UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

MOHAMMADREZA IMAN

THESEUS: A Routing System for Shared Sensor Networks

Rio de Janeiro

2015

MOHAMMADREZA IMAN

THESEUS: A Routing System for Shared Sensor Networks

A dissertation submitted in partial fulfillment of the requirements for the degree of Master (Computer Science, Network and Distributed systems) in Programa de Pós-Graduação em Informática, Universidade Federal do Rio de Janeiro.

Advisors: Flávia Coimbra Delicato

Claudio Miceli de Farias

Rio de Janeiro

2015

FICHA CATALOGRÁFICA

```
        Iman, Mohammadreza
I31t       THESEUS: a routing system for shared sensor
        networks / Mohammadreza Iman. -- Rio de Janeiro,
        2015.
           123 f.

           Orientadora: Flávia Coimbra  Delicato.
           Coorientador: Claudio Miceli de Farias.
           Dissertação (mestrado) - Universidade Federal
        do Rio de Janeiro, Instituto Tércio Pacitti de
        Aplicações e Pesquisas Computacionais, Programa de
        Pós-Graduação em informática, 2015.

           1. Roteamento. 2. Redes de sensores
        compartilhadas. 3. Redes de sensores sem fio. 4.
        Eficiência energética. 5. Agregação de pacotes. I.
        Delicato, Flávia Coimbra , orient. II. Farias,
        Claudio Miceli de, coorient. III. Título.
```

MOHAMMADREZA IMAN

THESEUS: A routing system for Shared Sensor Networks

A dissertation submitted in partial fulfillment of the requirements for the degree of Master (Computer Science, Network and Distributed systems) in Programa de Pós-Graduação em Informática, Universidade Federal do Rio de Janeiro.

[illegible]

Prof. D.Sc. Flávia Coimbra Delicato – Adviser
COPPE/UFRJ, Brasil
UFRJ/PPGI

[illegible]

Prof. D.Sc. Claudio Miceli de Farias – co Adviser
UFRJ/PPGI

[illegible]

Prof. D.Sc. Paulo de Figueiredo Pires
COPPE/UFRJ, Brasil.
UFRJ/PPGI

[illegible]

Prof. Dr. Luiz Fernando Rust da Costa Carmo
UPS, Franca
UFRJ/PPGI

[illegible]

Prof. Dr. José Ferreira de Rezende
Université Pierre et Marie Curie
UFRJ/PESC

Rio de Janeiro

2015

DEDICATION

I take pleasure in dedicating this thesis to everyone in my family and friends. Each of them has helped to shape my path to success in a distinctive way.

ACKNOWLEDGEMENTS

I am an Iranian, a descendant of Cyrus the Great. This emperor proclaimed 2,500 years ago that “he would not reign over the people if they did not wish it". He promised not to force any person to change his religion and faith and guaranteed freedom for all. Cyrus the Great followed “Good words, good reflection and good deed”, the rules of Zoroastrianism (Persian ancient religion), this is how to keep it simple, forget about the rest and concentrate on the main parts. Accordingly, first, I thank the God for giving the wisdom, strength, support and knowledge in exploring things.

It is a pleasure to thank the many people who made this thesis possible. Though only my name appears on the cover of this dissertation, a great many people have contributed to its production. I owe my gratitude to all those people, who have made this dissertation possible and because of whom, my master degree experience has been one that I will cherish forever.

My deepest gratitude is to my advisor, Dr. Flavia C. Delicato. I had been amazingly fortunate to have an advisor who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. She taught me how to question thoughts and express ideas. Her patience and support helped me overcome many crisis situations and finish this dissertation. I hope that one day I would become as professional as her. I am also thankful to her for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on countless revisions of this manuscript.

My co-advisor, Dr. Claudio Miceli, has been always there to listen and give advice. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work. I am also thankful to him for reading my reports, commenting on my ideas and helping me understand and enrich my ideas.

Dr. Paulo F. Pires's insightful comments and constructive criticisms at different stages of my research were thought to provoke, and they helped me focus my ideas. Also, I am grateful to Dr. Paulo and Dr. Flavia for the opportunity of the one year experience of working on a research project at EMC Corporation.

Igor L. dos Santos, as well thought me everything I know about TinyOS programming, was the proofreaders of my document. In him, I have found a very good friend.

In my daily work, I have been blessed with a friendly and cheerful group of fellow students at UBICOMP laboratory. Jesus Talavera, Bruno Costa, Thomaz Barros, Pedro Silveira, Ricardo Caldeira, Wagner Lopez, and Taniro Rodrigues who helped me not just in my academic life even for handling the living issues in a new country. Their support and care helped me overcome setbacks and stay focused on my graduate study. I greatly value their friendship, and I deeply appreciate their belief in me.

This research would not have been possible without the financial assistance of CAPES (Brazilian Funding Agency), the Federal University of Rio de Janeiro (UFRJ), and the Department of PPGI at Federal University of Rio de Janeiro (Research Scholarships). I express my gratitude to those agencies.

Above all, I want to thank my family for the support they provided me through my entire life. Thanks also go to my former manager and friend, Mr. Shahrokh Tabibzadeh, my dear cousin, Dr. Leili Mohammad Khanli and my beloved sister Zahra Iman, who encouraged me to continue my academic studies as well as my dear parents.

RESUMO

IMAN, Mohammadreza. **THESEUS**: a routing system for shared sensor networks. 2015. 131 f. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

Monitoramento do ambiente a nossa volta nos mantém informados, nos ajuda a manter um ambiente saudável e sustentável, e nos alerta sobre os problemas futuros que possam surgir. Monitoramento nos diz o que está acontecendo; a pesquisa mostra por que algo está acontecendo, e modelagem ajuda a nos dizer o que pode acontecer. Progresso tecnológico permitiu o surgimento de vários tipos de sensores para medir fenômenos físicos. Recentes avanços em dispositivos sensores e tecnologias de comunicação sem fio permitiram a construção de sensores de baixo custo e pequeno porte. Redes de Sensores Sem Fio (RSSF) são compostos por um grande número desses dispositivos minúsculos que são alimentados por bateria e são equipados com uma ou mais unidades de sensoriamento, além de processador, memória e uma antena de transmissão. RSSFs são usados para coletar dados sobre fenômenos físicos. RSSFs tradicionais são tipicamente redes específicas de aplicação. A ideia de compartilhar a detecção e comunicação de infraestrutura de redes de sensores através de múltiplas aplicações surgiu recentemente. Este conceito é conhecido como Redes de Sensores Compartilhadas (RSCs). RSCs são capazes de lidar com mais de uma aplicação simultaneamente de forma eficiente. Da mesma forma que as RSSFs, o maior desafio em RSCs resulta das restrições quanto ao consumo de energia dos nós sensores, que incentiva o desenvolvimento de técnicas para poupar o máximo de energia dos nós sensores quanto possível. Neste contexto, algoritmos de roteamento poderiam desempenhar um papel fundamental para melhorar o tempo de vida da rede.

THESEUS é um sistema de roteamento eficiente em termos de consumo de energia para redes de sensores compartilhadas (RSCs), com o principal objetivo de estender o tempo de vida da rede. THESEUS tem duas características que o distinguem de outros trabalhos encontrados na literatura de redes de sensores sem fio (RSSFs) e Rede de Sensores Compartilhadas (RSCs). Em primeiro lugar, economiza energia de nós de uma RSC usando um algoritmo de agregação de pacotes, o que reduz o número de transmissões. Em segundo lugar, THESEUS equilibra o consumo de energia em toda a RSC, graças ao seu algoritmo de seleção de rota dinâmico ciente do QoS e da energia. Tal balanceamento do uso da energia evita a particionamento de rede devido ao esgotamento de energia de alguns nós utilizados extensivamente utilizados. Ambos os recursos de THESEUS resultam em prolongar o tempo de vida da RSC. Experimentos foram realizados a fim de avaliar a eficácia do THESEUS em melhorar o tempo de vida da RSC. Além disso, THESEUS suporta vários nós sorvedouros, portanto, é capaz de ajustar dinamicamente rotas sempre que um nó sorvedouro é adicionado ou removido. A avaliação realizada mostra melhorias significativas no consumo de energia e em relação ao equilíbrio do consumo de energia, em comparação com os trabalhos relacionados na literatura Rede de Sensores Sem Fio (RSSF).

Palavras-chave: Roteamento. Redes de sensores compartilhadas. Redes de sensores sem fio. Eficiência energética. Agregação de pacotes. Vários nós sorvedouros.

ABSTRACT

IMAN, Mohammadreza. **THESEUS**: a routing system for shared sensor networks. 2015. 131 f. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

Monitoring our surrounding environment keeps us informed, helps us to maintain a healthy and sustainable environment, and alerts us about future problems that may arise. Monitoring tells us what is happening; research shows why something is happening, and modeling helps to tell us what can happen. Technology progress provides several types of sensors for measuring physical phenomena. Recent advances in sensor devices and wireless communication technologies have enabled the building of low-cost and small-sized sensors. Wireless Sensor Networks (WSN) are composed of a large number of these tiny battery-operated devices equipped with one or more sensing units, processor, memory, and a wireless radio. WSNs are used to gather data about physical phenomena. Traditional WSNs are typically application specific networks. The idea of sharing the sensing and communication infrastructure of WSNs through multiple applications has recently emerged. This concept is known as Shared Sensor Networks (SSNs). SSNs are able to handle more than one application simultaneously in an efficient way. Similar to WSNs, the greatest challenge in SSNs arises from the energy-constrained nature of sensor nodes, which encourages the development of techniques to save as much energy from sensor nodes as possible. In this context, routing algorithms could play a key role to improve the network lifetime.

THESEUS is an energy-efficient routing system for Shared Sensor Networks (SSNs), with the primary goal of extending the network lifetime. THESEUS has two features that distinguish it from other works found in the Wireless Sensor Network (WSN) and Shared Sensor Network (SSN) literature. First, it saves energy of SSN nodes by using a packet aggregation algorithm, which reduces the number of transmissions. Second, THESEUS balances energy usage in the whole SSN thanks to its dynamic, QoS and energy aware route selection algorithms. Such energy usage balancing avoids network partitioning due to the energy depletion of some more extensively used nodes. Both features of THESEUS result in prolonging the SSN lifetime. Experiments were performed with the purpose of evaluating THESEUS effectiveness for improving the SSNs lifetime. Furthermore, THESEUS supports multiple sink nodes, thus it is able to adjust routes whenever a sink node is added or removed, dynamically. The conducted evaluation shows significant improvements in the energy usage and the energy balance metrics, compared to the related work in the Wireless Sensor Network (WSN) literature.

Keywords: Routing. Shared sensor networks. Wireless sensor networks. Energy efficiency. Packet aggregation. Multiple sink nodes.

LIST OF FIGURES

FIGURE 1. SAMPLE OF NETWORK TOPOLOGY 40
FIGURE 2. THESEUS PACKET AGGREGATION ALGORITHM 43
FIGURE 3. THESEUS SINK ARCHITECTURE 48
FIGURE 4. THESEUS NODE ARCHITECTURE 49
FIGURE 5. THESEUS SINK ALGORITHM 51
FIGURE 6. THESEUS NODE ALGORITHM 52
FIGURE 7. COMPONENTS AND INTERFACES OF THESEUS SINK MANAGER APPLICATION (TINYOS) 56
FIGURE 8. COMPONENTS AND INTERFACES OF THESEUS NODE MANAGER APPLICATION (TINYOS) 57
FIGURE 9. GRAPH OF DATA FLOWS DURING A CYCLE OF THESEUS SIMULATION (100 NODES) 64
FIGURE 10. RESULTS OF THE ENERGY AVERAGE METRIC 66
FIGURE 11. RESULTS OF THE ENERGY PSTD METRIC 67
FIGURE 12. RESULTS OF THE CPU ACTIVE METRIC 68
FIGURE 13. RESULTS OF THE NUMBER OF PACKETS USED IN ROUTE CONSTRUCTION METRIC 69
FIGURE 14. RESULTS OF THE ROUTE CONSTRUCTION TIME METRIC 69
FIGURE 15. RESULTS OF THE NET. RATE METRIC 70
FIGURE 16. RESULTS OF THE AGGREGATION RATE METRIC 71
FIGURE 17. AGGREGATION DETAIL 71
FIGURE 18. RESULTS OF THE PACKET LOSS METRIC 72
FIGURE 19. RESULTS OF THE SAMPLE LOSS METRIC 72
FIGURE 20. RESULTS OF THE DELAY TIME METRIC 73
FIGURE 21. TOPOLOGY OF 100 NODES AND 4 APPLICATIONS 76
FIGURE 23. RESULTS OF THE ENERGY PSTD METRIC 77
FIGURE 22. RESULTS OF THE ENERGY AVERAGE METRIC 77
FIGURE 24. GRAPHS OF DATA FLOWS DURING A CYCLE OF THESEUS SIMULATION (100 NODES) FOR 3, 5, 10, AND 15 METERS DISTANCES 79
FIGURE 25. GRAPH OF DATA FLOWS DURING A CYCLE OF THESEUS SIMULATION (100 NODES) FOR ONE OF NODES RANDOM POSITIONS 80
FIGURE 26. GRAPH OF DATA FLOWS DURING A CYCLE OF THESEUS SIMULATION (100 NODES) FOR THE CENTRAL SINK NODE 81
FIGURE 27. GRAPHS OF DATA FLOWS OF SIX CYCLES OF MULTI SINK TEST, WHERE THE FIRST SINK REMOVED IN CYCLE IDS: 3, 4, AND 5 87
FIGURE 28. GRAPH OF DATA FLOWS DURING A CYCLE OF SIMULATING 99 NODES AND 4 SINKS 88
FIGURE 29. GRAPH OF DATA FLOWS DURING A CYCLE OF REAL TEST SCENARIO 89

LIST OF TABLES

TABLE 1. RELATED WORKS COMPARISON...37
TABLE 2. RESULTS OF NUMBER OF RUNNING APPLICATIONS VARIATION TESTS...76
TABLE 3. RESULTS OF NODES DISTANCE VARIATION TESTS...78
TABLE 4. RESULTS OF THE NODES RANDOM POSITION VARIATION TESTS...80
TABLE 5. RESULTS OF THE SINK NODE POSITION TESTS...82
TABLE 6. RESULTS OF THE CYCLE TIME VARIATION TESTS...82
TABLE 7. RESULTS OF THE MDTC VARIATION TESTS...83
TABLE 8. RESULTS OF THE SIMULATION TIME VARIATION TESTS...84
TABLE 9. RESULTS OF THE ADD AND REMOVE SINK NODE...86
TABLE 10. RESULTS OF THE ADD AND REMOVE SINK NODE...86
TABLE 11. RESULTS OF THE REAL AND SIMULATED TESTS...90

LIST OF ABBREVIATIONS

AMCR	Adaptive Multi-Criteria Routing
APTEEN	Adaptive Periodic TEEN
BS	Base Station
CAPES	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
CDMA	Code Division Multiple Access
GAF	Geographic Adaptive Fidelity
GEAR	Geographic and Energy Aware Routing
GPS	Global Positioning System
GQM	Goal Question Metric
ID	Identification/Identity/Identifier
IP	Internet Protocol
LEACH	Low Energy Adaptive Clustering Hierarchy
MAC	Medium Access Control
MCFA	Minimum Cost Forwarding Algorithm
MDT	Maximum accepted Delay Time
OS	Operating System
PEGASIS	Power-Efficient GAthering in Sensor Information Systems
PPGI	Programa de Pós-Graduação em Informática
PROC	Proactive ROuting with Coordination
QoS	Quality of Service
RSC	Redes de Sensores Compartilhada
RSSF	Redes de Sensores Sem Fio
SN	Sink Node
SPIN	Sensor Protocols for Information via Negotiation
SSN	Shared Sensor Network
TDMA	Time Division Multiple Access
TEEN	Threshold-sensitive Energy Efficient sensor Network protocol
UFRJ	Universidade Federal do Rio de Janeiro
USB	Universal Serial Bus
WSN	Wireless Sensor Network

TABLE OF CONTENTS

1. INTRODUCTION 15
1.1 MOTIVATION 16
1.2 CONTRIBUTIONS 18
1.3 ORGANIZATION 19
2. BASIC CONCEPTS 20
2.1 SHARED SENSOR NETWORKS 20
2.2 ROUTING ON SSNS 26
3. RELATED WORK 35
4. PROPOSAL 38
4.1 PROC OVERVIEW 38
4.2 SSN MODEL AND ASSUMPTIONS 40
4.3 THESEUS 41
4.3.1 THESEUS packet aggregation 43
4.3.2 THESEUS application function 45
4.3.3 THESEUS architecture 48
4.3.4 THESEUS operation 50
5. IMPLEMENTATION 55
6. EVALUATION 60
6.1 COMPARING THESEUS AND PROC 60
6.1.1 GQM 60
6.1.2 Evaluation methodology and scenarios 63
6.1.3 Tests and analysis of results 65
6.2 ANALYSIS OF THE IMPACTS OF VARIATION OF IMPORTANT PARAMETERS 74
6.2.1 GQM 74
6.2.2 Evaluation methodology and scenarios 75
6.2.3 Tests and analysis of results 75
6.3 ANALYSIS OF THE IMPACTS OF USING MORE THAN ONE SINK NODE ON THESEUS 85
6.3.1 GQM 85
6.3.2 Evaluation methodology and scenarios 85
6.3.3 Tests and analysis of results 85
6.4 COMPARISON BETWEEN REAL AND SIMULATED NODES 89
7. CONCLUSIONS AND FUTURE WORK 91
7.1 FUTURE WORK 92
REFERENCES 94

APPENDIX A, THE PUBLISHED ARTICLE: 97
APPENDIX B, THE IMPLEMENTED CODES FOR TINYOS: 105
I) THESEUS SINK MANAGER APPLICATION 105
1) Makefile 105
2) message.h 105
3) THESEUS_SinkManagerAppC.nc 106
4) THESEUS_SinkManagerC.nc 106
II) THESEUS NODE MANAGER APPLICATION 108
1) Makefile 108
2) message.h 108
3) THESEUS_NodeManagerAppC.nc 110
4) THESEUS_NodeManagerAppC.nc 111

1. INTRODUCTION

Monitoring our surrounding environment keeps us informed, helps us to maintain a healthy and sustainable environment, and alerts us about future problems that may arise. We use monitoring in several activities, from simple tasks of everyday life till advanced industrial applications to keep track of weather, traffic patterns, process productivity etc (LOVETT *et al.*, 2007). We make decisions based on this information (LOVETT *et al.*, 2007). Through such careful observation, we can make science-based management decisions. For example, Charles David Keeling's long-term measurements of atmospheric carbon dioxide at Mauna Loa, HI provided the first unmistakable evidence that carbon dioxide emissions from human activities were warming the Earth. As a result of Keeling's and other scientists' careful and consistent monitoring, global climate change is now widely accepted as scientific fact (LOVETT *et al.*, 2007).

Monitoring, research, and modeling are three legs of a stool that provides scientific support for ecosystem restoration and management (LOVETT *et al.*, 2007). Monitoring tells us what is happening; analysing shows why something is happening, and modeling helps to tell us what can happen (LOVETT *et al.*, 2007). Long-term observations also expose trends and patterns that can improve evaluation of experimental results or yield new research hypotheses. In this context, technologies progress provides various types of sensors for measuring physical phenomena, thus effectively performing monitoring tasks. Recent advances in microelectromechanical systems and wireless communication technologies have enabled the building of sensor devices that are low-cost, small-sized, energy and resource constrained equipped with sensing interfaces and wireless links. A Wireless Sensor Network (WSN) is composed of a large number of such sensors (DELICATO *et al.*, 2012).

Wireless Sensor Networks (WSNs) is one of the subclasses of ad hoc networks. WSNs are used to gather data about physical phenomena and send them to client applications through one or more network exit points often called sink nodes or base stations. Sink nodes are powerful devices, for instance, a personal computer, connected to a constant and reliable power source, that are responsible for collecting sensed data from all sensors, further processing them, and making them available to applications and information systems via external networks such as the Internet. Sensor nodes act in a cooperative way to complete sensing tasks, providing data with both spatial and temporal resolutions which would be very difficult (or even impossible) to achieve by using other monitoring techniques such as wired sensors (DELICATO *et al.*, 2012).

Traditional WSNs are generally *application specific networks*, i.e. they are designed and deployed to serve a single application (LEONTIADIS *et al.*, 2012). Considering the amount of nodes scattered in an area with each one usually having various sensing units, the idea of sharing the sensing and communication infrastructure of WSNs through multiple applications has recently emerged. This concept is known as Shared Sensor Networks (SSNs) (LEONTIADIS *et al.*, 2012). SSNs are able to handle more than one application simultaneously in an efficient way by avoiding performing redundant tasks among applications, and by exploiting data aggregation and other functionalities of the network to increase resource utilization while meeting application requirements. Essentially, a SSN can be defined as a platform that allows the execution of multiple virtual sensor networks on top of a single physical infrastructure (LEONTIADIS *et al.*, 2012). Similarly to WSNs, the greatest challenge in SSNs arises from the energy-constrained nature of sensor nodes, which encourages the development of techniques to save as much energy from sensor nodes as possible, so as to prolong the network operational lifetime. On top of this, SSNs have new requirements and pose many new challenges that require adapting techniques/algorithms/protocols already successfully used in traditional WSNs. Among the several challenges raised by the SSN paradigm (LEONTIADIS *et al.*, 2012) (MADRIA *et al.*, 2014), to the best of our knowledge, there is no work about routing algorithms for these networks. Routing algorithms is one of the key challenges to be tackled in order to foster the widespread use of the SSN paradigm and it is the focus of our work.

1.1 MOTIVATION

Routing in WSNs is very challenging due to the inherent characteristics that distinguish these networks from other wireless networks. First, due to the relatively large number of sensor nodes, it is challenging to build a global addressing scheme. In addition, for WSN applications, often getting the data is more important than knowing the topological IDs of which nodes sent the data. Threfore, it is not relevant to address nodes by their topological ID (such as IP), instead the WSN applications are interested in addressing nodes by their attributes (geographical location, type of sensing units, etc). Furthermore, also due to their relatively large number, sensor nodes need to be self-organizing, especially as the operation of sensor networks should be unattended (or requiring a minimal human intervention). Second, in contrast to typical communication networks, almost all applications of sensor networks require the sensed data to be carried from multiple sources to either a single or few sink nodes. Third, sensor nodes are tightly constrained in terms of energy, processing, and storage capacities. Thus, they require careful resource management. Fourth, in most application scenarios, nodes in WSNs are generally stationary after deployment except for maybe a few mobile nodes. Nodes in other traditional wireless networks are free to move, which results in unpredictable and frequent topological changes. However, in some applications such as target tracking and visual sensor network applications (CHEN *et al.*, 2007), some sensor nodes may be allowed to move and change their location. Fifth, sensor networks are application-specific (i.e., design requirements of a sensor network change with application). For example, the challenging problem of forest fire detection is different from that of a periodic weather monitoring task. Sixth, position awareness of sensor nodes is important since data collection is normally based on the location. Finally, data collected by many sensors in WSNs is typically based on a common phenomena, so there is a high probability that this data has some redundancy. Such redundancy needs to be exploited by the routing protocols to improve energy and bandwidth utilization (AL-KARAKI *et al.*, 2004).

Due to such differences, many new algorithms have been proposed to address the routing problem in WSNs. These routing mechanisms have taken into consideration the inherent features of WSNs along with the application and architecture requirements. The task of finding and maintaining routes in WSNs is nontrivial since energy restrictions, and sudden changes in node status (e.g., failure) cause frequent and unpredictable topological changes (AL-KARAKI *et al.*, 2004).

Routing algorithms for WSNs typically have strong impacts on the network lifetime since radio communication is often regarded as the major source of energy consumption in these networks (AKKAYA *et al.*, 2005) (DIETRICH *et al.*, 2009). Routing protocols can balance the energy usage of sensor nodes by selecting appropriate routes dynamically, and also prolong network lifetime by reducing the number of message transmissions, since sending messages has a higher energy cost than processing in sensor platforms (AKKAYA *et al.*, 2005). Therefore, any routing protocol for WSNs must work in an energy-efficient way.

The routing challenges posed by the emergent scenario of SSN concerns the aspect that (i) executing a larger number of applications generates an enormous number of network transmissions and, consequently, increases energy consumption thus potentially reducing the network lifetime; (ii) multi-application demands may impose additional burden on some set of nodes and jeopardize the energy usage balance in the network, consequently reducing the whole network lifetime while many powerful nodes remains alive, characterizing a network partition. Therefore, routing protocols for WSNs are potentially inefficient for SSNs since most of such protocols are designed based on a single application using the network. Consequently, the development of routing algorithms for SSNs should be designed from scratch by considering the importance of reducing the number of message transmissions and balancing energy usage to prolong the network lifetime.

To minimize energy consumption, routing techniques proposed in the literature for WSNs employ some well-known routing strategies as well as tactics specifically tailored to WSNs, such as data aggregation and in-network processing, clustering, different node role assignment, and data-centric methods (AL-KARAKI *et al.*, 2004). Most of such routing protocols operate at the application level, using information fusion (NAKAMURA *et al.*, 2007) and data aggregation (RAJAGOPALAN *et al.*, 2006) techniques. Such solutions aim at reducing the number of transmitted packets (network transmissions), and consequently reducing energy consumption. Traditionally, most of these fusion techniques are performed at the application level by analyzing the sensed measures/data. We believe that it is possible to achieve further benefits and leverage the energy efficiency of SSN by providing a solution at the network (routing) level that acts in cooperation with application level strategies.

1.2 CONTRIBUTIONS

In this context, we propose THESEUS as an application-aware routing system for SSNs along with methods for reducing the number of message transmissions and balancing energy usage that are necessary to prolong the network lifetime. Such routing method is said to be application-aware because it uses techniques to select routes based on applications and QoS requirements. THESEUS routing algorithm is inspired by Proactive Routing with Coordination (PROC) protocol (MACEDO *et al.*, 2006), incorporating all the strengths of the original algorithms, extending and leveraging them for usage in the specific context of SSNs.

The fundamental idea behind PROC is to select a set of forwarding nodes (called coordinators) based on application demands. Such selection process is repeated periodically to balance the energy usage of coordinators, being important to mention that coordinators may show more energy usage than the regular nodes, because they receive and send more messages than those. The coordinators create the backbone for the routing process (MACEDO *et al.*, 2006).

Considering the specific features of SSNs, THESEUS detailed contributions are:

(i) THESEUS implements a technique for packet aggregation independent of the packet contents with the goal of prolonging the network lifetime. The advantages of using such technique are: to avoid a dependency on the data content from the aggregation process; and to work completely on the network layer.

(ii) THESEUS updates the coordinator selection process to meet the demands of multiple applications in order to better balance the energy usage. The aim of improving the PROC app function is to modify it for SSNs conditions such as considering the number of applications and network traffic during the coordinator selection to select more coordinators among the set of nodes that generate more data samples.

(iii) THESEUS makes use of QoS parameters and applications' requirements to adapt the routing paths.

(iv) THESEUS supports multiple sink nodes dynamically, bringing the ability of using more than one sink node in nework (which will be probably the typical case for SSN), and also bringing the ability to add or remove sink nodes while the network is working.

1.3 ORGANIZATION

The rest of this document is structured as follows: Chapter 2 describes the basic concepts; Chapter 3 discusses related work; Chapter 4 describes the proposal included the detail of our contributions; Chapter 5 details the implementation; Chapter 6 presents the evaluations, and finally, Chapter 7 contains conclusions and future work.

2. BASIC CONCEPTS

Our research is focused on routing techniques for SSN environment. Therefore, in this chapter we introduce the fundamental concepts of SSNs and routing techniques for wireless sensor networks, the two main areas of our research.

2.1 SHARED SENSOR NETWORKS

A Wireless Sensor Network (WSN) is a network composed of smart sensors, devices that are endowed with processing, storage, sensing and wireless communication resources. The communication capability allows the sensor nodes to be grouped, offering as benefits: (i) redundancy of communication channels that advantages fault tolerance (which does not occur with wired sensing systems); (ii) flexibility of installation and configuration and (iii) low maintenance costs. WSNs' nodes are devices with an energy source (usually non-rechargeable batteries) and limited computational capabilities. WSNs encompass a sheer number of such devices, often in the order of hundreds or thousands that act collaboratively with the purpose to monitor physical and environmental variables such as temperature, humidity, vibration and light intensity. Sensing devices typically used in the context of Smart Spaces are accelerometers, temperature sensors, humidity sensors and magnetometers. The data acquired by these sensing devices are sent to one or more sink nodes, which are computing devices that do not have the power limitations of the sensors and have higher processing capabilities. Sink nodes are part of a WSN architecture and also act as entry points for submitting application requests and as sensing data collection points (DELICATO *et al.*, 2012) (TILAK *et al.*, 2002).

In the context of conventional wireless sensor network (WSN), in order to maintain the network, a user needs to own a WSN, program the wireless sensors, deploy them and spend time and resources. The user is also limited to one application per sensor network. In recent years, the WSNs field has undergone several changes that influenced the design and operation of these networks. Among these changes, there is the emergence of Shared Sensor Networks (SSN) (LEONTIADIS *et al.*, 2012). Which, instead of taking into account a fit-for-purpose design with the primary aim of supporting a single application that belongs to a single authority (usually the owner of the infrastructure), allows the communication and sensing infrastructure to be shared by multiple applications that may belong to different users, thus optimizing the use of resources. The fact that SSN share the same sensing and communication infrastructure among several applications makes this kind of network one of the most promising solutions for

Smart Spaces applications. Without the infrastructure sharing, there would be unnecessary replication of the sensing and communication infrastructure as the number of applications increase (EFSTRATIOU *et al.*, 2010).

A shared sensor network is a WSN, which serves as a flexible infrastructure capable of supporting resource sharing between applications. Users can submit new applications to the shared sensor network through the sink node. Applications can be deployed dynamically at different times based on user demand. Furthermore, different applications can have different priorities according to their importance. Node sensors can be heterogeneous in terms of supported sensors and available energy. A shared sensor network works as a highly flexible infrastructure that supports different levels of resource sharing between applications. For example, multiple applications can share (1) one sensing unit in a sensor node e.g., a magnetometer can be used to detect parked cars and to detect of moving vehicles, (2) a sensor node with multiple sensing units and (3) one network with multiple applications on different sensor nodes. The SSN represent a total decoupling of applications and physical infrastructure of sensing and transmission (LEONTIADIS *et al.*, 2012).

Sharing WSNs is also known as Sensor cloud and virtual sensor networks (MADRIA *et al.*, 2014) with small differences in concepts. Sensor cloud is a concept of using virtual sensors that constructed on top of physical wireless sensors. The virtualization could change dynamically and automatically based on users applications' requests. Sensor clouds bring a number of benefits. Three main groups of these advantages are: providing better sensor management capability, sharing same sensed data between users, and removing the need for users to go through low-level details and challenges inside the network. A customized view of physical sensors that gathers filtered data for a user or a specified application is the concept of virtual sensors. As a matter of fact, resource-constrained sensors cannot handle multiple tasks as multiple VMs in the concept of cloud computing. Thus, sensor cloud uses "virtual sensors as an image in the software of the corresponding physical sensors and the user currently holding that virtual sensor" (MADRIA *et al.*, 2014).

In the following, several properties of both SSNs and application-specific WSNs are introduced and compared. By doing so, we can obtain a better understanding of the specific differences between SSN and application-specific WSN design. The analyzed properties are (1) Ownership; (2) Platform Dependency; (3) Code Modularization; (4) Resource Sharing; and (5) Application Information Sharing.

Ownership: The ownership property concerns the relationship between the hardware owner and the hardware user. These two parties are from the same authority or agreement is established between the two stakeholders before the hardware deployment. This is because no third party is allowed to get involved in the system operation after the system deployment.

As mentioned earlier, the application-specific WSN aims to run a single application on top of the infrastructure during the whole system lifetime. In such designs, all the resources are reserved for the one application. Moreover, the application objectives as well as the application composition rarely change, and requests from other applications are unlikely to be served at runtime. These design principles make the application-specific WSN behave like a closed system. This approach limits the possibility of outsourcing the hardware ownership to an external authority, as well as preventing third-party users from invoking services provided by the system. On the contrary, SSN must be built so that multiple applications are able to run on the same infrastructure, sharing the underlying hardware resources. More importantly, applications can be dynamically submitted to the system at runtime, regardless of whether any arrangement is established beforehand (LI *et al.*, 2014). With such design principles, the deployed system works like an open system offering a high degree of flexibility in hardware management (HUGHES *et al.*, 2009), system maintenance (EFSTRATIOU *et al.*, 2010) and application processing (RAICU *et al.*, 2008).

Platform Dependency: The property of platform dependency concerns how dependent the application is on the underlying infrastructure. The design and implementation of application components are based on the prerequisite that they should use the same node hardware, operation system (OS), and programming language.

In application-specific WSN, for reducing the complexity of the application development, the supported hardware is usually selected from the same manufacturer; all the nodes are preferably of the same model, run the same OS and communication protocols and have identical sensing capabilities. This design principle implies that application-specific WSNs are normally homogeneous. In this sense, when two system components communicate with each other, no complex and expensive bridging solution is required. However, this kind of system is usually bounded to specific requirements and restrictions posed by the platforms in which it is deployed, such as data rate, radio specifications and radio frequency (EFSTRATIOU *et al.*, 2010). Moreover, building a WSN with homogenous nodes regarding their sensing capabilities constrains the types of applications that can use such networks. If it is necessary to run the same application on a different type of node hardware, all code must be rewritten from scratch to accommodate the new programming language and the primitives provided by the new platform operating system (RODRIGUES *et al.*, 2013).

On the other hand, SSNs are often heterogeneous, which means the system is composed of multiple types of sensor hardware that are manufactured by different vendors, or even use different operating systems (OS), and programming languages. To cope with such heterogeneity, SSNs must provide appropriate mechanisms (FLORES-CORTÉS *et al.*, 2007) that hide the hardware-specific details from end-users and make the SSN operate as a homogeneous platform (JAYASUMANA *et al.*, 2007) for each running application. In this sense, (i) applications are not tied to the underlying sensor platform and (ii) the same infrastructure encompasses nodes with different hardware/software. The interaction between nodes from different platforms and even between multiple networks, designed with different technologies and protocol stacks, is required in SSN design. Such interoperability issues are often addressed by the insertion of an intermediary software layer that can be implemented following different approaches.

Code Modularization: Code modularization property concerns the relationship between the code at the application level and the code belonging to the underlying levels (communication protocols and OS).

In general, the development of application-specific WSNs is carried out under the assumption that the particular application is the owner of the physical network and this application is the only one that uses the hardware infrastructure. Therefore, all the application requirements are known a priori and WSN applications are developed as monolithic code installed on the nodes before the network deployment in the target area. This strong coupling leverages the customization of all the software layers in the network stack, and mostly aims at providing a high efficiency in terms of energy consumption. However, the design strategies for building the code for application-specific WSN are often ad-hoc and impose direct interaction of the application with the underlying embedded operating system, or even with hardware components of sensor nodes. Although such an approach is energy efficient, it generates rigid systems that are difficult to maintain, update and change, besides not promoting any reuse of software artifacts.

In the SSN, the presence of two user roles (LEONTIADIS *et al.*, 2012) (FARIAS *et al.*, 2014) is generally considered, namely, the infrastructure owner and the application owner. The infrastructure owner is assumed to have full control over the hardware infrastructure, while the application owners are assumed to have basic knowledge of the geography of the monitoring area and the functionalities provided by the network. One major requirement (DELICATO *et al.*, 2013) for SSN design is to enable newly arrived applications to be performed on the shared hardware infrastructure without interrupting the operation of previously running applications. To address such requirements, it is first necessary to provide solutions allowing, at development time, the clear separation of the application code and the underlying layers of code. Thus, the code to be installed on the sensor nodes can be built as a set of cohesive modules/components, with well-defined functions, instead of a monolithic piece of code encompassing multiple functions belonging to different abstraction levels. Second, it is necessary to break the tight coupling between the binary code and the physical hardware, built at compile and deployment times. However, both types of coupling (among the different layers of software to be deployed on a node and between the code and the underlying hardware) were adopted in application-specific WSN for the purpose of energy efficiency. Therefore, techniques used to break such couplings would allow the required flexibility and separation of concerns at the expense of lower energy efficiency. Such trade-off between flexibility/reusability/extensibility and energy efficiency is a challenge to be tackled in SSN design. Finally, it is important to isolate different applications in terms of the runtime environment inside the sensor node. Ultimately, code modularization in SSN aims at providing independent execution environments for each independently built application and making it operate in the same way as it would in an application-specific WSN.

Resource Sharing: The property of resource sharing (YU *et al.*, 2006) refers to the fact that the resources of a node can be used for different applications so that there is no need to replicate infrastructure to attend multiple applications. The property we address here concerns whether or not the available node resources are completely devoted to a single application during the system lifetime.

In application-specific WSNs, the idea of a single application utilizing the entire system causes fixed bonding between the nodes and the application, that is, all the available resources are reserved for satisfying the needs of a single application. Therefore, the resource allocation can be determined as early as possible in the network operational lifecycle. Unless the underlying hardware is changed at runtime (e.g. due to node movement, node replacement) and the real-time resource lookup is actually needed, the resource allocation can be statically done at compile time or deployment time (BHATTACHARYA *et al.*, 2010) (WU *et al.*, 2012). With these characteristics, the application-specific WSN shows a tight coupling regarding the resource sharing property.

In SSN, resource allocation happens at runtime instead, and sometimes at the latest possible time. This approach addresses a problem that makes the static resource allocation inefficient in SSN. The problem lies in the fact that resource contention could happen when multiple applications are running simultaneously within the same system. SSN allows multiple applications on top of the same infrastructure, with all the available resources opened for applications' dynamic arrival and thus requiring runtime decisions about which application to execute at each time. When applications arrive in the SSN, they will be dynamically allocated to a set of selected sensor nodes for further processing according to different factors, such as the latest node status, user demands and the priority of the applications (BHATTACHARYA *et al.*, 2010) (WU *et al.*, 2012) (LI *et al.*, 2013). The allocation of node resources (sensing, computation and communication) must not only meet the needs of simultaneously running multiple applications without causing interruption, but also comply with policies specified by different stakeholders. This further indicates that all the nodes have the chance to be used by any incoming application.

Application Information Sharing: This property concerns whether the network design assumes that the intermediate data produced by sensors can be shared among different applications running on the same system.

The major responsibility of an application-specific WSN is to transmit the collected data back to a device with sufficient computing and storage resource for further processing, in any of continuously, periodically or an event-based fashion. Data transmission has been widely recognized as one of the major energy consumers in WSNs. The in-network processing (e.g. data compression, data aggregation) is thus often used to reduce the size of the transmitting data to prolong the system lifetime. This imposes that all nodes between source and destination have to process the intermediate data with the same method or tool for the purpose of data encoding and decoding. By doing so, the messages are handled as a serialization of the same in-network processing technique in the system and are processed by the same method.

SSNs by default enable multiple applications to run simultaneously on the same system. Sharing application information (LE *et al.*, 2009) offers a great potential to save substantial energy in SSN due to the fact that tasks from different applications could simultaneously require the same data (and at the same rate) provided by a single sensor. Motivated by such necessity, information-sharing techniques are applied to SSN to achieve better energy conservation. Information sharing mechanisms used in SSN are generally designed as a cross-layer approach (VIJAY *et al.*, 2011), aiming to overcome the limitations of the layered protocol architecture by including more available information in a single message. However, the intermediate data of different applications might not adopt the same format and thus sharing information is dependent on the application format, which brings out interoperability issues. In order to enable information with different formats to be shared by applications, a commonly accepted format has to be adopted and all other formats need to be converted to it.

Finally, despite all mentioned potentials, the adoption of shared sensor networks poses new challenges, which must be surpassed to enjoy fully their envisioned benefits.

2.2 ROUTING ON SSNS

One of the critical issues for constructing SSNs at the network level is routing. Even in WSNs, routing was very challenging due to several characteristics that distinguish them from contemporary communication and wireless ad hoc networks. First of all, it is not possible to build a global addressing scheme for the deployment of sheer number of sensor nodes. Therefore, classical IP-based protocols is not efficient for sensor networks. Second, in contrary to typical communication networks almost all applications of sensor networks require the flow of sensed data from multiple regions (sources) to a particular sink. Third, generated data traffic has significant redundancy in it since multiple sensors may generate same data within the vicinity of a phenomenon. Such redundancy needs to be exploited by the routing protocols to improve energy and bandwidth utilization. Fourth, sensor nodes are tightly constrained in terms of transmission power, onboard energy, processing capacity and storage and thus require careful resource management (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Due to such differences, many algorithms have been proposed for the problem of routing data in WSNs. These routing mechanisms have considered the characteristics of sensor nodes along with the application and architecture requirements. Routing protocols in WSN can be categorized depending on the network structure, the protocol operation and as proactive or reactive (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Network structure: Flat-based (data-centric routing), Hierarchical routing, and Location-based routing protocols.

Flat network structure means that every node has the same role. Nodes cooperate to exchange packets. Since the network consists of a large number of nodes, it is not efficient to assign an address or a kind of identifier for each node. In that case, the routing works using queries. The node sends queries to a certain region of nodes waiting for data from these specific nodes. Therefore, the routing is data-centric utilizing the attribute-based naming queries to realize the communication. Usually in this category of protocols data aggregation is used during relaying. Data aggregation is a technique to aggregate the data collected through the network by combining similar data. That way, the amount of data to be transmitted is reduced and, as a result, the total cost is reduced. Several protocols fall into this category, the most important of them are *Flooding & Gossiping In flooding, Sensor Protocols for Information via Negotiation (SPIN), Directed Diffusion, Rumor routing, Minimum Cost Forwarding Algorithm (MCFA), Gradient-based routing, COUGAR, ACQUIRE,* and *Energy-Aware Routing* (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Flooding & Gossiping: In flooding, the node broadcasts the data to all its neighbors until destination is reached or until packet's TTL value equals zero. In gossiping, the data is sent to one randomly selected neighbor. The advantages of these ideas are that they are very simple and there is no need for state maintenance. Nevertheless, there are many redundant packets in the network, a lot of additional traffic, data overlapping and resources are not taken into account at all. Especially in gossiping there is an additional delay because it selects a random node to forward the data so, the delay appears in the propagation of data in the network (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Sensor Protocols for Information via Negotiation (SPIN) (AKKAYA *et al.*, 2005) is a group of adaptive protocols. This protocol disseminates the data from one or more nodes to the whole network. The nodes that are relatively close, maintain similar data. Therefore, the data is sent to nodes that are further away, which do not have it. SPIN protocols transmit the data into the network with three different types of messages. This way of communication is an improvement comparing to simple flooding due to the fact that it takes advantage of negotiation between the nodes and resource adaptation. When a node has new data to transmit, it sends the ADV (Advertisement) message to advertise this data to the neighbors via the data's metadata. The receiver of the message compares with what is known about it from this data and requests unknown data through the REQ message. The requested data is transmitted through the DATA message. SPIN offers significant energy saving mechanism comparing it to simple flooding. Additionally, due to the negotiation, significantly less redundant data is added to the network.

Last advantage of SPIN is that the topological changes do not have to bother all the nodes since only the adjacent nodes will know and learn the new topology. The family of the SPIN protocols consists of SPIN 1: a simple version of the protocol mentioned above, SPIN 2: extension to SPIN 1 using threshold in the resources, SPIN-PP: for point-to-point communication, SPIN-EC: similar to SPIN-PP but with energy heuristics added, and SPIN-BC: special for broadcast network (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Directed Diffusion is another data-centric protocol, which has no need for global identifier. It uses attributes combined with the values. Data from different nodes is combined inside the network. Thus, the redundant packets are fewer and the number of transmissions is decreased. Due to this idea, there is a decrease in energy consumption. In directed diffusion, a base station requests data by broadcasting interests. An interest is a task that the network needs to fulfill. Sensor nodes create gradients specifying value and direction. As interests are broadcasted and propagated through the network, gradients are updated serving the queries of the node. Eventually the query will reach the destination node. The intermediate nodes forward the data to reach the destination and create gradients to the source of the data. Along the way, the data is aggregated. When the node has new data, it updates the interest and retransmits it. The network reinforces one or a small number of specific paths (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Rumor routing seems like a different version of directed diffusion. In directed diffusion, the query is propagated in the network through flooding when there is no other information about the geographical position of the recipients. However, if the data being requested is small, flooding is not necessary. The concept of rumor routing is to send the queries to the nodes, which have recorded an event and not flood the queries to the whole network. For that reason, rumor routing uses long-lived packets, which are called agents. The moment the node discovers an event, it records the event to the events table and creates an agent. The agent goes through the network to reach the distant nodes and informs them about the events. Therefore, there is no flooding, a fact that is a significant improvement of directed diffusion. However, only one route between the source and the destination is used, a fact that can result in failure of communication if one node stops working. Rumor routing performs better when the events are relatively few. In the case of many events, it becomes infeasible to maintain so many agents and event tables (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Minimum Cost Forwarding Algorithm (MCFA) assumes the direction of the routes is known. Each sensor node maintains a least cost to the base station. Therefore, no routing table is needed. When a node receives a message, it checks if it comes from the least cost path between the source and the base station. If this is true, the node forwards the message to its neighbors (AL-KARAKI *et al.*, 2004).

Gradient-based routing stores the number of hops, which have been passed when the interest is diffused through the network. Therefore, each node can count the height of the node. The height is the minimum number of hops to reach the base station. The gradient on a link is the difference of the height between the node and the height of its neighbor. The packet is forwarded through the link with the greatest gradient (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

COUGAR uses declarative querying and tasking. The computing is distributed and it is done on the network. It uses in-network data aggregation for saving more energy (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

ACQUIRE comes from Active Query forwarding in sensor networks. Similar to COUGAR, considers the network as a distributed database. The query is sent by the sink node and each node that receives the query answers by processing the existing information. After that, it forwards the query to a neighboring sensor. If the existing information in the node needs to be updated, the node looks for the information from the neighbors who are at most d hops far away. When the query is resolved completely, it is sent back to the sink (AL-KARAKI *et al.*, 2004) (AKKAYA *et al.*, 2005).

Energy-Aware Routing is a protocol, which aims at saving as much energy as possible. It is a reactive protocol and destination initiated. The idea behind this protocol is to use different paths at different times. As a result, each path's energy will last longer. To achieve this, the protocol maintains several paths instead of one optimal comparing it to directed diffusion. The paths are selected taking into account probability of energy consumption. In fact, network lifetime is the only metric in this protocol and what only matters. It starts with localized flooding to discover all the routes between each pair of nodes, build the routing tables and find the cost of each route. Then it drops the high cost paths. An important disadvantage of this protocol is the setup phase, which can take more time than directed diffusion (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Hierarchical routing (based on network structure)**:** In wireless sensor networks, as to in other kind of networks, the idea of hierarchy is a very useful technique to use in routing. Higher energy nodes can be higher in the hierarchy of the protocol having the role of processing the information and transmitting it. Clusters and a gateway (cluster head) are created for each cluster. In a single-tier network, the gateway can be overloaded if the number of nodes is increased. If the gateway cannot handle all the nodes latency in communication will occur. Cluster heads aggregate and merge data so that fewer messages are sent to the base station. Since the nodes communicate within a cluster having a limited number of hops, the energy consumption is less. Generally, the main goals of hierarchical routing is what was just mentioned, fewer messages to the base station and less energy consumption. This concept was used in routing protocols for WSN; the most important ones are *LEACH (Low Energy Adaptive Clustering Hierarchy), PEGASIS (Power-Efficient Gathering in Sensor Information Systems),* and *Threshold-Sensitive Energy Efficient Protocols (TEEN and APTEEN)* (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

LEACH (Low Energy Adaptive Clustering Hierarchy) is a cluster-based hierarchical protocol. Each node uses a stochastic algorithm at each round to determine if it will become a cluster head for this round. Nodes that have been cluster head cannot be cluster heads for P rounds where P is the desired percentage of cluster heads. The probability for each node to become a cluster head in each round is 1/P. Thus, there is a rotation of cluster heads in order to evenly share the energy consumption between the nodes. Each node that is not a cluster head looks for the closest cluster head to become a member of this cluster. The cluster head creates a schedule of how to communicate with each node in its cluster to transmit the data. The cluster head compresses the data arriving from the nodes aggregates it and then sends it to the base station. The aggregation is realized in order to send less information to the base station. Each node communicates with the cluster head through TDMA (Time Division Multiple Access), the way the cluster head has decided. The communication happens with the least energy possible and when no communication is needed, the radio is turned off. CDMA (Code Division Multiple Access) is used with different CDMA codes in order to avoid interference among the cluster heads (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

PEGASIS (Power Efficient Gathering in Sensor Information Systems) is a chain-based protocol. The protocol is considered as an improvement to LEACH. In PEGASIS, each node communicates only with the closest neighbor. To communicate with the base station, each node has to wait for its turn. In one round, all the nodes have to communicate with the base station and after that, a new round begins. Thus, the energy consumption is fairly distributed among the nodes since all the nodes communicate with the base station one by one. As explained above, there is no cluster structure in this protocol; instead, there is the chain structure where each node has to wait its turn to collaborate in sharing the energy consumption. PEGASIS intends first to increase network lifetime by forcing the nodes to collaborate in energy consumption and second to decrease bandwidth consumption by forcing each node to communicate with the closest neighbor. To detect which neighbor is closest, the node uses signal strength indication to discover the distance to every node. Then, it adjusts the signal strength so that it can communicate with only one node, the closest one. Thus, a chain will be constructed consisting nodes that are closest to each other creating a route to the base station (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Threshold-Sensitive Energy Efficient Protocols are two protocols best suited for time-critical networks: *TEEN (Threshold-Sensitive Energy Efficient Sensor Network Protocol)* and *APTEEN (Adaptive Periodic TEEN)* (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

In *TEEN*, the sensor nodes are always in contact with the medium but they transmit data only when it is needed. The cluster head defines a hard threshold, which represents the threshold value of the sensed attribute, and a soft threshold, which represents a change in the value of the sensed attribute. If that change happens, the sensor node will turn on the radio and send the new data. By using the hard threshold, the protocol avoids unnecessary transmissions when the attribute is out of the range of interest. The soft threshold helps to reduce the number of transmissions by avoiding sending data when there is a little or no change in the sensed attribute. If a smaller soft threshold is used, there will be more data that are precise while more energy will be needed. By the time a new cluster head takes place new thresholds are broadcasted. The disadvantage of this protocol is that if the values are not received, the sensor nodes will not exchange data and the user will receive no data (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

APTEEN is similar to TEEN adapted to user’s needs. The user can choose how often each value will be used. The important characteristics of APTEEN are that it leaves the user to decide what to do and that it combines reactive and proactive routing (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Location-based routing protocols: Location-based routing happens when the protocol takes into account the location of the node. The location of the node can be specified by using signal strength indicators. A node can approximately calculate the relative distance between its neighbors by examining how high or low the signal strength from a neighbor is. If the signal strength is high, the neighbor is close. By examining all the signal strengths, a node can calculate relative coordinates. Another way to specify nodes' location is from a satellite by using a GPS (Global Positioning System) if the nodes have a GPS receiver installed. Two samples of such protocol are *Geographic Adaptive Fidelity (GAF)* and *Geographic and Energy Aware Routing (GEAR)* (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Geographic Adaptive Fidelity (GAF) is a location-based routing protocol with energy-aware characteristics. The area of the network is split into zones. Inside the zone the nodes work together to save energy. They choose one node to stay in wake up mode to monitor the network behavior and report what is happening back to the base station what happens while the rest of the nodes are in sleep mode. GAF saves total energy by turning off nodes that are not used. Each node specifies its position in the zone using GPS indication. Three modes are used in the protocol: *Discovery*, for discovering the neighbors in the area; *Active*, showing if the node is currently participating in the routing; and *Sleep*, the radio is turned off. The protocol handles mobility of the nodes by forcing the node to report what time it will leave the area. When the node is about to leave, the other nodes wake up and decide a new one to stay in wake up mode (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Geographic and Energy Aware Routing (GEAR) is a protocol that uses heuristics to define the position of the nodes and their energy state for routing. The concept is to decrease the data being sent in the network by transmitting the data in specific regions and not to the entire network. Each node maintains an estimated cost and a learning cost of reaching a destination. The estimated cost is calculated from the energy left and the distance to the destination. The learned cost is the estimated cost but with taking into account holes that may appear meaning not having any close neighbor in a specific area. If a hole appears, the route has to be changed (AL-KARAKI *et al.*, 2004) (AKKAYA *et al.*, 2005).

Routing Protocols based on Protocol Operation: Another classification of routing protocols in Wireless Sensor Networks is by protocol operation. The categories are presented below:

Multipath routing protocols: In this case, the routing protocol uses multiple paths between two nodes for routing. This idea can help in terms of redundancy and energy consumption. If there are two paths available, when the primary link breaks, the routing protocol will route the packets through the secondary link and the communication will continue flawlessly. Concerning energy consumption if there are several paths available the energy of the nodes of each path will not be consumed so quickly since the paths will change. However, the switching of paths can consume energy as well (AL-KARAKI *et al.*, 2004).

Query-based routing: In the query based routing, the node sends a query to the network asking for data. The node, which has the sensing data asked, will transmit it back to the node, which initiated the query. Directed Diffusion is an example of this type of routing with queries. Data aggregation is a good solution for saving energy (AL-KARAKI *et al.*, 2004).

Negotiation-based routing: In this routing negotiation is used among the nodes in order to minimize transmitting data that has already been transmitted. Data descriptors are used for this negotiation and the result is increase of network lifetime. SPIN is an example of that kind of routing (AL-KARAKI *et al.*, 2004).

Quality of Service routing: The network in this case has to balance between energy consumption and QoS parameters like delay, bandwidth (AL-KARAKI *et al.*, 2004).

Coherent and Non-coherent Processing: This is a data processing technique, which is closely connected with the routing in a protocol. In non-coherent processing, the main processing is done by the nodes locally and then from other nodes for more processing. In coherent routing, the nodes apply a minimal processing and send it to the aggregators for the main processing (AL-KARAKI *et al.*, 2004).

Proactive and Reactive Routing: If someone thinks of how the source finds the route to destination, the protocols can be categorized in reactive, proactive and hybrid mode.

Reactive routing: In this type of routing routes are computed on demand. A node sends a request that wants to communicate and by receiving a route reply message the communication can begin. The disadvantage of this concept is high latency may appear during the procedure of finding routes. In addition, the network can be overloaded if the flooding is heavy. On the other hand, this type of routing is bandwidth efficient (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Proactive routing protocols create and keep routing information for all the nodes whether this information is needed or not. The information is collected by using control messages periodically. Proactive routing protocols are not bandwidth efficient since there are a lot of messages being exchanged without all of them being useful. The main advantage of proactive routing is that it is easy to get routing information and easy to establish a session. The drawbacks are first the heavy load of unnecessary data saved for routing and the difficulty in restructuring the communication when there is a link failure (AKKAYA *et al.*, 2005) (AL-KARAKI *et al.*, 2004).

Hybrid routing (Reactive and Proactive) is a type of routing, which combines advantages of both reactive and proactive routing. The routing is initially established with some prospected routes and then serves the demand from additionally activated nodes through reactive flooding (AKKAYA *et al.,* 2005) (AL-KARAKI *et al.*, 2004).

When sensor nodes are static, it is preferable to have table-driven routing protocols rather than reactive protocols. A significant amount of energy is used in route discovery and setup of reactive protocols. Another class of routing protocols is called cooperative. In cooperative routing, nodes send data to a central node where data can be aggregated and may be subject to further processing, hence reducing route cost in terms of energy use. Many other protocols rely on timing and position information (AL-KARAKI *et al.*, 2004).

Routing in WSN uses several different techniques to be energy efficient and scalable. That is why there are so many different protocols. Each protocol may use techniques from different categories trying to be more efficient. It depends on the network and which are the user's goals to specify which protocol is the best for this occasion. The conventional routing protocols developed for application-specific WSNs usually attempt to achieve routing efficiency by exploiting the application layer query semantics and proposing an all-in-one solution that weaves the routing concern with other application layer concerns, such as data-centric and service-centric routing. They also tend to optimize routing performance for a specific communication pattern inspired by a specific class of WSN applications. In a SSN, where multiple applications run within the same network infrastructure, each application presents its own set of requirements that must be dealt with and exploited by routing protocols in order to guarantee energy efficiency while forwarding data.

3. RELATED WORK

Several different routing protocols were proposed for WSNs in the last decade, all of them designed based on single application demands (according to the original definition of WSNs). The context of SSNs typically comprises several different applications, each one with potentially different requirements (such as different sampling rates, sensing coverage, data accuracy, delay and sensing demands). A limited number of papers were found in the literature proposing routing protocols specifically tailored for SSNs (ELTARRAS *et al.*, 2010) (HEFEIDA *et al.*, 2011) (SHAH *et al.*, 2012) (INOUE *et al.*, 2014), since sharing a WSN among multiple applications is a recent paradigm.

The adaptive multi-criteria routing (AMCR) (ELTARRAS *et al.*, 2010) was proposed as a routing framework for SSNs, and one of its primary goals was to be a generic routing framework for multi-application demands. Its authors, Eltarras and Eltoweissy (ELTARRAS *et al.*, 2010) state that changes in the network underlying resources, connectivity, mission, or QoS requirements demand the design of adaptable SSN architectures and protocols to increase the network lifetime. AMCR adopts a descriptive criterion based on an addressing scheme for improving application scalability and reducing broadcasts overhead by using index tables and updating them on demand. AMCR allows destination addresses to be specified as a qualitative reference to node capabilities, administrative settings, and/or application published criteria. THESEUS differs from AMCR in several aspects. First, AMCR is a conceptual routing framework, and so it does not present details of the specific algorithms used for achieving its main proposed features while THESEUS is a concrete system with a particular architecture and algorithms. Second, while AMCR uses its addressing scheme based on descriptive criteria for improving scalability, THESEUS relies on the characteristics of PROC for ensuring scalability as a distributed system. Third, THESEUS includes a packet aggregation algorithm, which helps improving network lifetime. Such aggregation strategy is not found in AMCR. Fourth, THESEUS has its own algorithm for selecting dynamic and QoS-aware routes, while AMCR is said to be able of exploiting the message semantics and adapting itself to the observed application characteristics in order to support the efficient operation of the SSN. In addition, because of exploiting application semantics, the conception of AMCR is more tied to the application than THESEUS. In the development of THESEUS, we specified a well-defined interface between the application and network layers.

The context-aware protocol proposed in (HEFEIDA *et al.*, 2011) was developed for supporting collaborative sensor network applications. In (HEFEIDA *et al.*, 2011), the application, network, and physical layers were combined and modeled as a single protocol stack (cross-layer development), to use the information of each layer in another one (what is said to result in context awareness). The primary goal was to promote load balancing in the network and the ability to use data aggregation. This type of integration brings a high level of energy efficiency but at the expense of losing the generality and makes the protocol more complex for implementation. The authors mentioned the benefits of their proposal as: (i) getting the running application(s) involved in lower layer decisions, and so giving them the ability to control/tailor the network behavior; (ii) integrating context parameters of interest in WSNs (e.g. battery life, delay, mobility) into a single framework allowing nodes to make better decisions; (iii) communicating context parameters among all layers (interlayer context sharing); (iv) communicating context parameters and node state among nodes (inter-nodal context sharing); (v) distributing the load over the entire network to achieve load balancing and prolong network lifetime. THESEUS shares these benefits with this related work, but THESEUS differs from this work in its clear separation of layers, with well-defined interfaces for integration. THESEUS has been defined at the network layer with an explicit interface to the application layer only in the sink node, what makes THESEUS agnostic to the specific application running on the network, so that it can be used in several application scenarios and for different types of applications.

The aforementioned works were found more strictly related to THESEUS, considering that both exploit applications semantics for routing purposes. The following works do not explicitly use such semantics of applications (application level information) for routing purposes.

The work in (SHAH *et al.*, 2012) is based on prioritizing the applications and on pricing different paths so that each application will select the most appropriate route for its data. For instance, an application with low priority will try to find the cheapest possible route, but without considering the delay of this route; another application with high priority will attempt to find the fastest routes, but without considering route prices. In our solution, we divided applications into two categories for establishing priorities: applications based on event-driven data (high priority) and applications based on continuous dissemination data (normal priority). A QoS parameter in the sink node defines the maximum accepted delay time for each category of application data.

The authors in (INOUE *et al.*, 2014) study a particular case for SSNs. They explored the sharing of nodes from different WSNs for a single application when two or more overlapping WSNs have been deployed in the same geographical area. The solution proposed by the authors is based on selecting and sharing some nodes within the WSNs to build energy efficient routes. However, the radio component of those nodes is required to support multi-channel TX/RX for the proposed approach to work correctly. Such radio component is more expensive, in terms of energy consumption and monetary cost, than most radio components commonly used in WSN platforms. In comparison with THESEUS, our solution differs by aiming at sharing the same WSN for multiple applications and not multiple WSNs for a single application. Therefore, the work in (INOUE *et al.*, 2014) does not support multiple applications, as THESEUS does. Thus, the concept of SSN used in (INOUE *et al.*, 2014) is different from the concept of SSNs introduced in our work and all other related work. We claim that supporting multiple applications (our concept of SSNs) is mandatory, because it is possible to avoid performing redundant tasks among applications to increase resource utilization while meeting application requirements.

Finally, it is worth to remind that none of these related research supports multiple sink nodes in the network, however, using more sink nodes in SSNs could play a key role for balancing and reducing the energy usage of the nodes.

Table 1 shows a brief review of these related works.

Table 1. Related works comparison

	THESEUS	**(ELTARRAS *et al.*, 2010)**	**(HEFEIDA *et al.*, 2011)**	**(SHAH *et al.*, 2012)**	**(INOUE *et al.*, 2014)**
Energy Aware	Yes	No detail	No detail	No detail	Yes
QoS Aware	Yes	Yes	No detail	Yes, but not directly	No detail
Main Goal	Energy saving and balancing	Energy saving	Load balancing in the network	Segregates traffic flows	Prolonging network lifetime
Architecture	Yes	No	Yes	No	No
Aggregation	Packet aggregation	No	Able to use	No	No
Implementation	TinyOs	No	NS-2.29	NS-2	No detail
Evaluation	Real scenario and simulated by AVRORA	Analytical	Simulated by NS-2	Simulated by NS-2	QualNet 4.5.1 simulator
Multiple sinks	Yes, dynamically	Not mention	Not mention	Not mention	Yes

4. PROPOSAL

The goal of this work is proposing a multi-application-aware and *proactive* routing system for shared sensor networks (SSNs), called THESEUS. The proposed system is defined as a routing method that has an interface with applications in order to adapt the chosen paths according to their demands. THESEUS is specifically tailored for the scenario of SSN, where multiple application demands need to be met without jeopardizing the scarce resources of the networks nodes.

As part of the methodology adopted in this work to design THESEUS, the first step was to search, in the literature, for application-aware routing protocols in the context of WSNs to find the existing solutions for the challenges related to routing in WSNs and SSNs. Through this investigation, Proactive Routing with Coordination (PROC) was found as a well-defined routing protocol for WSNs (MACEDO *et al.*, 2006), which is classified as a proactive, cooperative, dynamic and application-aware protocol. PROC is one of the first WSN routing protocols that interact with the application for determining which nodes are more suitable to route data (MACEDO *et al.*, 2006). In the following, PROC is briefly described, since it was the starting point for our proposal. Next, the network model and assumptions considered in this work are defined. Finally, our proposal, THESEUS routing system is presented, which is specially tailored to SSN.

4.1 PROC OVERVIEW

The fundamental idea behind PROC is to select (periodically) a set of forwarding nodes (called *coordinators*) based on application demands. The coordinators create the routing backbone and all other nodes (called *regular* nodes) will directly connect to one of them forming a treelike structure. Therefore, in PROC each node can be either a *coordinator* or a *regular node*.

The routing establishment is based on the node role and the information about its neighbors. Coordinator nodes must select their respective parent nodes among their neighbors (which may also be coordinators or only regular nodes). The parent of a regular node is selected based on the following priorities, in descending order: (1) nodes located at the shortest hop distance to the sink, (2) nodes holding a coordinator role, and (3) nodes with the greater available energy within their neighbors. The parent of a coordinator node is selected based on the following priorities, in descending order: (1) nodes holding a coordinator role, (2) nodes at a shortest hop distance to the sink, and (3) nodes with the greater available energy within their neighbors. Such prioritizing is used to minimize the number of coordinators while finding the shortest paths from each sensor node towards the sink node. Using such prioritizing scheme, along with choosing to start routes' creation from the nodes inside the network (and not from the sink node) are features that avoid cycling in PROC. Cycling occurs whenever a route created by a routing protocol gets into a loop inside the network, and its creation process never ends. Therefore, due to its features, PROC does not require any additional mechanism for cycle detection.

According to the authors, PROC is an application-aware routing protocol. This awareness is achieved by considering the application requirements for calculating a value that is used as the probability of a node becoming a coordinator. This calculation is performed by a function called *app function*. The app function is deployed on all nodes and is invoked every time the routes need to be created, in order to calculate the mentioned probability value based on each node parameters (such as number of neighbors and remaining energy). The app function in PROC is defined by three rules. These rules are considered as application requirements with the main goal of saving energy. First, recent coordinators will have a lower probability of being again a coordinator for a given predefined period (an adaptation of LEACH (HEINZELMAN *et al.*, 2000) technique). Second, having more neighbors reduces the chance of being a coordinator, in order to avoid selecting more coordinators in denser areas. Third, nodes near to the sink node have a higher likelihood of being a coordinator in order to divide the high network traffic near to the sink among more nodes (since this is a region with heavy traffic).

Considering energy consumption, Macedo et al. (MACEDO *et al.*, 2006) state that it is possible to save more energy (and then increase the network lifetime) by shutting down the regular nodes (nodes that do not play the role of coordinator) whenever the application does not require information from such nodes. However, it is important to mention that such shutting down feature was not detailed, implemented, or evaluated in PROC. PROC always tries to minimize the number of coordinators based on the selection process. PROC proved guaranteeing the network connectivity of the monitored region based on its route establishment process. Therefore, topology control algorithms only need to care about sensing coverage. Periodical backbone reconstruction is a fault tolerance mechanism that makes PROC a robust protocol.

4.2 SSN MODEL AND ASSUMPTIONS

In this work, the following assumptions are made. The SSN is composed of a set of sensor nodes and one or more sink node(s), all of which are organized in a flat network topology (Figure 1). We also considered that all nodes (either sensor or sink nodes) in the SSN are static. THESEUS does not depend on the information of nodes' positions for its operation. However, once the network is deployed, nodes must remain at the same positions during the whole execution of THESEUS (node mobility is not considered). Moreover, sensor nodes are homogeneous in terms of processing units and memory capacities. However, each node can have different types of sensing units (meaning that the nodes are able to sense different types of environment variables). In addition, THESEUS supports sensor nodes with different energy sources, i.e. nodes can differ in terms of their energy power.

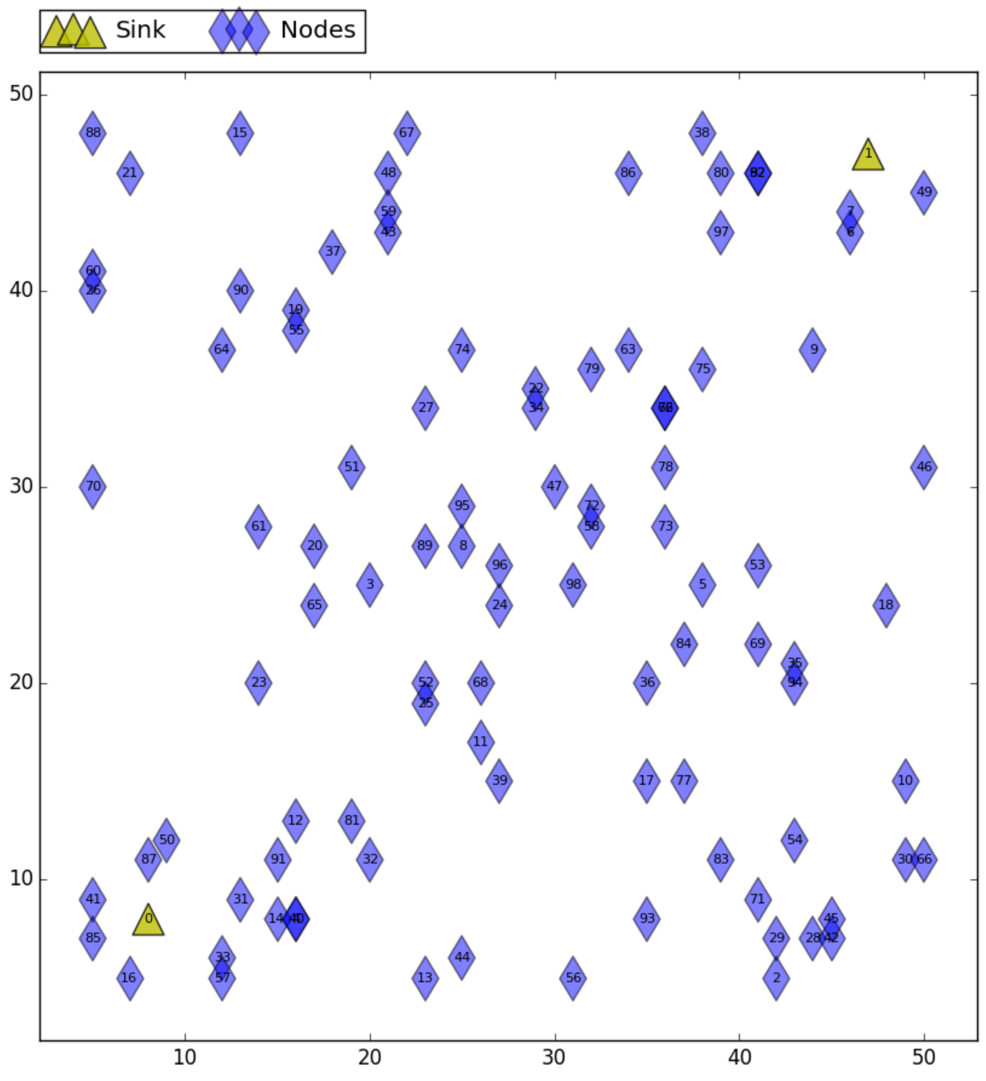

Figure 1. Sample of network topology

The sink node has more powerful hardware components than sensor nodes, and it is connected to a constant and unlimited energy source. In our SSN model, nodes could be deployed in any random position in all three dimensions. However, each node must be in the wireless range of at least another one that can reach a sink node potentially through multiple hops, thus ensuring the connectivity of all nodes. Therefore, we assume the network density is high enough to assure radio connectivity during the whole network lifetime, since we believe this will be typically the case in SSNs.

4.3 THESEUS

THESEUS is a routing system for SSNs that dynamically updates the routes based on applications' demands and nodes' conditions (node contextual information). By nodes conditions we denote the node hop distance towards the sink node, its number of neighbors and available energy.

Similar to PROC, THESEUS routing algorithm operates based on selecting a set of forwarding nodes as coordinators. Coordinators create the backbone for routing; all other nodes will directly connect to one of the coordinators forming a treelike structure. The sink node is responsible to periodically trigger the coordinator selection and route creation processes, but such processes are performed by the sensor nodes, in a distributed way (taking advantage of in-network processing). The reconstruction of the routes (backbone) happens in time intervals called cycles.

THESEUS is defined completely in the network layer and has one interface to interact with the application layer in the sink node, thus avoiding many cross-layer communications. In addition, THESEUS is implemented as a routing protocol for different applications in an SSN environment, without requiring any customization in its architecture and algorithms (it is just necessary to adjust the parameters based on the specific requirements of each applications). In this sense, THESEUS routing system is considered agnostic to the specific application running in the network.

THESEUS supports both types of data delivery models commonly existing in WSNs (TILAK *et al.*, 2002): (i) EVENT-DRIVEN and (ii) CONTINUOUS. In the EVENT-DRIVEN data delivery model, data is transmitted whenever an event of interest (for the application) occurs. In CONTINUOUS data delivery model, data is sent periodically according to a time interval, which is predefined by the application (TILAK *et al.*, 2002). In comparison, PROC supports a single sink node in the network and only continuous data delivery model (TILAK *et al.*, 2002). The data routing method in PROC is store-and-forward, which only forwards each data packet to the respective parent node. THESEUS manages to support both types of data delivery models by using the respective priorities of each type of data delivery models based on the application QoS in terms of maximum tolerated delay.

Another feature of THESEUS is supporting multiple sink nodes on the network, also supporting adding and removing sink nodes dynamically. Multiple sink nodes could improve the network functionality since it can reduce the number of hops from sensor nodes towards the sink node, which helps to reduce the nodes energy usage and probability of packet collisions on the network. Moreover, the ability to adjust the network for adding or removing sink nodes dynamically brings the option of adding more sink nodes in case of necessity such as more applications arrival. This ability makes THESEUS more adjustable and scalable for dynamic usage of SSNs.

THESEUS improves PROC in four ways: (i) it implements packet aggregation independent of the packet contents with the goal of saving energy, thus prolonging the network lifetime; (ii) the coordinator selection process is modified to cover multi-application demands (so it is a process tailored for SSNs) in order to better balancing energy usage; (iii) it uses QoS parameters (such as maximum accepted delay for each type of data delivery model) and application requirements to adapt the routes accordingly, thus saving further energy while meeting application needs; and (iv) it allows the presence of more than one sink node in the network with the option of adding or removing sink nodes dynamically, thus increasing the scalability of the solution.

In this section, we first describe the details of our packet aggregation algorithm (concerning item (i)), and THESEUS coordinator selection process (the app function, which concerns item (ii)). The item (iii), regarding the use of QoS parameters, is addressed in both packet aggregation algorithm and THESEUS app function. Next, THESEUS architecture is presented. Finally, the behavior of each sub-system and component that comprises THESEUS architecture is described, including the details of how THESEUS supports multiple sink nodes (regarding item (iv) above).

4.3.1 THESEUS packet aggregation

THESEUS packet aggregation algorithm works at the network level and it is independent of the packet data content (see Figure 2). The main goal of our proposed aggregation algorithm is to maximize the packet data field usage as follows. Network packet formats have different fields and sizes based on the specific WSN platform. Generally, network packet formats have three main parts with a limited size for each one: packet header, data, and trailer. The packet header and trailer (MAC layer information) are used for packet transmission by physical network layer. If a data size is bigger than the platform given data field size, then the physical layer is able to divide it into several packets for transmitting. On the other hand, if a data size is smaller than the data field size then the packet is transmitted with a smaller size than the maximum supported packet size. Therefore, sending more number of small packets uses more bits of data for transmitting them because of the packet header and trailer overhead. On the other hand, merging small data into one packet and transmitting them reduce the transmission number of bits and number of the packets, resulting energy saving.

All data packets in a WSN/SSN should be eventualy delivered to the sink node(s). Each network packet has some MAC layer information (such as packet CRC, source mac address, and destination mac address). Therefore, in case of aggregation, the source address (application

```
procedure Data-Manager ( )
1:   if MData.aggflag = false then                                    // not possible to aggregate data
2:       send(MData, parent.address);
3:   else if MData.emptyspace <= 4 bytes then
4:       MData.aggflag = false;
5:       send(MData,parent.address);
6:   else
7:       THESEUS-packet-aggregation ( );
8:   end if
end procedure

procedure THESEUS-packet-aggregation ( )
9:   if MData.typeflag = 0 then                                       // for continuous data type
10:       if storedp.continuous = ∅ then
11:           storedp.continuous = MData; start timer.C;
12:       end if
13:       while (timer < MDTC) & (stored.continuous.emptyspace > 4) do    // from  Sync.QoSparam MDTC:
                                                          Maximum accepted Delay Time for Continuous data type
14:           try add (MData.sourceaddress, MData.appdata) to (storedp.continuous.appdata)
15:               catch error {send(MData, parent.address);};
16:       end while
17:       stop timer.C; timer.C = 0; stored.continuous.aggflag = false;
18:      Data-Manager (stored.continuous);
19:  end if
20:// same from line 9 to 19 for event driven data with related maximum accepted delay (MDTE)
end procedure
```

Figure 2. THESEUS Packet aggregation algorithm

level address, node ID), and the application data (sensed data) fields should be stored, and other fields could be excluded (MAC layer information). When the aggregated packet is sent, the radio component adds new MAC layer information for the aggregated packet. The use of two bytes for the source address (application level address) could support 65535 nodes in a network. Since large-scale WSNs are considered using thousands of sensor nodes (LI *et al.*, 2011) we considered two bytes for the source address in this work. However, this size is adjustable for other use cases. Moreover, most of the sensors measure environmental data in maximum two bytes size (MICAZ & MICA2, Accessed May 2015). Therefore, the minimum unused size of a packet data field should be 4 bytes (2 bytes of source address + 2 bytes of sensed data) to be able to aggregate another packet inside itself. In addition, we considered the first two bytes of a packet data field for representing a flag field. The flag bits are used for controlling the aggregation process when aggregated packets are received by the next hop in the path towards the sink. Consequently, the packet data field of THESEUS includes the flag (2 bytes), the first monitored data address (2 bytes), the first monitored data (2 bytes), the second monitored data address (2 bytes), the third monitored data (2 bytes), and so on, until the data field size is reached (depending on the maximum data field size supported by the WSN platform).

It is possible to prioritize the two types of data delivery models (continuous and event-driven) according to the expected delivery delay time in the sink node as follows: event-driven data has higher priority (lower delay time tolerated) than continuous data type. Therefore, THESEUS packet aggregation operates based on a QoS parameter, called the “maximum accepted delay time (MDT)”, which represents the highest amount of time tolerable by applications for having their data delivered. The MDT has different values for each type of data delivery models: the MDT for continuous data type (called MDTC), which should be the minimum accepted delay time through all applications; and the MDT for the event driven data type (called MDTE).

The coordinator checks the data field of the received data packet; if there is not enough empty space, then the aggregation flag bit changes to false (to inform the next coordinator to just forward this packet without further analysis) and this packet is forwarded through all coordinators to the sink node without any delay (*Data-Manager* Procedure, Figure 2). Otherwise, the packet is stored in the node memory. The aggregation algorithm tries to add other received packets inside the stored packet during the maximum delay time defined by applications. After the specified time has elapsed, or if the maximum packet size is reached, then the aggregation flag bit changes to false and the packet is forwarded to the sink node without any further delay. This process is the same for both data delivery models types, continuous dissemination, and event-driven, with the respective QoS parameters and variables (*THESEUS-packet-aggregation* procedure, Figure 2).

4.3.2 THESEUS application function

Similarly to PROC, in THESEUS it is possible to define different objective functions (called app functions) responsible for calculating the probability (in percentage) of a node becoming a coordinator. This probability is computed based on the node conditions (such as the number of neighbors and remaining energy), applications' demands and QoS requirements (such as maximum accepted delay for delivering data samples). In traditional WSNs, a single application would require a simple app function. In SSNs, multi-application demands bring the necessity of combining all applications requirements, thus increasing the complexity of the app function. Many techniques, such as linear programming or AI could be used to define the app function for properly combining applications requirements. However, the use of complex techniques is out of scope of this work.

In this work, the app function is defined according to the following steps: (i) specifying a set of rules based on the applications' requirements, (ii) defining equations for each rule, and finally (iii) creating the app function using the defined equiations. We considered the balancing energy usage within all nodes as the main goal for our app function. We defined rules in our work following a heuristic-based approach. In such approach, we considered the WSN behavior regarding the nodes energy usage. Thus, the set of rules of THESEUS (R1, R2, R3 and R4 in Equations (1), (2), (3) and (4)) are generated aiming to balance the energy usage in the network to prolong the system lifetime. The four default rules (R1, R2, R3, and R4) for THESEUS are as following.

First, R_1 in Equation (1) calculates the probability (in percentage) of a node being coordinator, considering if this node has been recently coordinator and/or if it has been coordinator for many cycles. The coordinator nodes use more energy than other nodes since they are responsible to receive data messages from other nodes and forward them to the next coordinators, besides executing their monitoring tasks (generating samples). Therefore, regarding network transmissions, a node with coordinator role uses more energy than other nodes. In this case, we expect that these nodes should have a lower probability of being a coordinator for another cycle. This method is an adaptation of LEACH cluster heads selection technique (HEINZELMAN *et al.*, 2000).

$$R_1 = 100 - \left[(coord(curCycle - 1) * 50) + \left(\frac{coord.size}{curCycle} * 50\right)\right] \quad (1)$$

where, *coord* (dynamic parameter) is an array of cycle ids during which the node was coordinator; *curCycle* (dynamic parameter) is the current cycle id and *coord.size* is the number of cycles during which the node was the coordinator (size of coord array). So, in Equation (1) if the node was not coordinator in the last cycle nor during all the previous cycles, then the probability of the node being coordinator, (R_1) is 100%. On the other hand, we can observe in Equation (1) that if the node was coordinator in the last cycle, then the probability of the node being coordinator, (R_1) is subtracted by 50%. Additionally, if a node has been coordinator for many cycles, the probability of being coordinator (R_1) is subtracted by 50% of the ratio between the number of cycles that the node was coordinator and the total number of cycles. For example, if a node was coordinator in the last cycle and during all the existing 6 cycles (since the network deployment) it was a coordinator in 3 of these 6 cycles; so it has 25% (100-(50+25)) of chance of being coordinator according to the first rule.

Second, R_2 in Equation (2) calculates the probability (in percentage) of a node being coordinator when it is in a dense area. In this case, we expect that nodes having more neighbors (higher density area) should have lower probability of being coordinator in order to avoid selecting more coordinators in denser areas. This rule, along with THESEUS packet aggregation algorithm, creates a synergy that did not exist in PROC. Reducing the number of coordinators in a denser area (by rule R2) fosters the aggregation because fewer coordinators will have a higher chance to aggregate packets of more neighbors. In other words, a limited number of coordinators (within a higher number of regular nodes) receives more samples at the same time, resulting improvement of the number of aggregated samples per packet. This synergy results in a more efficient use of each coordinator in the aggregation process.

$$R_2 = 100 - \left(\frac{neighbors.size}{max.neighbor + 1} * 100\right) \quad (2)$$

where *max.neighbor* (static parameter) is the maximum number of neighbors that a node can have. Parameter *max.neighbor* is calculated based on the amount of memory required for storing information from a neighbor. Therefore, it must be configured according to the adopted platform. The parameter *neighbors.size* (dynamic value) is the number of neighbors a node has. For example, if the implementation imposes *max.neighbor* = 20 and a node has 15 neighbors (*neighbors.size*), therefore R2 = (100-((15/20)*100)) = 25%.

Third, R_3 in Equation (3) calculates the probability (in percentage) of a node being coordinator when it is near to the sink. In this case, we expect that nodes near to sink node have higher probability of being a coordinator in order to balance the high network traffic near to the sink among more nodes.

$$R_3 = \frac{100}{hops + 1} \tag{3}$$

where *hops* (dynamic parameter) is the number of hops from the sink node (each node discovers this value based on the information of its parent, described in section 4.3.4). Nodes directly connected to the sink node have zero hop distance. For example, if a node was connected directly to the sink, then it will have 100% chance of being a coordinator according to R_3 while a node with four hops distance to sink will have 20% of chance.

Forth, R_4 in Equation (4) calculates the probability (in percentage) of a node being coordinator when it is located in areas with higher rate of data generation (the source area for a monitored phenomenon). In almost all SSNs, continuous data dissemination from sensor nodes to sink contributes to the biggest part of data flow (KULKARNI, 2004) (DEMIRKOL *et al.*, 2006). Therefore, rule (R_4) states that the node that generates more data, i.e. the node that has more application requests, should have a higher possibility to become a coordinator. This rule is the same rationale used for R_3, contributes to sharing the network traffic among nodes, balancing the energy usage in the network. For this rule, it is necessary to calculate the sampling rate for each area and send these values to the nodes. The concept of *area* in the context of this work is used to denote the list of nodes required by an application (meaning the nodes deployed in the application target area).

$$R_4 = \frac{SPpMTI(i)}{mSPpMTI} * 100 \tag{4}$$

where *SPpMTI* (dynamic parameter) is an array of the sample rates per maximum time interval for each area (list of nodes used by an application) of the network; *mSPpMTI* (dynamic parameter) is the highest sample rate per maximum time interval; and *i* is the node area id. For example, consider that one area should generate 14 samples per 20 minutes (meaning 21 samples per 30 minutes), another area 10 samples per 30 minutes and the last area 10 samples per 10 minutes (meaning 30 samples per 30 minutes). In this case, *mSPpMTI* is 30 samples per 30 minutes. Therefore, the probabilities of a node becoming coordinator by R_4, for the three mentioned areas are respectively: 70% (21/30*100), 33% (10/30*100) and 100% (30/30*100).

Based on these rules, the app function and QoS parameters are defined by the following formulas:

$$Appparam: apx.networksize, SPpMTI, mSPpMTI, a, b, c, d \quad (5)$$

$$Appfunc(coord, curCycle, neighbors, hops, i) = \left(\frac{R_1 + R_2 + R_3 + R_4}{4}\right) \quad (6)$$

where *Coord*, *curCycle*, *neighbors*, *hops* and *i* are input local parameters of the node, and are the same parameters used in equations (1) to (4). $R_1 \ldots R_4$ are the rules equations. R1, R2, R3 and R4 rules calculate the probability of a node being coordinator.

As an example, if we assume R1= 25%, R2 = 25%, R3 = 20%, R4 = 70% for a node, the app function will return ((25+25+20+70)/4) = 35% as the probability of a given node being coordinator.

4.3.3 THESEUS architecture

THESEUS logical architecture encompasses software components to be deployed in two different types of nodes: Sink and Sensor nodes. THESEUS Sink architecture (Figure 3) encompasses the *SSN Manager* component and *Sink Manager* (SM) system that contains *App Function & QoS parameters database*. The *SSN Manager* is responsible for gathering the requirements from all applications and creating a set of rules to build the *app function* and QoS

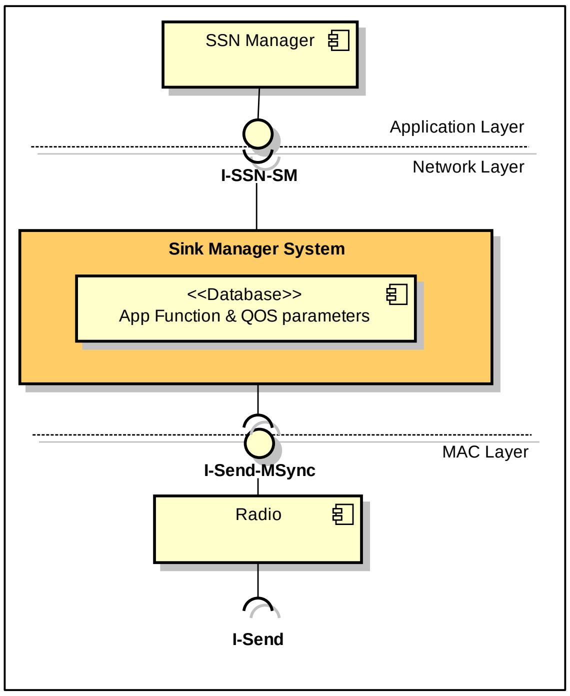

Figure 3. THESEUS Sink Architecture

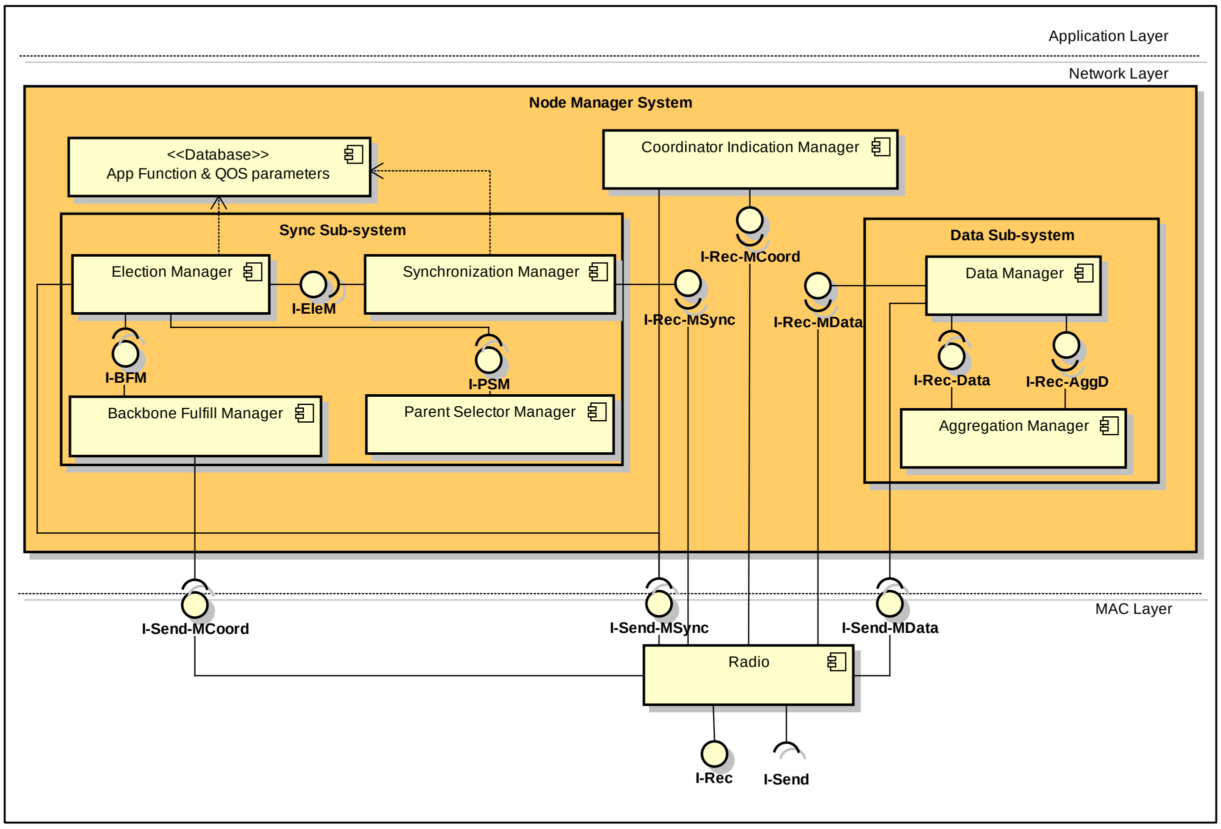

Figure 4. THESEUS Node Architecture

parameters (described in section 4.3.2). In addition, it is responsible for updating the app function parameters and QoS parameters, sending them to the network . The SM starts the construction of routes by broadcasting synchronization messages (Sync message) to the network. These messages carry the app function and QoS parameters (previously described in section 4.3.2). SM broadcasts Sync messages to the network periodically, and each new Sync message starts rebuilding the routing paths in the whole network (same approach used by PROC). In addition, whenever the arrival of a new application changes the app function and/or QoS parameters during a cycle, it is possible to update the network by disseminating a new Sync message.

THESEUS Node architecture, as shown in Figure 4, includes *Node Manager* (NM) system that contains *Sync sub-system*, *Data sub-system*, *Coordinator indication manager* component, and *App Function & QoS parameters database*. NM is responsible for initializing the sensor node and forwarding received messages by the sensor node to the related sub-system. There are three different types of messages: (i) Sync message, (ii) Data message and (iii) Coord message, which are delivered to *Synchronization sub-system*, *Data sub-system* and *Coordinator indication manager* respectively.

Sync sub-system using *Synchronization manager* component, *election manager, parent selector manager* and *backbone fulfill manager* is responsible for the process of coordinator selection and the route construction process. When the coordinator selection process finishes, the *election manager* component broadcasts a Sync message to inform other nodes about its new role.

Data sub-system contains *data manager* and *Aggregation manager* components, which perform the data packet aggregation algorithm, based on the data delivery models (event-driven or continuous dissemination) and related QoS parameters. The *Data manager* component sends the data to the next hop (parent node) when the aggregation process is done.

The *Coordinator indication manager* is responsible for forcing the node to be a coordinator when a Coord message is received. Then it broadcasts a Sync message to inform other nodes about the node new role.

The details of THESESU operation are described in the next section (4.3.4).

4.3.4 THESEUS operation

Figure 5 shows the algorithm running on the sink node representing the operation of *Sink Manager* (SM) system. Figure 6 shows the algorithm that represents the logic executed by the *Node Manager* (NM) system.

The operation sequence starts with the *route establishment process* after the deployment of the physical nodes in the target area and run the SM and NM systems on the sink node(s) and sensor nodes, respectively. This process starts by *SSN-Manager procedure* and calls (I-SSN-SM interface) the *Sink-Manager procedure* (Figure 5) to broadcast the Sync message to the network (I-Send-MSync interface), starting routes construction. The *Sink-Manager procedure* uses an incremental variable, named *nextCycle*, representing sink cycle id. This cycle id starts from one and increments each time the sink node broadcasts a new Sync message (Figure 5, line: 3 and 12). In the implementation time, the variable of cycle id need to be handled well for the time that it reaches to the maximum variable size (the variable starts to count from zero again). The Sync message updates node’s neighbor list with current cycle ID, the related sink cycle time (valid time), number of hops to the sink node, the node status (whether it is a coordinator or not), residual energy, app function and QoS parameters. Each sink node on the network can have a different value for its cycle time. The sensor nodes use the cycle time (valid time) to verify the selected sink node broadcasts the Sync message in next cycles, or the sensor nodes need to select another active sink node. Each sensor node stores the cycle id and valid time of received Sync message from each sink node into a table (sink nodes table).

Inside the network (sensor nodes), the *Node-Manager procedure* initializes the sensor node and waits to receive a message (Figure 6). When a Sync message is received, the *Node Manager* (NM) forwards it to the *synchronization manager* (I-Rec-MSync interface), which is a part of the *synchronization sub-system* (Figure 4). *Sync-Manager Procedure* in Figure 6 shows the running algorithm of *synchronization manager* (Figure 4) and how a node selects its sink node, and the *Coord messages* and new *Sync messages* are generated. The *synchronization manager* receives the Sync messages, and checks the cycle id of the message to prevent the node from repeating the further steps (controlling the Sync messages flooding).

If the cycle id of the sender sink node is the same as the related record on the sink nodes table of the node, the *synchronization manager* just completes the node neighbor list. Otherwise, if the cycle id is older than node's related record on the sink nodes table, it means the node has already done the route construction process (decided to be a coordinator and selected the node parent) and there is no more activity to be done. If the cycle id is newer than the node related record on the sink nodes table, it means a new cycle of route reconstruction has been started by the sender sink node. First, the node will update its sink nodes table by cycle id, valid time (cycle time), and app and QoS parameters. Then, the node checks the sink id to verify if it is the same sink id that nodes used in the last cycle, if so, then the node calls the *Election Manager* (I-EleM interface). If the sink id is not the same sink id the node already used, the node compares the number of hops distance towards the new sink with the old sink.

```
procedure SSN-Manager ( )
1  : Set (CycleTime,appparam, QoSparam);
2  : Sink-Manager (CycleTime,appparam, QoSparam);
end procedure
procedure Sink-Manager ( )
3  : nextCycle ← 1;
4  : loop
5  :     MSync.cycle = nextCycle;
6  :     MSync.hops = 0;
7  :     MSync.coord = true;
8  :     MSync.energy =∞;
9  :     MSync.appfunc = appparam;
10:     MSync.appparam = QoSparam;
11:     send(MSync, broadcast);
12:     nextCycle ← nextCycle + 1;
13:     wait CycleTime seconds or update of app function parameters or QoS parameters;
14: end loop
end procedure
```

Figure 5. THESEUS sink algorithm

```
procedure Node-Manager ( )
1  : parent ← null;
2  : SinkCycle← ∅;
3  : Neighbors ← ∅;
4  : coordinator ← false;
5  : Require: receive (MSync message);
6  :     Sync-Manager (MSync);
7  : Require: receive(MCoord message);
8  :     Coordinator-Indication-Manager (MCoord);
9  : Require: receive(MData message);
10:     Data-Manager (MData);
end procedure
procedure Sync-Manager ( )
11: if SinkCycle < MSync.cycle then  //New cycle
12:     Neighbors.sink ← ∅;
13:     Neighbors ← Neighbors ∪ MSync.{hops, cycle, coord, energy};
14:     Update SinckCycle (cycle, validtime);
15:     Update (appparam, QoSparam);
16:     if selectedsink = MSync.sink then //New cycle from the sink already used
17:           Election-Manager ( );
18:     else if MSync.hops < currenthops then //selcect another sink because of less distance
19:           selectedsink = MSync.sink;
20:           Election-Manager ( );
21:     else if selectedsink.validtime is expired then //select another sink because the old sink is not valid any more
22:           Neighbors.sink ← ∅;
23:           selectedsink = MSync.sink;
24:           Election-Manager ( );
25:     end if
26: else if SinkCycle = MSync.cycle then  //Same cycle
27:     Neighbors ← Neighbors ∪ MSync.{hops, cycle, coord, energy};
28: end if
end procedure
procedure Election-Manager ( )
29: prob ← AppFunc(app parameters);
30: coordinator ← (random ( ) < prob);
31: send(MSync(parent.hops+1,cycle,sinkid,coord,getEnergy(),app&QoS.param),BROADCAST);
32: wait Backoff Time ( );
33: parent ← Parent-Selector-Manager ( );
34: Backbone-Fulfill-Manager ( );
end procedure
procedure Parent-Selector-Manager ( )
35: if coordinator = true then
36:      return min(Neighbors, {hops, coord, 1/energy});
37: else
38:      return min(Neighbors, {coord, hops, 1/energy});
39: end if
end procedure
procedure Backbone-Fulfill-Manager ( )
40: if parent.coordinator = false then
41:     send(MCoord(parent.hops+1,cycle,sinkid,coord,getEnergy(),app&QoS.param), parent);
42:     Neighbors.parent .coord ← true;
43: end if
end procedure
procedure Coordinator-Indication-Manager ( );
44: Neighbors ← Neighbors ∪MSync.{hops, cycle, coord, energy};
45: coordinator ← true;
46: send(MSync(parent.hops+1,cycle,sinkid,coord,getEnergy(),app&QoS.param),BROADCAST);
end procedure
```

Figure 6. THESEUS node algorithm

The node will select the new sink and call (via I-EleM interface) the *Election Manager* for the new sink if the new sink has fewer hops distance. Moreover, the nodes select a new sink in case of expiration of old sink valid time (cycle time) and not receive the new cycle message during the appropriate time (valid time). This process makes THESEUS support adding and removing sink nodes dynamically.

The *Election Manager* calls the app function with relevant local and QoS parameters in order to calculate the probability of the node being a coordinator. *Election-Manager procedure* in Figure 6 shows the related algorithm. A simple random function (that returns a random floating number between 0 and 1), along with the value of the app function (a value defined by the app function which states a probability between 0 and 1) decide if the node should be a coordinator or not. For example, for a 61% chance calculated by app function, the node will be a coordinator if the random number is less than 0.61. Next, a *Sync message* is broadcasted (I-Send-MSync interface) to inform the network about the updated node status. This rebroadcasting of Sync messages by each sensor node guarantees that all nodes were reached by the first Sync message broadcasted by the sink node, therefore ensuring the network connectivity (same approach as PROC).

After sending a *Sync message,* sensor nodes wait for a random time to receive information about the states of other nodes. This random time, named *Backofftime,* avoids the whole network from selecting more coordinators than necessary. The *SSN manager* should define the duration of this time based on QoS requirements and the network conditions (such as network size) before starting the network. After the backofftime has elapsed (I-PSM interface), the *parent selector manager* selects the parent within the updated neighbor list (the *Parent-Selector-Manager procedure* in Figure 6 show the related algorithm). At this point, *Backbone Fulfill Manager* checks (I-BFM interface) the status of the selected parent. As mentioned, all nodes should connect to a coordinator. Therefore, if the selected parent is not a coordinator, a Coord message will be sent to the selected parent (I-Send-MCoord interface), and the node updates the parent status to the coordinator in its neighbor list. As shown in Figure 6, *Backbone-Fulfill-Manager procedur*e runs in the *Backbone Fulfill Manager* component and they represent the sending of Coord messages.

The *parent selector manager* (*Parent-Selector-Manager procedure* in Figure 6) selects, within the neighbor list, the best node to be the parent of a node based on the following criteria, in descending order: (i) holding a coordinator role, (ii) shortest hop distance to the sink, and (iii) greater available energy. Like PROC, this selection process and sink selection process avoid looping by considering the shortest hop distance towards the sink. Therefore, THESEUS does not need to use any loop discovery algorithms. In addition, based on checking the valid time for the sink nodes in the route selection process of nodes, THESEUS supports adding and removing of more sink nodes dynamically.

The *Coordinator-Indication-Manager procedure* of Figure 6 runs in the *Coordinator Indication Manager* and configure the node to be a coordinator (I-Rec-MCoord interface) and broadcast a Sync message (I-Send-MSync interface) to inform other nodes about the new status of the node.

The *data dissemination process* starts when the sensor nodes begin collecting data from the environment and sending the sensed data to their selected coordinators using Data messages. Within each sensor node, the NM will forward the received Data messages to the *data manager* component (I-Rec-MData interface), in the Data Sub-system. In the Data Sub-system of the coordinator node, specifically within the *Aggregation Manager* component, our proposed packet aggregation algorithm (whose operation was described in section 4.3.4) takes place. After the aggregated packet is ready, it is forwarded (I-Send-MData interface) through the tree-like structure until reaching the sink node.

Sync messages are broadcasted periodically and/or eventually for each cycle, starting routes reconstruction. Such periodic reconstruction of routes makes THESEUS fault tolerant in case of node failures and helps balancing the energy usage within all nodes. Furthermore, if any new application demand comes up in the network, which changes the app function parameters or QoS parameters, then the nodes will be updated immediately. It is also important to mention that the network connectivity is guaranteed in every cycle, as well as in the first cycle, by rebroadcasting Sync messages by each node.

5. IMPLEMENTATION

THESEUS and PROC routing protocols were implemented on TinyOS 2.1.0 (TINYOS, Accessed May 2015), using the nesC programming language (an extension of the C language), which adopts an event-driven programming model (TINYOS, Accessed May 2015). TinyOS is a component-based operating system, designed specifically for WSN application development. TinyOS provides a number of interfaces to abstract the underlying communication services and a number of components that implement these interfaces (TINYOS, Accessed May 2015). We used MICAz platform (MICAZ & MICA2, Accessed May 2015) (4kB RAM, 128 kB flash memory for program storage) for our implementation.

The implemented codes followed THESEUS architecture and algorithms previously described (sections 4.3.3 and 4.3.4). First, we implemented THESEUS, and then we implemented a simple version of PROC for comparison purposes by excluding the packet aggregation function, the fourth formula of the app function, and supporting multiple sink nodes.

As mentioned, TinyOS programming is based on components, events, and tasks. Some of the procedures of THESEUS are based on more than one event and/or task. Therefore, we were forced to divide such procedures to separate tasks and/or events. In the following, first, we describe the components of TinyOS that we used in our implementation. Then, we described the detail of how we implement our algorithms on TinyOS.

TinyOS provides the MainC component, which is responsible to boot the node, and call the ActiveMessageC component and SplitControl interface afterwards in order to initialize the node and start the application. Since it is very common to have multiple services using the same radio to communicate, TinyOS provides the Active Message (AM) layer (ActiveMessageC component) to multiplex access to the radio. A number of components implement the basic communications and active message interfaces in TinyOS (TINYOS, Accessed May 2015). In our implementation, we used the Packet, AMSenderC, and AMReceiverC TinyOS components, which are responsible to create messages, send messages, and receive messages respectively. Sending a message by AMSenderC creates an event of sendDone in TinyOS, which is used for continuing the process of an algorithm that should run after a message was sent (TINYOS, Accessed May 2015). In this version of our work, the Packet Acknowledgements (Ack) interface of TinyOS 2.1.0 was used to manage simple message confirmation, and assist the retransmission in case of losses of unicast messages (Coord and Data messages). (TINYOS, Accessed May 2015). RandomC and VoltageC are two other TinyOS components, which are used to create random numbers and read the battery voltage respectively (TINYOS, Accessed May 2015). Finally, the last component of TinyOS used in our implementation is TimerC. When a timer component in TinyOS is called, it raises a "timer-fired" event after a given time (provided as a parameter to this call, in milliseconds) (TINYOS, Accessed May 2015).

We implemented two applications by following the described architecture and algorithms (sections 4.3.3 and 4.3.4): (i) THESEUS sink manager (Sink Manager system of THESEUS sink architecture, Figure 3), and (ii) THESEUS node manager (Node Manager system of THESEUS node architecture, Figure 4). Figure 7 shows the components and interfaces of THESEUS sink manager (TSM) application and Figure 8 shows the components and interfaces of THESEUS node manager (TNM) application, which these figures were created by TinyOS documentation (TINYOS, Accessed May 2015). The AM_MData_R component in TSM is responsible to receive data messages. The detail of this component is out of the scope of our work. We created it with the sole purpose of testing our routing protocol in case of data aggregation and delivering data to the sink node.

TSM application was implemented by one task and two events based on the *Sink-Manager procedure* (Figure 5), representing the Sink Manager component of THESEUS sink manager architecture (Figure 3): MSync_broad_task task, MilliTimer.fired and MSync_S.sendDone events. In our implementation, we assumed that parameters of the application and QoS (such as cycle time and maximum accepted delay for aggregation) are already defined as variables in initializing time of the sink node. After the sink node boots and is initialized by MainC and ActiveMessageC components, it calls the TimerMiliC components by cycle time value and periodic mode ("call MilliTimer.startPeriodic(CycleTime)"). As we described, calling timer makes an event of the timer.fired (MilliTimer.fired). MilliTimer.fired event calls MSync_broad_task task ("post MSync_broad_task();"). MSync_broad_task

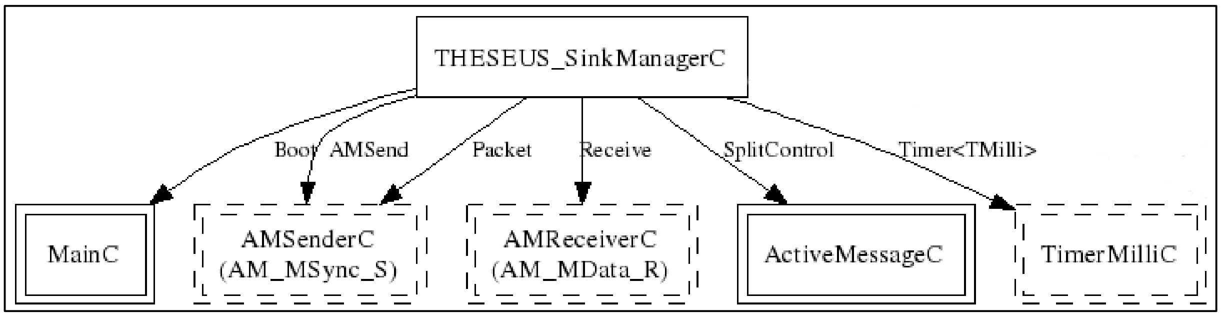

Figure 7. Components and interfaces of THESEUS sink manager application (TinyOS)

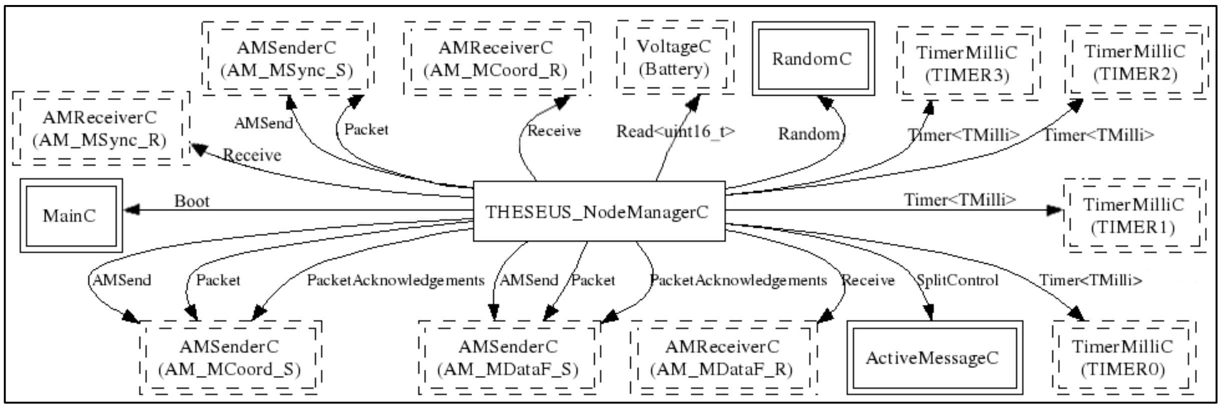

Figure 8. Components and interfaces of THESEUS node manager application (TinyOS)

broadcasts a Sync message to the network that contains the defined variables (application and QoS parameter), using AM_MSync_S component (instantiate of AMSenderC, representing the I-Send-MSync interface of THESEUS sink architecture, Figure 3). After the message is sent, the process goes to MSync_S.sendDone event, which increments the cycle id (“nextCycle++;”).

TNM application was implemented in a similar way of TSM application. Some procedures are implemented by dividing them into tasks and events. TNM application was implemented by 4 functions, 3 tasks and 11 events based on the *Node-Manager, Sync-Manager, Election-Manager, Parent-Selector-Manager, Backbone-Fulfill-Manager, Coordinator-Indication-Manager, Data-Manager, THESEUS-packet-aggregation procedures,* and the app function (Figure 6, Figure 2 and equation (6)), representing the Node Manager component and all its components of THESEUS node manager architecture (Figure 4). After the sensor node boots and is initialized by MainC and ActiveMessageC components, it waits to receive a message: Sync, Coord, or data messages. Each message type makes an event and brings the process to the related event (*Node-Manager procedure).*

Sync messages are received by AM_MSync_R (instantiate of AMReceiverC, representing the I-Rec-MSync interface of THESEUS node architecture, Figure 4), which creates the MSync_R.receive event. This event implemented the *Sync-Manager procedure* (Synchronization Manager component of THESEUS node architecture) and it calls the Election_Manager function based on the algorithm whenever it is needed.

Election-Manager procedure (Election Manager component of THESEUS node architecture) was implemented in three parts. First part was implemented by a function named Election_Manager, which calls the App_Function. The app function was implemented by App_Function function. Election_Manager calls the RandomC component to get a random number for its process (described in section 4.3.4). Afterwards, it calls the MSync_broad_task (a similar task of sink application). The second part was implemented in the MSync_S.sendDone event (a similar event of sink application) based on the sendDone event, which uses RandomC and Timer0 (instantiate of TimerC) in order to implement the wait time. The third part was implemented in the Time0.fired event based on the timer.fired event, which calls the Parent_Selector_Manager, Backbone_Fulfill_Manager functions, and Timer1 (instantiate of TimerC) to implement the sink valid-time timer (we considered 10 seconds additional time besides the sink valid-time to be sure the new cycle message is not delayed through the network). *Parent-Selector-Manager procedure,* the Parent Selector Manager component of THESEUS node architecture, was implemented by a function named Parent_Selector_Manager. *Backbone-Fulfill-Manager procedure,* the Backbone Fulfill Manager component of THESEUS node architecture, was implemented by a function named Backbone_Fulfill_Manager. Based on the algorithm Backbone_Fulfill_Manager calls the task of MCoord_uni_task. This task unicasts a Coord message to the parent node, using AM_MCoord_S component (instantiate of AMSenderC, representing the I-Send-MCoord interface of THESEUS node architecture, Figure 4).

If the event of Timer1.fired happens, it means the selected sink node is not valid anymore. We implemented the algorithm of lines 22 to 24 of the *Sync-Manager procedure* (Figure 6) in this event to node change its selected sink node to another sink node.

Coord messages are received by AM_MCoord_R (instantiate of AMReceiverC, representing the I-Rec-MCoord interface of THESEUS node architecture, Figure 4), which creates the MCoord_R.receive event. The *Coordinator-Indication-Manager procedure* (Coordinator Indication Manager component of THESEUS node architecture) was implemented in this event. It calls the task of MSync_broad_task based on the algorithm whenever it is needed.

The default data field size of TinyOS packet is 28 bytes (TINYOS, Accessed May 2015). Therefore, we divide these 28 bytes to 14 two bytes parts. The first used for the flag; second for the source node id; third for the destination id (the sink node id); and fourth for the sample value. The next ten two bytes were considered for five groups of node id and its sample value for the aggregation purpose (more detail in section 4.3.1).

Data messages are received by AM_MDataF_R (instantiate of AMReceiverC, representing the I-Rec-MData interface of THESEUS node architecture, Figure 4), which creates the MDataF_R.receive event. The *Data-Manager* and *THESEUS-packet-aggregation procedures* (Data Manager and Aggregation Manager components of THESEUS node architecture) were implemented by MDataF_R.receive, Timer2.fired, and Timer3.fired events. When the aggregation is terminated because of the timer or reaching the maximum packet data field size, those events calls the task of forward MDataToParentTask. Timer2 and Timer3 (instantiate of TimerC) are used to control the aggregation time (MDTC and MDTE) regarding the data delivery models (continuous and event-driven). The task of forwardMDataToParentTask sends the aggregated data by AM_MDataF_S component (instantiate of AMSenderC, representing the I-Send-MData interface of THESEUS node architecture, Figure 4).

It is worth mentioning that sixteen real MICAz sensor nodes were used to verify and debug the implementation functionality of THESEUS in the Ubiquitous Computing Laboratory of PPGI-UFRJ, including the following tasks: route establishment, continuous data type delivery, and event-driven data type delivery.

The implemented codes are available at Appendix B and online: “https://github.com/mrezaim/THESEUS”.

6. EVALUATION

In this Chapter, we present the experiments performed to evaluate THESEUS. Our evaluation is divided into four parts. The first evaluation is a comparison between THESEUS and PROC. Since THESEUS routing algorithm is inspired by PROC, we compared the results of their functionalities in the same scenarios with the goal of analyzing the improvements and overheads of using THESEUS in SSNs. To the best of our knowledge, no other practical related works were found in the literature of routing solutions specifically for SSNs. Therefore, it was not possible to perform such a comparison with other routing protocol for SSNs. The second evaluation consists of an analysis on the impact of the variation of important parameters on THESEUS behavior. The third evaluation is an analysis of the impact of using more than one sink nodes on THESEUS performance. Finally, we present a comparison between Real and Simulated Nodes running THESEUS to verify the validation of the simulated results.

6.1 COMPARING THESEUS AND PROC

The main goal of the first set of performed experiments is to prove that THESEUS is suitable for the SSN scenario, achieving satisfactory values of network lifetime in such scenario. In this part of the evaluation, we compare PROC and THESEUS in order to highlight the achievement of improving the network lifetime by THESEUS. We also performed a thorough analysis of pros and cons of using THESEUS packet aggregation algorithm. Macedo et al. (MACEDO *et al.*, 2006) have already proved several essential characteristics of PROC such as fault tolerance in case of “Transient and isolated failures”, “Permanent and isolated failures” and “Permanent and grouped failures” (MACEDO *et al.*, 2006). Since THESEUS uses the same process for routing establishment as PROC, experiments to assess such features will not be performed with THESEUS: we assume that such characteristics are inherited by THESEUS.

6.1.1 GQM

Considering the objectives of the first part of the experiments and following the Goal, Question, and Metric (GQM) methodology proposed by Basili et al. (BASILI *et al.*, 1994), the following goals were defined:

(i) Analysis of the improvements and overheads of using THESEUS instead of PROC in SSNs regarding the SSN lifetime in the context of routing protocols, with the purpose of comparing with PROC.

(ii) Analysis of the pros and cons of using THESEUS packet aggregation algorithm, with the purpose of comparing with PROC.

These goals can be expressed by five questions. Q1, Q2, and Q3 are related to the first goal, Q4 and Q5 are related to the second goal:

Q1: How much THESEUS improves the energy usage of the nodes, in comparison with PROC?

Q2: How much THESEUS improves the energy usage balance through nodes, in comparison with PROC?

Q3: How much overhead, in terms of memory, processing, time spent to build the routes, and number of packets sent in the phase of routes construction, THESEUS imposes to the SSN in relation to PROC?

Q4: What are the advantage(s) of using the packets aggregation technique, compared with PROC, which does not use it?

Q5: What are the drawback(s) of using the packets aggregation technique, compared with PROC, which does not use it?

Metrics were defined to support the answers to these questions. In the following, Mij denotes the metric, where i corresponds to the question identifier and j is a counter used whenever there is more than one metric per question. Moreover, all the following metrics are calculated for each simulated scenario for both THESEUS and PROC, to allow comparing the retrieved values of the metrics in order to analyze the improvements or worsening of using THESEUS instead of PROC on SSNs, regarding each metric.

Regarding Q1, we defined the **Energy Average**, **M11** as the average of the energy consumption values of all nodes during each simulated scenario. Therefore, smaller values for this metric mean that more energy is saved.

In relation to Q2, we defined the **Energy Population Standard Deviation, M21.** This statistical metric shows the deviation (from average) of the energy consumption of all nodes in a network. In this case, all the sensor nodes of the SSN form the statistical population and **M21** is the standard deviation of such population. Therefore, the routing protocol that returns a value (for M21) more close to zero achieves better energy usage balance among sensor nodes.

In relation to Q3, we defined four metrics: **M31**: **Memory usage**, **M32**: **CPU Active**, **M33**: **Number of Packets used in Route Construction**, and **M34**: **Route Construction Time**. **M31** is defined as the number of bytes of RAM used by the routing protocol. **M32** is defined as the average of the percentage of time that the CPU of a node remains active. **M33** is defined as the number of packets (Sync and Coord messages) exchanged among nodes that are required to construct the routes (in average among all cycles). **M34** is defined as the duration of routes construction (in average among all cycles, and measured in milliseconds). AVRORA simulator output log file provides the timestamps of each packet, and such values were used for keeping track of the first and last packet used for constructing routes. Smaller values for **M31, M32, M33,** and **M34** mean improvements regarding these metrics. Therefore, we will compare the values of these metrics on each simulated scenario by THESEUS and PROC to find the appropriate answer to Q3.

In relation to Q4, we defined **M41: Network traffic rate per samples**, **M42: Aggregation Rate**, and **M43**: **Packet Loss** metrics. **M41** is defined as the average of packets sent to deliver samples to the sink node inside the network, considering all repeated packets (because of not receiving acknowledge message). It is calculated by dividing the number of sent data packets per number of samples generated during the simulations. A smaller value for this metric shows less network traffic, which means less energy usage on the network and less probability of packet collision. **M42** is defined as the average of samples aggregated in a packet. It is calculated by dividing the number of samples per number of packets delivered in the sink node during the simulation. A higher value for this metric means more samples are aggregated, which means less number of packets has been transmitted on the network, resulting in improvement of network lifetime. **M43** is defined as the percentage of packets lost (any type of packets) in the network during the simulation. A value closer to zero shows less packet loss on the network.

Finally, in relation to Q5, we defined **M51: Sample Loss** and **M52: Delay Time** metrics. **M51** is the percentage of lost samples, calculated by equation (7):

$$Sample\ loss = \left(\frac{B-A}{B}\right) * 100 \qquad (7)$$

where A is the number of received samples in the sink node and B is the number of samples generated by all nodes. Using packet aggregation, the number of network transmissions is reduced and consequently the chance of losing packet gets lower. On the other hand, each packet could contain more than one sample and could produce a higher value of sample loss. Therefore, analyzing sample loss will give a better answer to the fifth question instead of analyzing the packet loss. The last metric, **M52** is defined as the average time each sample took from generating until delivering at the sink node. Again, similar to **M34**, we used the AVRORA simulator output log file for calculating this metric. A smaller value of **M51** and **M52** show more improvement regarding these metrics.

6.1.2 Evaluation methodology and scenarios

The experiments for comparing THESEUS and PROC were performed using AVRORA 1.7.117 simulator (AVRORA SIMULATOR, Accessed May 2015), in similar conditions. For performing its simulation process, AVRORA uses the same implementation code produced to be deployed on real WSN platforms. We used the AVRORA default radio model (AEON). This radio model is basically a distance-attenuation model where radio signal strength drops off with the square of the distance (AVRORA SIMULATOR, Accessed May 2015).

In this set of experiments, we fixed the following parameter values (in the next set of tests we analyzed the behaviour of THESEUS by varying such parameter values):

(i) **Cycle time:** in Macedo et al. (MACEDO *et al.*, 2006), the authors mentioned that they used **180** seconds for cycle time based on empirical observations of PROC behavior, so the same value was used in our evaluation.

(ii) **Maximum neighbor list size:** considering the limited memory of sensor nodes, this list should be restricted, and we used the same value as PROC, which was **20.**

(iii) **Minimum and maximum values of Backofftime:** if these parameters are set to form a short period, then the nodes cannot receive enough information from their neighbors, while a long period results in a longer required time for routes formation. Based on the results obtained by variation tests, we selected **120ms** and **512ms** for these parameters in our evaluation (smaller values, as well as a shorter period could be used in a denser network).

(iv) **Packet Acknowledgments:** similar to PROC, we used acknowledgments for unicast packets. We also considered one more try for sending the packet, in case of a packet loss (packet not acknowledged by the receiver).

In our evaluation, we varied the number of nodes in **50, 100, 150 and 200 nodes**, and in all of these scenarios, the first node (in the bottom left corner) was configured as the sink node (the observation that we assessed the presence of more than one sink node in section6.3). In each topology, we placed the nodes on a **square grid** (Figure 9) with a distance of **5 meters between nodes** in each horizontal and vertical direction in the Cartesian plane. In addition, to bring more neighbors closer to the sink node, the sink node was placed slightly displaced from the central point of the grid formed by the three nodes in the bottom left corner of the topology (**inside the grid formed by these three nodes**) as seen in Figure 9.

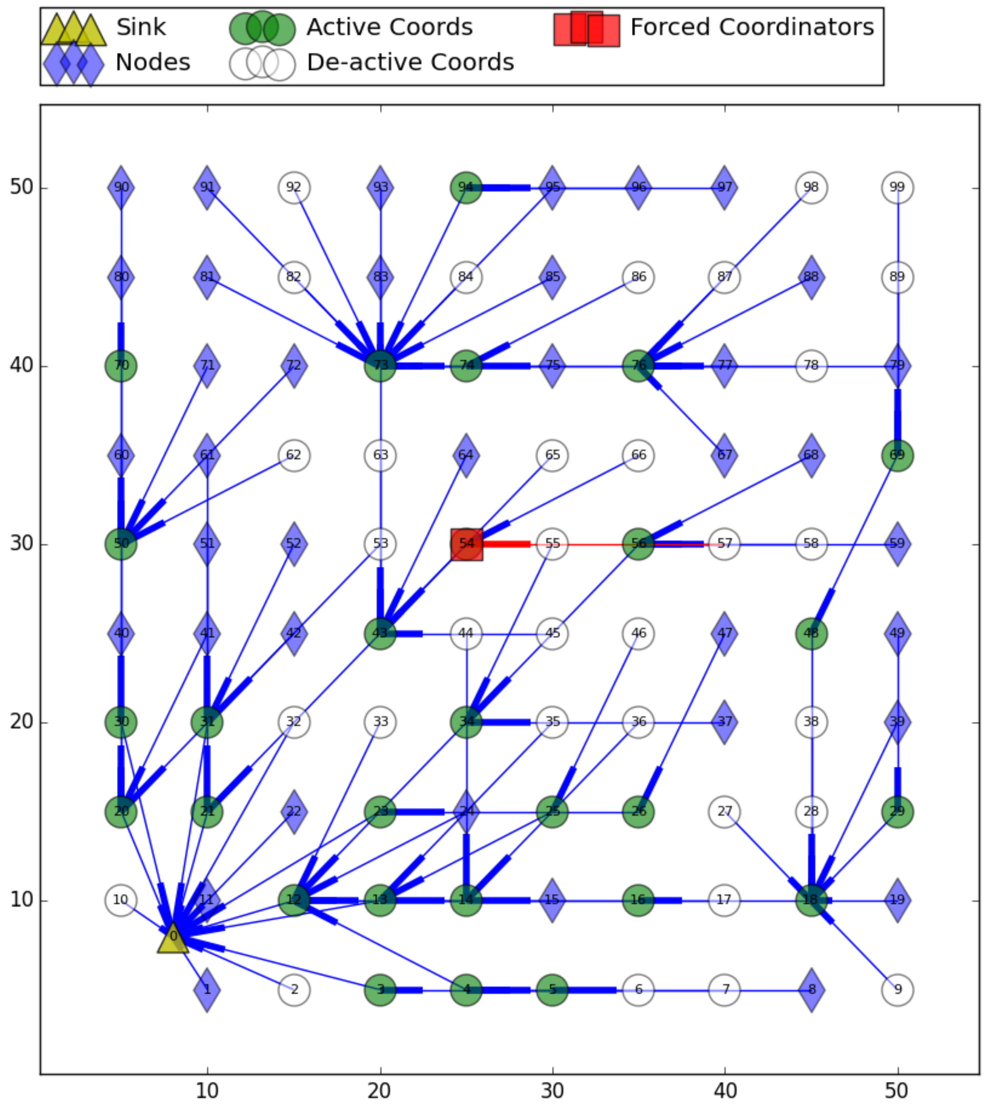

Figure 9. Graph of data flows during a cycle of THESEUS simulation (100 nodes)

To simulate the conditions of a SSN, we created two areas within each topology. PROC only supports continuous data delivery model. However, the event-driven data delivery model does not have a considerable effect on energy usage comparing to the effect of the continuous data delivery model. This is due to the fact that each sample of event-driven model refers to an unusual (rare) demand of sampling by applications (i.e. on-demand sampling) (AKKAYA *et al.*, 2005). Therefore, to perform the tests in more demanding conditions regarding data transmissions, we decided to use two applications based on continuous data delivery model. One application (monitoring humidity) runs in both areas with data sample rate time of 28 seconds, and another application (monitoring temperature) runs in just one of the areas with same data sample rate time of 28 seconds, but starts monitoring with 14 seconds of delay after the first application. We decided to use 28 seconds interval in order to avoid matching the same value of the cycle time (180 seconds). Avoiding this matching is important to balance the number of packet collisions in the network. In other words, **the nodes in the first half of the network (area 1) generate samples, each 14 seconds, and the nodes in the second half of the network (area 2) generate samples, each 28 seconds**. For instance, in a topology of 100 nodes, the nodes with ID from 1 to 49 generate samples at each 14 seconds (once humidity and once temperature), and nodes with ID from 50 to 99 generate samples at each 28 seconds (Figure 9 shows node positions).

Maximum accepted delay time for continuous dissemination samples (MDTC) should not be bigger than minimum data sample rate and should be long enough for the nodes to be able to aggregate packets. In this part of evaluation, we considered **MDTC as 10 seconds**.

6.1.3 Tests and analysis of results

In our evaluations, we selected an energy model to provide more generic comparison results for energy related metrics. We chose the energy model described in (LI *et al.*, 2014) since it is a well-defined and generic energy model for wireless sensor nodes. As suggested by (LI *et al.*, 2014) we calculated energy consumption for each node based on sent packets. Our simulation log files provided the necessary information on transmitted packets, regarding the size, sender and receiver node IDs, and distance between sender and receiver (calculated by using the related topology). The energy consumption of transmitting l-bit data over distance d is defined as Etx(l, d):

$$\mathrm{Etx}(l, d) = \mathrm{Eelec} \times l + \varepsilon\mathrm{amp} \times l \times d^2 \qquad (8)$$

where Eelec and εamp are hardware-related parameters (MICAZ & MICA2, Accessed May 2015).

In this evaluation, we made eight experiments (with 50, 100, 150 and 200 nodes for both PROC and THESEUS) during five hours. In following, we analyze the results of each metric and its impact in each scenario. The five hours of simulation comprise the execution of 100 cycles, which were all considered in the calculation of the presented metrics (in average among all cycles performed).

Regarding **Energy Average (M11)**, THESEUS shows better results in comparison to PROC in all topologies. Regarding **Q1**, M11 for THESEUS was 10.70% lower than M11 for PROC, in the 50 nodes topology. This advantage of THESEUS increases as the amount of nodes in each topology rises, reaching 20.83% for the 200 nodes topology. Therefore, THESEUS is more advantageous than PROC for larger networks (Figure 10). This result is explained because, in relation to PROC, THESEUS has an aggregation algorithm, and so, it transmits aggregated data in packets, requiring the transmission of a smaller number of packets than PROC. For instance, a fully data aggregated packet in THESEUS is 20 bytes larger than a PROC data packet. However, PROC would require more than one packet transmission for delivering this same amount of data. So, PROC needs to activate its radio device more times than THESEUS, spending more energy. Also, PROC spends more energy with header/trailer bytes transmissions than THESEUS. Since radio transmissions have a significant impact on energy usage in the radio component (LI *et al.*, 2014), THESEUS shows a considerable

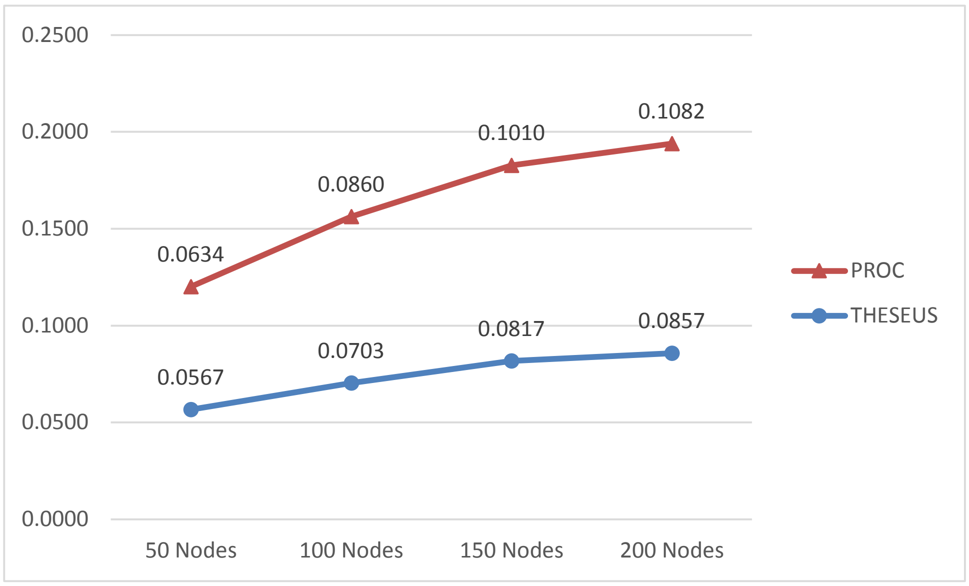

Figure 10. Results of the Energy Average metric

improvement in relation to PROC, even with the larger size of packets, because THESEUS generates a smaller number of packets.

Regarding the **Energy Population Standard Deviation (M21)**, THESEUS shows better results in comparison to PROC in all topologies (Figure 11). Regarding **Q2**, M21 for THESEUS was 32.33% lower than M21 for PROC, in the 50 nodes topology, and respectively 40.76%, 28.74% and 35.18% lower for 100, 150 and 200 nodes topologies. In average among all topologies, THESEUS showed M21 values 34.25% lower than PROC. As the number of nodes increase in the topology, the M21 values increase at a lower rate for THESEUS than for PROC. These facts prove that THESEUS is more capable of balancing the energy usage in the network than PROC. This is because THESEUS considers the specificities of a SSN environment (i.e. multiple applications running on nodes) for selecting coordinators, thus resulting in better choices of coordinators than PROC.

Memory usage (M31): This value was retrieved from TinyOS at compile time for real nodes (it is the same for all simulations). Regarding this metric, THESEUS used 757 bytes of memory, which corresponds to 18.48% of node memory in MICAz platform (4096 bytes). In relation to PROC, THESEUS used 195 bytes more of the node memory, representing less free memory as an overhead (worsening 4.76%). However, the remaining free memory (3339 bytes) is adequate to be used by any other required protocol, application code, etc.

CPU Active (M32): AVRORA provides the values of the time the CPU of the nodes remain active during the performed simulations (AVRORA SIMULATOR, Accessed May

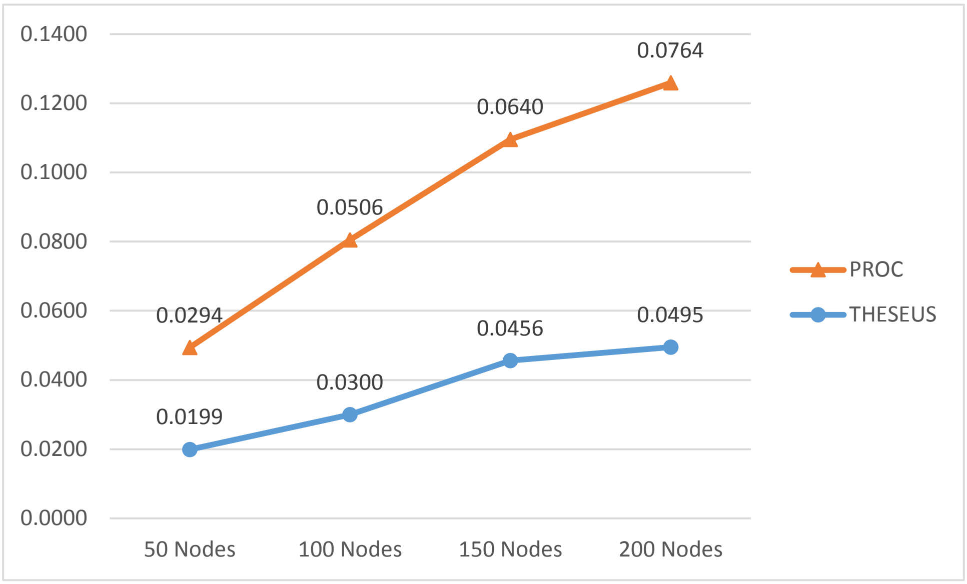

Figure 11. Results of the Energy PSTD metric

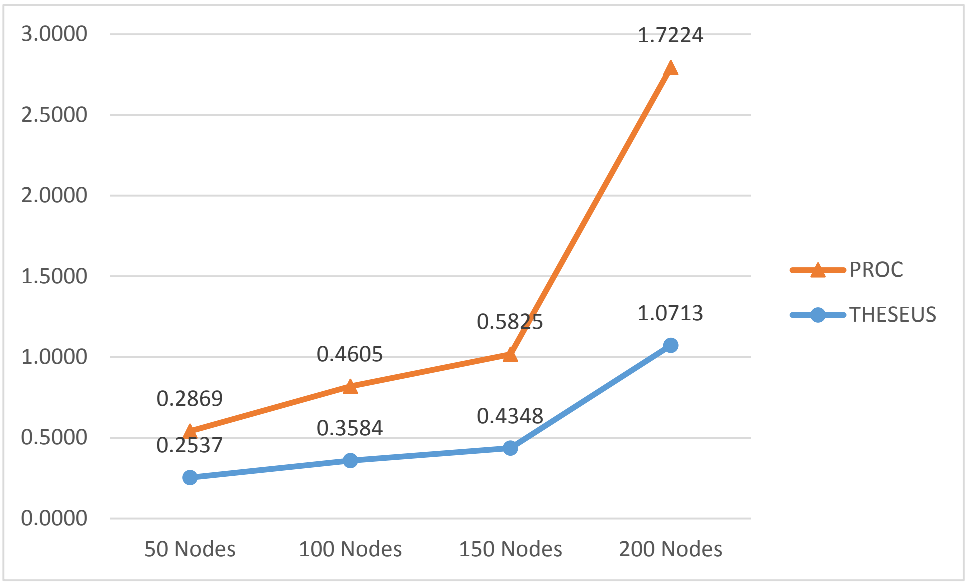

Figure 12. Results of the CPU Active metric

2015)**.** An overhead was expected regarding this metric, based on the higher complexity of THESEUS, mainly driven by its aggregation algorithm. However, the results show that the impact of sending more packets (as in PROC) on the CPU usage is greater than the effect caused by THESEUS packet aggregation algorithm. In an average of all tests with THESEUS, nodes CPU were active 0.53% of the time instead of 0.76% for the average of all tests with PROC. In one of the scenarios, THESEUS improved this metric by 37.80%. In average of tests, the improvement is 24.22% in relation to PROC (Figure 12). So, THESEUS proved to be better than PROC regarding this metric.

Number of Packets used in Route Construction (M33) and Route Construction Time (M34): as expected, M33 and M34 showed almost the same values for THESEUS and PROC in each simulation since the route construction processes of both are similar (Figure 13 and Figure 14 respectively for M33 and M34). Furthermore, we tested THESEUS for 500 and 1000 nodes, regarding these metrics, and the results were 617 and 1350 packets respectively for M33, 740 and 868 milliseconds respectively for M34. These results are important because they show that THESEUS (as well as PROC) route construction procedure is scalable regarding the number of nodes. Regarding M33, THESEUS spends only 54 packets for constructing routes in a 50 nodes topology. This value increases to 235 packets in topologies with 200 nodes. Therefore, it is possible to assume that THESEUS is scalable to the number of nodes, because as the network size increases, a proportional (linear) number of packets is required for constructing routes. This number of packets was measured for the highest possible amount of

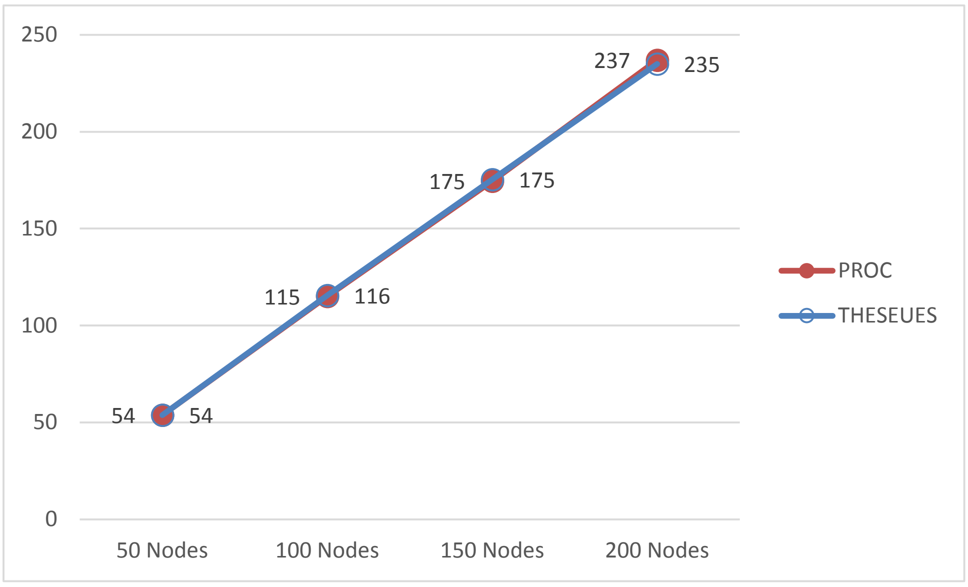

Figure 13. Results of the Number of Packets used in Route Construction metric

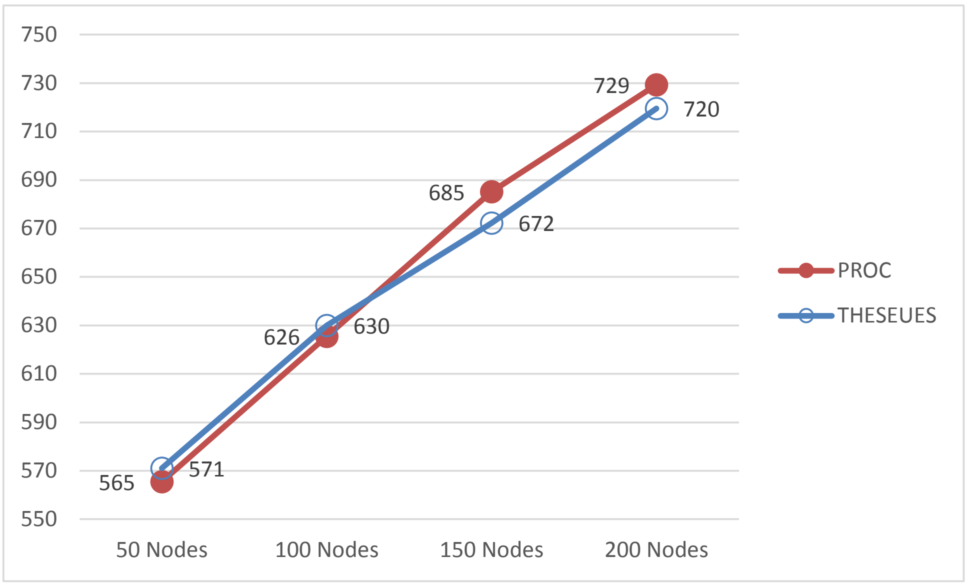

Figure 14. Results of the Route Construction Time metric

sensor nodes in simulations (1000 nodes), and THESEUS still worked well, following this linear relation. And finally, M34 also helps supporting these conclusions.

Therefore, regarding **Q3**, the overhead of using THESEUS instead of PROC, in terms of memory, is 4.76%, but for processing THESEUS improved more than 24% in average. Regarding the duration and the number of packets sent on routes construction phase, the results showed that THESEUS and PROC route construction procedure is scalable.

Network traffic Rate per samples (M41): This parameter showed an improvement between 19.57% and 32.33%, getting better as the network size increases (Figure 15). Because

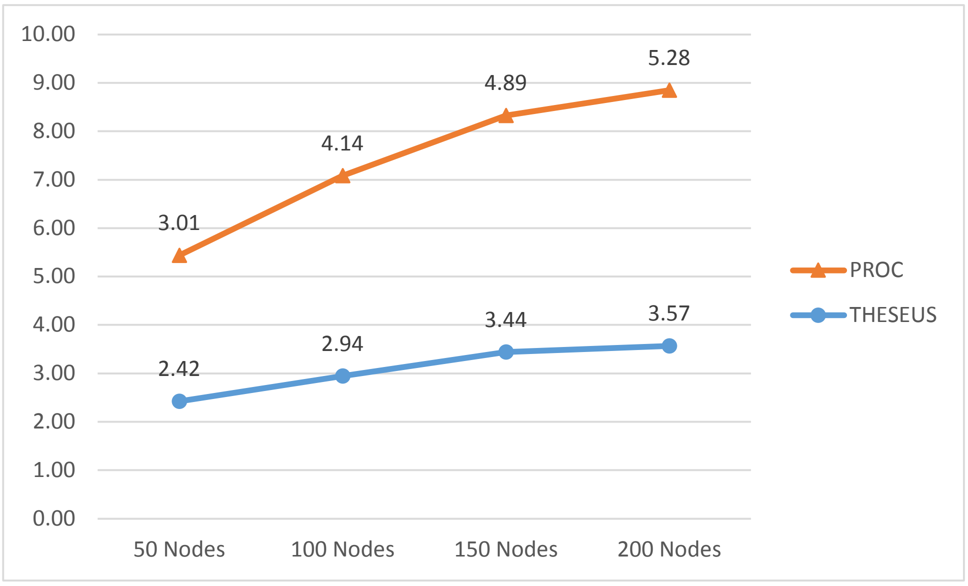

Figure 15. Results of the Net. Rate metric

of less traffic in the network, THESEUS is less prone to packet loss and energy usage than PROC. The reduced traffic rate in THESEUS, in relation to PROC, is explained by the packet aggregation algorithm used in THESEUS. THESEUS packet aggregation imposes a smaller number of packet, as well as less byte transmissions by each sensor node than PROC.

Aggregation Rate (M42): This parameter showed 2.28 samples were aggregated in each data packet in the average of all five scenarios during five hours of simulation time (Figure 17). In this implementation of THESEUS, each data packet could carry up to six samples (Figure 17). Moreover, the results showed more number of nodes in a network could improve the aggregation chance of more number of samples in a data packet during the same limited time (MDTC). In other words, in a 50 nodes topology there are less aggregated packets with 6 samples, but in a 200 nodes topology, the percentage of aggregated packets with 6 samples increases. This can be perceived in Figure 16, since the orange bars in the outer circles are increasingly bigger than in the inner circles. On the other hand, the percentage of aggregated packets with one sample is reduced in the outer circles (blue bars in Figure 17), so the proportion of large aggregated packets (4-6 samples per packet) increases and the proportion of small aggregated packets (1-3 samples per packet) decreases, as the network size increases. It means THESEUS could work more efficient in a bigger network.

Packet Loss (M43): AVRORA provides the packet loss report in the performed simulations (AVRORA SIMULATOR, Accessed May 2015). This packet loss includes all types of packets sent over the network (Sync, Coord, and Data messages). This parameter showed that THESEUS improved packet loss 11.51% in average of all tests in comparison with

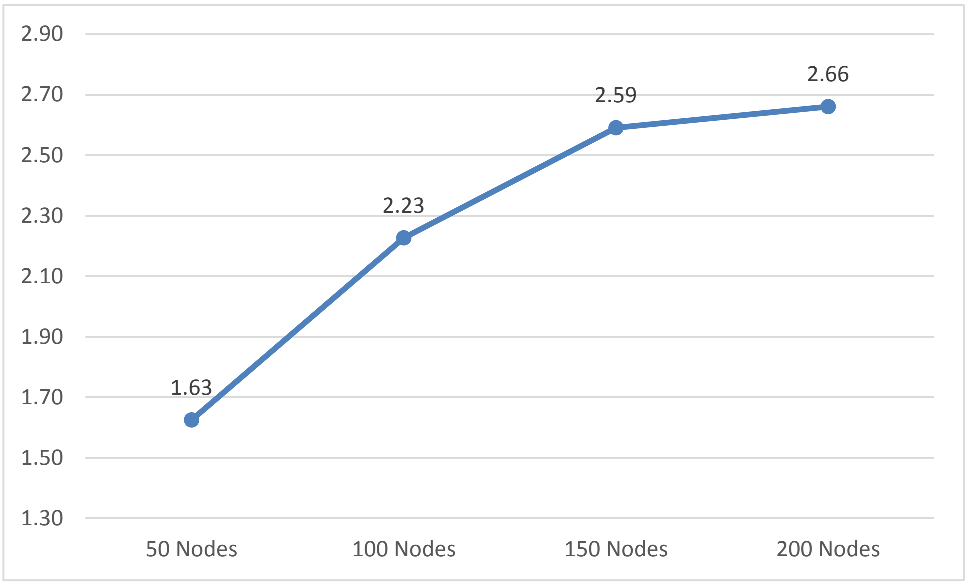

Figure 16. Results of the Aggregation Rate metric

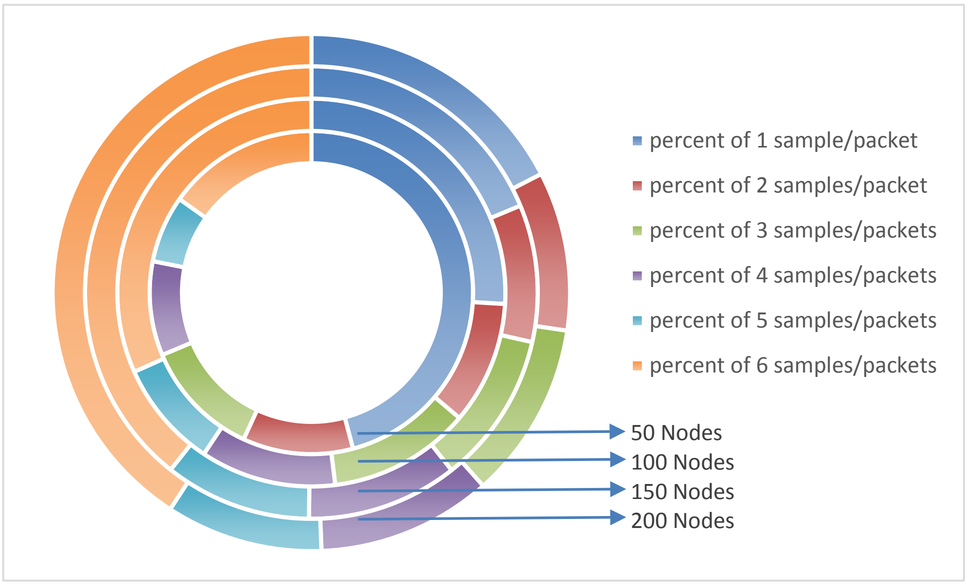

Figure 17. Aggregation detail

PROC (Figure 18). Furthermore, larger networks showed more improvement. Such improvement happens because of THESEUS reducing the number of packets, and consequently, reducing the chance of packet collisions.

Regarding **Q4**, about advantage(s) of using the packet aggregation technique, THESEUS reduced network traffic in 27.59% and packet loss 11.51% in average of all tests, compared with PROC. Moreover, THESEUS showed it works more efficient than PROC when the number of nodes increases. This is a promising and expected result since THESEUS is tailored for SSNs, which are potentially large-scale networks.

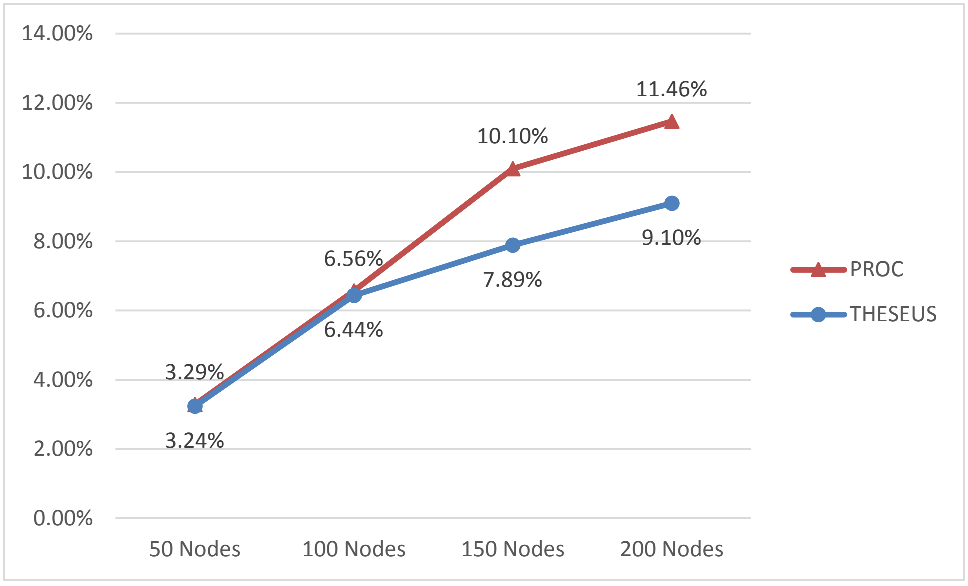

Figure 18. Results of the Packet Loss metric

Sample Loss (M51): In the average of all simulations, THESEUS improved sample loss by 51.29% in comparison to PROC. This happens because THESEUS aggregation algorithm helps reducing the amount of network traffic, so it reduces packet loss. Therefore, as the network size increases, this metric shows better results (Figure 19). In PROC, the network traffic increases faster than in THESEUS when the network size increases. It is worth mentioning that in our THESEUS implementation, a packet loss could cause six samples losses (so, packet losses in THESEUS are more critical). However, even with this potential drawback, the packet losses in THESEUS were reduced, and sample losses result improved.

Delay Time (M52): In the average of all simulations, THESEUS delivered samples with a delay of 6435ms and PROC with 153ms of delay. Such delay in THESEUS is acceptable

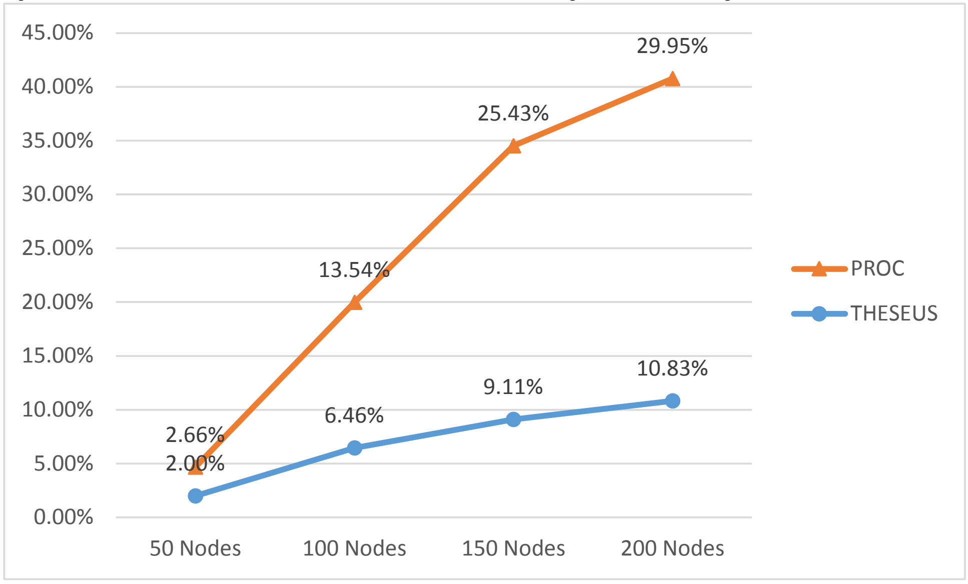

Figure 19. Results of the Sample Loss metric

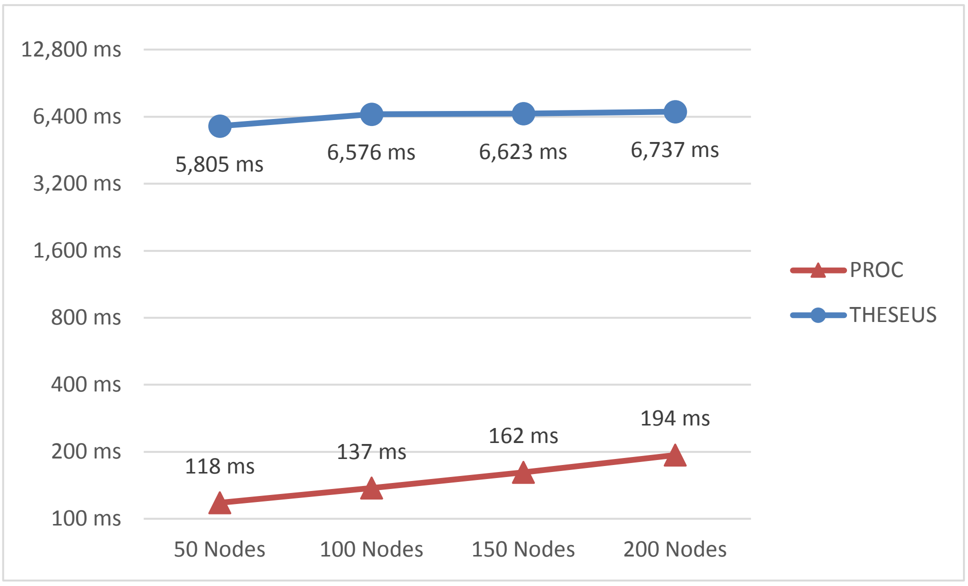

Figure 20. Results of the Delay Time metric

and is not considered as a disadvantage, since the maximum accepted delay time (MDTC) was set to 10 seconds and THESEUS delivered samples in less than 65% of MDTC. In other words, THESEUS improved the network lifetime and efficiency using a delay time that was acceptable by applications, thus without sacrificing the provided QoS.

Finally, Regarding **Q5**, about drawback(s) of using the packet aggregation technique, using larger packet size (consequently using more energy and imposing a higher risk of losing more samples per packet loss) could be a drawback. Nonetheless, the evaluation showed that, since THESEUS reduces network traffic, such potential drawback was transformed into an advantage. THESEUS improved the sample loss metric 51.29% in average of all tests. Moreover, THESEUS showed it delivers packet in a time smaller than the maximum accepted delay defined.

At the end, these results prove that we achieved our goal of improving the SSNs lifetime and confirming the advantages of using THESEUS packet aggregation algorithm.

6.2 ANALYSIS OF THE IMPACTS OF VARIATION OF IMPORTANT PARAMETERS

The main goal of this evaluation is to analyze the impact of important parameters on THESEUS that were fixed in the previous evaluation. Moreover, the results of this set of evaluation show the functionality of THESEUS on SSNs in absolute terms.

6.2.1 GQM

Similar to the previous evaluation (section 6.1), following the Goal, Question, and Metric (GQM) methodology (BASILI *et al.*, 1994), the following goal was defined for this set of experminets:

(i) Analysis of THESEUS by varying the number of applications, nodes distance, nodes random position, sink node position, cycle time, MDTC, and simulation time with the purpose of finding the impact of such parameters on the network behavior.

This goal can be expressed by seven questions:

Q1: What is the impact of the number of applications on THESEUS?

Q2: What is the impact of the distance between nodes on THESEUS?

Q3: What is the impact of the displacement of nodes on THESEUS (in comparison to the grid position format)?

Q4: What is the impact of the sink node position on THESEUS?

Q5: What is the impact of the cycle time on THESEUS?

Q6: What is the impact of the MDTC on THESEUS?

Q7: What is the impact of simulation time on THESEUS in regards to energy consumption of nodes?

We used the same 11 metrics defined in the previous section (6.1) on this evaluation to answer each question. However, **Memory usage** (M31) metric has the same value, independent of the test scenario (since this metric depends on the implementation code, which did not change); therefore, we did not analyze this metric here (it is available in section 6.1.3).

6.2.2 Evaluation methodology and scenarios

In this evaluation, we compared the result of variation of each parameter to find its impacts on THESEUS using AVRORA 1.7.117 simulator (AVRORA SIMULATOR, Accessed May 2015). We considered the evaluation methodology of the perivious section as a basic configuration for this set of experiments. We used the topology of 100 nodes that was defined in section 6.1.3. In each set of following tests, we varied one of the aforementioned parameters (all other parameters were similar to the last evaluation).

6.2.3 Tests and analysis of results

In this evaluation, we performed six group of experiments, each group for assessing the variation of one of the aforementioned parameters. The simulations for all these experiments ran for five hours. In the following section, we analyze the results of each group of tests and the impact(s) of each parameter.

6.2.3.1 number of applications

In the previous tests, we considered two applications in two areas of the network. In the scenario of 100 nodes, the first application (app1, monitoring humidity) runs in all sensor nodes (with ID from 1 to 99) and sample rate of 28 seconds. The second application (app2, monitoring temperature) runs in first half of the sensor nodes (with ID from 1 to 49) and with the same sample rate of 28 seconds. For this evaluation, we add two other applications. The third application app3 was specified to monitor luminosity and runs in sensor nodes with ID from 30 to 69 and sample rate of 42 seconds. The fourth application (app4, monitoring air pressure) runs in sensor nodes with ID from 30 to 49 and sample rate of 14 seconds (Figure 21).

Therefore, we considered four scenarios: (i) one application (app1); (ii) two applications (app1 and app2); (iii) three applications (app1, app2, and app3); and finally (iv) four applications (app1, app2, app3, and app4) running on the network. We performed the tests for each scenario using THESEUS and PROC to illustrate the difference of sharing the network with more applications (since we did not find any applicable routing protocol specific for SSNs). In the following, we compare and analyze the results using the same metrics that we defined in the previous evaluation.

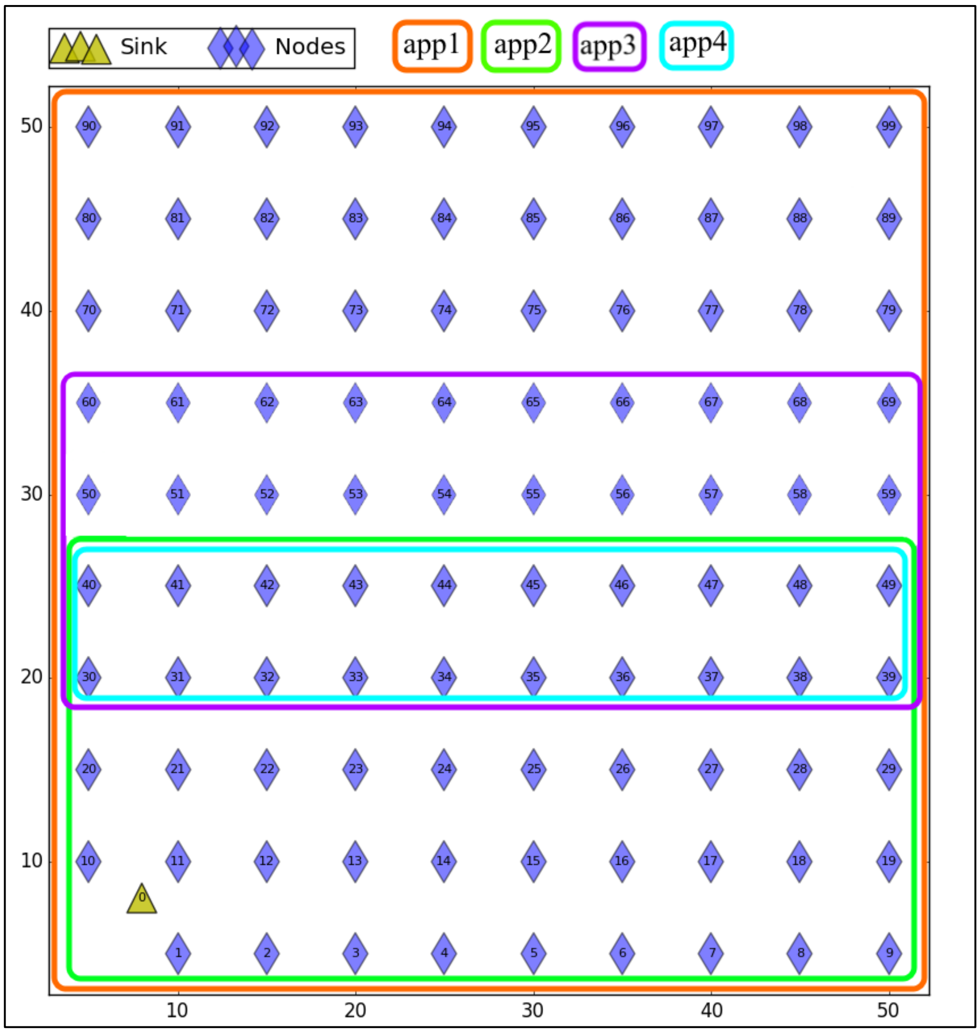

Figure 21. Topology of 100 nodes and 4 applications

Regarding Q1, Table 2 shows the values of the defined metrics for each scenario. As expected, the metrics, **Number of Packets used in route construction** (M33) and **Route Construction Time** (M34), showed almost the same values in all tests because the routes construction process of THESEUS and PROC are similar. Moreover, it shows the route

Table 2. Results of Number of running applications variation tests

100 Nodes 5 hours		M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
One App	THESEUS	0.0572	0.0191	0.29%	115.9	630 ms	3.16	2.396	7.89%	7.02%	6,811 ms
	PROC	0.0693	0.0359	0.38%	115.0	629 ms	4.61	0.000	8.31%	18.23%	73 ms
Two Apps	THESEUS	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.226	6.44%	6.46%	6,576 ms
	PROC	0.0860	0.0506	0.46%	115.0	626 ms	4.14	0.000	6.56%	13.54%	137 ms
Three Apps	THESEUS	0.0729	0.0307	0.37%	113.2	632 ms	2.98	2.255	6.17%	6.42%	6,621 ms
	PROC	0.0867	0.0509	0.47%	113.8	634 ms	4.05	0.000	7.30%	16.29%	166 ms
Four Apps	THESEUS	0.0744	0.0331	0.38%	114.0	629 ms	3.05	2.264	6.17%	6.64%	6,220 ms
	PROC	0.0888	0.0526	0.47%	114.1	633 ms	4.24	0.000	6.98%	16.45%	178 ms

construction process is independent of number of applications. Regarding the **Energy Average** (M11) and **Energy Population Standard Deviation** (M21) metrics, THESEUS shows more improvement when the number of applications increases than PROC (Figure 22 and Figure 23). Even, in the case of single application scenario, THESEUS was more energy efficient than PROC based on its aggregation ability. In these scenarios, the app3 and app4 have not a big impact on increasing the sample rates; therefore, the graphs does not show a tangible improvement of M11 and M21 metrics for THESEUS using three applications and four applications. Analyzing this metrics by considering more applications and more demanding rate is a considerable future work to show better the THESEUS ability regarding handling multiple applications. The **CPU Active** (M32) metric shows that more number of applications brings

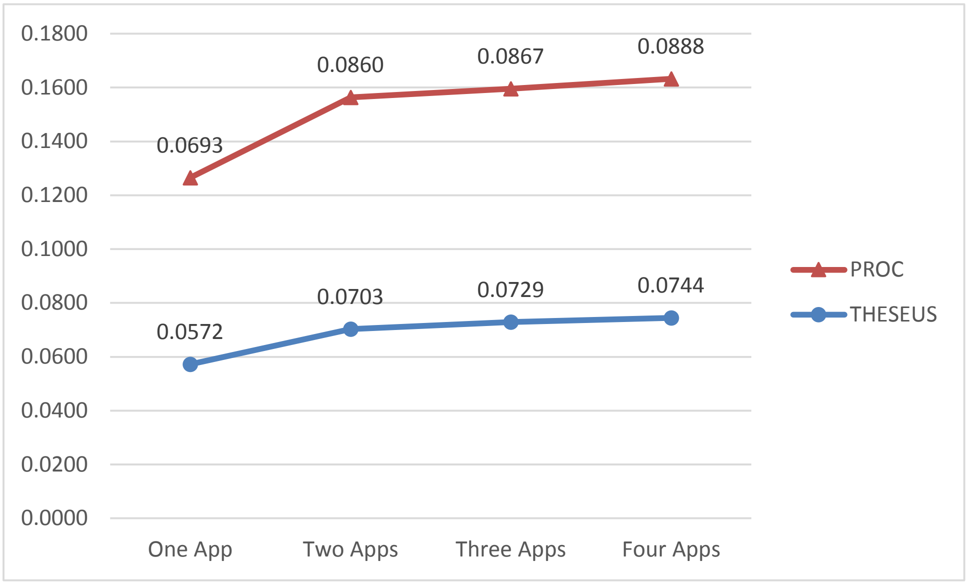

Figure 22. Results of the Energy Average metric

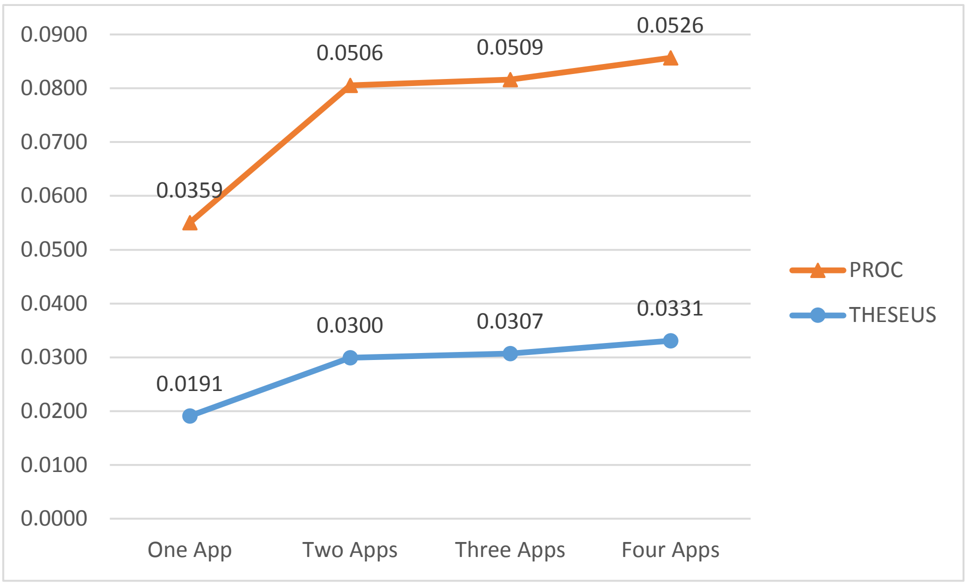

Figure 23. Results of the Energy PSTD metric

more CPU activity on nodes in both protocols. Moreover, the result shows almost in all cases that THESEUS uses nodes' CPU 20% less than PROC. About the **Network traffic rate per sample** (M41), **Packet Loss** (M43) and **Sample Loss** (M51) metrics results show similar values in all scenarios, showing THESEUS worked better than PROC in all of them (we analyzed these metrics in section 6.1.3). The **Delay Time** (M52) metric shows THESEUS performs better by raising the number of applications. More application means more samples were generated during the defined limited time (MDTC). Therefore, the aggregation process finished faster since the maximum packets size were reached before the maximum accepted delay elapsed. On the other hand, the **Delay Time** metric increases by raising the number of applications in PROC, since the probability of packet collisions increases as more data samples are simultaneously generated.

6.2.3.2 distance between Nodes

In our previous tests, we placed the nodes on a **square grid** with a distance of **5 meters between nodes** in each horizontal and vertical direction in the Cartesian plane. In this group of experiments, we varied such distance by 3, 10, and 15 meters in order to find the impact of this parameter on THESEUS. The 3, 10, and 15 meters distances were chosen based on the MICAz datasheet (MICAZ & MICA2, Accessed May 2015). The indoor wireless range of this platform is 20 to 30 meters, but experimental tests by varying the distance between nodes in AVRORA show that this simulator considers coverage rate for this platform around 15 meters.

Regarding Q2, Table 3 shows the results of distance variation tests. All metrics improved as the nodes got closer to each other. Such improvement happens due to the participation of more nodes in the process of aggregating the samples and routing packets. Denser networks bring more neighbors for each node, which is important for selecting different parent node in order to balance energy in each cycle. Figure 24 shows the routes in a cycle of each scenario, showing how a denser network brings more chance for aggregating samples and balancing energy.

Table 3. Results of Nodes Distance variation tests

100 Nodes 5 hours	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
3 meters	0.0669	0.0194	0.53%	102.4	607 ms	2.45	1.93	5.30%	4.66%	3,154 ms
5 meters	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.23	6.44%	6.46%	6,576 ms
10 meters	0.1312	0.0950	0.38%	133.7	682 ms	5.95	2.14	4.03%	15.58%	9,561 ms
15 meters	0.2570	0.1242	0.55%	150.6	713 ms	11.26	1.49	5.37%	37.85%	9,611 ms

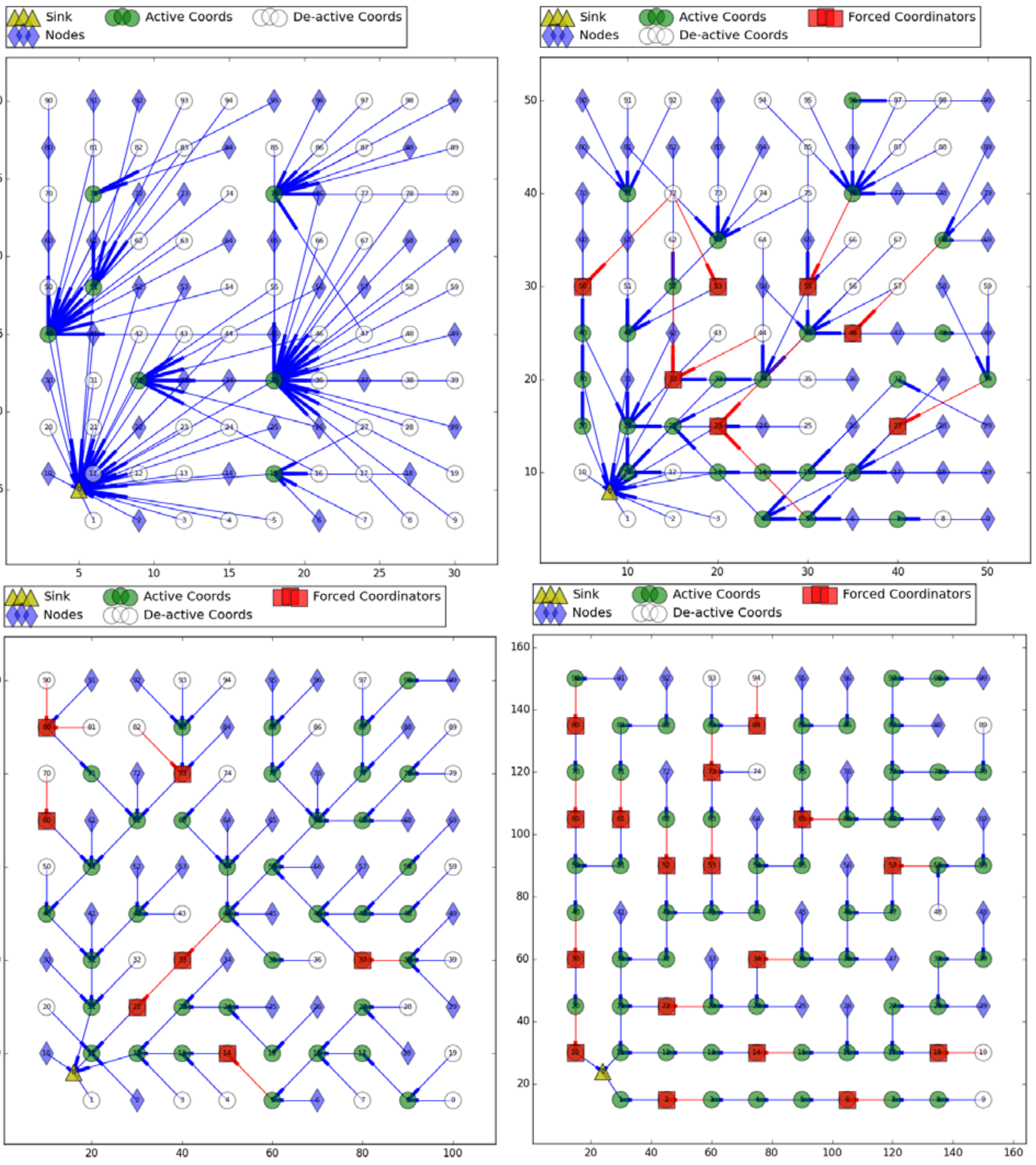

Figure 24. Graphs of data flows during a cycle of THESEUS simulation (100 nodes) for 3, 5, 10, and 15 meters distances

6.2.3.3 random Position of Nodes

In our previous tests, we placed the nodes on a **square grid** in the Cartesian plane. In this group of experiments, we used three different random positions for nodes in order to show the impact of the displacement of nodes on THESEUS We compared the results of these three random positions with the previous scenario of square grid positions in the same area size to find the impact of this change on THESEUS.

Regarding Q4, the results in Table 4 show no tangible differences regarding any assessed metrics, which proves that networks with similar density have similar results. Based

Table 4. Results of the Nodes Random Position variation tests

100 Nodes 5 hours	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
5m Sq. Gr.	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.23	6.44%	6.46%	6,576 ms
Rnd.P. 1	0.0799	0.0430	0.44%	111.8	618 ms	3.24	2.53	5.04%	8.28%	7,021 ms
Rnd.P. 2	0.0706	0.0331	0.39%	111.4	623 ms	2.89	2.29	5.34%	8.05%	5,989 ms
Rnd.P. 3	0.0733	0.0330	0.39%	111.4	620 ms	3.00	2.51	5.46%	7.50%	7,175 ms

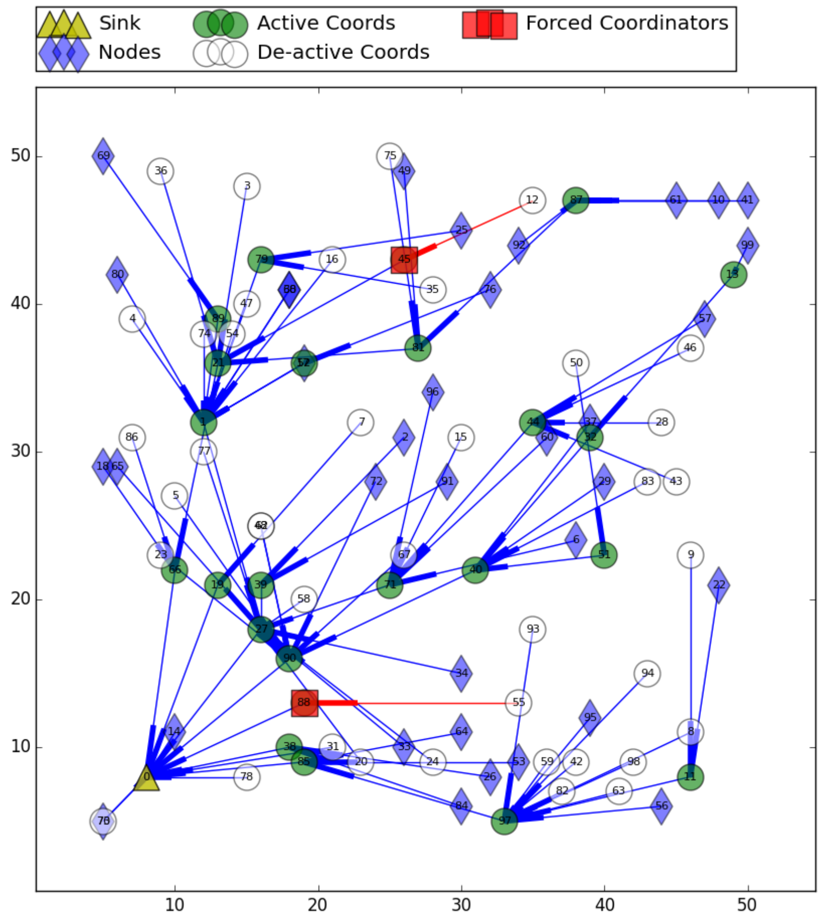

Figure 25. Graph of data flows during a cycle of THESEUS simulation (100 nodes) for one of nodes random positions

on the results from variation of nodes distance (previous tests) and nodes random positions, it is possible to conclude that the network density i.e. the average of number of nodes per square meter could change the network behavior, but nodes displacement with similar density does not change network behavior. Moreover, THESEUS proved that it guarantees network connectivity in any configuration of nodes' positions. This is based on the tests' results (sample loss and no rise in Nrj. PSTD) and considering Figure 25, which shows routes that THESEUS created for a cycle of one of random position tests.

6.2.3.4 sink node position

In all of our experiments, we placed the sink node in the corner of the topology to maximize the network depth. In this experiment, we compare the result of 100 nodes scenario from previous section with the new experiment of placing the sink node near the center of the network topology (we changed the place of node with ID 44 by the sink node, Figure 26). Table 5 shows the results of previous and new tests.

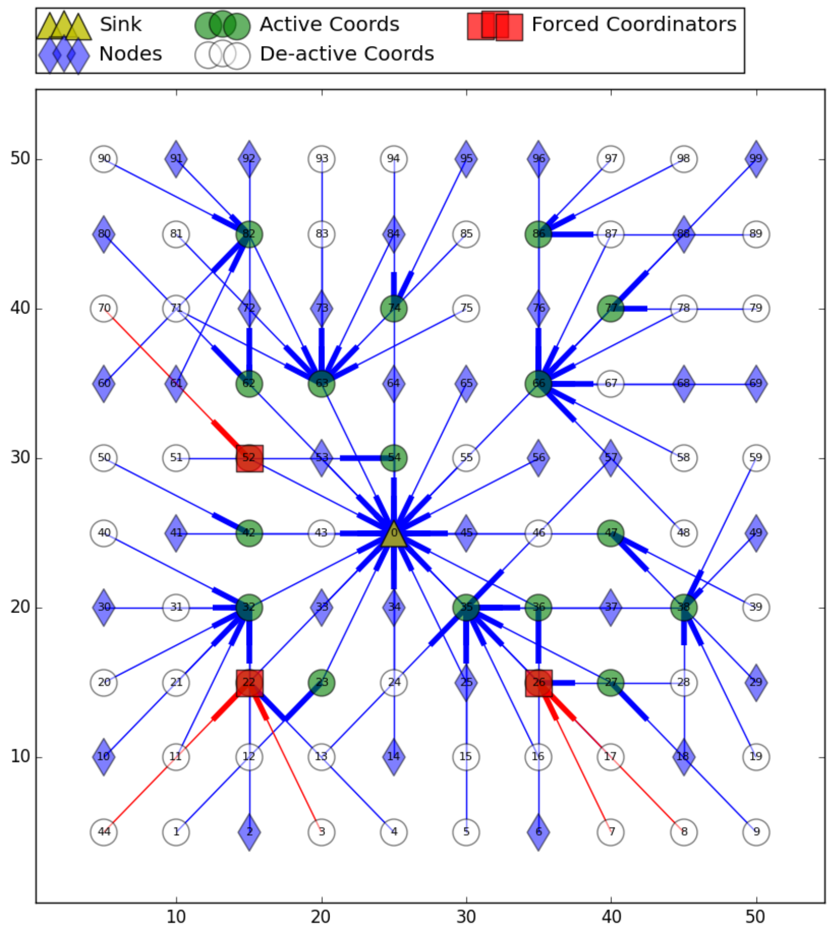

Figure 26. Graph of data flows during a cycle of THESEUS simulation (100 nodes) for the central sink node

Table 5. Results of the Sink Node Position tests

100 Nodes 5 hours	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
Corner	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.23	6.44%	6.46%	6,576 ms
Center	0.0608	0.0193	0.32%	107.1	613 ms	2.61	1.91	6.16%	3.13%	5,777 ms

Regarding Q4, the results indicate that all metrics (excluding aggregation rate metric) were improved when the sink node was placed in the center of the network. It happened because when the sink was placed in the center, there are more routes from more directions to access it. Therefore THESEUS balanced the energy better than in the previous test. In addition, the number of hops to deliver messages to the sink node is smaller than before, so the energy consumption of nodes was improved. Regarding the aggregation rate metric, which had better value in the test with the sink node in the corner, we can conclude that many nodes delivered the data samples directly to the sink node without passing through coordinators, since they have been placed around the sink. Therefore, coordinators had less chance to aggregate samples.

6.2.3.5 cycle time

In the previous evaluation, we fixed this parameter to 180 seconds. In this group of experiments, we varied this parameter by 60, 1800, and 3600 seconds, and performed another test with just one cycle of route construction (without reconstruction of routes during five hours of simulation) to verify the impact of variation of this parameter on THESEUS.

Regarding Q5, Table 6 shows the results of these variation tests. Analytically, as we discussed previously, a long period of cycle time damages the energy usage balance among the nodes and increases the probability of losing more samples if some coordinator nodes fail during that cycle time. On the other hand, a short period of cycle time increases the energy usage of nodes and the probability of packet collisions. The results of energy balance metric (M21) shows that the value of 180 seconds for the cycle time in our scenarios (based on sample

Table 6. Results of the Cycle time variation tests

100 Nodes 5 hours	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
60 sec.	0.1041	0.0347	0.46%	113.5	630 ms	3.06	2.24	11.41%	6.71%	6,485 ms
3 min.	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.23	6.44%	6.46%	6,576 ms
30 min.	0.0519	0.0341	0.31%	117.6	628 ms	2.95	2.23	1.23%	6.24%	6,183 ms
1 hour	0.0495	0.0398	0.30%	116.8	619 ms	2.88	2.25	0.84%	5.28%	6,511 ms
one cycle	0.0476	0.0607	0.29%	117.0	619 ms	2.83	2.23	0.53%	5.45%	6,516 ms

rates) is the optimum value for this parameter. However, considering all other metrics such as energy average, packet loss, and delay time the value of 1800 seconds (30 minutes) of cycle time shows the best option for our scenario. This is becauses instead of losing negligible energy usage balance; the energy usage, packet loss, and delay time were more improved.

The improvement of this metric is related to the number of cycles in tests because more cycles of reconstruction of routes mean changes in the node that is playing the role of coordinator. As the roles of the nodes change, THESEUS can better balance energy consumption among the SSN nodes.

6.2.3.6 MDTC

This parameter was initially fixed to 10 seconds. Now, we varied it by 250 milliseconds, 1, 5, 30, and 60 seconds to analyze the impacts of variation of this parameter on THESEUS. Regarding Q6, Table 7 shows the obtained results. The results of tests with 30 and 60 seconds shows more sample loss since most of the delivered packets carried the maximum number of samples (losing a packet means losing more samples), but these tests show less energy usage. The 1-second test proves that our packet aggregation method could play a crucial role to save more energy considering packets delivered to the sink node with a delay smaller than one second. For instance, this test (1-second test with THESEUS) improved the node energy usage around 20% in comparison to the similar test with PROC (one second data delivery delay is negligible for most of existing application demands).

Table 7. Results of the MDTC variation tests

100 Nodes 5 hours	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
250 ms.	0.0727	0.0336	0.38%	114.3	629 ms	3.13	1.967	6.64%	7.76%	249 ms
1 sec.	0.0720	0.0320	0.37%	114.1	631 ms	3.04	2.229	6.20%	6.31%	956 ms
5 sec.	0.0731	0.0358	0.38%	113.5	633 ms	3.09	2.203	6.12%	5.95%	3,563 ms
10 sec.	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.226	6.44%	6.46%	6,576 ms
30 sec.	0.0646	0.0232	0.30%	115.1	635 ms	2.59	3.095	7.78%	12.72%	10,383 ms
60 sec.	0.0645	0.0254	0.30%	113.9	635 ms	2.63	3.236	7.46%	13.04%	13,298 ms

6.2.3.7 simulation Time

In order to observe the changes in the metrics with variation of simulation time, we experimented different simulation times of 1, 10, and 20 hours for the same 100 nodes scenario.

The Table 8 shows the changes in energy consumption corresponding to each simulation time. As it is observed the energy usage average and balance metrics increase linearly with simulation time. Other metrics provide similar values in all tests which proves that the proposed routing protocol is resilient with simulation time variations.

Regarding Q7, the reason for these experiments was to show the resiliency/flexibility of routing protocol in regard to longer simulation time. Hence, we showed that energy consumption level increases linearly with simulation time and the other metrics remain the same.

Table 8. Results of the Simulation time variation tests

100 Nodes	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
1 hour	0.0152	0.0106	0.36%	114.2	624 ms	3.20	2.15	5.10%	5.95%	6,699 ms
5 hours	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.23	6.44%	6.46%	6,576 ms
10 hours	0.1436	0.0613	0.36%	115.5	629 ms	3.03	2.27	6.38%	6.58%	6,703 ms
20 hours	0.2905	0.1181	0.37%	114.4	628 ms	3.12	2.27	6.09%	7.73%	6,609 ms

6.3 ANALYSIS OF THE IMPACTS OF USING MORE THAN ONE SINK NODE ON THESEUS

The main goal of this set of evaluations is to analyze the impact of the number of sink nodes on THESEUS, which is one of the features that THESEUS provides and PROC does not.

6.3.1 GQM

The following goal was defined for this evaluation:

(i) Analysis of THESEUS by varying the number of sink nodes with the purpose of finding the impact of using more than one sink node on a network.

Two questions can express this goal:

Q1: Does THESEUS support more than one sink node, and does it support adding and removing sink nodes from the network dynamically?

Q2: What is the impact of the number of sink nodes on THESEUS?

6.3.2 Evaluation methodology and scenarios

In this evaluation, we used similar conditions and parameters of the scenario with 100 nodes from our previous evaluation. About Q1, we modified our previous scenario by adding second sink node, also turning off the first sink node for three cycles in the middle of the simulation, and compare the result with similar test without turning off the sink node. This will enable us to analyze the THESEUS behavior regarding adding and removing sink nodes dynamically. Regarding the Q2, we varied the number of sink nodes by two and four in the same scenario with 100 nodes. We used metrics that were defined in the section 6.1.1 to compare the results of this set of test for answering the aformentioned questions.

6.3.3 Tests and analysis of results

The tests regarding verifying the THESEUS ability to add and remove the sink node were done during ten cycles (1800 seconds). In our implementation, we considered around 10 seconds tolerance for checking valid time of the last received Sync message (section 4.3.4), based on considering network latency for delivering Sync messages. Therefore, removing a sink node could cause some sample loss for those 10 seconds.

Regarding Q1, Table 9 shows the results of the test without removing the sink node (normal) and the other one that first sink node was removed during three cycles (Figure 27). The result shows that removing and adding a sink node brings some limited impacts on the nodes such as using more energy, losing the energy usage balance and losing some samples. However, even in the short time of tests, these impacts are not considerable. Such impacts are not tangible during long time tests either.

Table 9. Results of the add and remove Sink Node

2 sinks + 99 Nodes 30 minutes	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
Normal	0.0067	0.0022	0.37%	116.9	618 ms	2.76	1.67	6.26%	2.45%	6,408 ms
3 cycles by one sink	0.0073	0.0032	0.40%	116.5	696 ms	2.91	1.76	5.69%	5.06%	5,800 ms

Regarding Q2, the impacts of using more than one sink node, we tested THESEUS by adding sink nodes to the scenario of 100 nodes from previous evaluation (section 6.1). We simulated that same scenario with two sink nodes and four sink nodes (Figure 28). Table 10 shows the results of varying the number of sink nodes. As results show, network lifetime increased since the energy usage average and balance improved by increasing the number of sink nodes. Such improvements happen because the presence of more sink nodes allows dividing the network into smaller network with less network depth. Consequently, the packets are delivered to the sink node by traversing fewer hops. Moreover, the sample loss and delay time metrics improved when the network had more sink nodes to deliver samples.

Table 10. Results of the add and remove Sink Node

sinks + 99 Nodes 5 hours	M11	M21	M32	M33	M34	M41	M42	M43	M51	M52
1 Sink	0.0703	0.0300	0.36%	115.6	630 ms	2.94	2.23	6.44%	6.46%	6,576 ms
2 Sinks	0.0672	0.0242	0.30%	115.1	645 ms	2.82	1.71	6.64%	4.35%	6,497 ms
4 Sinks	0.0580	0.0190	0.25%	112.9	665 ms	2.50	1.29	7.74%	4.48%	3,888 ms

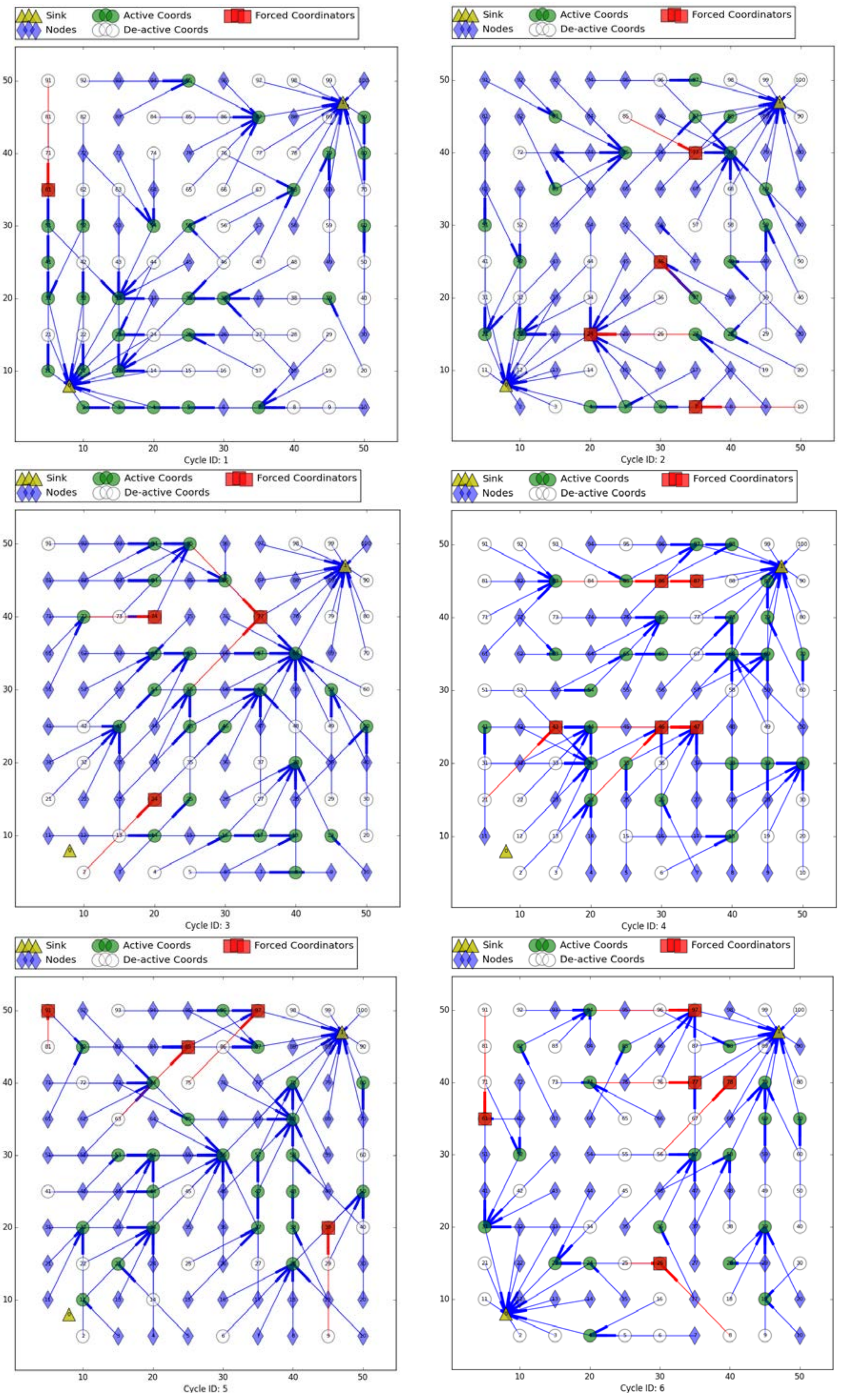

Figure 27. Graphs of data flows of six cycles of multi sink test, where the first sink removed in cycle ids: 3, 4, and 5

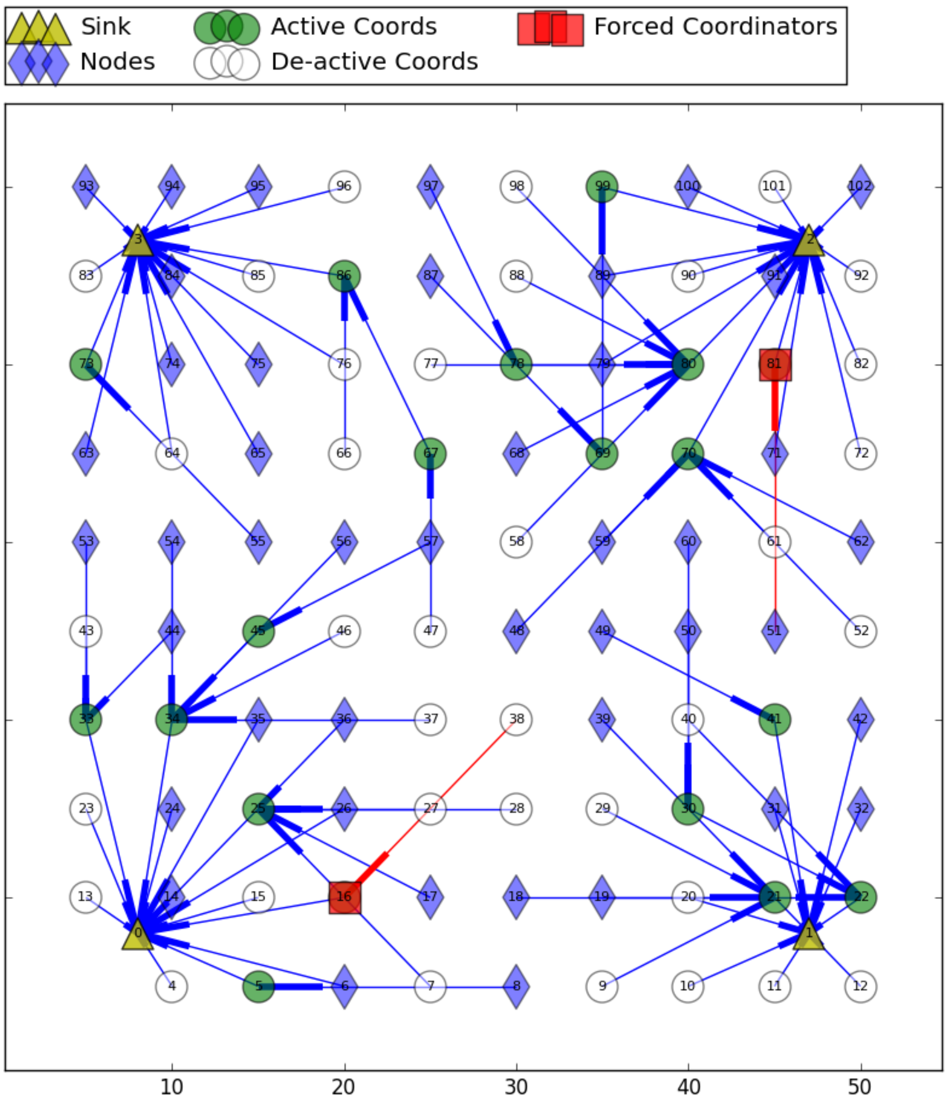

Figure 28. Graph of data flows during a cycle of simulating 99 nodes and 4 sinks

6.4 COMPARISON BETWEEN REAL AND SIMULATED NODES

In this experiment, we analyzed the differences of behavior of real nodes and simulated nodes to verify the validity of our previous simulated results. Since our main goal is prolonging the network lifetime, comparing the results of M11 (Energy Average) and M21 (Energy PSTD) metrics are our main concern in this section.

Sixteen real MICAz sensor nodes were used to make a real test of THESEUS in the Ubiquitous Computing Laboratory of PPGI-UFRJ. We placed the nodes as shown in Figure 29 in our Laboratory. The previously defined app1 was set for nodes with ID 1 to 16 and app2 was set for nodes with ID 1 to 12. We connected the sink node and nodes with ID 1 to 5 through a USB cable and proper board (crossbow) to the computer. PrintF function of TinyOS was used to save the log of all performed activities. During one hour of test, 3525 samples were received in the sink node. We simulated a same scenario (to the real test) with same parameters and

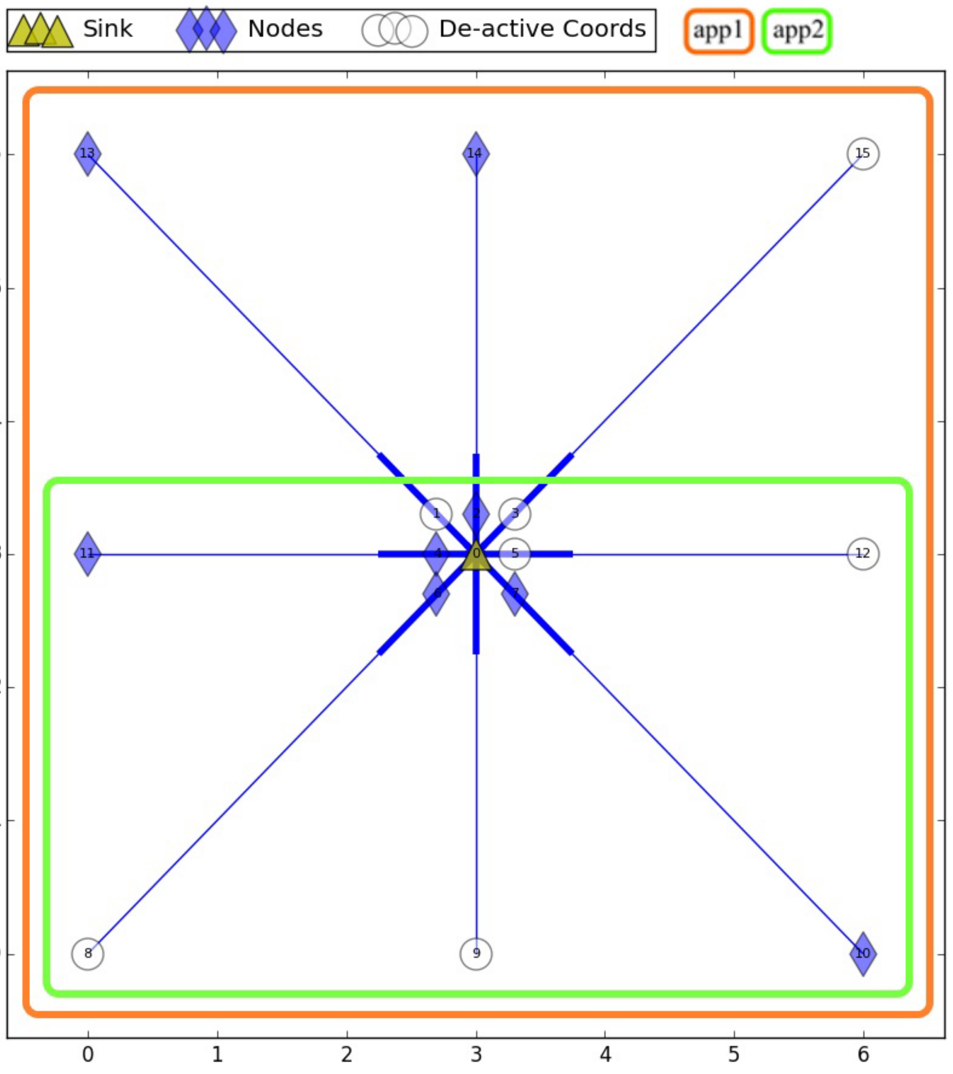

Figure 29. Graph of data flows during a cycle of real test scenario

conditions (such as node positions) in AVRORA simulator. Table 11 shows the results of real and simulated tests. M32 (CPU Active), M34 (Route construction time), and M52 (Delay Time) metrics were not measurable in our real nodes. Regarding energy usage average (M11) and energy usage balance (M21) metrics, our real test showed that AVRORA simulation results are well promising and our simulated results are valid. Moreover, we realized AVRORA considers more packet loss (M43) than real nodes. However, sample loss (M51) was zero for both tests because of using packet acknowledgement feature.

Table 11. Results of the real and simulated tests

sink + 15 Nodes 1 hour	**M11**	**M21**	**M32**	**M33**	**M34**	**M41**	**M42**	**M43**	**M51**	**M52**
Real	0.0079	0.0013	NA	16.0	NA	1.01	0.00	0.11%	0.00%	NA
AVRORA	0.0086	0.0012	0.18%	16.0	<1 ms	1.10	0.00	0.75%	0.00%	<1 ms

7. CONCLUSIONS AND FUTURE WORK

In this work, we proposed THESEUS as an application generic routing system for SSNs. THESEUS routing algorithm is inspired by PROC. The primary goal of THESEUS is to extend the lifetime of the network by reducing the amount of energy usage and balancing energy usage among all nodes for the multi-application environments. To the best of our knowledge, THESEUS is the first applicable routing protocol for SSNs with support of dynamic multiple sink nodes.

Considering the specific features of SSN, our main contribution is creating an energy efficient and application generic routing system for SSNs. Regarding the routing technique THESEUS detailed contributions are:

(i) THESEUS packet aggregation is independent of the packet. The goals of such technique are: to avoid a dependency on the data content in the aggregation process; and to work completely inside the network layer.

(ii) THESEUS app function. The main goal of this function is to select the best nodes to be a coordinator. To better match the functionality of this function by SSN conditions, we considered the number of applications and network traffic during the coordinator selection to select more coordinators among the set of nodes that generate more data samples.

(iii) THESEUS makes use of QoS parameters and applications' requirements to adapt the routing paths.

(iv) THESEUS supports multiple sink nodes dynamically, bringing the ability of using more than one sink node in nework (which will be probably the typical case for SSN), and also bringing the ability to add or remove sink nodes while the network is working.

We introduced a well-detailed software architecture for the proposed Routing system. Moreover, the running algorithms are presented for all software components. To verify our proposed system, we implemented a routing protocol based on THESEUS routing system for MICAz platform to assess its performance. We also implemented PROC routing protocol by making changes in THESEUS implementation. All the performed evaluation followed the "Goal, Question, Metric" (GQM) methodology. We evaluated THESEUS in four group of tests:

(i) Comparing THESEUS and PROC. As to the best of our knowledge, no other practical related work were found in the literature of routing solutions for SSNs for performing such a comparison. The results showed great improvements of THESEUS, in relation to PROC, when operating in a shared sensor network scenario. Such improvements are related to energy saving and balancing in the network, in the face of a negligible increase of memory usage.

(ii) Analysis of the impact of the variation of important parameters on THESEUS behavior. The variation tests prove the functionality, efficiency, and scalability of THESEUS features, such as routes construction and THESEUS packet aggregation techniques.

(iii) Analysis of the impact of using more than one sink node on THESEUS performance, which the results proved the THESEUS ability to adapt the network by adding and removing sink nodes dynamically.

(iv) Comparison between Real and Simulated Nodes. This test proved the validity of our simulated results.

Based on the analysis of aforementioned tests, our proposed routing system is an efficient, stable and scalable solution for routing on SSNs.

It is worth to mention that our article of THESEUS (THESEUS: A Routing System for Shared Sensor Networks) was accepted on Computer and Information Technology (CIT-IEEE) 2015 conference (CAPES quails B1) successfully (Appendix A).

7.1 FUTURE WORK

As proposals for future work, we emphasize the following three developments regarding our routing system.

First, we identified in our experiments that in certain topologies, some coordinators are idle after elected, i.e. they are not chosen as part of the path for routing messages. Therefore, it is suggested to investigate solutions for improving the efficiency with which the decision of setting each nodes' role is taken, avoiding idle coordinators. In our system, the election of coordinator nodes is guided by the result of the app function, which is performed within each node and returns the probability of this node becoming a coordinator. It is possible to improve this value returned by the app function, seeking better coordinator selection. This could be done by keeping track of recent history regarding the roles that each node is assumed to have, and thus using simple and applicable machine learning techniques on nodes for analyzing this history, in order to make better decisions.

Second, we realized, in the experiments performed in this work, the network running THESEUS could behave better by configuring it with lower values of Backoff Time and higher value of Cycle Time. This would allow THESEUS to save time and network resources when few applications are sharing the network. For performing this configuration, while keeping the ability to restore it when more applications arrive, it is required to add to THESEUS the capability of self-adaptating to the number of running applications and network density. Similar to the previous suggested development, it is possible to use simple and applicable machine learning techniques that will respond to the number of applications running in the network and network density, selecting most adequate values of Backoff Time and Cycle Time.

Third, we suggest a thorough research on different auxiliary protocols for SSN, useful for supporting THESEUS and more suitable to its requirements. A class of protocols that can be investigated are the class of time synchronization protocols, seeking to find protocols that are more suitable to SSN requirements. It is important to investigate solutions for providing THESEUS with the capability of time synchronization, so that it can become more efficient and capable. In future works, some key concepts of several existing time synchronization protocols in the literature can be easily implemented in THESEUS by changing (adding fields to) the content of already defined route messages.

REFERENCES

AKKAYA, K.; YOUNIS, M. A survey on routing protocols for wireless sensor networks. **Ad hoc networks,** v. 3, n. 3, p. 325-349, 2005.

AL-KARAKI, J. N.; KAMAL, A. E.. Routing techniques in wireless sensor networks: a survey. **IEEE Wireless Communications**, New York, v. 11, n. 6, p. 6-28, 2004.

AVRORA SIMULATOR. Disponível em: <http://compilers.cs.ucla.edu/avrora>. Acesso em: 14 mar. 2015.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. Goal question metric paradigm. In: ______. **Encyclopedia of Software Engineering**. Hoboken: Wiley, 1994. p. 528-532. 2 v

BHATTACHARYA, S. et al. Multi-application deployment in shared sensor networks based on quality of monitoring. IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM (RTAS)16., 2010, Stockholm. **Proceedings**… New York: IEEE, 2010. p. 259-268.

CHEN, M.; GONZALEZ, S.; LEUNG, V. C. M.. Applications and design issues for mobile agents in wireless sensor networks. **IEEE Wireless Communications**, New York, v. 14, n. 6, p. 20-26, 2007.

DELICATO, F. C. et al. Energy Awareness and Efficiency in Wireless Sensor Networks: From Physical Devices to the Communication Link. In: ZOMAYA, Albert Y. **Energy-Efficient Distributed Computing Systems**. Hoboken: Wiley, 2012. p. 673-707.

DELICATO, F. C. et al. MARINE: MiddlewAre for resource and mIssion-oriented sensor NEtworks. **Mobile Computing and Communications Review**, New York, v. 17, n. 1, p. 40-54, 2013.

DEMIRKOL, I.; ERSOY, C.; ALAGOZ, F. MAC protocols for wireless sensor networks: a survey. **IEEE Communications Magazine**, New York, v. 44, n. 4, p. 115-121, 2006.

DIETRICH, I.; DRESSLER, F. On the lifetime of wireless sensor networks. **ACM Transactions on Sensor Networks**, New York, v. 5, n. 1, p. 1-38, 2009.

EFSTRATIOU, C. et al. A shared sensor network infrastructure. ACM CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, 8., 2010, Zurich. **Proceedings**… New York: ACM, 2010. p. 367-368.

ELTARRAS, R.; ELTOWEISSY, M. Adaptive Multi-Criteria Routing for Shared Sensor-Actuator Networks. In: IEEE GLOBAL TELECOMMUNICATIONS CONFERENCE, 2010, Miami. **Proceedings**… New York: IEEE, 2010, p. 1-6.

FARIAS, C. et al. Multisensor data fusion in Shared Sensor and Actuator Networks. INTERNATIONAL CONFERENCE ON INFORMATION FUSION, 17., 2014, Salamanca. **Proceedings**… New York: IEEE, 2014, p. 1-8.

FLORES-CORTÉS, C. A.; BLAIR, G. S.; GRACE. P. An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. **IEEE Distributed Systems Online**, v. 8, n. 7, p. 1-11, 2007.

HEFEIDA, M. et al. Context modeling in collaborative sensor network applications. In: INTERNATIONAL CONFERENCE COLLABORATION TECHNOLOGIES AND SYSTEMS, 2011, Philadelphia. **Proceeding**… New York: IEEE, p. 274-279, 2011.

HEINZELMAN, W. R.; CHANDRAKASAN, A.; BALAKRISHNAN, H. Energy-efficient communication protocol for wireless microsensor networks. In: ANNUAL HAWAII INTERNATIONAL CONFERENCE SYSTEM SCIENCES, 33, 2000. **Proceedings**… New York: IEEE, 2000. p. 1-10.

HUGHES, D. et al. LooCI: a loosely-coupled component infrastructure for networked embedded systems. In: INTERNATIONAL CONFERENCE ON ADVANCES IN MOBILE COMPUTING AND MULTIMEDIA, 7. 2009, Kuala Lumpur. **Proceedings**… New York: ACM, 2009, p. 195-203.

INOUE, N. et al. A cooperative routing method with shared nodes for overlapping wireless sensor networks. In: INTERNATIONAL WIRELESS COMMUNICATIONS AND MOBILE COMPUTING CONFERENCE, 2014, Nicosia. **Proceedings**… New York: IEEE, 2014, p. 1106-1111.

JAYASUMANA, A. P.; HAN, Q.; ILLANGASEKARE, T. H. Virtual Sensor Networks - A Resource Efficient Approach for Concurrent Applications. In: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY, 4., 2007, Las Vegas. **Proceedings**… New York: IEEE, 2007. p. 111-115.

KULKARNI, S. S. TDMA service for sensor networks. In: INTERNATIONAL CONFERENCE DISTRIBUTED COMPUTING SYSTEMS WORKSHOPS, 24., 2004, **Proceedings**… New York: IEEE, 2004. p. 604-609.

LE, T.; NORMAN, T. J.; VASCONCELOS, W. Agent-based sensor-mission assignment for tasks sharing assets. In: INTERNATIONAL WORKSHOP ON AGENT TECHNOLOGY FOR SENSOR NETWORKS, 3., 2009, Budapest. **Proceedings**… 2009. p. 33-40.

LEONTIADIS, I. et al. SenShare: transforming sensor networks into multi-application sensing infrastructures. In: EUROPEAN CONFERENCE ON WIRELESS SENSOR NETWORKS, 9., 2012, Trento. **Proceedings**…Berlin: Springer, 2012. p. 65-81.

LI, C. et al. A survey on routing protocols for large-scale wireless sensor networks. **Sensors**, v. 11, n. 4, p. 3498-3526, 2011.

LI, W.; DELICATO, F. C.; ZOMAYA, A. Adaptive energy-efficient scheduling for hierarchical wireless sensor networks. **ACM Transactions on Sensor Networks**, New York, v. 9, n. 3, p. 33, 2013.

LI, W. et al. Efficient allocation of resources in multiple heterogeneous Wireless Sensor Networks. **Journal of Parallel and Distributed Computing**, New York, v. 74, n. 1, p. 1775-1788, 2014.

LOVETT, G. M. et al. Who needs environmental monitoring? **Frontiers in Ecology and the Environment**, v. 5, n. 5, p. 253-260, 2007.

MACEDO, D. F. et al. A rule-based adaptive routing protocol for continuous data dissemination in WSNs. **Journal of Parallel and Distributed Computing**, New York, v. 66, n. 4, p. 542-555, 2006.

MADRIA, S.; KUMAR, V.; DALVI, R. Sensor cloud: a cloud of virtual sensors. **IEEE Software**, Los Angeles, v. 31, n. 2, p. 70-77, 2014.

MICAZ & MICA2. Disponível em: <http://www.memsic.com/wireless-sensor-networks>. Acesso em: 14 mar. 2015.

NAKAMURA, E. F.; LOUREIRO, A. A. F.; FRERY, A. C. Information fusion for wireless sensor networks: methods, models, and classifications. **ACM Computing Surveys**, New York, v. 39, n. 3, p. 9, 2007.

NS-2. Disponível em: <http://www.isi.edu/nsnam/ns/>. Acesso em: 14 maio 2015.

QUALNET SIMULATOR. Disponível em: <http://web.scalable-networks.com/>. Acesso em: 12 maio 2015.

RAICU, I. et al. Toward loosely coupled programming on petascale systems. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Austin. **Proceedings**… New York: ACM, 2008. p. 22.

RAJAGOPALAN, R.; VARSHNEY, P. K. Data aggregation techniques in sensor networks: a survey. **IEEE Communications Surveys & Tutorials** , v. 8, n. 4, p. 48-63, 2006.

RODRIGUES, T. et al. Model-driven approach for building efficient wireless sensor and actuator network applications. In: INTERNATIONAL WORKSHOP SOFTWARE ENGINEERING FOR SENSOR NETWORK APPLICATIONS, 4., 2013, San Francisco. **Proceedings**… New York: IEEE, 2013. p. 43-48.

SHAH, S. Y.; SZYMANSKI, B. K. Dynamic multipath routing of multi-priority traffic in wireless sensor networks. In: ANNUAL CONFERENCE OF INTERNATIONAL TECHNOLOGY ALLIANCE, 6., 2012, Southampton. **Proceedings**… 2012.

TILAK, S.; ABU-GHAZALEH, N. B.; HEINZELMAN, W. A taxonomy of wireless micro-sensor network models. **Mobile Computing and Communications Review**, New York, v. 6, n. 2, p. 28-36, 2002.

TINYOS. Disponível em: <http://www.tinyos.net>. Acesso em: 14 mar. 2015.

VIJAY, G.; BEN ALI BDIRA, E.; IBNKAHLA, M. et al. Cognition in wireless sensor networks: a perspective. **IEEE Sensors Journal**, v. 11, n. 3, p. 582-592, 2011.

WU, C. et al. Submodular game for distributed application allocation in shared sensor networks. In: IEEE INFOCOM, 2012, Orlando. **Proceedings**… New York: IEEE, 2012. p. 127-135.

YU, Y. et al. Supporting concurrent applications in wireless sensor networks. In: INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, 4., 2006, Boulder. **Proceedings**… New York: ACM, 2006. p. 139-152.

APPENDIX A, THE PUBLISHED ARTICLE:

THESEUS: A Routing System for Shared Sensor Networks

Mohammadreza Iman, Flávia C. Delicato, Claudio M. de Farias, Luci Pirmez, Igor L. dos Santos, Paulo F. Pires
PPGI-DCC/IM, Federal University of Rio de Janeiro, UFRJ (Universidade Federal do Rio de Janeiro), Brazil
mohammadreza.iman@hotmail.com, {fdelicato, cmicelifarias, luci.pirmez, igorlsantos, paulo f.pires}@gmail.com

***Abstract*—THESEUS is an energy-efficient routing system for Shared Sensor Networks (SSNs), with the primary goal of extending network lifetime. THESEUS has two features that distinguish it from other works found in the literature. First, it saves SSN nodes energy by using a packet aggregation algorithm, which reduces the number of transmissions. Second, THESEUS balances energy usage in the whole SSN by its dynamic, QoS and energy aware route selection algorithm. Such energy usage balancing avoids network partitioning. Both features of THESEUS result in prolonging the SSN lifetime. Experiments were performed with the purpose of evaluating THESEUS effectiveness for improving the SSNs lifetime. Furthermore, THESEUS supports multiple sink nodes dynamically that means it adjusts routes in case of a sink node added or removed.**

Keywords-routing; shared sensor networks; wireless sensor networks; energy efficiency; packet aggregation; multiple sink nodes

I. INTRODUCTION

Wireless Sensor Networks (WSN) [1] are wireless networks consisting of spatially distributed autonomous devices using sensors to monitor physical or environmental conditions. Traditional WSNs are generally *application specific networks*, i.e. they are designed and deployed to serve a single application [2]. Considering the amount of nodes scattered in an area with each one usually having various sensing units, the idea of sharing the sensing and communication infrastructure of WSNs through multiple applications has recently emerged. This concept is known as Shared Sensor Networks (SSNs) [2]. SSNs are able to handle more than one application simultaneously in an efficient way by avoiding performing redundant tasks among applications, and by exploiting data aggregation and other functionalities of the network to increase resource utilization while meeting application requirements. Similarly to WSNs, the greatest challenge in SSNs arises from the energy-constrained nature of sensor nodes, which encourages the development of techniques to save as much energy from sensor nodes as possible, so as to prolong the network operational lifetime. On top of this, SSNs have new requirements and pose many new challenges that require adapting techniques / algorithms / protocols already successfully used in traditional WSNs. Routing algorithms is one of the key challenges to be tackled in order to foster the widespread use of the SSN paradigm and it is the focus of our work.

Routing algorithms for WSNs typically have strong impacts on network lifetime since radio communication is often regarded as the major source of energy consumption in these networks [3] [4]. Routing protocols can balance the energy usage of sensor nodes by selecting appropriate routes, and also prolong network lifetime by reducing the number of message transmissions. The routing challenges posed by the emergent scenario of SSN concerns the aspect that (i) executing a larger number of applications generates an enormous number of network transmissions and, consequently, increases energy consumption that potentially reduces the network lifetime; (ii) multi-application demands may impose additional burden on some set of nodes and jeopardize the energy usage balance in the network, consequently reducing the whole network lifetime while many powerful nodes remains alive, characterizing a network partition. Therefore, routing protocols for WSNs are potentially inefficient for SSNs since most of such protocols are designed based on one single application.

There are several techniques that routing algorithms could use to minimize energy consumption, such as data aggregation and in-network processing, clustering, different node role assignment, and data-centric methods [5]. Most of the routing algorithms use Information fusion [6] and Data aggregation [7] techniques. Such solutions aim at reducing the number of transmitted packets (network transmissions), and consequently reducing energy consumption. Traditionally, most of these fusion techniques are performed at the application level by analyzing the sensed measures / data. We believe that it is possible to achieve further benefits and leverage the energy efficiency of SSNs by providing a solution at the network (routing) level that acts in cooperation with application level strategies.

In this context, we propose THESEUS as a generic application-aware routing protocol for SSNs along with methods for reducing the number of message transmissions and balancing energy usage that are necessary to prolong the network lifetime. THESEUS was inspired by Proactive Routing with Coordination (PROC) protocol [8], incorporating all the strengths of the original proposal, extending and leveraging them for usage in a context of SSNs. THESEUS's contributions are: (i) it implements a technique for packet aggregation independent of the packet contents with the goal of prolonging the network lifetime; (ii) it updates the coordinator selection process to cover multi-application demands in order to better balance energy usage; (iii) it makes use of QoS parameters and applications' requirements to adapt the routing paths and (iv) THESEUS supports multiple sink nodes to deliver and receive data. The sink nodes can dynamically enter or leave the SSN.

II. RELATED WORK

Different routing protocols were proposed for WSNs, all of them designed based on single application demands (according to the original definition of WSNs). The environment of SSNs may comprise several different applications, each with potentially different requirements. A limited number of papers were found in the literature proposing routing protocols for SSNs [9] [10] [11], since sharing a WSN among multiple applications is a recent concept.

The adaptive multi-criteria routing (AMCR) [9] was proposed as a routing framework for SSNs, and one of its primary goals was to be a generic routing framework for multi-application demands. Eltarras and Eltoweissy [9] state that changes in the network underlying resources, connectivity, mission, or QoS requirements demand the design of adaptable SSN architectures and protocols to increase the network lifetime. THESEUS differs from AMCR in several aspects. First, AMCR is a conceptual routing framework, and so it does not present details of the specific algorithms used for achieving its main proposed features while THESEUS is a concrete system with a particular architecture and algorithms. Second, while AMCR uses its addressing scheme based on descriptive criteria for improving scalability, THESEUS relies on the characteristics of PROC for ensuring scalability as a distributed system. Third, THESEUS includes a packet aggregation algorithm, which helps improving network lifetime. Fourth, THESEUS has its own algorithm for selecting dynamic and QoS aware routes, while AMCR is said to be able of exploiting the message semantics and adapting itself. In addition, because of exploiting application semantics, the conception of AMCR is more tied to the application than THESEUS. In the development of THESEUS, we specified a well-defined interface between the application and network layers.

The context-aware protocol proposed in [10] was developed for supporting collaborative sensor network applications. In [10], all layers were combined and modeled as a single protocol stack (cross-layer development), to use the information of each layer in another one (what is said to result in context awareness). The primary goal was to promote load balancing in the network and the ability to use data aggregation. This type of integration brings a high level of energy efficiency but loses the generality and makes the protocol more complex for implementation. THESEUS shares the characteristics of this related work, but THESEUS differs from this work in its clear separation of layers, with well-defined interfaces for integration. THESEUS has been defined at the network layer with an explicit interface to the application layer just in the sink node, what makes THESEUS a generic and adaptable routing system for SSNs.

The aforementioned works were found more strictly related to THESEUS, considering that both exploit applications semantics for routing purposes. The work in [11] does not explicitly use such definition of applications (application level information) for routing purposes. The solution in [11] is based on prioritizing the applications and on pricing different paths so that each application will select the most appropriate route for its data. For instance, an application with low priority will try to find the cheapest possible route, but without considering the delay of this route; another application with high priority will attempt to find the fastest routes, but without considering route prices. In our solution, we divided applications into two categories for establishing priorities: applications based on event-driven data (high priority) and applications based on continuous dissemination data (normal priority). Two QoS parameters in the sink node define the maximum accepted delay time for each category of application data. At the end, it is worth to remind, none of these related research supports multiple sink nodes in SSNs.

III. PROPOSAL

The goal of this work is proposing a multi-application-aware routing system for SSNs, called THESEUS. THESEUS has an interface with applications to adapt the paths according to the demands of multiple applications and nodes' conditions. THESEUS works at the network layer level and has one interface to interact with the application layer (at the sink node), thus avoiding many cross-layer communications and making THESEUS a generic routing system. Such generality enables THESEUS to be implemented as a routing system for different applications in a SSN environment, without requiring any customization in its architecture and algorithms. THESEUS operates based on selecting a set of forwarding nodes (called *coordinators*) based on application demands. Coordinators create the backbone for routing; all other nodes (called *regular* nodes) will directly connect to one of the coordinators forming a treelike structure. Therefore, each node can be either a *coordinator* or a *regular node*. The sink node triggers the coordinator selection and route creation processes periodically, but these methods are performed on the nodes, in a distributed way (in-network processing). The reconstruction of the routes (backbone) happens in time intervals called cycles.

THESEUS is an application-aware routing system. This awareness is achieved by considering the application requirements for calculating a value that is used as the probability of a node becoming a coordinator. This calculation is performed by a function called *app function*. The app function is deployed on all nodes and is called every time the routes need to be created to calculate the mentioned probability value based on each node parameters. The routing establishment is based on the node role and the information about its neighbors. Coordinator nodes must select their respective parent nodes among their neighbors. In addition, THESEUS supports multiple sink nodes and both types of data flow commonly existing in WSNs, defined based on the communication frequency [8]: (i) EVENT-DRIVEN and (ii) CONTINUOUS. In the EVENT-DRIVEN data flow, data is transmitted whenever an event of interest occurs, and in CONTINUOUS data flow, data is sent periodically according to a time interval, which is predefined by the application [8]. THESEUS manages to support both types of data flow by using the respective priorities to each type of data flow based on the application QoS in terms of maximum tolerated delay.

A. *SSN Model*

The SSN is composed of sensor nodes and one or more sink nodes. We also considered that every node in the SSN is static. THESEUS does not depend on nodes position information for its operation. However, once the network is deployed, nodes must remain at the same positions during the whole execution of THESEUS. Moreover, sensor nodes are homogeneous in terms of processing units and memory capacities. However, each node can have different types of sensing units and different energy sources (energy levels). The sink node is connected to an unlimited energy source. In our SSN model, nodes could be deployed in any random position in all three dimensions.

B. *THESEUS Packet Aggregation*

THESEUS uses its aggregation algorithm that works at the network level, independent of the packet data content (see Fig. 1). Network packet formats have different fields and size based on the WSN platform. Generally, packet format has three main parts with a limited size for each one: packet header, data, and trailer. The main goal of our proposed aggregation algorithms is to maximize the packet data field usage. All data packets should be delivered to a sink node. Each packet, when received by a node, has some MAC layer information (such as packet CRC). Therefore, in case of aggregation, the source address (application level address, node ID), and the application data (sensed data) fields should be stored, and other fields could be excluded. When the aggregated packet is sent, radio component adds new MAC layer information for the aggregated packet. The use of two bytes for source address (application level address) could support 65535 nodes in a network. Moreover, most of the sensors measure environment data in maximum two bytes size [12]. Therefore, the minimum unused size of a packet data field should be 4 bytes (2 bytes of source address + 2 bytes of sensed data) to be able to aggregate another packet inside itself.

```
procedure Data-Manager ( )
1:   if MData.aggflag = false then // not possible to aggregate data
2:      send(MData, parent.address);
3:   else if MData.emptyspace <= 4 bytes then
4:      MData.aggflag = false;
5:      send(MData,parent.address);
6:   else
7:      THESEUS-packet-aggregation ( );
8:   end if
end procedure
procedure THESEUS-packet-aggregation ( )
9:   if MData.typeflag = 0 then      // for continuous data type
10:      if storedp.continuous = ∅ then
11:         storedp continuous = MData; start timer.C;
12:      end if
13:       while (timer < MDTC) & (stored.continuous.emptyspace > 4) do //
from Sync.QoSparam MDTC, maximum accepted delay
14:                  try add (MData.sourceaddress, MData.appdata) to
(storedp.continuous.appdata)
15:            catch error {send(MData, parent.address);};
16:      end while
17:      stop timer.C; timer.C = 0; stored.continuous.aggflag = false;
18:     Data-Manager (stored.continuous);
19:  end if
20:// same from line 9 to 19 for event driven data with related maximum
accepted delay (MDTE)
end procedure
```

Figure 1. THESEUS Packet aggregation algorithm

THESEUS packet aggregation operates based on one QoS parameter that has different values for each type of data flow application: first, the maximum accepted delay time for continuous data type, which should be the minimum accepted delay time through all applications QoS parameters; and second, the maximum accepted delay time for the event-driven data type as well. The coordinator checks the data field of the first received packet; if there is not enough empty space, then the aggregation flag bit changes to false (to inform the next coordinator to just forward this packet without further analysis) and this packet is forwarded through all coordinators to the sink node without any delay (Data-Manager Procedure, Fig. 1). Otherwise, the packet is stored in the node memory. The aggregation algorithm tries to add other received packets inside the stored packet during the maximum delay time defined by applications. After the specified time has passed, or if the maximum packet size is reached, then the aggregation flag bit changes to false and the packet will be forwarded to the sink node without any further delay. This process is the same for both data flow types, continuous dissemination, and event-driven, with the respective QoS parameters and variables (THESEUS-packet-aggregation procedure, Fig. 1).

C. *THESEUS Application Function*

In THESEUS it is possible to define different objective functions (called app functions) that calculate the probability (in percentage) of a node becoming a coordinator. This probability is calculated based on the node conditions, applications' demands and QoS requirements. Multi-application demands bring the necessity of combining all applications requirements, thus increasing the complexity of the app function. Many techniques, such as linear programming or AI could be used to define the app function for properly combining applications requirements.

In this work, the app function is defined according to the following steps: (i) specifying a set of rules based on the applications' requirements, (ii) defining equations for each rule, and finally (iii) creating the app function. We considered the balancing energy usage within all nodes as our main goal for our app function. The rules were defined heuristic-based considering the WSNs behavior. The set of rules of THESEUS R1, R2, R3 and R4 in (1), (2), (3) and (4) are generated. First, R_1 in (1) calculates the probability (in percentage) of a node being coordinator, considering if this node has been recently coordinator and/or if it has been coordinator for many cycles. In this case, we expect that these nodes should have lower probability of being coordinator for a period (a LEACH's adaptation [13]).

$$R_1 = 100 - \left[\left(coord(curCycle - 1) * 50 \right) + \left(\frac{coord.size}{curCycle} * 50 \right) \right] \quad (1)$$

where, *coord* (dynamic parameter) is an array of cycle ids during which the node was coordinator; *curCycle* (dynamic parameter) is the current cycle id and *coord.size* is the number of cycles during which the node was the coordinator (size of coord array).

Second, R_2 in (2) calculates the probability (in percentage) of a node being coordinator when it is in a dense area. In this case, we expect that nodes having more neighbors should have lower probability of being coordinator in order to avoid selecting more coordinators in denser areas. This rule, along with THESEUS packet aggregation algorithm, creates a synergy. Reducing the number of coordinators in a denser area (by rule R2) fosters the aggregation because fewer coordinators will have a higher chance to aggregate packets of more neighbors.

$$R_2 = 100 - \left(\frac{neighbors.size}{max.neighbor + 1} * 100\right) \qquad (2)$$

where *max.neighbor* (static parameter) is the maximum number of neighbors that a node can have. Parameter *max.neighbor* is calculated based on the amount of memory required for storing information from a neighbor. Therefore, it must be configured according to the adopted platform. The parameter *neighbors.size* (dynamic value) is the number of neighbors a node has.

Third, R_3 in (3) calculates the probability (in percentage) of a node being coordinator when it is near to the sink. In this case, we expect that nodes near to sink node have higher probability of being a coordinator in order to balance the high network traffic near to the sink among more nodes.

$$R_3 = 100/hops + 1 \qquad (3)$$

where *hops* (dynamic parameter) is the number of hops from the sink node (each node discovers this value based on the information of its parent, described in section III.E). Nodes directly connected to the sink node have zero hop distance.

Forth, R_4 in (4) calculates the probability (in percentage) of a node being coordinator when it is located in areas with higher rate of data generation. In almost all SSNs, continuous data dissemination from sensor nodes to sink contributes to the biggest part of data flow [14] [15]. Therefore, rule (R_4) states that the node that generates more data, i.e. the node that has more application requests, should have a higher possibility to become a coordinator. This rule is the same rationale used for R_3, contributes to sharing the network traffic among nodes, balancing the energy usage in the network. For this rule, it is necessary to calculate the sampling rate for each area and send these values to the nodes. The concept of *area* in the context of this work is used to denote the list of nodes required by an application (meaning the nodes deployed in the application target area).

$$R_4 = (SPpMTI(i)/mSPpMTI) * 100 \qquad (4)$$

where *SPpMTI* (dynamic parameter) is an array of the sample rates per maximum time interval for each area of the network; *mSPpMTI* (dynamic parameter) is the highest sample rate per maximum time interval; and *i* is the node area id.

Based on these rules, the app function and QoS parameters are defined by the following formulas:

$$Appparam\colon apx.networksize, SPpMTI, mSPpMTI \qquad (5)$$

$$Appfunc(coord, curCycle, neighbors, hops, i) = (R_1 + R_2 + R_3 + R_4)/4 \qquad (6)$$

where *Coord, curCycle, neighbors, hops* and *i* are input local parameters of the node, and are the same parameters used in (1) to (4). $R_1 \ldots R_4$ are the rules equations, which calculate the probability of a node being coordinator.

D. *THESEUS Architecture*

THESEUS logical architecture is organized into two parts: (i) THESEUS Sink architecture and (ii) THESEUS Node architecture. THESEUS Sink architecture (Fig. 2) is composed of the SSN Manager and Sink Manager (SM) components. The SSN Manager should gather all requirements from all applications and create a set of rules to build the app function and QoS parameters to send them to the network by the Sink Manager component.

THESEUS Node architecture, as shown in Fig. 3, includes *Node Manager* (NM), *Synchronization sub-system,*

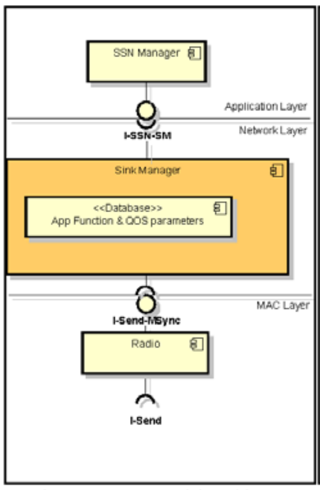

Figure 2. THESEUS Sink Architecture

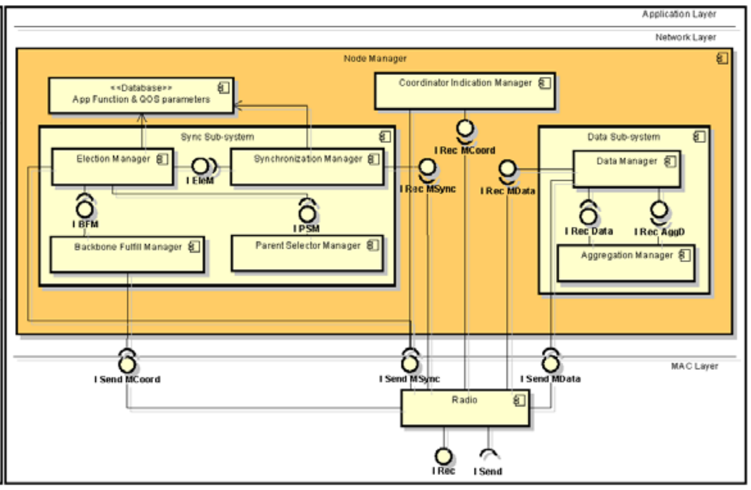

Figure 3. THESEUS Node Architecture

Data sub-system, Coordinator indication manager component, and App Function & QoS parameters database. NM is responsible for initializing the sensor node and forwarding received messages by the sensor node to the related sub-system. There are three different types of messages: (i) Sync message, (ii) Data message and (iii) Coord message, which are delivered to *Synchronization sub-system*, *Data sub-system* and *Coordinator indication manager* component respectively. *Synchronization sub-system* contains *Synchronization manager* component, *election manager, parent selector manager* and *backbone fulfill manager,* which are responsible for coordinator selection process and the route construction process. *Data sub-system* contains *data manager* and *Aggregation manager* components, which perform the data packet aggregation algorithm. *Coordinator indication manager* is responsible for forcing the node to be a coordinator when a Coord message is received.

E. *THESEUS Operation*

The operation sequence starts with the *route establishment process*. This process starts when a sink node is deployed by *SSN-Manager* and calls the *Sink-Manager* (Fig. 2) to broadcast the Sync message to the network, starting routes construction. This message updates node's neighbor list with current cycle ID, number of hops to the sink node, the node status (whether it is a coordinator or not), residual energy, app function and QoS parameters. Inside the network (nodes), the *Node-Manager procedure initializes the sensor node and* waits to receive a message (Fig. 4). When a Sync message is received, the *Node Manager* (NM) forwards it to the *synchronization manager,* which is a part of the *synchronization sub-system* (Fig. 3). *Sync-Manager Procedure* in Fig. 4 shows the running algorithm of *synchronization manager* (Fig. 3) and how a node selects its sink node, and the *Coord messages* and new *Sync messages* are generated.

The *synchronization manager* receives the Sync messages, checks the cycle id of the message to prevent the node from repeating the further steps. If the cycle id is same as the node current cycle id, the *synchronization manager* just completes the node neighbor list. If the cycle id is newer than the node current cycle id (each sink node sends its cycle id and valid time, all nodes save these parameters), it means a new cycle of route reconstruction started by a sink node. First, the node will update its cycle id, valid time (cycle time), and app and QoS parameters. Then, the node checks the sink id if it is the same sink id that nodes used in the last cycle; the node calls the Election Manager. If the sink id is not the same sink id the node already used, the node compares the number of hops distance towards the new sink with old sink. The node will select the new sink and call the Election Manager for the new sink if the new sink has fewer hops distance. Moreover, the nodes select new sink in case of expiration of old sink valid time (cycle time). This process makes THESEUS support adding and removing sink nodes dynamically.

Election Manager calls the app function with relevant local and QoS parameters in order to calculate the probability of the node being a coordinator. *Election-Manager procedure* in Fig. 4 shows the related algorithm. A simple random function along with the value of the app function decide if the node should be a coordinator or not. Next, a *Sync message* is broadcasted to inform the network about the updated node status. This rebroadcasting of Sync messages by each sensor node guarantees that all nodes were reached by the first Sync message broadcasted by the sink node, therefore ensuring the network connectivity. After receiving a *Sync message*, sensor nodes wait for a random time to receive information about the states of other nodes. This random time, named *Backofftime*, avoids the whole network from selecting more coordinators than necessary. The *SSN manager* should define

```
procedure Node-Manager ()
initialize parameters and variables;
Require: receive (MSync message);
    Sync-Manager (MSync);
Require: receive(MCoord message);
    Coordinator-Indication-Manager (MCoord);
Require: receive(MData message);
    Data-Manager (MData);
end procedure
procedure Sync-Manager ()
if SinkCycle < MSync.cycle then  //New cycle
    Neighbors.sink ← ∅;
    Neighbors ← Neighbors ∪ MSync.{hops, cycle, coord, energy};
    Update SinckCycle (cycle, validtime);
    Update (appparam, QoSparam);
    if selectedsink = MSync.sink then //New cycle already used
        Election-Manager ();
    else if MSync.hops < currenthops then //selcect another sink
        selectedsink = MSync.sink;
        Election-Manager ();
    else if selectedsink.validtime is expired then //select another sink
        Neighbors.sink ← ∅;
        selectedsink = MSync.sink;
        Election-Manager ();
    end if
else if SinkCycle = MSync.cycle then  //Same cycle
    Neighbors ← Neighbors ∪ MSync.{hops, cycle, coord, energy};
end if
end procedure
procedure Election-Manager ()
prob ← AppFunc(app parameters);
coordinator ← (random () < prob);
send(MSync(parent.hops+1,cycle,sinkid,coord,getEnergy(),
app&QoS.param),BROADCAST);
wait Backoff Time ();
parent ← Parent-Selector-Manager ();
Backbone-Fulfill-Manager ();
end procedure
procedure Parent-Selector-Manager ()
if coordinator = true then
    return min(Neighbors, {hops, coord, 1/energy});
else
    return min(Neighbors, {coord, hops, 1/energy});
end if
end procedure
procedure Backbone-Fulfill-Manager ()
if parent.coordinator = false then
    send(MCoord(parent.hops+1,cycle,sinkid,coord,getEnergy(),
app&QoS.param), parent);
    Neighbors.parent .coord ← true;
end if
end procedure
procedure Coordinator-Indication-Manager ();
Neighbors ← Neighbors ∪MSync.{hops, cycle, coord, energy};
coordinator ← true;
send(MSync(parent.hops+1,cycle,sinkid,coord,getEnergy(),
app&QoS.param),BROADCAST);
end procedure
```

Figure 4. THESEUS node algorithm

the duration of this time based on QoS requirements and the network conditions (such as network size) before starting the network. After the backofftime has elapsed, the *parent selector manager* selects the parent within the updated neighbor list (*Parent-Selector-Manager procedure* in Fig. 4 show the related algorithm). At this point, *Backbone Fulfill Manager* checks the status of the selected parent. As mentioned, all nodes should connect to a coordinator. Therefore, if the selected parent is not a coordinator, a *Coord message* will be sent to the selected parent, and the node updates the parent status to the coordinator in its neighbor list. As shown in Fig. 4, *Backbone-Fulfill-Manager procedure* runs in the *Backbone Fulfill Manager* component and they represent the sending of *Coord messages*.

The parent selector manager (*Parent-Selector-Manager procedure* in Fig. 4) selects, within the neighbor list, the best node to be the parent of a node. The parent of a regular node is selected based on the following priorities, in descending order: (1) nodes located at the shortest hop distance to the sink, (2) nodes holding a coordinator role, and (3) nodes with the greater available energy within their neighbors. The parent of a coordinator node is selected based on the following priorities, in descending order: (1) nodes holding a coordinator role, (2) nodes at a shortest hop distance to the sink, and (3) nodes with the greater available energy within their neighbors. Such prioritizing is used to minimize the number of coordinators while finding the shortest paths from each sensor node towards the sink node. Using such prioritizing scheme, along with choosing to start routes' creation from the nodes inside the network are features that avoid cycling. Therefore, due to its features, THESEUS does not require any additional mechanism for cycle detection. In addition, based on checking the valid time for the sink nodes in the route selection process of nodes, THESEUS supports adding and removing of more sink nodes dynamically.

The *Coordinator-Indication-Manager procedure* (shown in Fig. 4) runs in the *Coordinator Indication Manager* and configures the node to be a coordinator and broadcast a Sync message to inform other nodes about the new status of the node. The *data dissemination process* starts when the sensor nodes begin collecting data from the environment and sending the sensed data to their selected coordinators using Data messages. Within each sensor node, the NM will forward the received Data messages to the *data manager* component, in the Data Sub-system. In the Data Sub-system of the coordinator node, specifically within the *Aggregation Manager* component, our proposed packet aggregation algorithm takes place. After the aggregated packet is ready, it is forwarded through the tree-like structure until reaching the sink node.

Sync messages are broadcasted periodically and/or eventually for each cycle, starting routes reconstruction. Such periodic reconstruction of routes makes THESEUS fault tolerant in case of node failures and helps balancing the energy usage within all nodes. Furthermore, if any new application demand comes up in the network, which changes the app function parameters or QoS parameters, then the nodes will be updated immediately. It is also important to mention that the network connectivity is guaranteed in every cycle, as well as in the first cycle, by rebroadcasting Sync messages by each node.

IV. EXPERIMENTS

A. GQM

Considering the objectives of this work and following the Goal, Question, Metric (GQM) methodology [16], the following goals were defined: (i) evaluate THESEUS effectiveness regarding SSNs lifetime; (ii) analysis of the pros and cons of using THESEUS packet aggregation algorithm. These goals can be expressed by five questions. Q1, Q2, and Q3 are related to the first goal, and Q4 and Q5 are related to the second one: Q1: How much THESEUS improves the energy usage of the nodes, in relation to PROC? Q2: How much THESEUS improves the energy usage balance through nodes, in relation to PROC? Q3: How much overhead, in terms of memory and processing, THESEUS imposes to the SSN in absolute terms, and in relation to PROC? Q4: What are the advantage(s) of using the packets aggregation technique? Q5: What are the drawback(s) of using the packets aggregation technique?

Metrics were also defined to support the answers to these questions. Regarding Q1, we defined the **Energy Average** (Nrj. Avg.) metric. The **Energy** is defined as the simple average of the energy consumption values of all nodes during each simulation. In relation to Q2, we defined the **Energy Population Standard Deviation** (Nrj. PSTD) **metric.** This statistical metric shows the deviation (from average) of the energy consumption of all nodes. In this case, all the sensor nodes of the SSN form the statistical population and the Nrj. PSTD is the standard deviation of such population. In other words, the more this value approaches zero, the better the energy consumption balance is among nodes. In relation to Q3, we defined two metrics: **Memory usage** (Mem.) and **CPU Active. Memory usage** is defined as the number of bytes of RAM used by the protocol. **CPU Active** is defined as the average of the percentage of time that the CPU of a node remains active. In relation to Q4, we defined **Network traffic rate per samples** (Net. Rate) metric. This metric is defined as the average of packets sent to deliver samples to the sink node inside the network. It is calculated by dividing the number of sent data packets per number of samples generated during the simulations. Finally, In relation to Q5, we defined **Sample Loss** metric as the percentage of lost samples, calculated by (7):

$$Sample\ loss = ((B - A)/B) * 100 \qquad (7)$$

where A is the number of received samples in the sink node and B is the number of samples generated by all nodes. Using packet aggregation, the number of network transmissions is reduced and consequently the chance of losing packet gets lower but on the other hand, each packet could contain more than one sample and could make higher sample loss.

B. Implementation

THESEUS and PROC were implemented on TinyOS 2.1.0 [17], using the nesC programming language (an extension of the C language), which adopts an event-driven programming model [17]. TinyOS is a component-based operating system, designed specifically for the development of applications for WSNs. We used MICAz platform [12] (4kB RAM, 128 kB flash memory for program storage) for our implementation. The implemented codes followed THESEUS architecture and algorithms previously described. It is worth mentioning that fifteen real MICAz sensor nodes were used to verify and debug the implementation functionality of THESEUS in the Ubiquitous Computing Laboratory of PPGI-UFRJ. In addition, the results of similar scenario tests by AVRORA and real nodes showed simulator results are well promising.

C. *Evaluation Methodology and Scenarios*

In this work, we compared THESEUS and PROC using the AVRORA 1.7.117 simulator [18], making similar conditions for fair comparison. AVRORA uses implementation code of real platforms for the simulation process. In order to evaluate energy consumption, we used the energy model described in [19] since it is a well-defined and generic energy model for wireless sensor nodes. In our tests, we fixed some parameter values: **(i) Cycle time:** In [8], the authors mentioned that they used **180** seconds for cycle time based on empirical observations of PROC, so the same value is used in our evaluation. **(ii) Maximum neighbor list size:** considering the limited memory of sensor nodes, this list should be restricted, and we used the same value used in PROC**- 20. (iii) Minimum and maximum of Backofftime:** if these parameters are set so that they form a short period, then nodes cannot receive enough information from neighbors, while a long period results in a longer time required for routes formation. Based on the results obtained in preliminary phase of calibration tests, we selected **120ms** and **512ms** for these parameters in our evaluation. **(iv) Packet Acknowledgments:** we used acknowledgments for unicast packets.

In our evaluation, we varied the number of nodes in **50, 100, 150 and 200 nodes**, and in all of them, the first node (in the left bottom corner) was configured as the sink node. In each topology, we placed the nodes on a **square grid** with a distance of **5 meters between nodes** in each horizontal and vertical direction in the Cartesian plan. The 5 meters distance was chosen based on the MICAz datasheet [12]. To simulate SSNs conditions, we decided to use two applications based on continuous data flow (since they are more demanding in terms of data flow). **The first application runs on all nodes (both area 1 and area 2) with sample rate of 28 seconds** and **the second application runs in first half of the network nodes (area 1) each 14 seconds**. Tests showed that THESEUS improves the network lifetime (also other metrics) more than PROC when the number of running applications in the network increases. **Maximum accepted delay time for continuous dissemination samples (MDTC)** should not be bigger than minimum sample rate time and should be long enough for the nodes to be able to aggregate packets. In our evaluation, we considered **MDTC as 10 seconds**. Tests showed that even for the case of 1 second, there is no significant change in the aggregation rate i.e. the average number of samples per delivered data messages on the sink node.

D. *Tests and Analysis of Results*

We made eight experiments (with 50, 100, 150 and 200 nodes for both PROC and THESEUS) during five hours. TABLE 1 shows the results of each metric and its improvement percentage in each scenario. **Energy Average** of THESEUS shows 10.70% to 20.83% improvement regarding PROC in different topologies. In addition, it shows more improvement in a bigger network and longer time. Packet size has a significant impact on energy usage in radio component. In relation to PROC, THESEUS has more application parameters and aggregated data packets. THESEUS shows a considerable improvement even with the larger size of packets (aggregated packets), because it generates a smaller number of packets. **Energy Population Standard Deviation** improved 34.25% in average of all scenarios. The improvement of this metric is related to number of cycles in tests because more cycles of reconstruction of routes mean changes in the node that is a coordinator. **Memory usage** was retrieved from TinyOS at compile time for real nodes. In this metric, THESEUS used 757 bytes of memory, which corresponds to 18.48% of node memory in MICAz platform (4096 bytes). In relation to PROC, THESEUS uses 195 bytes more memory (-4.76%), representing less free memory as an overhead. AVRORA provides the **CPU active time** values in simulation [18]. It was expected an overhead in case of this metric, based on the greater complexity of THESEUS, mainly driven by the aggregation algorithm. However, the results show that the impact of sending more packets on CPU usage is greater than the effect caused by the packet aggregation algorithm of THESEUS. THESEUS improved this metric in average of tests 24.22% in relation to PROC. So, THESEUS proved to be better than PROC regarding this metric. **Network traffic rate per samples** showed an improvement between 19.57% and 32.33%, getting better as the network size increases. It is important to mention that less traffic in the network brings two main benefits: less packet loss and less energy usage. **Sample Loss:** In the average of

TABLE 1. Evaluation of Results using GQM

		Q 1	Q 2	Q 3	Q 4	Q 5
Scenario	**Protocol**	Nrj. Avg.	Nrj. PSTD	CPU Active	Net. Rate	Sample loss
50 nodes 5 hours	THESEUS	0.0567	0.0199	0.2537	2.42	2.00%
	PROC	0.0634	0.0294	0.2869	3.01	2.66%
	Improvement	**10.70%**	**32.33%**	**11.58%**	**19.57%**	**24.88%**
100 nodes 5 hours	THESEUS	0.0703	0.0300	0.3584	2.94	6.46%
	PROC	0.0860	0.0506	0.4604	4.14	13.54%
	Improvement	**18.21%**	**40.76%**	**22.16%**	**28.86%**	**52.29%**
150 nodes 5 hours	THESEUS	0.0817	0.0456	0.4348	3.44	9.11%
	PROC	0.1010	0.0640	0.5825	4.89	25.43%
	Improvement	**19.11%**	**28.74%**	**25.35%**	**29.59%**	**64.17%**
200 nodes 5 hours	THESEUS	0.0857	0.0495	1.0713	3.57	10.83%
	PROC	0.1082	0.0764	1.7224	5.28	29.95%
	Improvement	**20.83%**	**35.18%**	**37.80%**	**32.33%**	**63.83%**
Average of improvement		**17.21%**	**34.25%**	**24.22%**	**27.59%**	**51.29%**

all simulations, THESEUS improved sample loss by 51.29% in comparison to PROC. This happens because the aggregation algorithm of THESEUS helps reducing the amount of network traffic, so it reduces packet loss. Therefore, as the network size increases, this metric shows better results. It is worth mentioning that in our THESEUS implementation, a packet loss could cause six samples losses. However, even with this restriction, the packet losses in THESEUS were reduced, and sample losses result improved.

Based on the analysis of metrics, and on the improvement percentages shown, we could answer the aforementioned questions. **Q1**: THESEUS improved the energy usage of the nodes on an average of all tests more than 17%. **Q2**: THESEUS improved the energy usage balance in the SSN in average 34.25%. **Q3**: The overhead of using THESEUS instead of PROC, in terms of memory, is 4.76%, but for processing THESEUS improved more than 22% in average. **Q4**: About advantage(s) of using the packet aggregation technique, THESEUS reduces network traffic 27.59% in average of all tests. **Q5**: Regarding drawback(s) of using the packet aggregation technique, using larger packet size could be a drawback. Nonetheless, the evaluation showed that, since THESEUS reduces network traffic, such potential drawback was transformed into an advantage. THESEUS improved the sample loss metric 51.29% in average of all tests. These results prove that we achieved our goal of improving the SSNs lifetime and confirming the advantages of using THESEUS packet aggregation algorithm.

V. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, THESEUS is the first applicable routing protocol for SSNs with support of dynamic multiple Sink nodes. Our experiments showed tangible improvements of THESEUS, in relation to PROC, when operating in a shared sensor network scenario. Furthermore, our variation and real tests proved the strength of THESEUS. For future work, we intend to support time synchronization among nodes in order to make THESEUS more energy efficient.

ACKNOWLEDGMENT

This work was partially supported by Brazilian Funding Agencies FAPERJ, CNPq, CAPES and CENPES. Flávia C. Delicato, Paulo F. Pires and Luci Pirmez are CNPq fellows.

REFERENCES

[1] Flávia C. Delicato, and Paulo F. Pires, "Energy Awareness and Efficiency in Wireless Sensor Networks: From Physical Devices to the Communication Link," *Energy-Efficient Distributed Computing Systems,* pp. 673-707, 2012.

[2] Leontiadis, Ilias, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft, "SenShare: transforming sensor networks into multi-application sensing infrastructures," *In Wireless Sensor Networks,* pp. 65-81. Springer Berlin Heidelberg, 2012.

[3] Akkaya, Kemal, and Mohamed Younis, "A survey on routing protocols for wireless sensor networks," *Ad hoc networks 3, no. 3,* pp. 325-349, 2005.

[4] Dietrich, Isabel, and Falko Dressler, "On the lifetime of wireless sensor networks," *ACM Transactions on Sensor Networks (TOSN) 5, no. 1,* 2009.

[5] Al-Karaki, Jamal N., and Ahmed E. Kamal., "Routing techniques in wireless sensor networks: a survey," *Wireless communications, IEEE 11, no. 6,* pp. 6-28, 2004.

[6] Nakamura, Eduardo F., Antonio AF Loureiro, and Alejandro C. Frery, "Information fusion for wireless sensor networks: Methods, models, and classifications," *ACM Computing Surveys (CSUR) 39, no. 3,* 2007.

[7] Rajagopalan, Ramesh, and Pramod K. Varshney, "Data aggregation techniques in sensor networks: A survey," 2006.

[8] Macedo, Daniel F., Luiz HA Correia, Aldri L. dos Santos, Antonio AF Loureiro, and José Marcos S. Nogueira, "A rule-based adaptive routing protocol for continuous data dissemination in WSNs," *Journal of Parallel and Distributed Computing 66, no. 4,* pp. 542-555, 2006.

[9] Eltarras, Ramy, and Mohamed Eltoweissy, "Adaptive Multi-Criteria Routing for Shared Sensor-Actuator Networks," *In Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE,* pp. 1-6, 2010.

[10] Hefeida, Mohamed, Turkmen Canli, A. Kshemkalyani, and A. Khokhar, "Context modeling in collaborative sensor network applications," *In Collaboration Technologies and Systems (CTS), 2011 International Conference, IEEE,* pp. 274-279, 2011.

[11] Shah, Syed Yousaf, and Boleslaw K. Szymanski, "Dynamic Multipath Routing of Multi-Priority Traffic in Wireless Sensor Networks," *In Proc. 6th Annual Conference of International Technology Alliance, Southampton, UK,* 2012.

[12] MICA2 & MICAZ, "Online," *http://www.memsic.com/wireless-sensor-networks,* Accessed August 2015.

[13] Heinzelman, Wendi Rabiner, Anantha Chandrakasan, and Hari Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," *In System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference, IEEE,* p. 10, 2000.

[14] Kulkarni, Sandeep S, "TDMA service for sensor networks," *In Distributed Computing Systems Workshops, 24th International Conference, IEEE,* pp. 604-609, 2004.

[15] Demirkol, Ilker, Cem Ersoy, and Fatih Alagoz, "MAC protocols for wireless sensor networks: a survey," *Communications Magazine, IEEE 44, no. 4,* pp. 115-121, 2006.

[16] V. Basili, G. Caldiera, and H. Rombach, "The goal question metric approach," *Encyclopedia of software Engineering, vol. 2,* pp. 528-532, 1994.

[17] TinyOS, "Online," *http://www.tinyos.net,* Accessed August 2015.

[18] AVRORA Simulator, "Online," *http://compilers.cs.ucla.edu/avrora,* Accessed August 2015.

[19] Wei Li, Flávia C. Delicato, Paulo F. Pires, Young Choon Lee, Albert Y. Zomaya, Claudio Miceli, and Luci Pirmez, "Efficient allocation of resources in multiple heterogeneous Wireless Sensor Networks," *Journal of Parallel and Distributed Computing 74, no. 1,* pp. 1775-1788, 2014.

APPENDIX B, THE IMPLEMENTED CODES FOR TINYOS:

I) THESEUS SINK MANAGER APPLICATION

1) Makefile

```
COMPONENT=THESEUS_SinkManagerAppC
include $(MAKERULES)
```

2) message.h

```
#ifndef THESEUS_H
#define THESEUS_H
typedef nx_struct MSync {
  nx_uint16_t Node_ID;
  nx_uint8_t hops;
  nx_uint16_t cycle;
  nx_uint8_t coord;
  nx_uint16_t energy;
  nx_uint16_t appparam_MDTC;
  nx_uint16_t appparam_MDTE;
  nx_uint8_t appparam_a;
  nx_uint8_t appparam_b;
  nx_uint8_t appparam_c;
  nx_uint8_t appparam_d;
  nx_uint8_t SinkID;
  nx_uint8_t appparam_SPpMTI_1;
  nx_uint8_t appparam_SPpMTI_2;
  nx_uint8_t appparam_SPpMTI_3;
  nx_uint8_t appparam_SPpMTI_4;
  nx_uint8_t appparam_mSPpMTI;
  nx_uint16_t Parent_ID;
  nx_uint16_t Valid_Time;
} MSync_t;
typedef nx_struct MData0 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
} MData0_t;
typedef nx_struct MData1 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
} MData1_t;
typedef nx_struct MData2 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
} MData2_t;
typedef nx_struct MData3 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
  nx_uint16_t app_data6;
  nx_uint16_t app_data7;
} MData3_t;
```

```
typedef nx_struct MData4 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
  nx_uint16_t app_data6;
  nx_uint16_t app_data7;
  nx_uint16_t app_data8;
  nx_uint16_t app_data9;
} MData4_t;
typedef nx_struct MData5 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
  nx_uint16_t app_data6;
  nx_uint16_t app_data7;
  nx_uint16_t app_data8;
  nx_uint16_t app_data9;
  nx_uint16_t app_data10;
  nx_uint16_t app_data11;
} MData5_t;
enum {
  AM_MSYNC = 1,AM_MDATA = 3,
};
#endif
```

3) THESEUS_SinkManagerAppC.nc

```
#include "messages.h"
configuration THESEUS_SinkManagerAppC {}
implementation {
  components MainC, THESEUS_SinkManagerC as App;
  components new TimerMilliC();
  components new TimerMilliC() as TIMER1;
  components ActiveMessageC;
  components new AMSenderC(AM_MSYNC) as AM_MSync_S;
  components new AMReceiverC(AM_MDATA) as AM_MData_R;
  App.Boot -> MainC.Boot;
  App.MSync_S -> AM_MSync_S.AMSend;
  App.MSync_P -> AM_MSync_S.Packet;
  App.MData_R -> AM_MData_R.Receive;
  App.AMControl -> ActiveMessageC;
  App.MilliTimer -> TimerMilliC;
  App.MilliTimerstart -> TIMER1;
}
```

4) THESEUS_SinkManagerC.nc

```
#include "Timer.h"
#include "messages.h"
#define CycleTime 0xF000 // cycle timer set to 61440 milliseconds (60 seconds)
#define         MCycle 0x00000003 //Cycle time in minutes, it will make new cycle every 3 times of CycleTime fires
// Values of app parameter should be set here:
#define f_appparam_MDTC 0x2800 // 10 seconds
#define f_appparam_MDTE 0x0400 // 1 second
#define f_appparam_a 0x0001
#define f_appparam_b 0x0001
#define f_appparam_c 0x0001
#define f_appparam_d 0x0001
#define f_appparam_SPpMTI_1 0x0002
#define f_appparam_SPpMTI_2 0x0001
#define f_appparam_SPpMTI_3 0x0001
#define f_appparam_SPpMTI_4 0x0001
```

```
#define f_appparam_mSPpMTI 0x0002

module THESEUS_SinkManagerC {
  uses {
    interface Boot;
    interface Timer<TMilli> as MilliTimer;
        interface Timer<TMilli> as MilliTimerstart;
    interface SplitControl as AMControl;
    interface Packet;
        interface AMSend as MSync_S;
        interface Packet as MSync_P;
        interface Receive as MData_R;
        interface Packet as MData_P;
  }
}
implementation {
  message_t packet;
  uint8_t cyclecounter = 0x0000;
  bool locked;
  uint16_t nextCycle = 0x0001;
  uint16_t flags = 0;
  uint16_t source_add;
  uint16_t destination_add = 0;
  uint16_t app_data1 = 0;
  uint16_t app_data2 = 0;
  uint16_t app_data3 = 0;
  uint16_t app_data4 = 0;
  uint16_t app_data5 = 0;
  uint16_t app_data6 = 0;
  uint16_t app_data7 = 0;
  uint16_t app_data8 = 0;
  uint16_t app_data9 = 0;
  uint16_t app_data10 = 0;
  uint16_t app_data11 = 0;
  uint16_t appparam_MDTC;
  uint16_t appparam_MDTE;
  uint8_t appparam_a;
  uint8_t appparam_b;
  uint8_t appparam_c;
  uint8_t appparam_d;
  uint8_t SinkID;
  uint8_t appparam_SPpMTI_1;
  uint8_t appparam_SPpMTI_2;
  uint8_t appparam_SPpMTI_3;
  uint8_t appparam_SPpMTI_4;
  uint8_t appparam_mSPpMTI;

  task void MSync_broad_task(){ //Task of broad-casting MSync message
        if (!locked) {
                        MSync_t* rcm = (MSync_t*)call MSync_P.getPayload(&packet, sizeof(MSync_t));
                        if (rcm == NULL) {return;}
                        rcm->Node_ID = TOS_NODE_ID;
                        rcm->cycle = nextCycle;
                        rcm->hops = 0;
                        rcm->coord = 0x0001;
                        rcm->energy = 0xFFFF; //energy of sink node is always full
                        rcm->appparam_MDTC = f_appparam_MDTC;
                        rcm->appparam_MDTE = f_appparam_MDTE;
                        rcm->appparam_a = f_appparam_a;
                        rcm->appparam_b = f_appparam_b;
                        rcm->appparam_c = f_appparam_c;
                        rcm->appparam_d = f_appparam_d;
                        rcm->SinkID = TOS_NODE_ID;
                        rcm->appparam_SPpMTI_1 = f_appparam_SPpMTI_1;
                        rcm->appparam_SPpMTI_2 = f_appparam_SPpMTI_2;
                        rcm->appparam_SPpMTI_3 = f_appparam_SPpMTI_3;
                        rcm->appparam_SPpMTI_4 = f_appparam_SPpMTI_4;
                        rcm->appparam_mSPpMTI = f_appparam_mSPpMTI;
                        rcm->Parent_ID = 0xFFFF; // sink does not have parent
                        rcm->Valid_Time = MCycle; //cycle time in minutes
                        if (call MSync_S.send(AM_BROADCAST_ADDR, &packet, sizeof(MSync_t)) == SUCCESS) {
                                locked = TRUE;
         }
        }
        else{post MSync_broad_task();}
    }

    event void Boot.booted() {
      call AMControl.start();
    }

    event void AMControl.startDone(error_t err) {//Sink Node start to work and startPeriodic(CycleTime)
      if (err == SUCCESS) {
                    call MilliTimer.startPeriodic(CycleTime);
      }
      else {
        call AMControl.start();
      }
    }

    event void AMControl.stopDone(error_t err) {//Sink Node doesn't start to work
      // do nothing
    }

    event void MilliTimer.fired() { // For each cycle broad-cast new MSync message
      cyclecounter++;
          if (cyclecounter == MCycle){
                    cyclecounter = 0;
                    post MSync_broad_task();
          }
    }

   event message_t* MData_R.receive(message_t* bufPtrD, //Receive data
                                        void* payload, uint8_t len) {
          //application should decide to what to do with data
      return bufPtrD;
    }

    event void MSync_S.sendDone(message_t* bufPtrS, error_t error) { //call new cycle
      if (&packet == bufPtrS) {
        locked = FALSE;
           nextCycle++;
      }
    }
  }
```

II) THESEUS NODE MANAGER APPLICATION

1) Makefile

```
COMPONENT=THESEUS_NodeManagerAppC
include $(MAKERULES)
```

2) message.h

```
#ifndef THESEUS_H
#define THESEUS_H
typedef nx_struct MSync {
  nx_uint16_t Node_ID;
  nx_uint8_t hops;
  nx_uint16_t cycle;
  nx_uint8_t coord;
  nx_uint16_t energy;
  nx_uint16_t appparam_MDTC;
  nx_uint16_t appparam_MDTE;
  nx_uint8_t appparam_a;
  nx_uint8_t appparam_b;
  nx_uint8_t appparam_c;
  nx_uint8_t appparam_d;
  nx_uint8_t SinkID;
  nx_uint8_t appparam_SPpMTI_1;
  nx_uint8_t appparam_SPpMTI_2;
  nx_uint8_t appparam_SPpMTI_3;
  nx_uint8_t appparam_SPpMTI_4;
  nx_uint8_t appparam_mSPpMTI;
  nx_uint16_t Parent_ID;
  nx_uint16_t Valid_Time;
} MSync_t;
typedef nx_struct MCoord {
  nx_uint16_t Node_ID;
  nx_uint8_t hops;
  nx_uint16_t cycle;
  nx_uint8_t coord;
  nx_uint16_t energy;
  nx_uint16_t appparam_MDTC;
  nx_uint16_t appparam_MDTE;
  nx_uint8_t appparam_a;
  nx_uint8_t appparam_b;
  nx_uint8_t appparam_c;
  nx_uint8_t appparam_d;
  nx_uint8_t SinkID;
  nx_uint8_t appparam_SPpMTI_1;
  nx_uint8_t appparam_SPpMTI_2;
  nx_uint8_t appparam_SPpMTI_3;
  nx_uint8_t appparam_SPpMTI_4;
  nx_uint8_t appparam_mSPpMTI;
  nx_uint16_t Parent_ID;
  nx_uint16_t Valid_Time;
} MCoord_t;
typedef nx_struct MData0 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
} MData0_t;
typedef nx_struct MData1 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
} MData1_t;
typedef nx_struct MData2 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
} MData2_t;
typedef nx_struct MData3 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
  nx_uint16_t app_data6;
  nx_uint16_t app_data7;
} MData3_t;
typedef nx_struct MData4 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
  nx_uint16_t app_data6;
  nx_uint16_t app_data7;
  nx_uint16_t app_data8;
  nx_uint16_t app_data9;
```

```
} MData4_t;
typedef nx_struct MData5 {
  nx_uint16_t flags;
  nx_uint16_t source_add;
  nx_uint16_t destination_add;
  nx_uint16_t app_data1;
  nx_uint16_t app_data2;
  nx_uint16_t app_data3;
  nx_uint16_t app_data4;
  nx_uint16_t app_data5;
  nx_uint16_t app_data6;
  nx_uint16_t app_data7;
  nx_uint16_t app_data8;
  nx_uint16_t app_data9;
  nx_uint16_t app_data10;
  nx_uint16_t app_data11;
} MData5_t;
typedef struct {
  uint16_t node_id;
  uint8_t hops;
  uint16_t cycle;
  uint8_t sink_id;
  bool coord;
  uint16_t energy;
} Neighbors_t;
typedef struct {
  uint16_t scycle;
  uint16_t valid_time;
} SinkCycle_t;
enum {
  AM_MSYNC = 1,AM_MCOORD = 2,AM_MDATA = 3,AM_AVRORA = 4,
};
#endif
```

3) THESEUS_NodeManagerAppC.nc

```
#include "messages.h"
configuration THESEUS_NodeManagerAppC {}
implementation {
  components MainC, THESEUS_NodeManagerC as App;
  components new TimerMilliC() as TIMER0;
  components new TimerMilliC() as TIMER1;
  components new TimerMilliC() as TIMER2;
  components new TimerMilliC() as TIMER3;
  components new TimerMilliC() as TIMER4;
  components new TimerMilliC() as TIMER5;
  components ActiveMessageC;
  components new AMSenderC(AM_MSYNC) as AM_MSync_S;
  components new AMReceiverC(AM_MSYNC) as AM_MSync_R;
  components new AMSenderC(AM_MCOORD) as AM_MCoord_S;
  components new AMReceiverC(AM_MCOORD) as AM_MCoord_R;
  components new AMSenderC(AM_MDATA) as AM_MData_S;
  components new AMSenderC(AM_MDATA) as AM_MDataF_S;
  components new AMReceiverC(AM_MDATA) as AM_MDataF_R;
  components new AMSenderC(AM_AVRORA) as AM_Avrora_S;
  components RandomC;
  components new VoltageC() as Battery;
  App.Boot -> MainC.Boot;
  App.MSync_R -> AM_MSync_R.Receive;
  App.MSync_S -> AM_MSync_S.AMSend;
  App.MSync_P -> AM_MSync_S.Packet;
  App.MCoord_R -> AM_MCoord_R.Receive;
  App.MCoord_S -> AM_MCoord_S.AMSend;
  App.MCoord_P -> AM_MCoord_S.Packet;
  App.PacketAcknowledgements -> AM_MCoord_S;
  App.MData_S -> AM_MData_S.AMSend;
  App.MData_P -> AM_MData_S.Packet;
  App.PacketAcknowledgements -> AM_MData_S;
  App.MDataF_R -> AM_MDataF_R.Receive;
  App.MDataF_S -> AM_MDataF_S.AMSend;
  App.MDataF_P -> AM_MDataF_S.Packet;
  App.PacketAcknowledgements -> AM_MDataF_S;
  App.AMControl -> ActiveMessageC;
```

```
 App.MilliTimer -> TIMER0;
 App.MilliTimerApp -> TIMER1;
 App.MilliTimerBuf -> TIMER2;
 App.MilliTimerBufE -> TIMER3;
 App.MilliTimerACK -> TIMER4;
 App.MilliTimerValidTime -> TIMER5;
 App.Random -> RandomC.Random;
 App.Battery -> Battery;
}
```

4) THESEUS_NodeManagerAppC.nc

```
#include "Timer.h"
#include "messages.h"
#define NeighborsLen 0x0014 //Maximum number of neighbours on the list is 20
#define SinknodesLen 0x000A //Maximum number of sink nodes on the list is 10
#define MAX_RANDOM_THRESHOLD 0x0200 //Maximum backoff time 512ms
#define MIN_RANDOM_THRESHOLD 0x0078 //Minimum backoff time 120ms
//Set bits for changing flags
#define B01_16 0x8000
#define B02_16 0x4000

module THESEUS_NodeManagerC { //interfaces
 uses {
  interface Boot;
      interface Timer<TMilli> as MilliTimer;
      interface Timer<TMilli> as MilliTimerApp;
      interface Timer<TMilli> as MilliTimerBuf;
      interface Timer<TMilli> as MilliTimerBufE;
      interface Timer<TMilli> as MilliTimerACK;
      interface Timer<TMilli> as MilliTimerValidTime;
  interface SplitControl as AMControl;
      interface Receive as MSync_R;
      interface AMSend as MSync_S;
      interface Packet as MSync_P;
      interface Receive as MCoord_R;
      interface AMSend as MCoord_S;
      interface Packet as MCoord_P;
      interface AMSend as MData_S;
      interface Packet as MData_P;
      interface Receive as MDataF_R;
      interface AMSend as MDataF_S;
      interface Packet as MDataF_P;
      interface PacketAcknowledgements;
      interface Random;
      interface Read<uint16_t> as Battery;
 }
}

implementation {
 //start: defining variables
 //Messages variables
 message_t packet;
 bool locked;
 //The nodes choose areas based on ID
 uint8_t node_area = 0x0000;
 uint8_t node_area1_start = 0x0001; //Node_ID 1 to 49 is area 1
 uint8_t node_area1_end = 0x0031;
 uint8_t node_area2_start = 0x0032; //Node_ID 50 to 200 is area 2
 uint8_t node_area2_end = 0x00C8;
 uint16_t node_area3_start = 0x00C9; //Node_ID 201 to 300 is area 3
 uint16_t node_area3_end = 0x012C;
 uint16_t node_area4_start = 0x012D; //Node_ID 301 to 400 is area 4
 uint16_t node_area4_end = 0x0190;
 //ACK variables
 int trycoord = 0;
 int trydata = 0;
 int trydataF = 0;
 uint8_t rep_mode = 0;
 //battery variables
 uint16_t counter_send = 0;
 //Arrays
 SinkCycle_t SinkCycle[SinknodesLen];//Array of SinksCycles
```

```
Neighbors_t neighbors[NeighborsLen]; //Array of neighbours
int ID = -1;
uint8_t i;
//process control variables
uint16_t Valid_Time;
bool coord = FALSE;
bool cont_sync_task = FALSE;
int parent_add = -2;
int SinkID = -1;
uint8_t parent_hops;
bool parent_coord;
bool find;
float randBackoffPeriod=0.0F;
int Sinks_n;
//App function variables
float F1;
float F2;
float F3;
float F4;
float prob;
float tmp_rnd;
uint8_t Neighbors_n = 0; //0 to NeighborsLen, count the number of neighbours in each cycle
uint16_t AppCycleTime = 0x3800; // 14 seconds
uint16_t currEnergy = 0x2800; // first cycle will not read energy, this is default startup energy just for first cycle
uint8_t Appcounter = 0x0000;
uint8_t appparam_a;
uint8_t appparam_b;
uint8_t appparam_c;
uint8_t appparam_d;
uint8_t appparam_SPpMTI_1;
uint8_t appparam_SPpMTI_2;
uint8_t appparam_SPpMTI_3;
uint8_t appparam_SPpMTI_4;
uint8_t appparam_mSPpMTI;
uint8_t last_cycle_coord = 0x0000;
uint8_t count_coord = 0x0000;
//DATA MANAGER VARIABLES
uint16_t flags = 0;
uint16_t source_add;
uint16_t destination_add = 0;
uint16_t app_data1 = 0;
uint16_t app_data2 = 0;
uint16_t app_data3 = 0;
uint16_t app_data4 = 0;
uint16_t app_data5 = 0;
uint16_t app_data6 = 0;
uint16_t app_data7 = 0;
uint16_t app_data8 = 0;
uint16_t app_data9 = 0;
uint16_t app_data10 = 0;
uint16_t app_data11 = 0;
uint16_t buff_data1 = 0;
uint16_t buff_data2 = 0;
uint16_t buff_data3 = 0;
uint16_t buff_data4 = 0;
uint16_t buff_data5 = 0;
uint16_t buff_data6 = 0;
uint16_t buff_data7 = 0;
uint16_t buff_data8 = 0;
uint16_t buff_data9 = 0;
uint16_t buff_data10 = 0;
uint16_t buff_data11 = 0;
uint16_t buff_source_add;
uint16_t appparam_MDTC = 0x0001; // 1ms default
uint16_t flags_buff = 0;
uint16_t buffE_data1 = 0;
uint16_t buffE_data2 = 0;
uint16_t buffE_data3 = 0;
uint16_t buffE_data4 = 0;
uint16_t buffE_data5 = 0;
uint16_t buffE_data6 = 0;
uint16_t buffE_data7 = 0;
uint16_t buffE_data8 = 0;
uint16_t buffE_data9 = 0;
```

```
	uint16_t buffE_data10 = 0;
	uint16_t buffE_data11 = 0;
	uint16_t buffE_source_add;
	uint16_t appparam_MDTE = 0x0001; // 1ms default
	uint16_t flags_buffE = 0;
	uint16_t rndDATA = 0;
	//finish: defining variables

	void count_neighbors(){ //count number of neighbours in the array
	  Neighbors_n = 0;
	  for (i=0;i<NeighborsLen;i++){ //number of neighbours
			if ((neighbors[i].cycle > 0)){
				Neighbors_n = Neighbors_n + 1;
			}
		}
	}

	void count_sinks(){ //count number of neighbours in the array
	  Sinks_n = 0;
	  for (i=0;i<SinknodesLen;i++){ //number of neighbours related to selected sink
			if (SinkCycle[i].scycle > 0) {
				Sinks_n = Sinks_n + 1;
			}
		}
	}

	void App_Function(){ //App Function
		count_neighbors();
		F1 = 100.0F-(((float)(last_cycle_coord) * 50.0F)+(((float)(count_coord)/(float)(SinkCycle[SinkID].scycle))*50.0F));
		F2 = 100.0F-(((float)(Neighbors_n)/(float)(NeighborsLen+1))*100.0F);
		F3 = 100.0F / (((float)(parent_hops) + 1.0F));
		if (node_area == 0x0001){F4 = 100.0F * ((float)(appparam_SPpMTI_1) / (float)(appparam_mSPpMTI+1));} //this is
for a node in area 1
		if (node_area == 0x0002){F4 = 100.0F * ((float)(appparam_SPpMTI_2) / (float)(appparam_mSPpMTI+1));} //this is
for a node in area 2
		if (node_area == 0x0003){F4 = 100.0F * ((float)(appparam_SPpMTI_3) / (float)(appparam_mSPpMTI+1));} //this is
for a node in area 3
		if (node_area == 0x0004){F4 = 100.0F * ((float)(appparam_SPpMTI_4) / (float)(appparam_mSPpMTI+1));} //this is
for a node in area 4
		prob = ((float)(appparam_a))*((float)(F1));
		prob += ((float)(appparam_b))*((float)(F2));
		prob += ((float)(appparam_c))*((float)(F3));
		prob += ((float)(appparam_d))*((float)(F4));
		prob = prob / ((float)(appparam_a + appparam_b + appparam_c + appparam_d));
	}

	task void msync_broad_task(){ //Task of broadcasting MSync message with updated fields from this node
		if (!locked) {
		  MSync_t* rcm = (MSync_t*)call MSync_P.getPayload(&packet, sizeof(MSync_t));
		  rcm->Node_ID = TOS_NODE_ID;
		  rcm->cycle = SinkCycle[SinkID].scycle;
		  rcm->hops = parent_hops + 1;
		  if (coord == TRUE){rcm->coord = 1;}else if(coord == FALSE){rcm->coord = 0;};
		  rcm->energy = currEnergy;
		  rcm->appparam_MDTC = appparam_MDTC;
		  rcm->appparam_MDTE = appparam_MDTE;
		  rcm->appparam_a = appparam_a;
		  rcm->appparam_b = appparam_b;
		  rcm->appparam_c = appparam_c;
		  rcm->appparam_d = appparam_d;
		  rcm->SinkID = SinkID;
		  rcm->appparam_SPpMTI_1 = appparam_SPpMTI_1;
		  rcm->appparam_SPpMTI_2 = appparam_SPpMTI_2;
		  rcm->appparam_SPpMTI_3 = appparam_SPpMTI_3;
		  rcm->appparam_SPpMTI_4 = appparam_SPpMTI_4;
		  rcm->appparam_mSPpMTI = appparam_mSPpMTI;
		  rcm->Parent_ID = parent_add;
		  rcm->Valid_Time = Valid_Time; //valid cycle time
		  if (call MSync_S.send(AM_BROADCAST_ADDR, &packet, sizeof(MSync_t)) == SUCCESS) {
			locked = TRUE;
			counter_send++;
		  }
		}
		else{post msync_broad_task();}
	}

	void Election_Manager(){ //Election Manager
		coord = FALSE;
		App_Function();
		// coordinator or not, random limited to prob from app function
		tmp_rnd = (float)(call Random.rand16());
		tmp_rnd = 100.0F * (tmp_rnd/65535.0F);
		if (tmp_rnd < prob) {coord = TRUE;} else {coord = FALSE;}
		last_cycle_coord = 0;
		if (coord == TRUE) {count_coord++;last_cycle_coord = 1;}
		cont_sync_task = TRUE;
		post msync_broad_task();
	}

	void Parent_Selector_Manager(){ //select parent within neighbours list
		if (coord == 1){
				//priority: 1)minimum hops, 2)being coordinator, 3)maximum energy
				uint8_t min_hop = 255;
				uint16_t max_energy = 0;
				for (i=0;i<NeighborsLen;i++){
					if (neighbors[i].hops < min_hop && (!neighbors[i].energy == 0) && (neighbors[i].energy > 0) &&
(SinkID == neighbors[i].sink_id)) {
						min_hop = neighbors[i].hops;
					}
				}
				for (i=0;i<NeighborsLen;i++){
					if ((neighbors[i].hops == min_hop && neighbors[i].coord == 1) && (neighbors[i].energy > 0)&&
(SinkID == neighbors[i].sink_id)) {
						if (max_energy < neighbors[i].energy) {max_energy = neighbors[i].energy;
parent_add = neighbors[i].node_id;
						parent_hops = neighbors[i].hops;parent_coord = neighbors[i].coord;}
					}
				}
				if (max_energy == 0){
					for (i=0;i<NeighborsLen;i++){
						if ((neighbors[i].hops == min_hop)  && (neighbors[i].energy > 0)&& (SinkID ==
neighbors[i].sink_id))  {
							if (max_energy < neighbors[i].energy) {max_energy = neighbors[i].energy;
parent_add = neighbors[i].node_id;
							parent_hops = neighbors[i].hops;parent_coord = neighbors[i].coord;}
						}
					}
				}
		}
		else{
				//priority: 1)being coordinator, 2)minimum hops, 3)maximum energy
				uint8_t min_hop = 255;
				uint16_t max_energy = 0;
				for (i=0;i<NeighborsLen;i++){
					if ((neighbors[i].coord == 1) && (neighbors[i].energy > 0)&& (SinkID == neighbors[i].sink_id))  {
						if (neighbors[i].hops < min_hop && (!neighbors[i].energy == 0)) {min_hop =
neighbors[i].hops;}
					}
				}
				for (i=0;i<NeighborsLen;i++){
					if ((neighbors[i].hops == min_hop) && (neighbors[i].coord == 1)  && (neighbors[i].energy >
0)&& (SinkID == neighbors[i].sink_id)) {
						if (max_energy < neighbors[i].energy) {max_energy = neighbors[i].energy;
parent_add = neighbors[i].node_id;
						parent_hops = neighbors[i].hops;parent_coord = neighbors[i].coord;}
					}
				}
				if (max_energy == 0){
					for (i=0;i<NeighborsLen;i++){
						if ((neighbors[i].hops < min_hop) && (neighbors[i].energy > 0)&& (SinkID ==
neighbors[i].sink_id))  {min_hop = neighbors[i].hops;}
					}
					for (i=0;i<NeighborsLen;i++){
						if ((neighbors[i].hops == min_hop) && (neighbors[i].energy > 0)&& (SinkID ==
neighbors[i].sink_id))  {
							if (max_energy < neighbors[i].energy) {max_energy = neighbors[i].energy;
parent_add = neighbors[i].node_id;
							parent_hops = neighbors[i].hops;parent_coord = neighbors[i].coord;}
```

```
                                        }
                                }
                        }
                }
        }

        int Index_finder(uint16_t id){ //Function to find the index of array related to given node_id or create a new index
                for (i=0;i<NeighborsLen;i++){
                        if ((neighbors[i].node_id == id)||(neighbors[i].energy == 0)) {
                                neighbors[i].node_id = id;
                                return i;
                        }
                }
                return -1;
        }

        event message_t* MSync_R.receive(message_t* bufPtrS, //Receive MSync message
                                                void* payload, uint8_t len) {
                MSync_t* rcm = (MSync_t*)payload;
                //start Sync-Manager
                if (SinkID == -1){SinkID = rcm->SinkID;SinkCycle[(rcm->SinkID)].scycle = (rcm->cycle);SinkCycle[(rcm-
>SinkID)].valid_time = (rcm->Valid_Time);} //for the first time
                if ((SinkCycle[(rcm->SinkID)].scycle < (rcm->cycle))||(Neighbors_n == 0)){ //MSync message of new cycle
                        SinkCycle[(rcm->SinkID)].scycle = (rcm->cycle);
                        SinkCycle[(rcm->SinkID)].valid_time = (rcm->Valid_Time);
                        //empty the list of neighbours for new cycle for the related sink
                        for (i=0;i<NeighborsLen;i++){
                                if ((neighbors[i].sink_id == rcm->SinkID)&&(neighbors[i].cycle > 0)){
                                        neighbors[i].node_id = 0;
                                        neighbors[i].hops = 0;
                                        neighbors[i].sink_id = 0;
                                        neighbors[i].energy = 0;
                                        neighbors[i].coord = 0;
                                        neighbors[i].cycle = 0;
                                }
                        }
                        //add new cycle data to neighbours and parameters
                        ID = Index_finder(rcm->Node_ID);
                        if (ID>-1){
                                neighbors[ID].hops = rcm->hops;
                                neighbors[ID].cycle = rcm->cycle;
                                neighbors[ID].sink_id = rcm->SinkID;
                                neighbors[ID].coord = rcm->coord;
                                neighbors[ID].energy = rcm->energy;
                        }
                        appparam_MDTC = rcm->appparam_MDTC;
                        appparam_MDTE = rcm->appparam_MDTE;
                        appparam_a = rcm->appparam_a;
                        appparam_b = rcm->appparam_b;
                        appparam_c = rcm->appparam_c;
                        appparam_d = rcm->appparam_d;
                        appparam_SPpMTI_1 = rcm->appparam_SPpMTI_1;
                        appparam_SPpMTI_2 = rcm->appparam_SPpMTI_2;
                        appparam_SPpMTI_3 = rcm->appparam_SPpMTI_3;
                        appparam_SPpMTI_4 = rcm->appparam_SPpMTI_4;
                        appparam_mSPpMTI = rcm->appparam_mSPpMTI;
                        Valid_Time = rcm->Valid_Time; //valid cycle time
                        if (SinkID == rcm->SinkID){ //same sink
                                //set default parent to the new MSync sender
                                parent_add = rcm->Node_ID;
                                parent_hops = rcm->hops;
                                parent_coord = rcm->coord;
                                call MilliTimerValidTime.stop();
                                Election_Manager();
                        }else if((rcm->hops) < parent_hops){ //new sink is better because it has less hops to sink
                                parent_add = rcm->Node_ID;
                                parent_hops = rcm->hops;
                                parent_coord = rcm->coord;
                                SinkID = rcm->SinkID;
                                call MilliTimerValidTime.stop();
                                Election_Manager();
                        }
                }else if (SinkCycle[(rcm->SinkID)].scycle == (rcm->cycle)) {//MSync message of same cycle, which completes the
neighbour list
```

```
                    ID = Index_finder(rcm->Node_ID);
                    if (ID>-1){
                              neighbors[ID].hops = rcm->hops;
                              neighbors[ID].cycle = rcm->cycle;
                              neighbors[ID].sink_id = rcm->SinkID;
                              neighbors[ID].coord = rcm->coord;
                              neighbors[ID].energy = rcm->energy;
                    }
          }
          //stop Sync-Manager
          return bufPtrS;
     }

     event message_t* MCoord_R.receive(message_t* bufPtrC, //Receive MCoord message, the node force to be
coordinator, complete the neighbour list
                                        void* payload, uint8_t len) {
          MCoord_t* rcm = (MCoord_t*)payload;
          //start Coordinator_Indication_Manager,  if a node receive MCoord, it will force to be a coordinator
          ID = Index_finder(rcm->Node_ID);
          if (ID>-1){
                    neighbors[ID].hops = rcm->hops;
                    neighbors[ID].cycle = rcm->cycle;
                    neighbors[ID].sink_id = rcm->SinkID;
                    neighbors[ID].coord = rcm->coord;
                    neighbors[ID].energy = rcm->energy;
          }
          appparam_MDTC = rcm->appparam_MDTC;
          appparam_MDTE = rcm->appparam_MDTE;
          appparam_a = rcm->appparam_a;
          appparam_b = rcm->appparam_b;
          appparam_c = rcm->appparam_c;
          appparam_d = rcm->appparam_d;
          appparam_SPpMTI_1 = rcm->appparam_SPpMTI_1;
          appparam_SPpMTI_2 = rcm->appparam_SPpMTI_2;
          appparam_SPpMTI_3 = rcm->appparam_SPpMTI_3;
          appparam_SPpMTI_4 = rcm->appparam_SPpMTI_4;
          appparam_mSPpMTI = rcm->appparam_mSPpMTI;
          Valid_Time = rcm->Valid_Time; //valid cycle time
          coord = TRUE;
          count_coord++;last_cycle_coord = 1;
          cont_sync_task = FALSE;
          post msync_broad_task();
          //stop Coordinator_Indication_Manager
          return bufPtrC;
     }

     task void mcoord_uni_task(){ //Task of uni-casting MCoord message with updated fields to the parent which is not a
coordinator
          if (!locked) {
            MCoord_t* rcm = (MCoord_t*)call MCoord_P.getPayload(&packet, sizeof(MCoord_t));
            rcm->Node_ID = TOS_NODE_ID;
            rcm->cycle = SinkCycle[SinkID].scycle;
            rcm->hops = parent_hops + 1;
            if (coord == TRUE){rcm->coord = 1;}else if(coord == FALSE){rcm->coord = 0;};
            rcm->energy = currEnergy;
            rcm->appparam_MDTC = appparam_MDTC;
            rcm->appparam_MDTE = appparam_MDTE;
            rcm->appparam_a = appparam_a;
            rcm->appparam_b = appparam_b;
            rcm->appparam_c = appparam_c;
            rcm->appparam_d = appparam_d;
            rcm->SinkID = SinkID;
            rcm->appparam_SPpMTI_1 = appparam_SPpMTI_1;
            rcm->appparam_SPpMTI_2 = appparam_SPpMTI_2;
            rcm->appparam_SPpMTI_3 = appparam_SPpMTI_3;
            rcm->appparam_SPpMTI_4 = appparam_SPpMTI_4;
            rcm->appparam_mSPpMTI = appparam_mSPpMTI;
            rcm->Parent_ID = parent_add;
            rcm->Valid_Time = Valid_Time; //valid cycle time
            if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                      if (call MCoord_S.send(parent_add, &packet, sizeof(MCoord_t)) == SUCCESS) {
                              trycoord = trycoord+1;
                              locked = TRUE;
                              counter_send++;
```

```
                        }
            }
          }
          else{post mcoord_uni_task();}
      }

      void Backbone_Fulfil_Manager(){ //Backbone Fulfill Manager
          if (parent_coord == 0){
                  //Send MCoord message to parent with residual info to force parent to be coordinator
                  post mcoord_uni_task();
                  //update neighbour list
                  parent_coord = TRUE;
                  for (i=0;i<NeighborsLen;i++){
                          if (neighbors[i].node_id == parent_add) {
                                  neighbors[i].coord = TRUE;
                          }
                  }
          }
      }

      event void Boot.booted() { //Node start to boot
          call AMControl.start();
      }

      event void AMControl.startDone(error_t err) { //Node start to work. Based on given ID, node choose the area number
and startPeriodic(AppCycleTime)
          if (err == SUCCESS) {
            call Battery.read(); //for first cycle reading battery
            //When node starts, based on ID, choose the area
            if (TOS_NODE_ID > (node_area1_start-1) && TOS_NODE_ID < (node_area1_end+1)){node_area = 0x0001;}
            if (TOS_NODE_ID > (node_area2_start-1) && TOS_NODE_ID < (node_area2_end+1)){node_area = 0x0002;}
            if (TOS_NODE_ID > (node_area3_start-1) && TOS_NODE_ID < (node_area3_end+1)){node_area = 0x0003;}
            if (TOS_NODE_ID > (node_area4_start-1) && TOS_NODE_ID < (node_area4_end+1)){node_area = 0x0004;}
            //define the array of SinkCycle
            for (i=0;i<SinknodesLen;i++){
                  SinkCycle[i].scycle = 0;
                  SinkCycle[i].valid_time = 0;
        }
            //define the array of neighbors
            for (i=0;i<NeighborsLen;i++){
                  neighbors[i].node_id = 0;
                  neighbors[i].hops = 0;
                  neighbors[i].sink_id = 0;
                  neighbors[i].energy = 0;
                  neighbors[i].coord = 0;
                  neighbors[i].cycle = 0;
        }
            call MilliTimerApp.startPeriodic(AppCycleTime);
          }
          else {
            call AMControl.start();
          }
      }

      event void AMControl.stopDone(error_t err) { //Node doesn't start to work
      }

      task void forwardMDataToParentTask(){ //Task of Forwarding Data packets to parent
          if (!(app_data10 == 0)){
                  if ((!locked)&&(parent_add > -1)) {
                    MData5_t* rcm = (MData5_t*)call MDataF_P.getPayload(&packet, sizeof(MData5_t));
                    rcm->flags = flags;
                    rcm->source_add = source_add;
                    rcm->destination_add = destination_add;
                    rcm->app_data1 = app_data1;
                    rcm->app_data2 = app_data2;
                    rcm->app_data3 = app_data3;
                    rcm->app_data4 = app_data4;
                    rcm->app_data5 = app_data5;
                    rcm->app_data6 = app_data6;
                    rcm->app_data7 = app_data7;
                    rcm->app_data8 = app_data8;
                    rcm->app_data9 = app_data9;
                    rcm->app_data10 = app_data10;
                    rcm->app_data11 = app_data11;

                    if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                            if (call MDataF_S.send(parent_add, &packet, sizeof(MData5_t)) == SUCCESS) {
                                    trydataF = trydataF+1;
                                    locked = TRUE;
                                    counter_send++;
                            }
                    }
                  }
                  else{post forwardMDataToParentTask();}
          }
          else if (!(app_data8 == 0)){
                  if ((!locked)&&(parent_add > -1)) {
                    MData4_t* rcm = (MData4_t*)call MDataF_P.getPayload(&packet, sizeof(MData4_t));
                    rcm->flags = flags;
                    rcm->source_add = source_add;
                    rcm->destination_add = destination_add;
                    rcm->app_data1 = app_data1;
                    rcm->app_data2 = app_data2;
                    rcm->app_data3 = app_data3;
                    rcm->app_data4 = app_data4;
                    rcm->app_data5 = app_data5;
                    rcm->app_data6 = app_data6;
                    rcm->app_data7 = app_data7;
                    rcm->app_data8 = app_data8;
                    rcm->app_data9 = app_data9;

                    if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                            if (call MDataF_S.send(parent_add, &packet, sizeof(MData4_t)) == SUCCESS) {
                                    trydataF = trydataF+1;
                                    locked = TRUE;
                                    counter_send++;
                            }
                    }
                  }
                  else{post forwardMDataToParentTask();}
          }
          else if (!(app_data6 == 0)){
                  if ((!locked)&&(parent_add > -1)) {
                    MData3_t* rcm = (MData3_t*)call MDataF_P.getPayload(&packet, sizeof(MData3_t));
                    rcm->flags = flags;
                    rcm->source_add = source_add;
                    rcm->destination_add = destination_add;
                    rcm->app_data1 = app_data1;
                    rcm->app_data2 = app_data2;
                    rcm->app_data3 = app_data3;
                    rcm->app_data4 = app_data4;
                    rcm->app_data5 = app_data5;
                    rcm->app_data6 = app_data6;
                    rcm->app_data7 = app_data7;

                    if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                            if (call MDataF_S.send(parent_add, &packet, sizeof(MData3_t)) == SUCCESS) {
                                    trydataF = trydataF+1;
                                    locked = TRUE;
                                    counter_send++;
                            }
                    }
                  }
                  else{post forwardMDataToParentTask();}
          }
          else if (!(app_data4 == 0)){
                  if ((!locked)&&(parent_add > -1)) {
                    MData2_t* rcm = (MData2_t*)call MDataF_P.getPayload(&packet, sizeof(MData2_t));
                    rcm->flags = flags;
                    rcm->source_add = source_add;
                    rcm->destination_add = destination_add;
                    rcm->app_data1 = app_data1;
                    rcm->app_data2 = app_data2;
                    rcm->app_data3 = app_data3;
                    rcm->app_data4 = app_data4;
                    rcm->app_data5 = app_data5;
```

```
                        if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                                if (call MDataF_S.send(parent_add, &packet, sizeof(MData2_t)) == SUCCESS) {
                                        trydataF = trydataF+1;
                                        locked = TRUE;
                                        counter_send++;
                                }
                        }
                       }
                       else{post forwardMDataToParentTask();}
               }
               else if (!(app_data2 == 0)){
                       if ((!locked)&&(parent_add > -1)) {
                        MData1_t* rcm = (MData1_t*)call MDataF_P.getPayload(&packet, sizeof(MData1_t));
                        rcm->flags = flags;
                        rcm->source_add = source_add;
                        rcm->destination_add = destination_add;
                        rcm->app_data1 = app_data1;
                        rcm->app_data2 = app_data2;
                        rcm->app_data3 = app_data3;

                        if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                                if (call MDataF_S.send(parent_add, &packet, sizeof(MData1_t)) == SUCCESS) {
                                        trydataF = trydataF+1;
                                        locked = TRUE;
                                        counter_send++;
                                }
                        }
                       }
                       else{post forwardMDataToParentTask();}
               }
               else if (!(source_add == 0)){
                       if ((!locked)&&(parent_add > -1)) {
                        MData0_t* rcm = (MData0_t*)call MDataF_P.getPayload(&packet, sizeof(MData0_t));
                        rcm->flags = flags;
                        rcm->source_add = source_add;
                        rcm->destination_add = destination_add;
                        rcm->app_data1 = app_data1;

                        if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                                if (call MDataF_S.send(parent_add, &packet, sizeof(MData0_t)) == SUCCESS) {
                                        trydataF = trydataF+1;
                                        locked = TRUE;
                                        counter_send++;
                                }
                        }
                       }
                       else{post forwardMDataToParentTask();}
               }
       }

       event message_t* MDataF_R.receive(message_t* bufPtrFD, //Receive MData message, performing aggregation, and
forwarding to parent
                                                void* payload, uint8_t len) {
         app_data1 = 0; app_data2 = 0;app_data3 = 0;app_data4 = 0;app_data5 = 0;app_data6 = 0;
               app_data7 = 0;app_data8 = 0;app_data9 = 0;app_data10 = 0;app_data11 = 0;
               if (len == sizeof(MData5_t)){
                       MData5_t* rcm = (MData5_t*)payload;
                       flags = rcm->flags;
                       source_add = rcm->source_add;
                       destination_add = rcm->destination_add;
                       app_data1 = rcm->app_data1;
                       app_data2 = rcm->app_data2;
                       app_data3 = rcm->app_data3;
                       app_data4 = rcm->app_data4;
                       app_data5 = rcm->app_data5;
                       app_data6 = rcm->app_data6;
                       app_data7 = rcm->app_data7;
                       app_data8 = rcm->app_data8;
                       app_data9 = rcm->app_data9;
                       app_data10 = rcm->app_data10;
                       app_data11 = rcm->app_data11;
               }
               else if (len == sizeof(MData4_t)){
                       MData4_t* rcm = (MData4_t*)payload;
```

```
			flags = rcm->flags;
			source_add = rcm->source_add;
			destination_add = rcm->destination_add;
			app_data1 = rcm->app_data1;
			app_data2 = rcm->app_data2;
			app_data3 = rcm->app_data3;
			app_data4 = rcm->app_data4;
			app_data5 = rcm->app_data5;
			app_data6 = rcm->app_data6;
			app_data7 = rcm->app_data7;
			app_data8 = rcm->app_data8;
			app_data9 = rcm->app_data9;
		}
		else if (len == sizeof(MData3_t)){
			MData3_t* rcm = (MData3_t*)payload;
			flags = rcm->flags;
			source_add = rcm->source_add;
			destination_add = rcm->destination_add;
			app_data1 = rcm->app_data1;
			app_data2 = rcm->app_data2;
			app_data3 = rcm->app_data3;
			app_data4 = rcm->app_data4;
			app_data5 = rcm->app_data5;
			app_data6 = rcm->app_data6;
			app_data7 = rcm->app_data7;
		}
		else if (len == sizeof(MData2_t)){
			MData2_t* rcm = (MData2_t*)payload;
			flags = rcm->flags;
			source_add = rcm->source_add;
			destination_add = rcm->destination_add;
			app_data1 = rcm->app_data1;
			app_data2 = rcm->app_data2;
			app_data3 = rcm->app_data3;
			app_data4 = rcm->app_data4;
			app_data5 = rcm->app_data5;
		}
		else if (len == sizeof(MData1_t)){
			MData1_t* rcm = (MData1_t*)payload;
			flags = rcm->flags;
			source_add = rcm->source_add;
			destination_add = rcm->destination_add;
			app_data1 = rcm->app_data1;
			app_data2 = rcm->app_data2;
			app_data3 = rcm->app_data3;
		}
		else if (len == sizeof(MData0_t)){
			MData0_t* rcm = (MData0_t*)payload;
			flags = rcm->flags;
			source_add = rcm->source_add;
			destination_add = rcm->destination_add;
			app_data1 = rcm->app_data1;
		}
		if ((flags & B01_16) == 0){ //Aggregation bit is 0, means the packet is not able to aggregate and should just forward
			//forward to parent
			post forwardMDataToParentTask();
		}
		else if (!((app_data11 == 0)&&(app_data10 == 0))){ //Means the packet is full and not able to aggregate and should
just forward
			flags = flags ^ B01_16; //Set the aggregation bit to 0
			//forward to parent
			post forwardMDataToParentTask();
		}
		else if ((flags & B02_16) == 0){ //Data type bit is 0, means continuous data type
			//start the timer of MDTC
			call MilliTimerBuf.startOneShot(appparam_MDTC);
			if (flags_buff == 0) {flags_buff = flags;}
			if (buff_data1 == 0) { buff_source_add = source_add; buff_data1= app_data1;}
			else if (buff_data2 == 0) { buff_data2 = source_add; buff_data3 = app_data1;}
			else if (buff_data4 == 0) { buff_data4 = source_add; buff_data5 = app_data1;}
			else if (buff_data6 == 0) { buff_data6 = source_add; buff_data7 = app_data1;}
			else if (buff_data8 == 0) { buff_data8 = source_add; buff_data9 = app_data1;}
			else if (buff_data10 == 0) { buff_data10 = source_add; buff_data11 = app_data1;}
			else {flags = flags_buff ^ B01_16;source_add = buff_source_add; app_data1 = buff_data1;
```

```
                                app_data2 = buff_data2;app_data3 = buff_data3;app_data4 = buff_data4;
                                app_data5 = buff_data5;app_data6 = buff_data6;app_data7 = buff_data7;
                                app_data8 = buff_data8;app_data9 = buff_data9;app_data10 = buff_data10;
                                app_data11 = buff_data11; buff_data1 = 0;  buff_data2 = 0;  buff_data3 = 0;  buff_data4 = 0;
                                buff_data5 = 0;  buff_data6 = 0;  buff_data7 = 0;  buff_data8 = 0;  buff_data9 = 0;  buff_data10
= 0;
                                buff_data11 = 0; flags_buff = 0; post forwardMDataToParentTask(); call MilliTimerBuf.stop();
                                }
            }
            else if ((flags & B02_16) == 1){ //Data type bit is 1, means event data type
                        //start the timer of MDTE
                        call MilliTimerBufE.startOneShot(appparam_MDTE);
                        if (flags_buffE == 0) {flags_buffE = flags;}
                        if (buffE_data1 == 0) { buffE_source_add = source_add; buffE_data1= app_data1;}
                        else if (buffE_data2 == 0) { buffE_data2 = source_add; buffE_data3 = app_data1;}
                        else if (buffE_data4 == 0) { buffE_data4 = source_add; buffE_data5 = app_data1;}
                        else if (buffE_data6 == 0) { buffE_data6 = source_add; buffE_data7 = app_data1;}
                        else if (buffE_data8 == 0) { buffE_data8 = source_add; buffE_data9 = app_data1;}
                        else if (buffE_data10 == 0) { buffE_data10 = source_add; buffE_data11 = app_data1;}
                        else {flags = flags_buffE ^ B01_16;source_add = buffE_source_add; app_data1 = buffE_data1;
                                app_data2 = buffE_data2;app_data3 = buffE_data3;app_data4 = buffE_data4;
                                app_data5 = buffE_data5;app_data6 = buffE_data6;app_data7 = buffE_data7;
                                app_data8 = buffE_data8;app_data9 = buffE_data9;app_data10 = buffE_data10;
                                app_data11 = buffE_data11; buff_data1 = 0;  buffE_data2 = 0;  buffE_data3 = 0;  buffE_data4 =
0;
                                buffE_data5 = 0;   buffE_data6 = 0;   buffE_data7 = 0;   buffE_data8 = 0;   buffE_data9 = 0;
buffE_data10 = 0;
                                buffE_data11    =    0;    flags_buffE    =    0;    post    forwardMDataToParentTask();    call
MilliTimerBufE.stop();
                                }
            }
            return bufPtrFD;
        }

        task void sendMDataTask(){ //Task of sending random value as monitored data
            if ((!locked)&&(parent_add > -1)) {
             MData0_t* rcm = (MData0_t*)call MData_P.getPayload(&packet, sizeof(MData0_t));
             flags = 0x8000;
             rcm->flags = flags;
             rcm->source_add = TOS_NODE_ID;
             rcm->destination_add = parent_add;
             rcm->app_data1 = rndDATA; // random sensed data
             if(call PacketAcknowledgements.requestAck(&packet)==SUCCESS){
                        if (call MData_S.send(parent_add, &packet, sizeof(MData0_t)) == SUCCESS) {
                                trydata = trydata+1;
                                locked = TRUE;
                                counter_send++;
                        }
             }
          }
            else{post sendMDataTask();}
        }

        event void MSync_S.sendDone(message_t* bufPtrS, error_t error) { //MSync broad-casting done
          if (&packet == bufPtrS) {
                        locked = FALSE;
                        if (cont_sync_task == TRUE){ //for case of MSync message after a new cycle, it call random backoff time
to receive neighbours packets
                                cont_sync_task = FALSE;
                                //wait random backoff time
                                randBackoffPeriod = (float)(call Random.rand16());
                                randBackoffPeriod = (randBackoffPeriod/65535.0F);
                                randBackoffPeriod = randBackoffPeriod * MAX_RANDOM_THRESHOLD;
                                if     (randBackoffPeriod     <     ((float)(MIN_RANDOM_THRESHOLD))){randBackoffPeriod     =
randBackoffPeriod +((float)(MIN_RANDOM_THRESHOLD));}
                                call MilliTimer.startOneShot((uint16_t)(randBackoffPeriod)); //start random backoff time
                        }
            }
        }

        event void MilliTimerACK.fired(){ //repeat message after 512ms
          if (rep_mode != 0){
                        if (rep_mode == 1){post mcoord_uni_task();rep_mode = 0;}
                        if (rep_mode == 2){post sendMDataTask();rep_mode = 0;}
```

```
			if (rep_mode == 3){post forwardMDataToParentTask();rep_mode = 0;}
		}
	}

	event void MData_S.sendDone(message_t* bufPtrD, error_t error) { //MData sending done
	  if ((&packet == bufPtrD)){locked = FALSE;}
		if ((&packet == bufPtrD) && (call PacketAcknowledgements.wasAcked(bufPtrD)==SUCCESS)){
			locked = FALSE;
			trydata = 1;
		}else{
			if (trydata < 3){
				trydata = trydata+1;
				rep_mode = 2;
				call MilliTimerACK.startOneShot(512); //repeat message after 512ms
			}
		}
	}

	event void MDataF_S.sendDone(message_t* bufPtrFD, error_t error) { //MData sending done
	  if ((&packet == bufPtrFD)){locked = FALSE;}
		if ((&packet == bufPtrFD) && (call PacketAcknowledgements.wasAcked(bufPtrFD)==SUCCESS)){
			locked = FALSE;
			trydataF = 1;
		}else{
			if (trydataF < 3){
				trydataF = trydataF+1;
				rep_mode = 3;
				call MilliTimerACK.startOneShot(512); //repeat message after 512ms
			}
		}
	}

	event void MCoord_S.sendDone(message_t* bufPtrC, error_t error) { //MCoord sending done
	  if ((&packet == bufPtrC)){locked = FALSE;}
		if ((&packet == bufPtrC) && (call PacketAcknowledgements.wasAcked(bufPtrC)==SUCCESS)){
			locked = FALSE;
			trycoord = 1;
		}else{
			if (trycoord < 4){
				trycoord = trycoord+1;
				rep_mode = 1;
				call MilliTimerACK.startOneShot(100); //repeat message after 100ms
			}
		}
	}

	event void MilliTimerValidTime.fired(){ //it means the selected sink is not valid any more
		for (i=0;i<NeighborsLen;i++){
			if ((neighbors[i].sink_id == SinkID)&&(neighbors[i].cycle > 0)){
				neighbors[i].node_id = 0;
				neighbors[i].hops = 0;
				neighbors[i].sink_id = 0;
				neighbors[i].energy = 0;
				neighbors[i].coord = 0;
				neighbors[i].cycle = 0;
			}
		}
		//change selected sink to another sink from neighbours list if there is
		find = 0;
		for (i=0;i<NeighborsLen;i++){
			if ((neighbors[i].sink_id != SinkID)&&(neighbors[i].cycle > 0)){
				SinkID = neighbors[i].sink_id;
				find = 1;
			}
		}
		//maybe it does not have info from other sink
		if (find == 1){
			Parent_Selector_Manager();
			Election_Manager();
		}else {
			SinkID = -1;
		}
	}
```

```
        event void MilliTimer.fired(){ //End of random backoff time
            //call valid time timer to find invalid sink, considering an amount of delay time and stagger of dissemination
            if ((SinkCycle[SinkID].valid_time) > 0){call MilliTimerValidTime.stop();
                    call
MilliTimerValidTime.startOneShot((uint32_t)(((SinkCycle[SinkID].valid_time)*61440)+10240+(5120/(parent_hops+2))));}
         Parent_Selector_Manager();
            Backbone_Fulfil_Manager();
        }

        event void MilliTimerBuf.fired(){ //If MDTC end, the aggregation stops and the packet forward to parent
            flags = flags_buff ^ B01_16;source_add = buff_source_add; app_data1 = buff_data1;
            app_data2 = buff_data2;app_data3 = buff_data3;app_data4 = buff_data4;
            app_data5 = buff_data5;app_data6 = buff_data6;app_data7 = buff_data7;
            app_data8 = buff_data8;app_data9 = buff_data9;app_data10 = buff_data10;
            app_data11 = buff_data11; buff_data1 = 0;  buff_data2 = 0;  buff_data3 = 0;  buff_data4 = 0;
            buff_data5 = 0;  buff_data6 = 0;  buff_data7 = 0;  buff_data8 = 0;  buff_data9 = 0;  buff_data10 = 0;
            buff_data11 = 0; flags_buff = 0;
            post forwardMDataToParentTask();
        }

        event void MilliTimerBufE.fired(){ //If MDTE end, the aggregation stops and the packet forward to parent
            flags = flags_buffE ^ B01_16;source_add = buffE_source_add; app_data1 = buffE_data1;
            app_data2 = buffE_data2;app_data3 = buffE_data3;app_data4 = buffE_data4;
            app_data5 = buffE_data5;app_data6 = buffE_data6;app_data7 = buffE_data7;
            app_data8 = buffE_data8;app_data9 = buffE_data9;app_data10 = buffE_data10;
            app_data11 = buffE_data11; buff_data1 = 0;  buffE_data2 = 0;  buffE_data3 = 0;  buffE_data4 = 0;
            buffE_data5 = 0;  buffE_data6 = 0;  buffE_data7 = 0;  buffE_data8 = 0;  buffE_data9 = 0;  buffE_data10 = 0;
            buffE_data11 = 0; flags_buffE = 0;
            post forwardMDataToParentTask();
        }

        event void MilliTimerApp.fired(){ //The application timer to call sample application, also in each period the current
energy of node updates
            // Updates battery
            call Battery.read();
            // Application sends MData
            Appcounter++;
            if (node_area == 0x0001){ //This timer fires every 14 second, area1 will monitor and send data every 14 seconds
                    if (Appcounter == 1){
                            rndDATA = call Random.rand16(); // random sensed data
                            post sendMDataTask();
                            Appcounter = 0x0000;
                    }
            }
            else if (node_area == 0x0002){          //This timer fires every 14 second, area2 will monitor and send data every
28 seconds
                    if(Appcounter == 2){
                            rndDATA = call Random.rand16(); // random sensed data
                            post sendMDataTask();
                            Appcounter = 0x0000;
                    }
            }else if (node_area == 0x0003){
            }else if (node_area == 0x0004){
            }
        }

        event void Battery.readDone(error_t result, uint16_t data){ //For update current energy of node
            if (result == SUCCESS) {
              currEnergy = data;
            }
        }
    }
```