwork = new System.Windows.Forms.PictureBox();
  this.groupBox1 = new System.Windows.Forms.GroupBox();
  this.label23 = new System.Windows.Forms.Label();
  this.label22 = new System.Windows.Forms.Label();
  this.label21 = new System.Windows.Forms.Label();
  this.label20 = new System.Windows.Forms.Label();
  this.label9 = new System.Windows.Forms.Label();
  this.label8 = new System.Windows.Forms.Label();
  this.label11 = new System.Windows.Forms.Label();
  this.label19 = new System.Windows.Forms.Label();
  this.label16 = new System.Windows.Forms.Label();
  this.label15 = new System.Windows.Forms.Label();
  this.textBox2 = new System.Windows.Forms.TextBox();
  this.textBox1 = new System.Windows.Forms.TextBox();
```

```
this.trackSensorRadius = new System.Windows.Forms.TrackBar();
this.trackReceiveCost = new System.Windows.Forms.TrackBar();
this.trackTransmissionCost = new System.Windows.Forms.TrackBar();
this.trackNetworkSize = new System.Windows.Forms.TrackBar();
this.trackSensorDelay = new System.Windows.Forms.TrackBar();
this.trackTransmitterDelay = new System.Windows.Forms.TrackBar();
this.trackSensorCost = new System.Windows.Forms.TrackBar();
this.trackTransmissionRadius = new System.Windows.Forms.TrackBar();
this.button1 = new System.Windows.Forms.Button();
this.btnDeploy = new System.Windows.Forms.Button();
this.groupBox2 = new System.Windows.Forms.GroupBox();
this.label14 = new System.Windows.Forms.Label();
this.radioDirected = new System.Windows.Forms.RadioButton();
this.radioRandom = new System.Windows.Forms.RadioButton();
this.trackUpdateFrequency = new System.Windows.Forms.TrackBar();
this.label10 = new System.Windows.Forms.Label();
this.trackResidualEnergy = new System.Windows.Forms.TrackBar();
this.label6 = new System.Windows.Forms.Label();
this.trackInitialEnergy = new System.Windows.Forms.TrackBar();
this.label4 = new System.Windows.Forms.Label();
this.trackEnergyCost = new System.Windows.Forms.TrackBar();
this.label3 = new System.Windows.Forms.Label();
this.label13 = new System.Windows.Forms.Label();
this.btnStart = new System.Windows.Forms.Button();
this.lblTime = new System.Windows.Forms.Label();
this.lblPower = new System.Windows.Forms.Label();
this.lblLivePackets = new System.Windows.Forms.Label();
this.lblStatus = new System.Windows.Forms.Label();
this.btnReplay = new System.Windows.Forms.Button();
this.timerUpdate = new System.Timers.Timer();
this.groupBox3 = new System.Windows.Forms.GroupBox();
this.groupBox5 = new System.Windows.Forms.GroupBox();
this.lblRecdPackets = new System.Windows.Forms.Label();
this.lblSensors = new System.Windows.Forms.Label();
this.toolTip = new System.Windows.Forms.ToolTip(this.components);
this.picNetwork1 = new System.Windows.Forms.PictureBox();
this.label17 = new System.Windows.Forms.Label();
this.label18 = new System.Windows.Forms.Label();
((System.ComponentModel.ISupportInitialize)(this.picNetwork)).BeginInit();
this.groupBox1.SuspendLayout();
((System.ComponentModel.ISupportInitialize)(this.trackSensorRadius)).BeginInit();
((System.ComponentModel.ISupportInitialize)(this.trackReceiveCost)).BeginInit();

((System.ComponentModel.ISupportInitialize)(this.trackTransmissionCost)).BeginInit();
((System.ComponentModel.ISupportInitialize)(this.trackNetworkSize)).BeginInit();
((System.ComponentModel.ISupportInitialize)(this.trackSensorDelay)).BeginInit();

((System.ComponentModel.ISupportInitialize)(this.trackTransmitterDelay)).BeginInit();
((System.ComponentModel.ISupportInitialize)(this.trackSensorCost)).BeginInit();
```

```
((System.ComponentModel.ISupportInitialize)(this.trackTransmissionRadius)).BeginInit();
            this.groupBox2.SuspendLayout();

((System.ComponentModel.ISupportInitialize)(this.trackUpdateFrequency)).BeginInit();

((System.ComponentModel.ISupportInitialize)(this.trackResidualEnergy)).BeginInit();
            ((System.ComponentModel.ISupportInitialize)(this.trackInitialEnergy)).BeginInit();
            ((System.ComponentModel.ISupportInitialize)(this.trackEnergyCost)).BeginInit();
            ((System.ComponentModel.ISupportInitialize)(this.timerUpdate)).BeginInit();
            this.groupBox3.SuspendLayout();
            this.groupBox5.SuspendLayout();
            ((System.ComponentModel.ISupportInitialize)(this.picNetwork1)).BeginInit();
            this.SuspendLayout();
            //
            // picNetwork
            //
            this.picNetwork.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
            this.picNetwork.Location = new System.Drawing.Point(466, 12);
            this.picNetwork.Name = "picNetwork";
            this.picNetwork.Size = new System.Drawing.Size(460, 274);
            this.picNetwork.TabIndex = 0;
            this.picNetwork.TabStop = false;
            this.picNetwork.Paint += new
System.Windows.Forms.PaintEventHandler(this.picNetwork_Paint);
            //
            // groupBox1
            //
            this.groupBox1.Controls.Add(this.label23);
            this.groupBox1.Controls.Add(this.label22);
            this.groupBox1.Controls.Add(this.label21);
            this.groupBox1.Controls.Add(this.label20);
            this.groupBox1.Controls.Add(this.label9);
            this.groupBox1.Controls.Add(this.label8);
            this.groupBox1.Controls.Add(this.label11);
            this.groupBox1.Controls.Add(this.label19);
            this.groupBox1.Controls.Add(this.label16);
            this.groupBox1.Controls.Add(this.label15);
            this.groupBox1.Controls.Add(this.textBox2);
            this.groupBox1.Controls.Add(this.textBox1);
            this.groupBox1.Controls.Add(this.trackSensorRadius);
            this.groupBox1.Controls.Add(this.trackReceiveCost);
            this.groupBox1.Controls.Add(this.trackTransmissionCost);
            this.groupBox1.Controls.Add(this.trackNetworkSize);
            this.groupBox1.Controls.Add(this.trackSensorDelay);
            this.groupBox1.Controls.Add(this.trackTransmitterDelay);
            this.groupBox1.Controls.Add(this.trackSensorCost);
            this.groupBox1.Controls.Add(this.trackTransmissionRadius);
            this.groupBox1.FlatStyle = System.Windows.Forms.FlatStyle.System;
```

```
            this.groupBox1.Font = new System.Drawing.Font("Microsoft Sans Serif", 8.25F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.groupBox1.Location = new System.Drawing.Point(8, 269);
            this.groupBox1.Name = "groupBox1";
            this.groupBox1.Size = new System.Drawing.Size(166, 28);
            this.groupBox1.TabIndex = 1;
            this.groupBox1.TabStop = false;
            this.groupBox1.Text = "Network Configuration";
            //
            // label23
            //
            this.label23.AutoSize = true;
            this.label23.Location = new System.Drawing.Point(468, 64);
            this.label23.Name = "label23";
            this.label23.Size = new System.Drawing.Size(104, 13);
            this.label23.TabIndex = 33;
            this.label23.Text = "Transmission Radius";
            //
            // label22
            //
            this.label22.AutoSize = true;
            this.label22.Location = new System.Drawing.Point(468, 26);
            this.label22.Name = "label22";
            this.label22.Size = new System.Drawing.Size(64, 13);
            this.label22.TabIndex = 32;
            this.label22.Text = "Sensor Cost";
            //
            // label21
            //
            this.label21.AutoSize = true;
            this.label21.Location = new System.Drawing.Point(231, 104);
            this.label21.Name = "label21";
            this.label21.Size = new System.Drawing.Size(92, 13);
            this.label21.TabIndex = 31;
            this.label21.Text = "Transmitter Period";
            //
            // label20
            //
            this.label20.AutoSize = true;
            this.label20.Location = new System.Drawing.Point(231, 64);
            this.label20.Name = "label20";
            this.label20.Size = new System.Drawing.Size(70, 13);
            this.label20.TabIndex = 30;
            this.label20.Text = "Network Size";
            //
            // label9
            //
            this.label9.AutoSize = true;
            this.label9.Location = new System.Drawing.Point(231, 26);
            this.label9.Name = "label9";
```

```
this.label9.Size = new System.Drawing.Size(71, 13);
this.label9.TabIndex = 29;
this.label9.Text = "Transmit Cost";
//
// label8
//
this.label8.AutoSize = true;
this.label8.Location = new System.Drawing.Point(6, 104);
this.label8.Name = "label8";
this.label8.Size = new System.Drawing.Size(76, 13);
this.label8.TabIndex = 28;
this.label8.Text = "Sensor Radius";
//
// label11
//
this.label11.AutoSize = true;
this.label11.Location = new System.Drawing.Point(6, 64);
this.label11.Name = "label11";
this.label11.Size = new System.Drawing.Size(71, 13);
this.label11.TabIndex = 27;
this.label11.Text = "Receive Cost";
//
// label19
//
this.label19.AutoSize = true;
this.label19.Location = new System.Drawing.Point(6, 26);
this.label19.Name = "label19";
this.label19.Size = new System.Drawing.Size(73, 13);
this.label19.TabIndex = 26;
this.label19.Text = "Sensor Period";
//
// label16
//
this.label16.AutoSize = true;
this.label16.Location = new System.Drawing.Point(575, 104);
this.label16.Name = "label16";
this.label16.Size = new System.Drawing.Size(31, 13);
this.label16.TabIndex = 23;
this.label16.Text = "Task";
//
// label15
//
this.label15.AutoSize = true;
this.label15.Location = new System.Drawing.Point(468, 104);
this.label15.Name = "label15";
this.label15.Size = new System.Drawing.Size(33, 13);
this.label15.TabIndex = 22;
this.label15.Text = "Node";
//
// textBox2
```

```
//
this.textBox2.Enabled = false;
this.textBox2.Location = new System.Drawing.Point(608, 101);
this.textBox2.Name = "textBox2";
this.textBox2.Size = new System.Drawing.Size(82, 20);
this.textBox2.TabIndex = 20;
this.textBox2.Text = "250";
//
// textBox1
//
this.textBox1.Enabled = false;
this.textBox1.Location = new System.Drawing.Point(507, 101);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(59, 20);
this.textBox1.TabIndex = 19;
this.textBox1.Text = "400";
//
// trackSensorRadius
//
this.trackSensorRadius.AutoSize = false;
this.trackSensorRadius.Location = new System.Drawing.Point(92, 104);
this.trackSensorRadius.Maximum = 120;
this.trackSensorRadius.Minimum = 20;
this.trackSensorRadius.Name = "trackSensorRadius";
this.trackSensorRadius.Size = new System.Drawing.Size(120, 30);
this.trackSensorRadius.TabIndex = 2;
this.trackSensorRadius.TickFrequency = 10;
this.toolTip.SetToolTip(this.trackSensorRadius, "The size of the area within which a
sensor can detect movement.");
this.trackSensorRadius.Value = 45;
//
// trackReceiveCost
//
this.trackReceiveCost.AutoSize = false;
this.trackReceiveCost.Location = new System.Drawing.Point(92, 62);
this.trackReceiveCost.Maximum = 100;
this.trackReceiveCost.Minimum = 1;
this.trackReceiveCost.Name = "trackReceiveCost";
this.trackReceiveCost.Size = new System.Drawing.Size(120, 30);
this.trackReceiveCost.TabIndex = 8;
this.trackReceiveCost.TickFrequency = 10;
this.toolTip.SetToolTip(this.trackReceiveCost, "The energy consumed by receiving a
packet.");
this.trackReceiveCost.Value = 15;
//
// trackTransmissionCost
//
this.trackTransmissionCost.AutoSize = false;
this.trackTransmissionCost.Location = new System.Drawing.Point(330, 22);
this.trackTransmissionCost.Maximum = 10001;
```

```
            this.trackTransmissionCost.Minimum = 1;
            this.trackTransmissionCost.Name = "trackTransmissionCost";
            this.trackTransmissionCost.Size = new System.Drawing.Size(120, 30);
            this.trackTransmissionCost.TabIndex = 7;
            this.trackTransmissionCost.TickFrequency = 1000;
            this.toolTip.SetToolTip(this.trackTransmissionCost, "The energy consumed by
sending a packet.");
            this.trackTransmissionCost.Value = 200;
            //
            // trackNetworkSize
            //
            this.trackNetworkSize.AutoSize = false;
            this.trackNetworkSize.Location = new System.Drawing.Point(330, 62);
            this.trackNetworkSize.Maximum = 410;
            this.trackNetworkSize.Minimum = 10;
            this.trackNetworkSize.Name = "trackNetworkSize";
            this.trackNetworkSize.Size = new System.Drawing.Size(120, 30);
            this.trackNetworkSize.TabIndex = 1;
            this.trackNetworkSize.TickFrequency = 40;
            this.toolTip.SetToolTip(this.trackNetworkSize, "The number of nodes in the
network.");
            this.trackNetworkSize.Value = 10;
            //
            // trackSensorDelay
            //
            this.trackSensorDelay.AutoSize = false;
            this.trackSensorDelay.Location = new System.Drawing.Point(92, 26);
            this.trackSensorDelay.Maximum = 101;
            this.trackSensorDelay.Minimum = 1;
            this.trackSensorDelay.Name = "trackSensorDelay";
            this.trackSensorDelay.Size = new System.Drawing.Size(120, 30);
            this.trackSensorDelay.TabIndex = 3;
            this.trackSensorDelay.TickFrequency = 10;
            this.toolTip.SetToolTip(this.trackSensorDelay, "How long a tripped sensor waits
before sending a subsequent event.");
            this.trackSensorDelay.Value = 15;
            //
            // trackTransmitterDelay
            //
            this.trackTransmitterDelay.AutoSize = false;
            this.trackTransmitterDelay.Location = new System.Drawing.Point(330, 104);
            this.trackTransmitterDelay.Maximum = 101;
            this.trackTransmitterDelay.Minimum = 1;
            this.trackTransmitterDelay.Name = "trackTransmitterDelay";
            this.trackTransmitterDelay.Size = new System.Drawing.Size(120, 30);
            this.trackTransmitterDelay.TabIndex = 6;
            this.trackTransmitterDelay.TickFrequency = 10;
            this.toolTip.SetToolTip(this.trackTransmitterDelay, "The amount of time required to
send a packet.");
            this.trackTransmitterDelay.Value = 10;
```

```
            //
            // trackSensorCost
            //
            this.trackSensorCost.AutoSize = false;
            this.trackSensorCost.Location = new System.Drawing.Point(570, 22);
            this.trackSensorCost.Maximum = 100;
            this.trackSensorCost.Minimum = 1;
            this.trackSensorCost.Name = "trackSensorCost";
            this.trackSensorCost.Size = new System.Drawing.Size(120, 30);
            this.trackSensorCost.TabIndex = 4;
            this.trackSensorCost.TickFrequency = 10;
            this.toolTip.SetToolTip(this.trackSensorCost, "The energy consumed by a sensor
activation.");
            this.trackSensorCost.Value = 20;
            //
            // trackTransmissionRadius
            //
            this.trackTransmissionRadius.AutoSize = false;
            this.trackTransmissionRadius.Location = new System.Drawing.Point(578, 62);
            this.trackTransmissionRadius.Maximum = 230;
            this.trackTransmissionRadius.Minimum = 30;
            this.trackTransmissionRadius.Name = "trackTransmissionRadius";
            this.trackTransmissionRadius.Size = new System.Drawing.Size(120, 30);
            this.trackTransmissionRadius.TabIndex = 5;
            this.trackTransmissionRadius.TickFrequency = 20;
            this.toolTip.SetToolTip(this.trackTransmissionRadius, "The maximum distance
between two connected nodes.");
            this.trackTransmissionRadius.Value = 130;
            //
            // button1
            //
            this.button1.Enabled = false;
            this.button1.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.button1.Location = new System.Drawing.Point(16, 351);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(892, 87);
            this.button1.TabIndex = 24;
            this.button1.Text = "Model Simulation Result";
            this.button1.UseVisualStyleBackColor = true;
            this.button1.Click += new System.EventHandler(this.button1_Click);
            //
            // btnDeploy
            //
            this.btnDeploy.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.btnDeploy.Location = new System.Drawing.Point(16, 437);
            this.btnDeploy.Name = "btnDeploy";
            this.btnDeploy.Size = new System.Drawing.Size(892, 76);
            this.btnDeploy.TabIndex = 0;
```

```
this.btnDeploy.Text = "Deploy Network";
this.toolTip.SetToolTip(this.btnDeploy, "Click to deploy a network with these
settings.");
this.btnDeploy.Click += new System.EventHandler(this.btnDeploy_Click);
//
// groupBox2
//
this.groupBox2.Controls.Add(this.label14);
this.groupBox2.Controls.Add(this.radioDirected);
this.groupBox2.Controls.Add(this.radioRandom);
this.groupBox2.Controls.Add(this.trackUpdateFrequency);
this.groupBox2.Controls.Add(this.label10);
this.groupBox2.Controls.Add(this.trackResidualEnergy);
this.groupBox2.Controls.Add(this.label6);
this.groupBox2.Controls.Add(this.trackInitialEnergy);
this.groupBox2.Controls.Add(this.label4);
this.groupBox2.Controls.Add(this.trackEnergyCost);
this.groupBox2.Controls.Add(this.label3);
this.groupBox2.FlatStyle = System.Windows.Forms.FlatStyle.System;
this.groupBox2.Font = new System.Drawing.Font("Microsoft Sans Serif", 8.25F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.groupBox2.Location = new System.Drawing.Point(8, 536);
this.groupBox2.Name = "groupBox2";
this.groupBox2.Size = new System.Drawing.Size(256, 136);
this.groupBox2.TabIndex = 2;
this.groupBox2.TabStop = false;
this.groupBox2.Text = "Routing Parameters";
this.groupBox2.Visible = false;
//
// label14
//
this.label14.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
this.label14.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.label14.Location = new System.Drawing.Point(8, 24);
this.label14.Name = "label14";
this.label14.Size = new System.Drawing.Size(80, 16);
this.label14.TabIndex = 15;
this.label14.Text = "Routing Method";
this.label14.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
this.label14.Visible = false;
//
// radioDirected
//
this.radioDirected.Checked = true;
this.radioDirected.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.radioDirected.Location = new System.Drawing.Point(16, 56);
this.radioDirected.Name = "radioDirected";
this.radioDirected.Size = new System.Drawing.Size(72, 24);
```

```
            this.radioDirected.TabIndex = 0;
            this.radioDirected.TabStop = true;
            this.radioDirected.Text = "Directed";
            //
            // radioRandom
            //
            this.radioRandom.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.radioRandom.Location = new System.Drawing.Point(16, 88);
            this.radioRandom.Name = "radioRandom";
            this.radioRandom.Size = new System.Drawing.Size(72, 24);
            this.radioRandom.TabIndex = 1;
            this.radioRandom.Text = "Random";
            //
            // trackUpdateFrequency
            //
            this.trackUpdateFrequency.AutoSize = false;
            this.trackUpdateFrequency.Location = new System.Drawing.Point(176, 96);
            this.trackUpdateFrequency.Maximum = 100;
            this.trackUpdateFrequency.Minimum = 1;
            this.trackUpdateFrequency.Name = "trackUpdateFrequency";
            this.trackUpdateFrequency.Size = new System.Drawing.Size(72, 30);
            this.trackUpdateFrequency.TabIndex = 5;
            this.trackUpdateFrequency.TickFrequency = 10;
            this.toolTip.SetToolTip(this.trackUpdateFrequency, "The amount of time between
packet-routing decision updates.");
            this.trackUpdateFrequency.Value = 1;
            this.trackUpdateFrequency.Visible = false;
            //
            // label10
            //
            this.label10.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
            this.label10.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.label10.Location = new System.Drawing.Point(176, 80);
            this.label10.Name = "label10";
            this.label10.Size = new System.Drawing.Size(72, 16);
            this.label10.TabIndex = 12;
            this.label10.Text = "Routing Period";
            this.label10.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
            this.label10.Visible = false;
            //
            // trackResidualEnergy
            //
            this.trackResidualEnergy.AutoSize = false;
            this.trackResidualEnergy.Location = new System.Drawing.Point(176, 40);
            this.trackResidualEnergy.Name = "trackResidualEnergy";
            this.trackResidualEnergy.Size = new System.Drawing.Size(72, 30);
            this.trackResidualEnergy.TabIndex = 3;
```

```
            this.toolTip.SetToolTip(this.trackResidualEnergy, "The emphasis placed on the
residual energy of the nodes in making routing decisio" +
        "ns.");
            this.trackResidualEnergy.Visible = false;
            //
            // label6
            //
            this.label6.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
            this.label6.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.label6.Location = new System.Drawing.Point(176, 24);
            this.label6.Name = "label6";
            this.label6.Size = new System.Drawing.Size(72, 16);
            this.label6.TabIndex = 10;
            this.label6.Text = "Residual Power";
            this.label6.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
            this.label6.Visible = false;
            //
            // trackInitialEnergy
            //
            this.trackInitialEnergy.AutoSize = false;
            this.trackInitialEnergy.Location = new System.Drawing.Point(96, 96);
            this.trackInitialEnergy.Name = "trackInitialEnergy";
            this.trackInitialEnergy.Size = new System.Drawing.Size(72, 30);
            this.trackInitialEnergy.TabIndex = 4;
            this.toolTip.SetToolTip(this.trackInitialEnergy, "The emphasis placed on the initial
energy of the nodes in making routing decision" +
        "s.");
            this.trackInitialEnergy.Visible = false;
            //
            // label4
            //
            this.label4.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
            this.label4.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.label4.Location = new System.Drawing.Point(96, 80);
            this.label4.Name = "label4";
            this.label4.Size = new System.Drawing.Size(72, 16);
            this.label4.TabIndex = 8;
            this.label4.Text = "Initial Power";
            this.label4.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
            this.label4.Visible = false;
            //
            // trackEnergyCost
            //
            this.trackEnergyCost.AutoSize = false;
            this.trackEnergyCost.Location = new System.Drawing.Point(96, 40);
            this.trackEnergyCost.Maximum = 1;
            this.trackEnergyCost.Name = "trackEnergyCost";
            this.trackEnergyCost.Size = new System.Drawing.Size(72, 30);
```

```
this.trackEnergyCost.TabIndex = 2;
this.toolTip.SetToolTip(this.trackEnergyCost, "Whether or not the cost of
transmitting packets is taken into account in determin" +
"ing routing.");
this.trackEnergyCost.Visible = false;
//
// label3
//
this.label3.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
this.label3.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.label3.Location = new System.Drawing.Point(96, 24);
this.label3.Name = "label3";
this.label3.Size = new System.Drawing.Size(72, 16);
this.label3.TabIndex = 6;
this.label3.Text = "Exchange Cost";
this.label3.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
this.label3.Visible = false;
//
// label13
//
this.label13.Location = new System.Drawing.Point(0, 0);
this.label13.Name = "label13";
this.label13.Size = new System.Drawing.Size(100, 23);
this.label13.TabIndex = 0;
//
// btnStart
//
this.btnStart.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.btnStart.Location = new System.Drawing.Point(8, 24);
this.btnStart.Name = "btnStart";
this.btnStart.Size = new System.Drawing.Size(378, 58);
this.btnStart.TabIndex = 0;
this.btnStart.Text = "Start Simulation";
this.toolTip.SetToolTip(this.btnStart, "Starts the simulation.");
this.btnStart.Click += new System.EventHandler(this.btnStart_Click);
//
// lblTime
//
this.lblTime.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
this.lblTime.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.lblTime.Location = new System.Drawing.Point(4, 42);
this.lblTime.Name = "lblTime";
this.lblTime.Size = new System.Drawing.Size(494, 18);
this.lblTime.TabIndex = 0;
this.lblTime.Text = "Time: 00:00.000";
this.lblTime.TextAlign = System.Drawing.ContentAlignment.MiddleLeft;
//
```

```
            // lblPower
            //
            this.lblPower.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
            this.lblPower.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.lblPower.Location = new System.Drawing.Point(4, 61);
            this.lblPower.Name = "lblPower";
            this.lblPower.Size = new System.Drawing.Size(494, 18);
            this.lblPower.TabIndex = 2;
            this.lblPower.Text = "Power: 0";
            this.lblPower.TextAlign = System.Drawing.ContentAlignment.MiddleLeft;
            //
            // lblLivePackets
            //
            this.lblLivePackets.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
            this.lblLivePackets.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.lblLivePackets.Location = new System.Drawing.Point(4, 99);
            this.lblLivePackets.Name = "lblLivePackets";
            this.lblLivePackets.Size = new System.Drawing.Size(494, 18);
            this.lblLivePackets.TabIndex = 4;
            this.lblLivePackets.Text = "Live Packets: 0";
            this.lblLivePackets.TextAlign = System.Drawing.ContentAlignment.MiddleLeft;
            //
            // lblStatus
            //
            this.lblStatus.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
            this.lblStatus.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.lblStatus.Location = new System.Drawing.Point(4, 23);
            this.lblStatus.Name = "lblStatus";
            this.lblStatus.Size = new System.Drawing.Size(494, 18);
            this.lblStatus.TabIndex = 5;
            this.lblStatus.Text = "Status: Ready";
            this.lblStatus.TextAlign = System.Drawing.ContentAlignment.MiddleLeft;
            //
            // btnReplay
            //
            this.btnReplay.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.btnReplay.Location = new System.Drawing.Point(8, 76);
            this.btnReplay.Name = "btnReplay";
            this.btnReplay.Size = new System.Drawing.Size(378, 54);
            this.btnReplay.TabIndex = 1;
            this.btnReplay.Text = "Replay Simulation";
            this.toolTip.SetToolTip(this.btnReplay, "Replays the previous simulation.");
            this.btnReplay.Click += new System.EventHandler(this.btnReplay_Click);
            //
            // timerUpdate
            //
```

```
            this.timerUpdate.Enabled = true;
            this.timerUpdate.Interval = 20D;
            this.timerUpdate.SynchronizingObject = this;
            this.timerUpdate.Elapsed += new
System.Timers.ElapsedEventHandler(this.timerUpdate_Elapsed);
            //
            // groupBox3
            //
            this.groupBox3.Controls.Add(this.btnStart);
            this.groupBox3.Controls.Add(this.btnReplay);
            this.groupBox3.FlatStyle = System.Windows.Forms.FlatStyle.System;
            this.groupBox3.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.groupBox3.Location = new System.Drawing.Point(8, 527);
            this.groupBox3.Name = "groupBox3";
            this.groupBox3.Size = new System.Drawing.Size(392, 145);
            this.groupBox3.TabIndex = 9;
            this.groupBox3.TabStop = false;
            this.groupBox3.Text = "Simulation Control";
            //
            // groupBox5
            //
            this.groupBox5.Controls.Add(this.lblRecdPackets);
            this.groupBox5.Controls.Add(this.lblSensors);
            this.groupBox5.Controls.Add(this.lblPower);
            this.groupBox5.Controls.Add(this.lblTime);
            this.groupBox5.Controls.Add(this.lblStatus);
            this.groupBox5.Controls.Add(this.lblLivePackets);
            this.groupBox5.FlatStyle = System.Windows.Forms.FlatStyle.System;
            this.groupBox5.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.groupBox5.Location = new System.Drawing.Point(412, 527);
            this.groupBox5.Name = "groupBox5";
            this.groupBox5.Size = new System.Drawing.Size(504, 145);
            this.groupBox5.TabIndex = 10;
            this.groupBox5.TabStop = false;
            this.groupBox5.Text = "Simulation Status";
            //
            // lblRecdPackets
            //
            this.lblRecdPackets.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
            this.lblRecdPackets.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            this.lblRecdPackets.Location = new System.Drawing.Point(4, 118);
            this.lblRecdPackets.Name = "lblRecdPackets";
            this.lblRecdPackets.Size = new System.Drawing.Size(494, 18);
            this.lblRecdPackets.TabIndex = 7;
            this.lblRecdPackets.Text = "Rec\'d Packets: 0";
            this.lblRecdPackets.TextAlign = System.Drawing.ContentAlignment.MiddleLeft;
            //
```

```
        // lblSensors
        //
        this.lblSensors.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
        this.lblSensors.Font = new System.Drawing.Font("Microsoft Sans Serif", 6.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
        this.lblSensors.Location = new System.Drawing.Point(4, 80);
        this.lblSensors.Name = "lblSensors";
        this.lblSensors.Size = new System.Drawing.Size(494, 18);
        this.lblSensors.TabIndex = 6;
        this.lblSensors.Text = "Sensors: 0";
        this.lblSensors.TextAlign = System.Drawing.ContentAlignment.MiddleLeft;
        //
        // picNetwork1
        //
        this.picNetwork1.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
        this.picNetwork1.Location = new System.Drawing.Point(0, 12);
        this.picNetwork1.Name = "picNetwork1";
        this.picNetwork1.Size = new System.Drawing.Size(460, 274);
        this.picNetwork1.TabIndex = 11;
        this.picNetwork1.TabStop = false;
        this.picNetwork1.Paint += new
System.Windows.Forms.PaintEventHandler(this.picNetwork1_Paint);
        //
        // label17
        //
        this.label17.AutoSize = true;
        this.label17.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
        this.label17.Location = new System.Drawing.Point(576, 291);
        this.label17.Name = "label17";
        this.label17.Size = new System.Drawing.Size(189, 20);
        this.label17.TabIndex = 12;
        this.label17.Text = "Existing Simulation Model";
        //
        // label18
        //
        this.label18.AutoSize = true;
        this.label18.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
        this.label18.Location = new System.Drawing.Point(111, 289);
        this.label18.Name = "label18";
        this.label18.Size = new System.Drawing.Size(200, 20);
        this.label18.TabIndex = 13;
        this.label18.Text = "Improved Simulation Model";
        //
        // WirelessSensorNetworkSimulator
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(928, 674);
        this.Controls.Add(this.label18);
```

```
        this.Controls.Add(this.label17);
        this.Controls.Add(this.picNetwork1);
        this.Controls.Add(this.groupBox5);
        this.Controls.Add(this.groupBox3);
        this.Controls.Add(this.groupBox1);
        this.Controls.Add(this.picNetwork);
        this.Controls.Add(this.groupBox2);
        this.Controls.Add(this.button1);
        this.Controls.Add(this.btnDeploy);
        this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.Fixed3D;
        this.MaximizeBox = false;
        this.MinimizeBox = false;
        this.Name = "WirelessSensorNetworkSimulator";
        this.Text = "Fault Tolerance Wireless Sensor Network Simulator- Inproved By
Francis Msc Studen" +
   "t ABU";
        this.Closing += new
System.ComponentModel.CancelEventHandler(this.WirelessSensorNetworkSimulator_Closi
ng);
        ((System.ComponentModel.ISupportInitialize)(this.picNetwork)).EndInit();
        this.groupBox1.ResumeLayout(false);
        this.groupBox1.PerformLayout();
        ((System.ComponentModel.ISupportInitialize)(this.trackSensorRadius)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.trackReceiveCost)).EndInit();

((System.ComponentModel.ISupportInitialize)(this.trackTransmissionCost)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.trackNetworkSize)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.trackSensorDelay)).EndInit();

((System.ComponentModel.ISupportInitialize)(this.trackTransmitterDelay)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.trackSensorCost)).EndInit();

((System.ComponentModel.ISupportInitialize)(this.trackTransmissionRadius)).EndInit();
        this.groupBox2.ResumeLayout(false);

((System.ComponentModel.ISupportInitialize)(this.trackUpdateFrequency)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.trackResidualEnergy)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.trackInitialEnergy)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.trackEnergyCost)).EndInit();
        ((System.ComponentModel.ISupportInitialize)(this.timerUpdate)).EndInit();
        this.groupBox3.ResumeLayout(false);
        this.groupBox5.ResumeLayout(false);
        ((System.ComponentModel.ISupportInitialize)(this.picNetwork1)).EndInit();
        this.ResumeLayout(false);
        this.PerformLayout();

                }
                protected override void Dispose(bool disposing) {
                        if (disposing) {
                                if (components != null) {
```

```
					components.Dispose();
				}
			}
			base.Dispose(disposing);
		}
		#endregion

		private void WirelessSensorNetworkSimulator_Closing(object sender,
System.ComponentModel.CancelEventArgs e) {
			if ((network != null) && (network.tSimulation != null))
				network.tSimulation.Abort();
	if ((network1 != null) && (network1.tSimulation != null))
	   network1.tSimulation.Abort();
		}

		[STAThread]
		static void Main() {
	Application.Run(new WirelessSensorNetworkSimulator());


      }

		#endregion

		#region Button-click event handlers

		private void btnDeploy_Click(object sender, System.EventArgs e) {

	button1.Enabled = false;
	//button2.Enabled = false;

			network = new WirelessSensorNetwork((int)trackNetworkSize.Value,
(int)trackSensorRadius.Value, (int)trackSensorDelay.Value,
(int)trackTransmissionRadius.Value, (int)trackTransmitterDelay.Value,
(float)trackTransmissionCost.Value / 100.0f, (int)trackReceiveCost.Value,
(int)trackSensorCost.Value, radioDirected.Checked, trackEnergyCost.Value,
trackResidualEnergy.Value, trackInitialEnergy.Value, 1, picNetwork.Width - 5,
picNetwork.Height - 5, picNetwork.Width - 45);
	network1 = new WirelessSensorNetwork( (int)trackNetworkSize.Value,
((int)trackSensorRadius.Value), 2*((int)trackSensorDelay.Value),
((int)trackTransmissionRadius.Value), 4+((int)trackTransmitterDelay.Value),
2*((float)trackTransmissionCost.Value / 100.0f), (int)trackReceiveCost.Value,
(int)trackSensorCost.Value, radioDirected.Checked, 2*(trackEnergyCost.Value),
trackResidualEnergy.Value, trackInitialEnergy.Value, 1, picNetwork1.Width - 5,
picNetwork1.Height - 5, picNetwork1.Width - 45);


	iSetupDisplay = 0;  // initiate the "deploying network" display
```

```
	}

		private void btnStart_Click(object sender, System.EventArgs e) {
			if (network != null && network1 != null) {
				if (network.tSimulation == null && network1.tSimulation ==
null) {  // no simulation running - start a new simulation (by spawning a new thread)

		network.Reset(true);
		network.tSimulation = new Thread(new ThreadStart(network.RunSimulation));
		network.tSimulation.Start();

		network1.Reset(true);
		network1.tSimulation = new Thread(new
ThreadStart(network1.RunSimulation));
		network1.tSimulation.Start();

					btnStart.Text = "Stop Simulation";
					btnStart.Refresh();
					lblStatus.Text = "Status: Operating";
					lblStatus.Refresh();
				}
				else {  // simulation running - tell the thread to stop running and
relabel buttons
					network.bAbort = true;
		network1.bAbort = true;
					network.tSimulation = null;
		network1.tSimulation = null;
					btnStart.Text = "Start Simulation";
					btnStart.Refresh();
					lblStatus.Text = "Status: Ready";
					lblStatus.Refresh();
				}
			}
		}

		private void btnReplay_Click(object sender, System.EventArgs e) {
			if (network != null) {
				if (network.tSimulation != null)  // immediately stop simulation
if it's running, so that we can start a new one
					network.tSimulation.Abort();
				else {
					btnStart.Text = "Stop Simulation";
					btnStart.Refresh();
					lblStatus.Text = "Status: Operating";
					lblStatus.Refresh();
				}
				network.Reset(false);  // "false" indicates that we're restarting
without picking a new seed (i.e., use the one from the last run)
				network.tSimulation = new Thread(new
ThreadStart(network.RunSimulation));
```

```
						network.tSimulation.Start();
					}
				}

	/*
				private void bMeanLifetimeTest_Click(object sender, System.EventArgs e) {
					if (network != null) {
						if (network.tSimulation != null) {  // abort any running
simulation
							network.tSimulation.Abort();
							network.tSimulation = null;
							btnStart.Text = "Start Simulation";
							btnStart.Refresh();
							lblStatus.Text = "Status: Ready";
							lblStatus.Refresh();
						}
						// start a new simulation and display the tester window
						Thread thread = new Thread(new
ThreadStart(network.RunTest));
						thread.Start();
						WirelessSensorNetworkTest test = new
WirelessSensorNetworkTest(network);
						test.ShowDialog();
					}
				}
	*/
				#endregion

				#region Paint and Timer event handlers

				private void picNetwork_Paint(object sender,
System.Windows.Forms.PaintEventArgs e) {
					// note that we're painting, so that the timer thread doesn't send another
Paint message
					if (network != null) {
						network.bPainting = true;
						network.bPaint = false;
					}
					// create drawing tools
					Font font = new Font("Times New Roman", 7.0f);
					StringFormat format = new StringFormat();
					format.Alignment = StringAlignment.Center;
					Graphics g = e.Graphics;
					g.SmoothingMode =
System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
					// draw background
			// g.FillRectangle(System.Drawing.Brushes.BlanchedAlmond, e.ClipRectangle);
					g.FillRectangle(System.Drawing.Brushes.Snow, e.ClipRectangle);
					// draw network objects (if network has been deployed)
					if (network != null) {
```

```
// draw sensor background
if (iSetupDisplay == -1) {
        ArrayList activatedSensors = new ArrayList();
        foreach (WirelessSensor sensor in network.aSensors) {
                if (sensor.iSensorDelay <= 0)
    //g.FillEllipse(new SolidBrush(Color.FromArgb(196, 196, 196))
                        g.FillEllipse(new
SolidBrush(Color.FromArgb(100, 100, 100)), sensor.x - sensor.iSensorRadius, sensor.y -
sensor.iSensorRadius, sensor.iSensorRadius * 2, sensor.iSensorRadius * 2);
                else
                        activatedSensors.Add(sensor);
        }
        foreach (WirelessSensor sensor in activatedSensors)
  //g.FillEllipse(new SolidBrush(Color.FromArgb(196, 196, 196 + 48 *
sensor.iSensorDelay
  //g.FillEllipse(new SolidBrush(Color.FromArgb(100, 100, 100 + 48 *
sensor.iSensorDelay
  g.FillEllipse(new SolidBrush(Color.FromArgb(100, 100, 100 + 48 *
sensor.iSensorDelay / network.iSensorDelay)), sensor.x - sensor.iSensorRadius, sensor.y -
sensor.iSensorRadius, sensor.iSensorRadius * 2, sensor.iSensorRadius * 2);


}
// draw end zone
g.DrawLine(Pens.Black, picNetwork.Width - 44, 0,
picNetwork.Width - 44, picNetwork.Height);
for (int i = 0; i < picNetwork.Height; i += 20)
        g.DrawLine(Pens.Black, picNetwork.Width - 44, i + 20,
picNetwork.Width, i);
// draw connections
foreach (WirelessSensor sensor in network.aSensors) {
        foreach (WirelessSensorConnection connection in
sensor.aConnections) {
                if ((connection.sReceiver != null) &&
((iSetupDisplay == -1) ||  (iSetupDisplay > connection.sReceiver.x)) &&
(connection.sSender.iResidualEnergy > 0) && (connection.sReceiver.iResidualEnergy > 0))
                        g.DrawLine(connection.iTransmitting >
0 ? Pens.Red : connection == sensor.connectionCurrent ? Pens.Blue : Pens.Black,
connection.sSender.x, connection.sSender.y, connection.sReceiver.x,
connection.sReceiver.y);
        }
}
// draw sensors
Brush sensorBrush = Brushes.DarkGray;
Pen sensorPen = Pens.Red;
foreach (WirelessSensor sensor in network.aSensors) {
        if ((iSetupDisplay == -1) || (iSetupDisplay > sensor.x))
{
                int color = sensor.iResidualEnergy <= 0 ? 0 :
(int) (255 * sensor.iResidualEnergy / WirelessSensorNetwork.iMaxEnergy);
```

```
                                                g.FillEllipse(new
SolidBrush(Color.FromArgb(color, color, color)), sensor.x - 4, sensor.y - 4, 9, 9);
                                                if (sensor.iResidualEnergy <= 0) {
                                                        if (sensor.iSensorRadius > 0)
                                                                sensor.iSensorRadius--;
                                                }
                                                g.DrawEllipse(sensor.iResidualEnergy <= 0 ?
Pens.Black : Pens.Red, sensor.x - 4, sensor.y - 4, 9, 9);
                                        }
                                }
                                // draw vectors
                                if ((network.bRunningSimulation == true) &&
(network.vectors != null)) {
                                        network.vectors.mutexVector.WaitOne();
                                        foreach (Vector vector in network.vectors.aVectors) {
                                                g.FillRectangle(Brushes.Green, vector.x - 1,
vector.y - 1, 3, 3);
                                        }
                                        network.vectors.mutexVector.ReleaseMutex();
                                }
                                // draw setup line, if network is being deployed
                                if (iSetupDisplay != -1)
                                        g.DrawLine(Pens.Black, iSetupDisplay, 0,
iSetupDisplay, picNetwork.Height);
                        }
                        // finish painting
                        if (network != null)
                                network.bPainting = false;
                }
        /*
                private void picRadar_Paint(object sender,
System.Windows.Forms.PaintEventArgs e) {
                        // create drawing tools
                        Font font = new Font("Times New Roman", 7.0f);
                        StringFormat format = new StringFormat();
                        format.Alignment = StringAlignment.Center;
                        Graphics g = e.Graphics;
                        g.SmoothingMode =
System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
                        // draw background
                        g.FillRectangle(System.Drawing.Brushes.Black, e.ClipRectangle);
                        // draw network radar objects
                        if ((network != null) && (network.vectors != null)) {
                                // draw sensors
                                int iRadius = network.iSensorRadius * picRadar.Width /
picNetwork.Width;

                                foreach (WirelessSensor sensor in network.aSensors) {
                                        if (sensor.iResidualEnergy > 0)
```

```
                                        g.DrawEllipse(Pens.Green, sensor.x *
picRadar.Width / picNetwork.Width - iRadius, sensor.y * picRadar.Height /
picNetwork.Height - iRadius, iRadius * 2, iRadius * 2);
                        }
                        // draw received packets
                        for (int i = 0; i < network.aRadar.Count;) {
                                if ((++((Packet) network.aRadar[i]).timestamp) >=
((Packet) network.aRadar[i]).lifespan)
                                        network.aRadar.RemoveAt(i);
                                else {
                                        int green = 255 - 191 * ((Packet)
network.aRadar[i]).timestamp / (((Packet) network.aRadar[i]).lifespan + 1);
                                        g.FillEllipse(new
SolidBrush(Color.FromArgb(0, green, 0)), ((Packet) network.aRadar[i]).x * picRadar.Width /
picNetwork.Width - iRadius, ((Packet) network.aRadar[i]).y * picRadar.Height /
picNetwork.Height - iRadius, iRadius * 2, iRadius * 2);
                                        i++;
                                }
                        }
                        // draw vectors
                        network.vectors.mutexVector.WaitOne();
                        foreach (Vector vector in network.vectors.aVectors)
                                g.DrawRectangle(Pens.White, vector.x *
picRadar.Width  / picNetwork.Width, vector.y * picRadar.Height / picNetwork.Height, 2, 2);
                        network.vectors.mutexVector.ReleaseMutex();
                        // draw radar line
                        int xRadarLine = picRadar.Width - Math.Abs((int)
(System.DateTime.Now.Ticks / 500000) % picRadar.Width);
                        for (int i = 0; i < 6; i++)
                                g.DrawLine(new Pen(Color.FromArgb(0, 255 - (255 * i
/ 7), 0)), xRadarLine - i, 0, xRadarLine - i, picRadar.Height);
                }
        }
   */
        private void timerUpdate_Elapsed(object sender,
System.Timers.ElapsedEventArgs e) {
                // check to see if the network is being deployed
                if (iSetupDisplay != -1) {
                        iSetupDisplay += 10;
                        if ((iSetupDisplay >= picNetwork.Width) && (iSetupDisplay
>= picNetwork1.Width))
                                iSetupDisplay = -1;
                        picNetwork.Refresh();
      picNetwork1.Refresh();
                }
                // check to see if network simulation is running
                else if ((network1 != null) && (network1.tSimulation != null) &&
(network1.bPaint == true) && (network1.bPainting == false)) {
                        // refresh objects
                        picNetwork.Refresh();
```

```
                picNetwork1.Refresh();
                                //picRadar.Refresh();
                                // display information in textboxes
                                if (network1.bRunningSimulation == true) {
                                        TimeSpan counter = new
TimeSpan(System.DateTime.Now.Ticks - network1.timeStart.Ticks);
                                        lblTime.Text = "Time: " +
counter.Minutes.ToString("d2") + ":" + counter.Seconds.ToString("d2") + "." +
counter.Milliseconds.ToString("d3");
                                        lblTime.Refresh();
                                        lblRecdPackets.Text = "Rec'd Packets: " +
network.iPacketsDelivered;
                                        lblRecdPackets.Refresh();
                                        int iPower = 0;int iSensors = 0;
                                        int iLivePackets = 0;
                                        foreach (WirelessSensor sensor in network1.aSensors) {
                                                iPower += sensor.iResidualEnergy;
                                                if (sensor.iResidualEnergy > 0) {
                                                        iSensors++;
                                                        iLivePackets += sensor.aPackets.Count;
                                                }
                                        }
                                        lblSensors.Text = "Sensors: " + iSensors + "/" +
network1.aSensors.Count;
                  if (iSensors == 0)
                  {
                     button1.Enabled = true;
                     //button2.Enabled = true;



                  }
                                        lblSensors.Refresh();
                                        lblPower.Text = "Power: " + iPower;
                                        lblPower.Refresh();
                                        lblLivePackets.Text = "Live Packets: " + iLivePackets;
                                        lblLivePackets.Refresh();
                                }
                                else {  // network has stopped running - throw away the
completed thread and relabel buttons
                                        network.tSimulation = null;
                                        btnStart.Text = "Start Simulation";
                                        btnStart.Refresh();
                                        lblStatus.Text = "Status: Ready";
                                        lblStatus.Refresh();
                                }
                        }
                }

                #endregion
```

```
        private void picNetwork1_Paint(object sender, System.Windows.Forms.PaintEventArgs
e)
        {
           // note that we're painting, so that the timer thread doesn't send another Paint message
           if (network1 != null)
           {
              network1.bPainting = true;
              network1.bPaint = false;
           }
           // create drawing tools
           Font font = new Font("Times New Roman", 7.0f);
           StringFormat format = new StringFormat();
           format.Alignment = StringAlignment.Center;
           Graphics g = e.Graphics;
           g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
           // draw background
           // g.FillRectangle(System.Drawing.Brushes.BlanchedAlmond, e.ClipRectangle);
           g.FillRectangle(System.Drawing.Brushes.Snow, e.ClipRectangle);
           // draw network objects (if network has been deployed)
           if (network1 != null)
           {
              // draw sensor background
              if (iSetupDisplay == -1)
              {
                 ArrayList activatedSensors = new ArrayList();
                 foreach (WirelessSensor sensor in network1.aSensors)
                 {
                    if (sensor.iSensorDelay <= 0)
                       //g.FillEllipse(new SolidBrush(Color.FromArgb(196, 196, 196))
                       g.FillEllipse(new SolidBrush(Color.FromArgb(100, 100, 100)), sensor.x -
sensor.iSensorRadius, sensor.y - sensor.iSensorRadius, sensor.iSensorRadius * 2,
sensor.iSensorRadius * 2);
                    else
                       activatedSensors.Add(sensor);
                 }
                 foreach (WirelessSensor sensor in activatedSensors)
                    //g.FillEllipse(new SolidBrush(Color.FromArgb(196, 196, 196 + 48 *
sensor.iSensorDelay
                    g.FillEllipse(new SolidBrush(Color.FromArgb(100, 100, 100 + 48 *
sensor.iSensorDelay / network1.iSensorDelay)), sensor.x - sensor.iSensorRadius, sensor.y -
sensor.iSensorRadius, sensor.iSensorRadius * 2, sensor.iSensorRadius * 2);
              }
              // draw end zone
              g.DrawLine(Pens.Black, picNetwork1.Width - 44, 0, picNetwork1.Width - 44,
picNetwork1.Height);
              for (int i = 0; i < picNetwork1.Height; i += 20)
                 g.DrawLine(Pens.Black, picNetwork1.Width - 44, i + 20, picNetwork1.Width,
i);
```

```
            // draw connections
            foreach (WirelessSensor sensor in network1.aSensors)
            {
               foreach (WirelessSensorConnection connection in sensor.aConnections)
               {
                  if ((connection.sReceiver != null) && ((iSetupDisplay == -1) || (iSetupDisplay
> connection.sReceiver.x)) && (connection.sSender.iResidualEnergy > 0) &&
(connection.sReceiver.iResidualEnergy > 0))
                     g.DrawLine(connection.iTransmitting > 0 ? Pens.Red : connection ==
sensor.connectionCurrent ? Pens.Blue : Pens.Black, connection.sSender.x,
connection.sSender.y, connection.sReceiver.x, connection.sReceiver.y);
               }
            }
            // draw sensors
            Brush sensorBrush = Brushes.DarkGray;
            Pen sensorPen = Pens.Red;
            foreach (WirelessSensor sensor in network1.aSensors)
            {
               if ((iSetupDisplay == -1) || (iSetupDisplay > sensor.x))
               {
                  int color = sensor.iResidualEnergy <= 0 ? 0 : (int)(255 *
sensor.iResidualEnergy / WirelessSensorNetwork.iMaxEnergy);
                  g.FillEllipse(new SolidBrush(Color.FromArgb(color, color, color)), sensor.x -
4, sensor.y - 4, 9, 9);
                  if (sensor.iResidualEnergy <= 0)
                  {
                     if (sensor.iSensorRadius > 0)
                        sensor.iSensorRadius--;
                  }
                  g.DrawEllipse(sensor.iResidualEnergy <= 0 ? Pens.Black : Pens.Red, sensor.x
- 4, sensor.y - 4, 9, 9);
               }
            }
            // draw vectors
            if ((network1.bRunningSimulation == true) && (network1.vectors != null))
            {
               network1.vectors.mutexVector.WaitOne();
               foreach (Vector vector in network1.vectors.aVectors)
               {
                  g.FillRectangle(Brushes.Green, vector.x - 1, vector.y - 1, 3, 3);
               }
               network1.vectors.mutexVector.ReleaseMutex();
            }
            // draw setup line, if network is being deployed
            if (iSetupDisplay != -1)
               g.DrawLine(Pens.Black, iSetupDisplay, 0, iSetupDisplay, picNetwork1.Height);
         }
         // finish painting
         if (network1 != null)
            network1.bPainting = false;
```

```
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Graph1 existing = new Graph1();
            existing.Show();
        }


    }
}
```

Appendix B: Codes for graphs of RFTAS

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Xml;

namespace Wireless_Sensor_Network_Simulator
{
    public partial class Graph2 : Form
    {
        public Graph2()
        {
            InitializeComponent();
        }

        private void Graph2_Load(object sender, EventArgs e)
        {
            try
            {
                XmlReader xmlFile;
                xmlFile = XmlReader.Create("record.xml", new XmlReaderSettings());
                DataSet ds = new DataSet();
                ds.ReadXml(xmlFile);
                //dataGridView1.DataSource = ds.Tables[0];   //RFTAS


                chart1.DataSource = ds.Tables[0];
                //set the member of the chart data source used to data bind to the X-values of the
series
                chart1.Series["RFTAS"].XValueMember = "EnergyNode";
                //set the member columns of the chart data source used to data bind to the X-values
of the series
                chart1.Series["RFTAS"].YValueMembers = "Energy";
                chart1.ChartAreas[0].AxisY.Maximum = 12;
                chart1.ChartAreas[0].AxisY.Minimum = 2;
                chart1.ChartAreas[0].AxisX.Maximum = 8; // 25;
                chart1.ChartAreas[0].AxisX.Minimum = 1;
                chart1.Titles.Add("RFTAS Chart");
                chart1.ChartAreas[0].AxisX.Title = "Node Number";
                chart1.ChartAreas[0].AxisY.Title = "Energy Consumption";
```

```
            chart2.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart2.Series["RFTAS"].XValueMember = "TimeConsumptionNode";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart2.Series["RFTAS"].YValueMembers = "TimeConsumption";
            chart2.ChartAreas[0].AxisY.Maximum = 10;
            chart2.ChartAreas[0].AxisY.Minimum = 4;
            chart2.ChartAreas[0].AxisX.Maximum = 8;
            chart2.ChartAreas[0].AxisX.Minimum = 1;
            chart2.Titles.Add("RFTAS Chart");
            chart2.ChartAreas[0].AxisX.Title = "Node Number";
            chart2.ChartAreas[0].AxisY.Title = "Time Consumption";


            chart3.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart3.Series["RFTAS"].XValueMember = "ReliabilityNode-U";         //
"ReliabilityNode";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart3.Series["RFTAS"].YValueMembers = "Reliability-U";           //"Reliability";
            chart3.ChartAreas[0].AxisY.Maximum = 500;
            chart3.ChartAreas[0].AxisY.Minimum = 150;
            chart3.ChartAreas[0].AxisX.Maximum = 8;
            chart3.ChartAreas[0].AxisX.Minimum = 1;
            chart3.Titles.Add("RFTAS Chart");
            chart3.ChartAreas[0].AxisX.Title = "Node Number";
            chart3.ChartAreas[0].AxisY.Title = "Reliability Cost";


            chart4.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart4.Series["RFTAS"].XValueMember = "Time";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart4.Series["RFTAS"].YValueMembers = "Failure";
            chart4.ChartAreas[0].AxisY.Maximum = 20;
            chart4.ChartAreas[0].AxisY.Minimum = 0;
            chart4.ChartAreas[0].AxisX.Maximum = 8;
            chart4.ChartAreas[0].AxisX.Minimum = 0;
            chart4.Titles.Add("RFTAS Chart");
            chart4.ChartAreas[0].AxisX.Title = "Node Number";
            chart4.ChartAreas[0].AxisY.Title = "Failure Node Number";
```

```
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }
        }
    }
}
```

Appendix C: Codes for graphs of mRFTAS

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Xml;

namespace Wireless_Sensor_Network_Simulator
{
    public partial class Graph3 : Form
    {
        public Graph3()
        {
            InitializeComponent();
        }

        private void Graph3_Load(object sender, EventArgs e)
        {
            try
            {
                XmlReader xmlFile;
                xmlFile = XmlReader.Create("record.xml", new XmlReaderSettings());
                DataSet ds = new DataSet();
                ds.ReadXml(xmlFile);
                //dataGridView1.DataSource = ds.Tables[0];   //RFTAS


                chart1.DataSource = ds.Tables[0];
                //set the member of the chart data source used to data bind to the X-values of the
series
                chart1.Series["m-RFTAS"].XValueMember = "EnergyNode-U";
                //set the member columns of the chart data source used to data bind to the X-values
of the series
                chart1.Series["m-RFTAS"].YValueMembers = "Energy-U";
                chart1.ChartAreas[0].AxisY.Maximum = 12;
                chart1.ChartAreas[0].AxisY.Minimum = 1;     //1
                chart1.ChartAreas[0].AxisX.Maximum = 8; // 25;
                chart1.ChartAreas[0].AxisX.Minimum = 1;
                chart1.Titles.Add("RFTAS Chart");
                chart1.ChartAreas[0].AxisX.Title = "Node Number";
                chart1.ChartAreas[0].AxisY.Title = "Energy Consumption";
```

```
            chart2.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart2.Series["m-RFTAS"].XValueMember = "TimeConsumptionNode-U";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart2.Series["m-RFTAS"].YValueMembers = "TimeConsumption-U";
            chart2.ChartAreas[0].AxisY.Maximum = 10;
            chart2.ChartAreas[0].AxisY.Minimum = 1;     //4
            chart2.ChartAreas[0].AxisX.Maximum = 8;
            chart2.ChartAreas[0].AxisX.Minimum = 1;
            chart2.Titles.Add("RFTAS Chart");
            chart2.ChartAreas[0].AxisX.Title = "Node Number";
            chart2.ChartAreas[0].AxisY.Title = "Time Consumption";


            chart3.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart3.Series["m-RFTAS"].XValueMember = "ReliabilityNode";
//ReliabilityNode-U
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart3.Series["m-RFTAS"].YValueMembers = "Reliability";       //"Reliability-U";
// Reliability-U
            chart3.ChartAreas[0].AxisY.Maximum = 500;
            chart3.ChartAreas[0].AxisY.Minimum = 150;
            chart3.ChartAreas[0].AxisX.Maximum = 8;
            chart3.ChartAreas[0].AxisX.Minimum = 1;
            chart3.Titles.Add("RFTAS Chart");
            chart3.ChartAreas[0].AxisX.Title = "Node Number";
            chart3.ChartAreas[0].AxisY.Title = "Reliability Cost";


            chart4.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart4.Series["m-RFTAS"].XValueMember = "Time-U";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart4.Series["m-RFTAS"].YValueMembers = "Failure-U";
            chart4.ChartAreas[0].AxisY.Maximum = 20;
            chart4.ChartAreas[0].AxisY.Minimum = 0;
            chart4.ChartAreas[0].AxisX.Maximum = 8;
            chart4.ChartAreas[0].AxisX.Minimum = 0;
            chart4.Titles.Add("RFTAS Chart");
            chart4.ChartAreas[0].AxisX.Title = "Node Number";
            chart4.ChartAreas[0].AxisY.Title = "Failure Node Number";
```

```
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }
        }
    }
}
```

Appendix D: Codes for graphs of both m-RFTAS and RFTAS

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Xml;


namespace Wireless_Sensor_Network_Simulator
{
   public partial class Graph1 : Form
   {
      public Graph1()
      {
         InitializeComponent();
      }

      private void Graph1_Load(object sender, EventArgs e)
      {
         try
         {
            XmlReader xmlFile;
            xmlFile = XmlReader.Create("record.xml", new XmlReaderSettings());
            DataSet ds = new DataSet();
            ds.ReadXml(xmlFile);
            //dataGridView1.DataSource = ds.Tables[0];   //RFTAS


            chart1.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart1.Series["RFTAS"].XValueMember = "EnergyNode";
            chart1.Series["m-RFTAS"].XValueMember = "EnergyNode-U";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart1.Series["RFTAS"].YValueMembers = "Energy";
            chart1.Series["m-RFTAS"].YValueMembers = "Energy-U";
            chart1.ChartAreas[0].AxisY.Maximum = 12;
            chart1.ChartAreas[0].AxisY.Minimum = 1;
            chart1.ChartAreas[0].AxisX.Maximum = 8; // 25;
            chart1.ChartAreas[0].AxisX.Minimum = 1;
            chart1.Titles.Add("RFTAS Chart");
            chart1.ChartAreas[0].AxisX.Title = "Node Number";
            chart1.ChartAreas[0].AxisY.Title = "Energy Consumption";
```

```
            chart2.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart2.Series["RFTAS"].XValueMember = "TimeConsumptionNode";
            chart2.Series["m-RFTAS"].XValueMember = "TimeConsumptionNode-U";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart2.Series["RFTAS"].YValueMembers = "TimeConsumption";
            chart2.Series["m-RFTAS"].YValueMembers = "TimeConsumption-U";
            chart2.ChartAreas[0].AxisY.Maximum = 10;
            chart2.ChartAreas[0].AxisY.Minimum = 1;
            chart2.ChartAreas[0].AxisX.Maximum = 8;
            chart2.ChartAreas[0].AxisX.Minimum = 1;
            chart2.Titles.Add("RFTAS Chart");
            chart2.ChartAreas[0].AxisX.Title = "Node Number";
            chart2.ChartAreas[0].AxisY.Title = "Time Consumption";


            chart3.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart3.Series["RFTAS"].XValueMember = "ReliabilityNode-U";
//ReliabilityNode
            chart3.Series["m-RFTAS"].XValueMember = "ReliabilityNode";
//ReliabilityNode-U
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart3.Series["RFTAS"].YValueMembers = "Reliability-U";          //Reliability
            chart3.Series["m-RFTAS"].YValueMembers = "Reliability";      //Reliability-U
            chart3.ChartAreas[0].AxisY.Maximum = 500;
            chart3.ChartAreas[0].AxisY.Minimum = 150;
            chart3.ChartAreas[0].AxisX.Maximum = 8;
            chart3.ChartAreas[0].AxisX.Minimum = 1;
            chart3.Titles.Add("RFTAS Chart");
            chart3.ChartAreas[0].AxisX.Title = "Node Number";
            chart3.ChartAreas[0].AxisY.Title = "Reliability Cost";


            chart4.DataSource = ds.Tables[0];
            //set the member of the chart data source used to data bind to the X-values of the
series
            chart4.Series["RFTAS"].XValueMember = "Time";
            chart4.Series["m-RFTAS"].XValueMember = "Time-U";
            //set the member columns of the chart data source used to data bind to the X-values
of the series
            chart4.Series["RFTAS"].YValueMembers = "Failure";
            chart4.Series["m-RFTAS"].YValueMembers = "Failure-U";
            chart4.ChartAreas[0].AxisY.Maximum = 20;
            chart4.ChartAreas[0].AxisY.Minimum = 0;
```

```
            chart4.ChartAreas[0].AxisX.Maximum = 8;
            chart4.ChartAreas[0].AxisX.Minimum = 0;
            chart4.Titles.Add("RFTAS Chart");
            chart4.ChartAreas[0].AxisX.Title = "Node Number";
            chart4.ChartAreas[0].AxisY.Title = "Failure Node Number";




          }
          catch (Exception ex)
          {
            MessageBox.Show(ex.ToString());
          }
        }

        private void existingToolStripMenuItem_Click(object sender, EventArgs e)
        {
          Graph2 gp1 = new Graph2();
          gp1.Show();
        }

        private void improvedToolStripMenuItem_Click(object sender, EventArgs e)
        {
          Graph3 gp2 = new Graph3();
          gp2.Show();
        }

        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
          this.Close();
        }
      }
    }
```

Appendix E: Simulation results table for 10 runs of sensor nodes energy consumption simulations

NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
100	8.7	8.9
	8.4	8.7
	8.4	10.1
	8.5	9.5
	8.3	9.4
	8.1	9.0
	8.8	9.1
	8.5	8.8
	8.7	8.9
	8.6	10.3
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
125	7.6	9.1
	7.4	9.0
	7.6	8.5
	7.5	8.1
	7.3	8.4
	7.5	8.5
	7.4	8.5
	7.7	8.4
	7.6	8.5
	7.6	8.4
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
150	6.6	7.3
	6.5	7.4
	6.7	7.8
	6.6	7.6
	6.5	7.4
	6.3	7.4
	6.5	7.3
	6.4	7.2
	6.3	7.3
	6.6	7.3
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
175	5.5	6.6
	5.4	5.7
	5.4	6.4
	5.4	5.8
	5.2	6.2
	5.4	6.2
	5.3	6.1
	5.6	6.4
	5.4	6.1
	5.4	6.3

NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
200	4.5	5.1
	4.5	5.4
	4.7	5.5
	4.6	4.9
	4.6	5.2
	4.5	5.1
	4.7	5.1
	4.6	5.2
	4.7	5.4
	4.6	5.2
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
225	3.7	4.5
	3.7	4.3
	3.8	4.4
	3.8	4.3
	3.7	4.5
	3.8	4.4
	3.8	4.5
	3.9	4.5
	3.7	4.4
	3.6	4.5
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
250	3.1	4.1
	2.8	4.2
	2.8	3.8
	2.9	3.9
	2.8	4.1
	3.1	4.3
	2.9	4.1
	2.9	3.9
	2.8	3.9
	2.9	4.0

Appendix F: Simulation results table for 10 runs of sensor nodes task execution time simulations

NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
100	8.4	7.6
	8.2	7.4
	8.3	7.0
	8.6	7.4
	8.1	7.3
	7.9	7.2
	8.5	7.1
	8.4	7.2
	8.5	7.2
	8.4	7.4
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
125	6.3	5.2
	6.2	5.2
	6.3	5.4
	6.0	5.6
	6.5	5.6
	6.3	5.5
	6.4	4.9
	6.2	5.4
	6.5	5.4
	6.1	5.5
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
150	5.2	3.9
	5.3	4.2
	5.1	4.1
	5.2	4.2
	5.1	4.0
	5.2	3.9
	5.2	4.0
	5.1	4.2
	5.3	4.1
	5.3	3.9
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
175	4.3	3.3
	4.3	3.2
	4.2	3.2
	4.0	3.0
	4.1	3.3
	4.0	3.1
	4.5	3.3
	4.2	3.4
	4.3	3.3
	4.3	3.2

NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
200	4.1	2.5
	4.0	2.3
	4.1	2.5
	4.2	2.5
	3.9	2.6
	3.9	2.6
	4.0	2.5
	4.2	2.6
	4.1	2.6
	4.1	2.6
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
225	3.9	2.3
	3.8	2.2
	3.7	2.3
	4.0	2.3
	3.8	2.4
	3.6	2.3
	3.8	2.3
	3.7	2.4
	3.8	2.3
	3.9	2.2
NUMBER OF NODES	RFTAS (10^4)	mRFTAS (10^4)
250	4.0	1.0
	3.5	2.0
	3.5	1.9
	3.4	2.1
	3.6	1.5
	3.5	2.1
	3.4	1.8
	3.4	1.9
	3.5	1.8
	3.9	1.5

Appendix G: Simulation results table for 10 runs of sensor nodes reliability cost simulations

NUMBER OF NODES	RFTAS	mRFTAS
100	495	475
	490	475
	500	485
	510	480
	505	480
	485	485
	500	480
	490	480
	495	480
	505	475
NUMBER OF NODES	RFTAS	mRFTAS
125	360	340
	355	350
	365	340
	355	345
	360	345
	360	340
	355	340
	360	350
	360	345
	365	340
NUMBER OF NODES	RFTAS	mRFTAS
150	305	285
	300	285
	300	280
	305	275
	310	280
	295	280
	300	275
	305	285
	300	280
	300	280
NUMBER OF NODES	RFTAS	mRFTAS
175	265	245
	265	245
	265	240
	265	240
	270	245
	260	245
	260	245
	265	235
	265	245
	265	240

NUMBER OF NODES	RFTAS	mRFTAS
200	245	230
	255	230
	245	230
	260	229
	240	225
	250	235
	255	230
	250	230
	245	235
	250	230
NUMBER OF NODES	RFTAS	mRFTAS
225	210	180
	210	185
	190	180
	190	180
	220	190
	215	180
	220	175
	210	175
	225	180
	200	180
NUMBER OF NODES	RFTAS	mRFTAS
250	155	150
	155	140
	150	140
	155	145
	145	140
	155	140
	160	140
	150	145
	150	155
	155	140

Appendix H: Simulation results table for 10 runs of sensor nodes network lifetime simulations

TIME	RFTAS (number of failed nodes)	mRFTAS (number of failed nodes)
50	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
TIME	RFTAS	mRFTAS
55	1	1
	1	1
	1	1
	1	1
	1	1
	1	1
	1	1
	1	1
	1	1
	1	1
TIME	RFTAS	mRFTAS
60	4	2
	4	2
	3	2
	3	2
	5	2
	4	2
	4	2
	4	2
	4	2
	5	2
TIME	RFTAS	mRFTAS
65	9	5
	8	7
	9	7
	9	6
	9	6
	9	6
	9	6
	9	5
	10	7

	9	5
TIME 70	RFTAS	mRFTAS
	16	10
	16	10
	16	10
	15	11
	14	10
	17	10
	18	9
	16	11
	16	9
	16	10
TIME 75	RFTAS	mRFTAS
	20	14
	22	14
	22	16
	22	14
	18	15
	20	15
	20	16
	18	14
	18	16
	20	16
TIME 80	RFTAS	mRFTAS
	20	18
	18	19
	18	18
	18	18
	20	17
	22	18
	22	17
	20	18
	22	19
	20	18