Technical University of Crete, Greece

School of Electronic and Computer Engineering

Design and Implementation of Algorithms for Aggregate Queries in the Presence of Duplicates

Marina Mavrikou

Thesis Committee

Professor Antonios Deligiannakis (ECE)

Professor Aggelos Bletsas (ECE)

Professor Minos Garofalakis (ECE)

Chania, December 2013

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδίαση και Υλοποίηση Αλγορίθμων για αποτίμηση Συναθροιστικών Επερωτήσεων υπό την παρουσία Διπλότυπων Μετρήσεων

Μαρίνα Μαυρίκου

Χανιά, Δεκέμβριος 2013

Abstract

Wireless Sensor Networks (WSNs) are nowadays widely used and form a rapidly growing area of research. There is a great number of environmental, economical, medical and even military applications that uses sensors for controlling and monitoring data and measurements. Faced with issues of low power consumption and wireless networking, proper operations and reliability of sensor results are very significant.

The goal of this thesis is to detect duplicate values inside a network. Duplicate values can be considered to be ids that are detected more than one time in the network. The identification of duplicates in a WSN is important in order to have accurate final results, while maintaining low power consumption and proper operation of the network. For aggregate queries such as MIN and MAX, which are monotonic and exemplary, the existence of duplicates is fault- tolerant. But for duplicate-sensitive aggregates such as COUNT, AVG or SUM, it gives incorrect final results.

By taking advantage of the hierarchical topology, which is created, during the dissemination of the query, nodes can detect if they have duplicate values at their sub-trees. If a node detects a duplicate value and is the highest in the hierarchy of the tree for this duplicate value, then it keeps track of this value and merges measurements of this duplicate value.

In order to keep energy consumption low, we approached our algorithm with three different methods and their combinations. These methods are Full Data, Delta and Bloom Filter methods. Through the experiments that we did, we present the gain for each one of these methods. Reliability and energy consumption are our quantities of comparison. Algorithms were tested and executed at the WSN of lab SoftNet.

Περίληψη

Τα ασύρματα δίκτυα αισθητήρων (ΑΔΑ) χρησιμοποιούνται ευρέως στη σημερινή εποχή και είναι μια περιοχή συνεχώς αναπτυσσόμενη. Υπάρχει μεγάλος αριθμός από περιβαλοντικές, οικονομικές, ιατρικές, ακόμη και στρατιωτικές εφαρμογές, οι οποίες χρησιμοποιούν αισθητήτες για τον έλεγχο και την παρακολούθηση δεδομένων και μετρήσεων. Επειδή αντιμετωπίζουν θέματα χαμηλής κατανάλωσης ενέργειας και ασύρματης δικτύωσης μεταξύ τους είναι σημαντική η εύρυθμη λειτουργία τους και η εγκυρότητα των αποτελεσμάτων τους.

Στόχος της διπλωματικής αυτής εργασίας είναι η ανίχνευση διπλότυπων τιμών μέσα στο δίκτυο. Διπλότυπα μπορούν να θεωρηθούν τα *ids*, τα οποία ανιχνεύονται πάνω από μία φορά μέσα στο δίκτυο. Η αναγνώριση διπλοτύπων σε ένα ΑΔΑ είναι σημαντική, ώστε να έχουμε αξιόπιστα τελικά αποτελέσματα, διατηρώντας, ταυτόχρονα, χαμηλή κατανάλωση ενέργειας και σωστή λειτουργία του δικτύου. Για τα συναθροιστικά επερωτήματα MIN και MAX, τα οποία είναι μονοτονικά και ενδεικτικά, η ύπαρξη διπλοτύπων σε αυτά είναι ανεκτική σε σφάλματα. Αλλά για συναθροιστικά επερωτήματα, τα οποία ειναι ευαίσθητα σε διπλότυπα, όπως τα *COUNT*, *AVG* και *SUM*, τα τελικά αποτελέσματα είναι εσφαλμένα.

Εκμεταλεύοντας την ιεραρχική τοπολογία, που δημιουργείται κατά την μετάδοση του επερωτηματος, οι κόμβοι μπορούν να ανιχνεύσουν αν έχουν κάποιο διπλότυπο στο υποδέντρο τους. Αν ένας κόμβος ανιχνεύσει ένα διπλότυπο και βρίσκεται στην πιο ψηλή ιεραρχία του δέντρου για το συγκεκριμένο διπλότυπο, τότε παρακολουθεί το διπλότυπο αυτό και συγχονεύει τις μετρήσεις του διπλοτύπου αυτού.

Για να διατηρήσουμε την κατανάλωση ενέργειας χαμηλή, προσεγγίσαμε τον αλγόριθμό μας με τρείς διαφορετικές μεθόδους και τους συνδιασμούς τους. Οι μέθοδοι αυτοί είναι οι *Full Data*, *Delta* και *Bloom Filter* μέθοδοι. Μέσω των πειραμάτων που πραγματοποιήθηκαν, παρουσιάζουμε τα οφέλη για την κάθε μία μέθοδο. Οι ποσότητες σύγκρισης είναι η αξιοπιστία και η κατανάλωση ενέργειας. Οι αλγόριθμοι δοκιμάστηκαν και εφαρμόστηκαν στο ΑΔΑ του εργαστηρίου *SoftNet*.

Acknowledgements

First of all, I would like to thank my Thesis supervisor Prof. Antonios Deligiannakis for his continuous guidance, support and trust during our cooperation. I would also like to thank not only the rest of the members of my examination committee Prof. Aggelos Bletsas and Prof. Minos Garofalakis for the time they spent on reading and evaluating this diploma thesis, but also Prof. Michail Lagoudakis for his useful advice and knowledge he shared with all of us during our studies in TUC.

Next, I would like to thank Antonios Igglezakis for his help and technical support, as well as his great ideas. I will not forget the assistance he gave me during the hard times of this thesis.

Mikro Mariaki is the next person that I would like to thank. She was always next to me, especially in the bad times. By her care and support I managed to finish University without having a heart attack! ☺

This work is also due to my friends from Chania, Bouklou, Paoki, Karma co. , Psari (or M.. M..) , Koukos, Bouklas, Billy, Ksanthia, Filitsa and Christos. They have assisted me during all the years of my studies. I would like to thank them for all of the unforgettable moments that we shared in this beautiful island all these years.

Last but not least, I would like to thank my family and especially Theano, for all of the support, constant encouragement and never-ending love that they show me every day. All of my work is dedicated to them.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Contribution

Wireless Sensor Network is composed of a great number of small sensor nodes and can be installed in a big terrain with one or more base stations. Sensor networks represent a rapidly growing network technology, because they have wireless communication, they can work with a small battery, they are low budget, and they can sense temperature, humidity, movement and many more abilities. However, they have limited memory and computing power.

Sensors, in today's society, can be used everywhere in many monitoring and controlling applications. Medicine uses sensors for medical observations and health. Military uses them for tracking and detection of enemy vehicles in the war zone. Moreover, sensors are applied in farming, production monitoring or even predict extreme weather conditions.

Some of these applications, though, are duplicate sensitive. For example, if nodes are applied in an environment and detect objects, there is a probability that two or more nodes are very close to each other and observe the same object. Nodes must understand, that the object they are detecting, is the same and they have to send one measurement of this object to the base station.

The contribution of this thesis is to detect duplicate values inside a network and merge their measurements into one. By taking advantage of the hierarchical topology, which is created during the dissemination of the query, nodes can detect, if they have duplicate values at their sub-trees. If a node detects a duplicate value and is the highest in the hierarchy of the tree for this duplicate value, then it keeps track of this value and merges duplicate value measurements.

Using three different methods and their combination we examine, which methods are preferred to be used most, and which are their drawbacks. The methods, which this algorithm uses are the Full Data method, the Delta method and the Bloom Filter method and their combination. Nodes that select to use Full data method send all received data to their parents or children, respectively, in phase one and two. When delta method is selected, nodes send their new ids of objects they observe and remove the old ones that do not observe anymore. By Bloom Filter method, nodes transmit bloom filters containing the ids of the objects, that nodes observe. An analysis of these methods will be shown in Chapters 4 and 5.

1.2 Roadmap

In Chapter 2, we describe Sensor networks and mote's architecture. In this Chapter, is also included the description of TinyOS and simulator TOSSIM, which simulates entire TinyOS appications by replacing components with simulation implementations. In the end of this Chapter, the Tiny Aggregation Service is noted as is essential for sensor networks. In Chapter 3, there is a brief survey on some related work of our algorithm. This includes the SenseJoin application from publication [1] and the application of Sketches in WSNs. At this point, Bloom Filters are introduced, as they will be used in Chapter 5. In Chapter 4, we present Full Data method, where nodes send all received data including their own. This Chapter is followed by Chapter 5, where the Delta and Bloom Filter methods are being analyzed. By these two methods, our goal is to transmit less data over the tree. Finally, Chapter 6 includes the experiments, which were accomplished for the requirements of this thesis, while Chapter 7 consists of not only of our outcome and conclusions, but also, some possible future work and applications.

Chapter 2

Sensor networks and TinyOs

2.1 Introduction

WSNs develop a technology, where many applications are being applied in. A network of sensors is usually consisted of a great number of sensor nodes placed in a large area. These sensors are low cost, but their lifetime is limited because of energy consumption. Sensors are used in a wide variety of applications, like medicine, environment, security and traffic jam. Apart from this, motes have restrictions in size and price and these limitations affect energy, memory and computing power. A nutshell of a sensor network can be seen in the Figure 2.1.

TinyOs is a free open source software component- based operating system, which is used in WSNs. NesC is the programming language, which is used in TinyOs, and it is a dialect of C optimized for the memory limits of sensor networks. As previously mentioned, there are components, that are connected to each other using interfaces. A component can use and can be used by other components.

There are some interfaces and components, which are provided by TinyOs, for common abstractions, such as packet communication, sensing, timing, storage and so forth. There is also the TinyOs API, by which a person can interact with the motes through computer by sending and gathering data from motes through serial communication. TinyOs, also, provides a simulator, where a programmer can execute code before installing it to the motes. Simulator is being analyzed at Section 2.5.

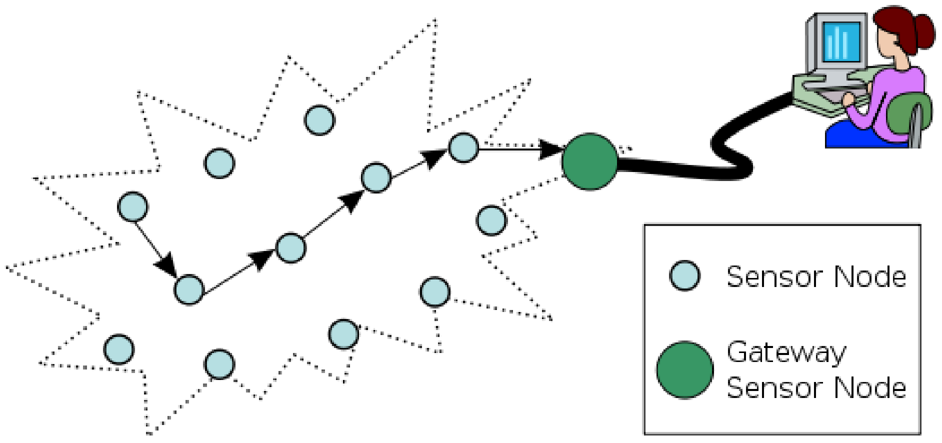


Figure 2.1: A Wireless Sensor Network

2.2 Architecture of Sensors

In this thesis, Iris motes are applied and an image of them can be seen in Figure 2.2. As mentioned in the previous paragraph, a mote does not include only sensors. Iris is a 2.4 GHz mote module used for enabling low- power wireless sensor networks. It, also, provides not only 250 Kbps high data rate Radio but also wireless communications with every node as Router Capability.

There is a RAM memory of $8K$ bytes and a flash memory of $128K$ bytes. A low-power micro controller ATmega1281 is being used, as well as, a collection of sensors that can be installed on sensorboard. The sensorboard, which was used, is MDA100CB 2.3 and it is consisted of two sensors, one photoresistanse and one thermistor. The first one measures brightness and the second one temperature values. A more detailed description about mote's characteristics is given at [2].

Furthermore, any mote can become a base station and gather all data onto a PC or other computer platform. When Iris is connected to a standard PC interface or gateway board, can function as a base station. An interface board which was used for this thesis is MIB520CA 2.4, which provides a serial/ USB interface for both programming and data communications. There is also MIB600 offered by MEMSIC that offers a stand-alone gateway solution for TCP/IP - based Ethernet networks.

The components of the wireless communication of Iris sensor are designed for low

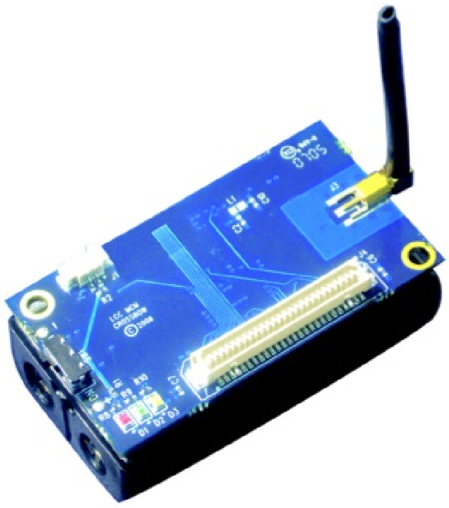


Figure 2.2: Memsic Iris mote

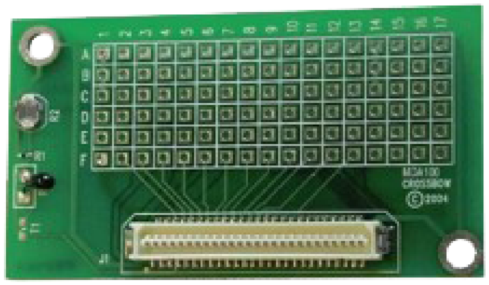


Figure 2.3: Sensorboard MDA100CB of Iris mote

energy consumption and can change frequency and transmitted energy through software. This particular sensor uses Atmel RF230 which works in frequencies between 2400MHz and 2483.5MHz. It is also compatible with protocol IEEE 802.15.4 and can achieve transmission speed up to 250$Kbps$. According to IEEE 802.15.4 the channels have bandwidth 5MHz, so Atmel RF230 can work in channels 11 (2405MHz) to 26(2480MHz). In a receiving node the current draw is equal to 16mA.

Figure 2.4: MIB520CA Mote Interface Board

2.3 Architecture of TinyOs

TinyOS is an open source, BSD-licensed operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitious computing, personal area networks, smart buildings, and smart meters. It has a component-based programming model, codified by the NesC language, a dialect of C. TinyOS is not an OS in the traditional sense; it is a programming framework for embedded systems and set of components that enable building an application-specific OS into each application. Traditional Operating Systems are too demanding to use them in sensors.

Sensors don't have a large memory and storage, which are the requirements of traditional OS. Additionally, a standard Operating System is undoubtedly too complex and power consuming for a wireless device. Hence, TinyOS is ideal for running on low power motes, as it is bundled with only the required components and only them.

A basic aspect of TinyOS and nesC is that there is a certain separation of construction and composition. Components include the names of their interfaces and their implementation. Apart from the applications, the libraries are also bundled in components. In more detail someone can see nesC language through this referenced Book [3].

2.3.1 Interfaces

A nesC interface definition specifies a bi-directional interaction between two components, known as the provider and the user. Interfaces describe a logical related set of commands and events. When a component provides an interface, it provides a functionality to the component it is used by it. The used interfaces represent the functionality components, which are needed in order to be able to perform the task they are intended to.

In most of the cases, the provider component provides some service, for example sending messages over radio, and commands represent requests, event responses. Commands are function calls from a user, who wants to use a provider component.

2.3.2 Modules and Configurations

Components can be separated into two different kinds, which are modules and configurations. Configurations describe the wiring between components. In general , they assemble all the components which are used. On the contrary, modules are implementations, which define functions, and commands and allocate state, while configurations only connect the declarations of different components. Typically, the process of connecting interfaces used by a set of components with interfaces provided by others is called wiring.

2.3.3 Singletons and Generic Components

Components can be separated in singletons and generic. Normally, components are singletons, which means that, the name of a component is a single entity in a global namespace. Generic components are not singletons, because they can have multiple instances as they can be instantiated within a configuration. In addition, singletons are unique and as a result they can only exist once.

2.3.4 Tasks and Events

Code in nesC is separated into synchronous code, where code is executed only inside tasks, and asynchronous code, where code can be executed not only inside tasks, as well as, in interrupt handlers. Asynchronous coding must be defined by key word async in interfaces and in modules, where it is applied.

Tasks cannot preempt each other, on the other side, events can preempt tasks since they have higher priority and events can preempt each other once they are enabled. This is a very crucial and critical property, because it can influence some tasks that have to be completed before an event. There is also a problem, when two or more events are fired at the same time.

For example, when two messages arrive to a mote at the same time, the appropriate event will get fired twice overriding the variables formerly containing useful information. In this way, a mote will lose data from one of the two messages leading to false results. On the other hand, if the data of the first received message are instantly copied to other memory locations, as soon as the events are fired and tasks are posted to process these data, they will not be overridden and no data loss will occur. TinyOS scheduler runs tasks one by one in the order they are posted until their completion.

2.4 Power Consumption

It should be clear until now that energy consumption is very important in sensors. This happens, because sensors are asked to work in remote areas for a long time, so they should be energy efficient. The elimination of power consumption is the main concern of all publications in the field of Wireless Sensor Networks. As it will be described later, in this thesis there are some mechanisms applied for minimizing energy consumption.

Towards minimization of power consumption, there exist two different mechanisms. First of all, as in Chapter 5 is mentioned, the limitation of transmitting data leads to less energy consumption by minimizing the number of calculations each mote has to accomplish. One other way to minimize power consumption is by increasing the amount of time the nodes remain inactive. This can be accomplished by turning on radio only when receiving and transmitting messages and turning it off the rest of the time.

2.5 Simulating in TinyOs (TOSSIM)

TOSSIM is a simulator of a Wireless Sensor Network, which is provided by TinyOS. It simulates entire TinyOS applications and works by replacing components with simulation implementations. Sensors, which are sometimes placed in uncontrolled physical environments like nature reserves or seismically threatened structures, need distributed algorithms for achieving efficient data processing. Their embedded nature makes controlled experiments a difficult task. TOSSIM simulates the TinyOS network stack at the bit level. This means that by the use of low-level protocols there can be experimentation in addition to top- level application systems can be achieved. By TOSSIM one can interact with sensors like they are in real world.

Therefore, TinyOs has developed a mote simulator, TOSSIM, to ease the development of sensor network applications. Thousands of nodes can be used through TOSSIM compiled with an additional parameter to make command code can be compiled. Compiling code to this simulator can only be applied to micaz platform. The command for compiling is:

$$\textit{make micaz sim} \tag{2.1}$$

Moreover, for the completion of the simulation it is necessary not only the topology of the network, but also the fading of the signal for each link and the noise model, which exists, when sending a message for each sensor. The topology, which is used in a network with 5 nodes is like this:

```
0 1 − 50.0
1 0 − 50.0

1 2 − 50.0
2 1 − 50.0

0 2 − 50.0
2 0 − 50.0

1 3 − 50.0
3 1 − 50.0

1 4 − 50.0
4 1 − 50.0
```

In the first and the second column there is a node id and in the third column there is the gain, meaning the fading of the signal. For the initialization of the simulation we use a python script which sets the given topology and creates noise model. There is a file named meyer-heavy.txt having a series of samples of noise, by which, through $createNoiseModel()$ function, occurs the creation of noise model for each node in network.

During simulation user can check the code with the help of debug messages, which are like printf messages. When code is installed to real sensors and not in simulation, these messages are ignored during compiling. This debug message can be defined as follows:

$$dbg(char^* \ stringID, \ const \ char^* \ format, ...)$$

By this function, user can print messages from the channel that is defined at $stringID$. Simulator disjoints these messages to channels depending on the value of this variable. For each different value of $stringID$, there is also a different channel. The choice of channels, that a user wants to observe, can be defined in python file, which initializes the simulator. The message of $dbg()$ function is created in the same way as $printf()$ works in C language. $Dbg()$ function also adds in front of each message the characteristic "DEBUG(TOS_NODE_ID) ", where TOS_NODE_ID represents the unique id of each sensor. Python script file can be executed through this command:

$$python \ mySimulation.py$$

2.6 TAG (Tiny AGregation Service for Ad-Hoc Sensor Networks)

Tiny Aggregation Service, or otherwise TAG [4], states that it is unnecessary a sensor to report its entire data stream in full fidelity. Moreover, as previously mentioned, in a sensor network each message transmission is an important energy- expending operation. Therefore, data aggregation can be used to summarize information which is collected from sensors. Having a workstation or a base station on a sensor working as a sink, the user can be connected to the sensor network.

Base station is disseminating a simple SQL-like aggregate query across all sensors of the network. This happens at an early phase. When all nodes have received this message,

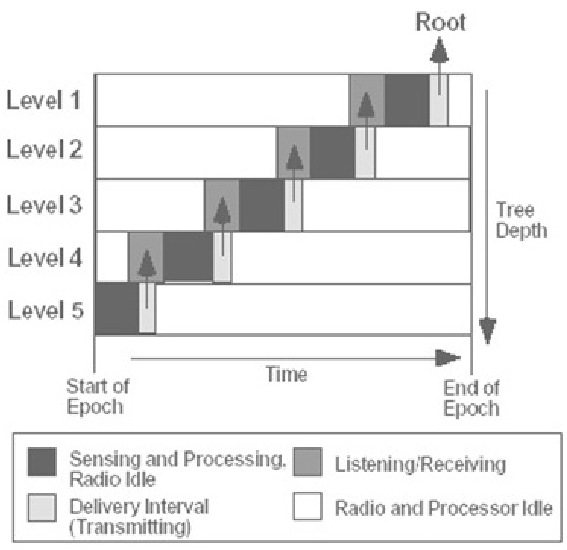


Figure 2.5: Transmitting Data in TAG

at a later phase each node transmits its data. Nodes that receive data of other sensors, combine their own data with the received ones and then they forward combined data over the spanning tree. The spanning tree consists of nodes, which are the sensors, and the root of the tree is the base station. A more detailed analysis of the construction of the tree is given in Section 4.6. The main motive of TAG is the processing of data inside the network.

Each node belongs to a level and time is divided in epochs. In each epoch, sensors route data or aggregate values of their subtree to user through the aggregation tree. Each epoch is divided into smaller slots. In the first slot the node listens and receives data from its children. In the second slot, the radio is turned off and it starts sensing and editing data. Finally, in the third slot the node transmits its data to its parent. In the following figure we can see time divided into epochs and epochs divided into smaller slots. In each level there are nodes and in the base station there is the root node.

Another thing that should be noted about Figure 2.5 is that time slots are not exactly sequential. Parents do not start listening at the exact second when their children start transmitting data. This happens because there are some limitations in the quality of the clock synchronization algorithms. Therefore, parents should start listening before the children start sending their data.

At this point, it should be mentioned that a node should choose very carefully the time slot in which radio remains active. This time slot should not be too long, because this is not efficient for energy consumption. On the other hand, time slot should not be very short either, because messages are very likely to be lost and not received. In conclusion, time slot should be very carefully chosen and tree length should also be taken into consideration.

2.7 Compiling and Programming motes

Concerning compiling the code, TinyOS uses a series of Makefiles. Basic Makefile is the one which is defined by the variable MAKERULES. Programmer must write a Makefile, defining the main configuration component of application, but also the sensorboards, which may be connected to sensors. Furthermore, #include and #define tags must be included in this file in order to use other libraries for this application. The Makefile which is used in this implementation is show in Algorithm 1.

Compiling the application we use the following command:

$$make\ \ < platform >$$

This command is used later when we want to install code to sensors. In platform we type the model of platform, which is used. In our case we typed iris. For the installation of the code in iris mote, which is connected to the computer via programming board, we use the following command in terminal:

$$make\ \ < platform >\ \ install, < nodeid > < progmethod >, < serialport >$$

For the sensors, which were used for this thesis, we ran these commands:

$$make\ iris$$
$$make\ iris\ install, < nodeid >\ \ mib520, /dev/ttyUSB0$$

Algorithm 1 Makefile

```
COMPONENT=SRTreeAppC

CFLAGS += -DSERIAL_EN -DTOSH_DATA_LENGTH=64
CFLAGS += -I$(TOSDIR)/lib/printf -DPRINTFDBG_MODE -DNEW_PRINTF_SEMANTICS
CFLAGS +=-I%T/sensorboards/mda100/cb

CFLAGS += -DBloom_En
CFLAGS += -DDELTA_MODE
CFLAGS+= -DFULL_MODE

CFLAGS += -I$(TOSDIR)/lib/ftsp/ -DTIMESYNC_RATE=1

##includes for simulation Extra implementations
CFLAGS += -I../simulation_extras/DummySync -I../simulation_extras/ConstantVectorSensor -I../simulation_extras/KelvinTempSensor -I../simulation_extras/RandomSensor

CFLAGS += -I$(TOSDIR)/lib/net/
CFLAGS += -I$(TOSDIR)/lib/net/ctp
CFLAGS += -I$(TOSDIR)/lib/net/le

SENSORBOARD=mda100
include $(MAKERULES)
```

While developing tinyOS applications, it is wise to mention Yeti. Yeti [5] is a very helpful plugin of eclipse IDE and it is useful for programmers to write code in nesC. Yeti isn't currently under development but it can help users compile and organize their code. Another way to do this is through gEdit or Kate. They are both editors compatible with nesc language, where code can be highlighted and indented.

Chapter 3

Related Work

The problem of energy efficiency is a familiar problem to all the algorithms in WSNs. Implementations, which are efficient, exist in data management systems for WSNs. In Section 3.1 Sens-Join will be described, which is an implementation that supports join operations well locally through nodes. At a later Section 3.2, Sketches will be described, an implementation which is familiar with Bloom Filters. Bloom filters are being used in this thesis for data compression (3.3). We review all the related work in the next sections.

3.1 Sens-Join

Sens-Join is an energy-efficient general- purpose join method for sensor networks. It is presented in the paper "Towards Efficient Processing of General- Purpose Joins in Sensor Network" [1]. The main scope of Sens-Join is to improve join queries using a minimum amount of communication. The main problem inside a network, when there is a join query, is that nodes do not know if their data contributes to query result. The first thing someone can think of is to have a global matching. But this is quite expensive and for that reason not efficient.

External Join, which is also presented in this paper, consolidates the data at the base station. Each node sends to its parent its own data combined with the received data and the filtering is done at the base station. This method is sometimes optimal in case when there is a very low selectivity. In addition, it sends too much unnecessary data and

Algorithm 2 SENS-Join (ref. [1])

```
//At the end of the query's dissemination;
sleepUntilNextStep(); //wait for beginning of SENS-Join

Join-Attribute-Collection:
ReceivedData = collectMessagesFromChildren();
T = constructTupleFromLocalSensorData();
//returns T = NULL if (T $\notin$ A) and (T $\notin$ B)
ForwardJoinAttrValues(ReceivedData, T); //cf. IV-B
sleepUntilNextStep();

Filter-Dissemination:
JoinFilter = receiveFromParent();
ForwardJoinFilter(JoinFilter); //cf. IV-C
sleepUntilNextStep();

Final-Result-Computation:
ReceivedData = collectMessagesFromChildren();
ForwardCompleteTuples(ReceivedData, T); //cf. IV-D
```

nodes that are higher at the tree must transmit more data. This means that their energy consumption is higher than nodes that are at a lower level of the tree.

Sens-Join uses a filtering inside the network with the purpose of sending less data at the base station. This method is split into two phases, pre-computation and subsequent final result computation. In the first phase, all nodes send their join attribute until all data has arrived at the base station. Base station joins all data and then yields a list of join attributes that contribute to the query result. In the meantime, it disseminates the filter, which contains the joined attributes that contribute to the query result to its children. Subsequent final result computation gathers complete tuples of nodes at the base station. Nodes, that their join attribute belongs to the received filter, disseminate complete tuples of their measurements to their parent. When all the data has arrived to base station, the final result is computed. Algorithms 2, 3 and 4 present Sens-Join from the point of view of a single node.

Algorithm 3 ForwardJoinAttrValues(Set $\{S_1, ..., S_n\}$, Tuple T) (ref. [1])

```
//Si: data received from child i

Set_Of_Full_Tuples FullTuples =$\emptyset$;
Join_Attr_Structure JoinAttTuples = $\emptyset$;
for all $S_i \in \{S_1, ...S_n\}$
     if ($S_i$ is Set_Of_Full_Tuples)
          FullTuples = $Union_{Full\_Tuples}$(FullTuples, $S_i$);
     else
          JoinAttTuples = $Union_{Join\_Atts}(JoinAttTuples, S_i)$;

if (Size($\{S_1, ..., S_n\}$) + Size(T) $\leq$ $D_{max}$) && ($\forall S_i \in \{S_1, ..., S_n\}$: $S_i$ is Set_Of_Full_Tuples)
     //use Treecut: hand over data to parent and go to sleep
     FullTuples = $Insert_{Full\_Tuples}$(FullTuples, T);
     send(FullTuples, parent);
     //query execution is complete:
     exitQuery();
else
     store FullTuples; //act as proxy for received complete tuples
     store JoinAttTuples as "SubtreeJoinAtts";
     ProxyJoinAttTuples = $\pi_{JoinAttr}$(FullTuples);
     JoinAttTuples = $Union_{Join\_Atts}$(JoinAttTuples, ProxyJoinAttTuples);
     T' = $\pi_{JoinAttr}$(T);
     JoinAttTuples = $Insert_{Join\_Atts}$(JoinAttTuples, T');
     send(JoinAttTuples, parent);
     //sleep until next step - cf. Figure 1
```

3.2 Sketches

In a sensor network, the sensors create streams of data from the observations and they have to transmit them to their parents. These streams can reach a great size, so it is inefficient to transmit all these data. This means that in many applications it is unnecessary for each node to report the whole stream in full fidelity. The purpose of

Algorithm 4 ForwardJoinFilter(Join_Attr_Structure Filter) (ref. [1])

```
1: SubtreeFilter = $Intersect_{Join\_Atts}$(Filter, SubtreeJoinAtts);
2: if (SubtreeFilter $\neq \emptyset$)
3:     //send join-attribute tuples of subtree to children
4:     broadcast(SubtreeFilter);
5: else
6:     //do nothing - the subtree won't be involved in final step
7:     //sleep until next step - cf. Figure 1
```

using Sketches is to minimize transmitted data for efficient energy consumption. This approach [6] is being used under networks that are ideal. Ideal networks are considered to be the ones that message loss rate is very low, which means that messages are not lost.

Sketches are being applied in order to estimate the number of distinct items in a database or a stream, while using only a small amount of space. Sketches have the same goal as bloom filters in this thesis, they both are being used to reduce data. The only difference is the way that are both used. Sketches here are used in COUNT and SUM queries.

When applying Sketches to COUNT queries, each sensor computes m independent sketches by using m different binary hash functions. With the use of routing algorithm each node transmits sketches towards root by sending them to its parents (a node can have more than one parents). When a node receives sketches from its children, unites them with its own k sketches and finally root makes the estimation of COUNT query result.

Flajolet Martin Sketch estimates the number of distinct items in a stream of values from $[0, ..., M-1]$. Given a multi-set M, the FM sketch of M, denoted as $S(M)[0, ..., k-1]$ are initialized to zero and are set to one using a random binary hash function h applied to the elements of M. Formally,

$$S(M)[i] \ \equiv \ 1 \ \ iff \ \exists \ x \ \in \ M \ s.t. \ min\{j \mid h(x,j) = 1\} = \ i \tag{3.1}$$

By this equation, each element x is able to set a bit of FMSketch $S(M)$ to one - the minimum i for which $h(x,i) = 1$. The least significant bit of $h(x)$, which is the rightmost, is equal to 1 and the remainings are equal to zero. The probability of each of the bits at positions $0..K-1$ being equal to zero is $\frac{1}{2}$. Because of uniformity, it is valid to prove:

$$Prob[BITMAP[k] = 1] = Prob[10^k] = \frac{1}{2^{k+1}} \tag{3.2}$$

Because of equation 3.2, we come to the conclusion, that the least significant bits have greater probability to be equally to one in addition to the most significant bits. In [7] it is shown that the estimation of the distinct elements that are added to sketch can be calculated like this:

- Let R be the position of the rightmost zero bit in BITMAP
- It is proven in [7] that $E[R] = log(\phi d)$ where $\phi = 0.7735$, or else $d = \frac{2^R}{\phi}$.

This estimation may have many errors, but if there are plenty of sketches there can be a better approach of d. The value of d can show the number of distinct values inside a set M.

When there is SUM query, we can calculate the result by using Sketches like when we had COUNT with some differences. If a node wants to sum a x value then instead of making one BITMAP, it makes x and unites them all together. This is a naive solution but it works just like COUNT queries in the end. This can be succeeded more efficiently and much more information is presented in Paper [6].

3.3 Bloom Filters

A bloom filter provides a simple space-efficient randomized data structure for representing a defined set in order to support membership queries. It is very useful in WSNs, because a bloom filter compresses data into a bitmap array and nodes send less data. Bloom filters allow false positives, but the space savings often outweigh this drawback, when the probability of an error is controlled. Testing the insertion and membership in bloom filters, implies an amount of randomization, since elements are transformed using one-way hash functions. Testing for the presence of elements, that have actually been inserted in the filter, will always give a positive result; there are no false negatives. In recent years they have become popular in the networking literature and are used in many related

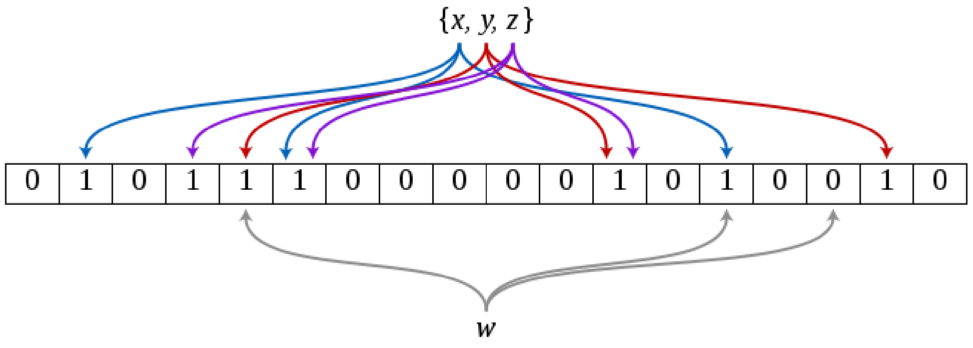


Figure 3.1: Bloom Filter in Action

applications. This article [8], represents the mathematical and practical framework of bloom filters and some other important variations. This implementation is influenced by this article, in order to choose the size of bloom filter and the necessary number of hash functions.

As it is already noted, bloom filters are used to represent a set $S = \{x_1, x_2, ..., x_n\}$ of n elements. The size of bloom filter can be set by m bits, where all bits are initialized by zero. With the use of the hash functions with range $\{1, ..., m\}$, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution. For each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

When it is necessary to be examined, if an element x is in a set S, it is inserted in k hash functions to get k array positions. If any of the bits at these positions are zero, the element is definitely not in the set. Otherwise, when it was inserted, all the bits would have been set to 1. However, there is a possibility of false positive, because bits have by chance been set to 1 during the insertion of other elements.

In Figure 3.1, there is an example of bloom filter with 18 bits initially set to zero. This bloom filter represents the set of $\{x, y, z\}$ elements. Each colored line represents the positions of each element in the bit array. The number of lines represent the number of hash functions used and in this case three hash functions are being used. The set of elements are mapped into bloom filter through the hash functions and set to 1. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit array position containing zero.

It is assumed that a hash function selects all array positions with equal probability.

After all elements of set S are hashed into the bloom filter, the probability that a specific bit is still 0 is :

$$p = e^{\frac{-kn}{m}} \tag{3.3}$$

the probability that it is 1 is therefore :

$$1 - p = 1 - e^{\frac{-kn}{m}}. \tag{3.4}$$

The probability of all bits set to 1, which means all hash functions lead to bits equal to 1 and claim that an element is in the set, is given from this:

$$f = (1-p)^k = (1 - e^{\frac{-kn}{m}})^k. \tag{3.5}$$

Considering the above, we have that the probability of false positives decreases as m, which is the number of bits in the array, increases. Probability increases if n increases too, which means, that when the number of inserted elements increases, the probability of false positive goes higher. The number of hash functions needed that minimizes the probability of false positive, depends on m and n. The optimal number that minimizes f as a function of k is found by taking the derivatives. More conveniently, note that $f = \exp(k \ln(1 - e^{\frac{-kn}{m}})) = k \ln(1 - e^{\frac{-kn}{m}})$. Minimizing the false positive rate f with respect to k there is :

$$\frac{\partial f}{\partial k} = \ln(1 - e^{\frac{-kn}{m}}) + \frac{kn}{m} \frac{e^{\frac{-kn}{m}}}{1 - e^{\frac{-kn}{m}}} \tag{3.6}$$

It is easy enough to discover that the derivative is 0, when $k = \frac{m}{n} \times \ln 2$. Consequently, in order to have low false positive rates, we have to consider how many hash functions we must have. The required number of bits m given n as well as a desired false positive probability p, and assuming the optimal value of k is used, can be computed by substituting k value in the probability expression above (3.5) :

$$f = (1 - e^{-(\frac{m}{n}\ln 2)\frac{n}{m}})^{(\frac{m}{n}\ln 2)} \Leftrightarrow \tag{3.7}$$

$$\ln f = -\frac{m}{n}(\ln 2)^2 \Leftrightarrow \tag{3.8}$$

$$m = -\frac{n \ln f}{(\ln 2)^2} \tag{3.9}$$

This means that, if there is a desired false positive probability f that bloom filter must have, then the length of the filter must be fixed. M, which is the length of filter, must be proportionate to the number of elements n being inserted in filter.

More Bloom Filters

There are more bloom filters proposed by many others, like Scalable Bloom Filters [9] or Stable Bloom Filters [10].

Scalable Bloom Filters can adapt dynamically to the increased number of elements stored, while retaining a low probability of false positive. This method ensures that the maximum false positive probability is being set from the start and it is independent from the number of elements which are inserted. It is based on standard bloom filters with increasing capacity.

Stable bloom filters are used in streaming data. The idea is to clear all stale data and make room for more recent elements. Streaming data cannot be stored, because the stream can be infinite. Since stale information is evicted, stable bloom filters create false negatives, which do not exist in standard bloom filters. This method is superior to standard bloom filters in terms of false positive rates and time efficiency, because it gives only a small space and an acceptable false positive rate.

Chapter 4

Design and Implementation

4.1 Introduction

In this Chapter the design and implementation of the application Duplicate Detection is described. As already mentioned, the main object of this thesis is to detect duplicate values of nodes inside a network. Sensors are assumed to be hierarchically organized in a collecting data tree. Each node has only one father. Nodes should be able to detect objects with a specific id and keep measurement for this id.

First of all, nodes have to be synchronized with each other. When this is accomplished, the creation of the tree, which will be the basis of our implementation for transmitting and receiving data, starts. Later, in this chapter, the execution flow and all the messages that are being exchanged among the neighbor nodes and the base station will be described. In order to execute the final results of a query, three phases are preceded, which are analyzed in a following section.

4.2 Duplicate Detection

As mentioned in Chapter 1 the main part of this thesis is to eliminate duplicate values. The existence of duplicates can lead us to false results in aggregate queries. For aggregate queries such as MIN and MAX which are monotonic and exemplary, the existence of duplicates is fault- tolerant. But for duplicate-sensitive aggregates such as COUNT, AVG or SUM, it gives incorrect results.

The energy consumption dominates today's sensor motes. Most of the algorithms used in sensor network try to minimize the number of messages transmitted in order to save power. In this section, we will not use the two methods Bloom Filter and Delta, that will be discussed in Chapter 5, but all sensors send all their data, in spite of what they sent in a previous epoch. This method will be called in later references as Full Data method. As will be mentioned in Section 4.7, each node stores the messages, it sends in previous epochs. If the message is the same as in previous epoch, the node does not transmit any message. Its parent or its children, depending on which phase we are in, keep a backup of messages received at each round. When a node does not receive a message from a specific node, it regains the message last received from this node. Consequently, the number of messages are transmitted and the power consumption is decreased.

While the tree is being constructed, a timer begins to count (in *milliseconds*) for every sensor. The time until timer *myTimer* fires an event, depends on the depth and node id of the sensor. When the event from *myTimer* occurs, there is a Finite-State-Machine (FSM), as shown in Figure 4.1, which separates the execution of the algorithm in four different states. These states are:

InitState : this state initializes variables of algorithm and subsystems

Phase1State : a sensor gets its measurement. In this state a new timer $Phase1Timer$ begins. When this timer fires, phase one begins of the algorithm, which is explained at the following subsection.

Phase2State : in this state a new timer $Phase2Timer$ begins. When this timer fires, phase two of the algorithm begins, which is explained at the following subsection.

Phase3State : in this state a new timer $TAGTimer$ begins. When this timer fires, phase three of the algorithm begins, where the execution of the query begins.

4.2.1 Phase 1 - Collection of data for the Detection of duplicate ids

In this phase, in a nutshell, nodes transmit the ids of the objects they track. Starting from the leaves, nodes send their own ids they observe to their parents. Intermediate nodes receive those id values transmitted from their children and unite them with their own ids of the objects they track. Once they have gathered values from all their children, they send only the distinct values of the united ids to their parents. This is repeated

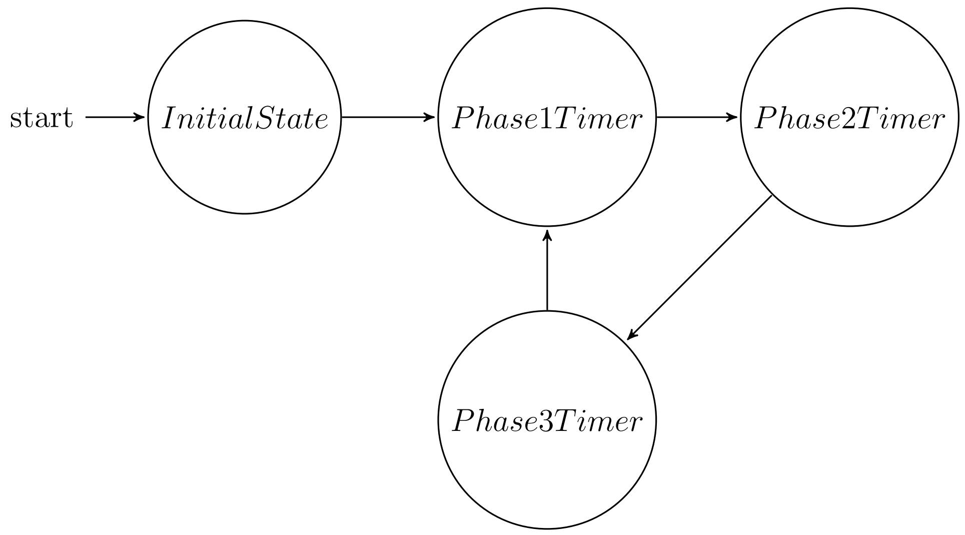


Figure 4.1: FSM of myTimer

recursively until all values are gathered at the base station. The type of message that is used in this phase is $SS\_Message$ that will be described later.

In particular, when timer $Phase1Timer$ fires, an event occurs. Initial state is STATE_START. In this state, the node does not do anything, but it just waits for $treceive$ time. It only contributes to the synchronization of sensors. After a few milliseconds, the timer fires again and the event re-occurs, we are at STATE_RECEIVE now. In this state a node receives measurements from its children and initializes the timer. When the timer fires, the state STATE_CALC is activated and the node, only if its not root, starts calculation tasks for the transmission of their data in the $sendSS\_Message()$ task. In the Algorithm 5 we can see the pseudocode and in Figure 4.2 the FSM of phase one.

In line 1 of Algorithm 5, there is a while-loop and node exits, when timer $Phase1Timer$ fires. In this loop, the node receives messages from its children and stores them to $joinAttr$ array. When the state changes to $STATE\_CALC$, if current node has received no message from one of its children, it retrieves backup values for those children it kept from the previous epoch, and saves them in $joinAttr$ (line 4). Later, each node saves its own id values, which it observes, in $joinSent$ vector. In this vector it also stores the distinct values of $joinAttr$ (line 6). By the time the calculation is finished, the node checks if vector $joinSent$ is equal to vector $previous\_joinsent$ (line 7), which is vector $joinSent$ in previous epoch. If these vectors are equal, then the node does not transmit

Algorithm 5 Phase1()

```
1: while ($synch\_Phase1 == STATE\_RECEIVE$) do
2:     Parse received message as vector $joinAttr$ of measurements
3: end while
4: For each child that didn't send message, store the ids of the previous epoch to array $joinAttr$
5: Store current Node's measurement in $joinSent$
6: Compute distinct values of $joinAttr$ and Store to $joinSent$
7: if $joinSent == previous\_joinsent$ then
8:     Doesn't transmit $joinSent$ to parent
9: else
10:    $previous\_joinsent = joinSent$
11:    Send $joinSent$ to parent
12: end if
13: sleepUntilNextPhase()
```

joinSent, otherwise vector *previous_joinSent* is replaced by the current *joinSent*, and the node sends *joinSent* vector to its parent (line 11). Finally, the node closes its Radio and waits for *MyTimer* timer to fire again for the next phase.

4.2.2 Phase 2 - Node with the minimum depth

In this phase, nodes search to discover, if they have duplicate values. For detecting a duplicate value, the algorithm must find the node, which is at highest hierarchy in the tree, that receives an id more than one time. When the root node receives all the messages from phase one, it gathers all data and detects which ids has duplicates. Thereafter, it sends this list to its children to inform them that it is the highest node for these ids. This action is, also repeated by all nodes when they reach phase two. The nodes in this phase send messages of type *ForwardingMsg*, which will be described later.

More specifically, each node creates its own array, named *Duplicates*, which stores ids that has received more than one time and believes that it is the highest of all the nodes, that receives this duplicate value. In order to be certain, that each node tracks the correct duplicate value and there is no other node in a higher hierarchy, that tracks the same duplicate value, each node receives a message from its parent. This message

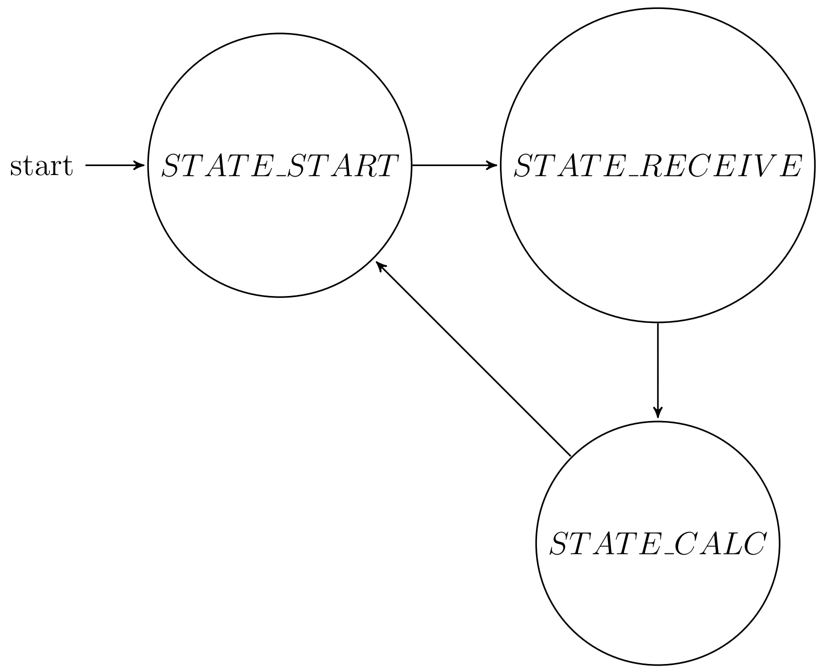


Figure 4.2: FSM of Phase1Timer, Phase2Timer and Phase3Timer

includes the duplicate ids that already exist at the upper tree. If ids are equal, the node erases those ids from array *Duplicates*, adds the remaining ones that exist in *Duplicates* and forwards the message to its children.

When timer $Phase2Timer$ fires, an event occurs. In this event there is FSM with initial state STATE_START. In this state, the node does not do anything, but just waits for *treceive* time. This only contributes to the synchronization of sensors. After a few milliseconds, when the timer fires again, the event re-occurs. We now are at STATE_RECEIVE state. Now in this state a node receives the duplicate list from its parent and initializes the timer. When the timer fires, state STATE_CALC is activated and the current node begins creating vector *Duplicates* with values of the previous phase. When this is completed, if node does not receive any message in this phase, then it assumes that message from the previous epoch is valid. Later, the node compares vector *Duplicates* to the duplicate list received from its parent. If an id from *Duplicates* is equal to the received list, we delete this value from the vector. When completing this task and *Duplicates* is not empty, the node unites this vector's values with the received list and forwards the new list to its children. FSM of this phase can be seen in Figure 4.2. This happens to each node until all nodes have forwarded their list. In the Algorithm 6 the pseudocode of phase two is presented.

Algorithm 6 Phase2()

```
1: **Parse** $joinAttr$ and **store** duplicates to $Duplicates$ including current node's ids
2: **while** ($synch\_Phase2 == STATE\_RECEIVE$) **do**
3:     **Save** received duplicate values at vector $receivedDups$
4: **end while**
5: **if** ( is root ) **then**
6:     $receivedDups = Duplicates$
7: **else**
8:     **for each** ($element$ in $Duplicates$) **do**
9:         If no message received from parent, **restore** $receivedDups$ from the previous
           epoch
10:        **if** $element \notin receivedDups$ **then**
11:            **Add** $element$ to $receivedDups$
12:        **else**
13:            **Erase** $element$ from $Duplicates$
14:        **end if**
15:    **end for**
16: **end if**
17: **if** $receivedDups == previous\_Dups$ **then**
18:     **Doesn't transmit** $receivedDups$ to children
19: **else**
20:     $previous\_Dups = receivedDups$
21:     **Sends** $receivedDups$ to children
22: **end if**
23: sleepUntilNextPhase()
```

From Algorithm 6 we can tell that this phase starts by detecting duplicates in the ids that were received at the previous phase from array *joinAttr*. Duplicate values are stored in vector *Duplicates* (line 1). In line 2, there is a while-loop and the node exits, when timer *Phase2Timer* fires. In this loop, node receives messages from its parent and stores data in the *receivedDups* vector. When the state changes to $STATE\_CALC$, if the current node has not received any message from parent and this node is not a root, it retrieves backup values that were kept from the previous epoch, and saves them in *receivedDups* (line 9). If the current node is root, then it stores vector *Duplicates* in *receivedDups*. In line 8 there is a for-loop, where it examines each element of vector *Duplicates*. If an element exists in *receivedDups* (line 12), then the node erases this element from vector *Duplicates*, because there is another node higher than this with duplicate to this value. Otherwise, the node adds this element to *receivedDups* (line 11). If *receivedDups* is not equal to the previous vector that the node sent, then it transmits the vector to its children (line 21), otherwise, the node does not send anything.

4.2.3 Phase 3 - Execution of Query

In this phase, there is some processing concerning the execution of the query. The implementation differs, if the query is a "*Select Star*" or if it is one of the rest of the aggregate queries. If the query is a "*Select Star*", then all nodes, except root, must transmit all ids that they observe, including their measurements. Otherwise, if the query is another aggregate, each node, except root node, must execute the query with measurements it receives. Here, it is important to notice that the node should not include measurements with ids that exist in *receivedDups*, which the node received in the previous phase. In other words, node should not include measurement of an id that is a duplicate, because this would lead to a wrong result. If the node is a root node, then it calculates the final result depending on the query. In this phase nodes send messages of type AvgMsg for "*Select Star*" queries and AvgMsgNotStar for the rest of the queries.

Starting from the leaves of the tree, they disseminate their data, depending on the query, to their parents. Intermediate nodes receive data from their children and do what was described earlier. This phase terminates, when all data has arrived at the base station, where the final result is calculated. Moreover, when a node receives a $AvgMsg$ or $AvgMsgNotStar$ message and in its data there is an id that exists in *receivedDups*, then the node calculates and stores the average value of measurements it has received for this id. When an id belongs to *receivedDups*, it means that it is a duplicate and the node must keep the average value of its measurements in order to avoid false results. Hence, if there is a "*Select Star*" query, the node forwards all ids and measurements that are received including the changes of possible duplicate values; otherwise, the node executes the query and forwards the query result.

In more detail, when timer *TAGTimer* fires, an event occurs. In this event there is FSM with initial state STATE_START. In this state, the node does not do anything but it just waits for *treceive* time. This only contributes to the synchronization of sensors. After a few milliseconds, when the timer fires again and the event re-occurs, we are in the STATE_RECEIVE state. In this state a node receives measurements from its children and initializes the timer. When the timer fires, state STATE_CALC is activated and the node, only if its not root, starts calculations for the executing query. As, previously described in Chapter 2, TAG [4], nodes try to minimize the payload length by executing the query. In this way, nodes send less information to the tree. This can only be applied to aggregate queries except for "*Select Star*". FSM of this phase can be seen in Figure 4.2. In the Algorithm 7 we can see the pseudocode of phase three.

Algorithm 7 describes the algorithm in phase three, when the query is a "*Select Star*". In line 1 the node is in STATE_RECEIVE and receives messages from its children storing them in vectors *avgid* and *avgvalue*. If its child did not send any message, then the current node would restore its data from a previous epoch. Received ids and measurements, including the current node's ids and measurements, are saved in vectors *query_listAvgid* and *query_listAvgvalue* (line 5). The node has to check if the received ids and its own ids are duplicates in order to calculate an average measurement of these ids. This is examined in lines 6 to 10, where the node compares ids of vector *query_listAvgid* and checks if they exist in vector *Duplicates*. If the current node is a root, then it only prints out the result. Otherwise, if *query_listAvgid* and *query_listAvgvalue* are equal to the vectors of the previous epoch, then the node does not disseminate any message (in line 15). In another case, the node adds to *value*1 and *value*2 fields of AvgMsgNotStar vectors *query_listAvgid* and *query_listAvgvalue* and transmits the message to its parent.

When the query is not a "*Select Star*", the algorithm differs, because inner and leaf nodes need to make some calculations depending on the query. Pseudocode of this part is shown in the Algorithm 8.

In Algorithm 8 is shown the algorithm in phase three, when the query is not a "Select Star". In line 1 the node is in STATE_RECEIVE and receives messages from its children storing them in vectors *avgid* and *avgvalue*. If its child does not send any message, then current node will restore its data from a previous epoch (line 4). Received ids and

Algorithm 7 Phase3 - “Select Star” query

```
1: while ($synch\_state == STATE\_RECEIVE$) do
2:     Parse received message as vectors $avgid$ of ids and $avgvalue$ of measurements
3: end while
4: For each child that didn't send message, restore the ids and measurements of the
   previous epoch to $avgid$ and $avgvalue$
5: Store $avgid$, $avgvalue$ and current Node's measurements and ids in $query\_listAvgid$
   and $query\_listAvgvalue$ respectively
6: for each ($element \in query\_listAvgid$) do
7:     if ( $element \in Duplicates$ ) then
8:         Calculate average value from $query\_listAvgvalue$ of this id
9:     end if
10: end for
11: if $is\ root$ then
12:     Show vectors $query\_listAvgid$ and $query\_listAvgvalue$
13: else
14:     if ( $query\_listAvgid == previous\_avgid$ ) && ( $query\_listAvgvalue ==$
        $previous\_avgvalue$ ) then
15:         Doesn't transmit to parent
16:     else
17:         $previous\_avgid = query\_listAvgid$
18:         $previous\_avgvalue = query\_listAvgvalue$
19:         Sends $query\_listAvgid$ and $query\_listAvgvalue$ to parent
20:     end if
21: end if
```

measurements including the current node's ids and measurements are saved in vectors *query_listAvgid* and *query_listAvgvalue* (line 5). The node has to check if the received ids and its own are duplicate in order to calculate an average measurement of this id. This is examined in lines 6 to 10, where node compares ids of vector *query_listAvgid* if they exist in vector *Duplicates*. Something that is different from the previous pseudocode is that the current node examines if each id, which is in *query_listAvgid*, exists in vector *receivedDups* (line 12). If this is true, then node inserts this id and the corresponding measurement in the $value1$ and $value2$ fields of AvgMsgNotStar message.

Algorithm 8 Phase3 - other queries

```
1: while ($synch\_state == STATE\_RECEIVE$) do
2:     Parse received message as vectors $avgid$ of ids and $avgvalue$ of measurements
3: end while
4: For each child that didn't send message, restore the ids and measurements of the
   previous epoch to $avgid$ and $avgvalue$
5: Store $avgid$, $avgvalue$ and current Node's ids and measurements in $query\_listAvgid$
   and $query\_listAvgvalue$ respectively
6: for each ($element \in query\_listAvgid$) do
7:     if ( $element \in Duplicates$ ) then
8:         Calculate average value from $query\_listAvgvalue$ of this id
9:     end if
10: end for
11: for each ($element \in query\_listAvgid$) do
12:     if $element \in receivedDups$ then
13:         Insert $element$ and its measurement to $value1$ and $value2$ fields of
            $AvgMsgNotStar$ message
14:     else
15:         Calculate result depending on the query and Store to $query\_result$
16:     end if
17: end for
18: if $is\ root$ then
19:     Show $query\_result$
20: else
21:     if ( $Current$ AvgMsgNotStar $msg$ == $Previous$ AvgMsgNotStar $msg$ ) then
22:         Doesn't transmit to parent
23:     else
24:         $previous\_avgid = query\_listAvgid$
25:         $previous\_avgvalue = query\_listAvgvalue$
26:         $previous\_query\_result = query\_result$
27:         Sends $query\_result$, $query\_listAvgid$ and $query\_listAvgvalue$ to parent
28:     end if
29: end if
```

Otherwise, node calculates the result depending on the query of the current epoch and stores the result in variable *query_result*. If the current node is a root, then it only prints out the result of the variable *query_result*. Otherwise, if *query_result*, *query_listAvgid* and *query_listAvgvalue* are equal to the variable and vectors of the previous epoch, then node doesn't disseminate any message (in line 22). In another case, node adds to *value*1 , *value*2 and *query_result* fields of AvgMsgNotStar vectors *query_listAvgid*, *query_listAvgvalue* and variable *query_result* and transmits the message to its parent.

4.3 Parameters of Program

Parameters, which are defined at the beginning of the program, are related to the construction of the sensor's tree and the basic characteristics that our algorithm needs.

In order to construct the sensor's tree, a value in some variables needs to be defined. First of all, there must be determined the maximum number of children and depth for each node. Moreover, when a node receives or transmits a message, it inserts it to a queue. It is important to define a size for these two queues, the sender and the receiver queue. All these parameters are settled on these variables : SENDER_QUEUE_SIZE, RECEIVER_QUEUE_SIZE, MAX_CHILDREN, MAX_DEPTH.

Detecting duplicates can be accomplished using three different methods. In order to choose which method to use, at the beginning of the program, we initialize some variables. The variables are : **FULL_MODE, BLOOM_EN, DELTA_MODE**. By the first variable, our algorithm uses the Full Data method, by the second variable, our algorithm chooses to use Bloom Filters and by the third, nodes send to their neighbours the delta values of the previous message they sent in previous epoch. These variables can be defined in file *Makefile*, which is presented in Algorithm 1.

There are, also, other variables, which are fundamental and it is worth mentioning them. Below we describe those parameters:

QUERY_ID : it defines the type of an aggregate query that needs to be executed. (1 for sum, 2 for max, 3 for min, 4 for count, 5 for count, 6 for "Select Star")

MEASUREMENT_TYPE : it defines a measurement that query needs for execution. (1 for photo, 2 for temperature)

EPOCH : this variable defines the number of epochs for each execution

```
typedef nx_struct RoutingMsg
{
        nx_uint8_t header;
        nx_uint16_t senderID;
        nx_uint8_t depth;
        nx_uint8_t query_id;
        nx_uint8_t epoch;
} RoutingMsg;
```

Figure 4.3: Definition of RoutingMsg

MAX_VLENGTH, MAX_VLENGTH_X, BFlength : these variables define sizes of arrays. BFlength represents the size of Bloom Filter.

4.4 Communication with nodes and base station

As previously stated, there will be three different phases before the calculation of the final result and, before that, there will be the creation of the collecting data tree. For all these proceedings there are different types of messages.

For constructing the tree and disseminating the query nodes we use the RoutingMsg message 4.3. The field **header** defines the type of message. In this case its value is equal to one. **SenderID** represents the id of the node that has sent this message. This field is necessary for the nodes that receive this kind of message. The nodes that receive it know that their parent is the node with id equal to **senderID**. With **depth**, a node can define its own depth based on this variable. **Query_id** and **epoch** are also needed for disseminating a query, so that all nodes will be informed about the type of the query that needs to be executed.

By the time nodes receive a $RoutingMsg$, they disseminate a $NotifyParent$ message until it reaches root node. Each node, that receives a $NotifyParentMsg$ from its subtree, forwards it to parent. This message includes the id of parent for each node that started disseminating the message and is essential for base station in order to learn the topology of the tree. The struct of this message is shown in Figure 4.4.

```
typedef nx_struct NotifyParentMsg
{
        nx_uint8_t header;
        nx_uint16_t senderID;
        nx_uint16_t parentID;
        nx_uint8_t depth;
} NotifyParentMsg;
```

Figure 4.4: Definition of NotifyParentMsg

```
typedef nx_struct SS_Message
{
        nx_uint8_t header;
        nx_uint8_t head;
        nx_uint16_t senderID;
        nx_uint8_t value[MAX_VLENGTH];
} SS_Message;
```

Figure 4.5: Definition of SS_Message

NotifyParentMsg includes **header**, which defines the type of message transmitted and it is equal to 11. Field **senderID** represents the id of the node that started disseminating this type of message and **parentID** defines the id of its parent. **Depth** represents the depth of the node.

When the tree is created, phase one begins, where all nodes send the ids of the objects that they observe. In this stage, all nodes send messages of type SS_Message 4.5. The field **header** defines the type of the message. In this case its value is equal to two. **Head** is a variable that defines the number of elements that are at the array **value**[**MAX_VLENGTH**]. **SenderID** represents the id of the node that has sent this message. This field is needed for the nodes that receive this kind of message. Nodes that receive it, store this id at the array children. By this array, nodes are aware of the nodes that are their children. Array **value**[**MAX_VLENGTH**] includes all the ids that each node has received from its children plus its own ids of the objects that it observes.

When phase one is completed, phase two begins, where nodes, starting from the root,

```
typedef nx_struct ForwardingMsg
{
        nx_uint8_t header;
        nx_uint16_t head;
        nx_uint8_t value[MAX_VLENGTH_X];
} ForwardingMsg;
```

Figure 4.6: Definition of ForwardingMsg

send the duplicate ids that exist. Details of this phase are being included at the above section. In this stage, all nodes send messages of type ForwardingMsg 4.6. The field **header** defines the type of the message. In this case its value is equal to three. **Head** is a variable that defines the number of elements that are at the array **value[MAX_VLENGTH]**. Array **value[MAX_VLENGTH]** includes all the duplicate ids that each node has received from its parent plus its own duplicate ids.

When phase two is completed, phase three begins, where nodes, starting from the leaves, send the ids and the measurements according to the query that they received. In this stage, all nodes send messages of type AvgMsg 4.7. If the query is not a "Select Star", then nodes send messages of type AvgMsgNotStar 4.7. The field **header** defines the type of the message. In this case its value is equal to four for all the queries except "Select Star" (when we do not have "Select Star" query, header is equal to nine). **Head** is a variable that defines the number of elements that are at the arrays **value1[MAX_VLENGTH]** and **value2[MAX_VLENGTH]**. Arrays value1 and value2 include the ids and the measurements that each node has received from its children plus its own ids and measurements. **SenderID** defines the id of the node that sends the message. In case where query is not the "Select Star", there is one more variable, **query_result**. Nodes execute the query, if possible, and store the result in this variable.

4.5 Synchronization of nodes

Synchronizing sensors is a basic condition for the right execution of an algorithm and for decreasing message loss rates. Moreover, with synchronization, nodes start sending routing messages by the time it has been defined to them, so that there is a stable list of

```
typedef nx_struct AvgMsg
{
        nx_uint8_t header;
        nx_uint8_t head;
        nx_uint16_t senderID;
        nx_uint8_t value1[MAX_VLENGTH];
        nx_uint16_t value2[MAX_VLENGTH];
} AvgMsg;


typedef nx_struct AvgMsgNotStar
{
        nx_uint8_t header;
        nx_uint8_t head;
        nx_uint16_t senderID;
        nx_uint32_t query_result;
        nx_uint8_t value1[MAX_VLENGTH];
        nx_uint16_t value2[MAX_VLENGTH];
} AvgMsgNotStar;
```

Figure 4.7: Definition of AvgMsg and AvgMsgNotStar

neighbours. In this implementation, nodes start phase one at the same time depending on their depth.

For this application, the implementation of Flooding Time Synchronization Protocol,which is included in TinyOs 's library, is used. In this subsection there will be a short description of this algorithm and of how synchronization of the nodes is accomplished.

4.5.1 Description of FTSP algorithm

The Flooding Time Synchronization Protocol (or FTSP) utilizes one broadcasting message. By this message, a sender obtains time synchronization reference points with its neighbours. There is a leader and all nodes have to synchronize their clocks to that leader. This can be achieved through broadcasting time synchronization messages periodically.

This algorithm can be used in single-hop, as well as in multi-hop network. The average error of the algorithm for a single hop case between two nodes is $1.48\mu s$. In the multi-hop case, the average error is $0.5\mu s$ per hop accuracy. When broadcasting a message, FTSP can estimate the time of the delay of the message until it is sent (offset) , but also the clock skew that exists from sensor to sensor.

FTSP is robust; it can handle topology changes well. Topology changes can be nodes entering or leaving a network, links failing or even mobile nodes. A detailed description of this algorithm can be read in this paper [11]. Each node maintains both a local and global time. Past and future time instances are translated between the two formats. Both the clock offset and clock skew between the local and global clocks are estimated using linear regression. Global time can be estimated from this equation:

$$globalTime = localTime + offset + skew * (localTime - syncPoint) \tag{4.1}$$

The skew is normalized to 0.0 to increase the machine precision. The syncPoint value is periodically updated to increase the machine precision of the floating point arithmetic as well as to allow time wrap.

4.5.2 Interface GlobalTime

GlobalTime, as mentioned before, is included in the library of TinyOs. Component TimeSyncC implements synchronization of nodes and provides interface GlobalTime. Interface GlobalTime, provides the following commands:

command error_t getGlobalTime(uint32_t *time) : it reads the current global time. Returns TRUE if this mote is synchronized and FAIL otherwise.

command uint32_t getLocalTime() : it returns the current local time of this mote. Returns the same value with the command Timer.getNow().

command error_t global2Local(uint32_t *time) : it converts the global time given in time into the correspoding local time and stores this again in time. It returns TRUE if this mote is synchronized and FAIL otherwise.

command error_t local2Global(uint32_t *time) : Converts the local time given in time into the corresponding global time and stores this again in time. Returns TRUE if this mote is synchronized and FAIL otherwise.

4.5.3 Real Synchronization

At the implementation of synchronization in our application, we used the commands of interface GlobalTime of the previous subsection. We used a common time to all the nodes, whereof, timers will start counting backwards for any different phase. We calculate a common time by using the command getGlobalTime and replacing the result into this formula:

$$stime = \lfloor (\frac{globalTime}{ROUND\_DURATION}) * ROUND\_DURATION \rfloor \qquad (4.2)$$

ROUND_DURATION variable is initialized at the beginning of the program with value equal to 2000 ns. Then, we convert the result to a local time with the use of command local2Global. In this way, all nodes have calculated the correspondence to the local time,in the common beginning of the period. This common time is being used as a timing reference for the calculation at the start of each timer.

4.6 Creation of the Hierarchical Tree

This section provides details of how the sensor tree is constructed. As already mentioned, the construction of the tree depends on the parameters that are defined at the beginning of the execution. The value of those parameters depends on the characteristics of the algorithm that we use to collect data from sensors.

The tree consists of nodes and these nodes represent sensors. The number of nodes is equal to the number of sensors that are inside a network. There is only one root node. This node is usually a node with id equal to zero and represents, also, the base station. Each node has only one parent and the number of children is equal to the parameter MAX_CHILDREN, as above mentioned. The construction of the tree happens as follows:

- Root node broadcasts a Routing message, which consists of the variables, that were previously shown.
- Each node that is inside the range of root and receives this message, stores the data of the message and sets root as its parent and calculates its depth to be equal to the depth of the senderID plus one.

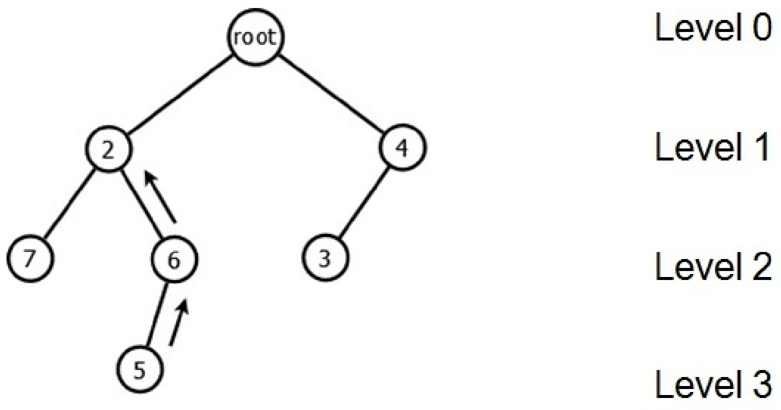


Figure 4.8: Binary tree with levels in each depth

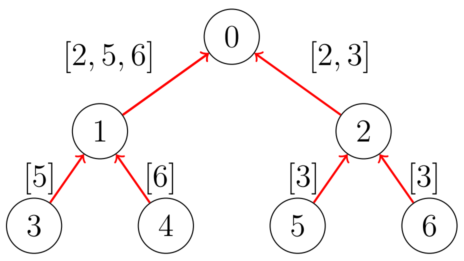
- It broadcasts a Routing message with its node id as senderID replaces the id of root.
- A node that receives a Routing message defines a sender as its parent, stores data of the message, calculates its depth to be equal to the depth of the senderID plus one and broadcasts a new Routing message.

This happens recursively, until there is no node that has no parent. In a network, where there is no message loss rate, there are no lost messages, and all nodes receive the messages that are sent to them. In Figure 4.8 there is a structure of a binary sensor tree with the depth of each level noted.

With Routing messages, nodes initialize their variables that are needed for the algorithm and the execution of the query.

4.7 Data of Previous Rounds

The algorithm for the Detection of Duplicates and the decrease of the total size of bits transmitted that is implemented requires the conservation of some data for each neighbour. As mentioned before, each node is considered to be a neighbour, which communicates bidirectionally with another node through the structure of the network. In general, with the term neighborhood, we mean the sum of nodes with which a sensor will compare



NodeId	*Id*	*Value*	*RecId*
0	8	30	[5,6,2,3,2]
1	2	6	[5,6]
2	2	30	[3,3]
3	5	10	[-]
4	6	15	[-]
5	3	7	[-]
6	3	14	[-]

Figure 4.9: Phase One in epoch t - Full Data method

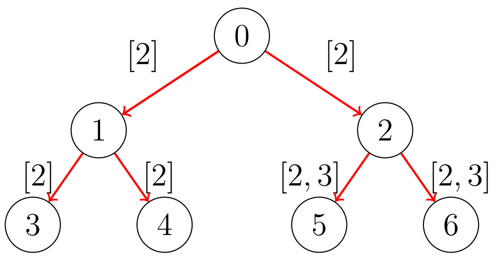
its measurements for detecting duplicates and achieve, at the same time, bidirectional communication.

For the decrease of the number of bits transmitted, there are arrays that store the previous message that a node has sent at a previous epoch. These arrays are : previous_joinsent, previous_Dups, previous_avgid and previous_avgvalue. The first array belongs to the first phase, the second to the second phase and the two other remaining belong to the third phase. There is also variable previous_query_result storing the query_result of the previous epoch.

4.8 Full Data method in Action

In Figure 4.9, phase one of epoch t is presented. In this topology there are 7 nodes, where nodes 3 and 4 have node 1 as their parent, nodes 5 and 6 have node 2 as their parent and so forth. In phase one, as it is previously mentioned, nodes send the ids that observe. In the attached array we can see the ids and values of all nodes in the network assuming that each node can have one id and one measurement for this id. In column *RecId* there are the received ids that each node has received from its children. Node 2, as we can see from Figure, sends only one time id 3, because it sends only distinct values.

In Figure 4.10, phase two of epoch t is presented. One can notice from 4.9, that node 0 is having a duplicate in id equal to 2 as it is shown from column *Duplicates* of Figure 4.10. Node 0 transmits its duplicates and their children receive it. If they have



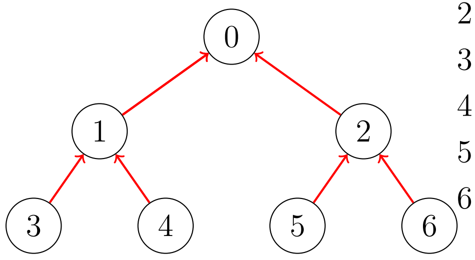
$NodeId$	Id	$Value$	$Duplicates$
0	8	30	[2]
1	2	6	[-]
2	2	30	[3]
3	5	10	[-]
4	6	15	[-]
5	3	7	[-]
6	3	14	[-]

Figure 4.10: Phase Two in epoch t - Full Data method

the same duplicate, they erase it from *Duplicates* array. If they have also duplicates that do not exist in array *Duplicates* they add it to the received duplicate values and forward them to their own children. Node 2 has duplicates in id equal to value 3. As we can see, it forwards values 2 and 3 to its children.

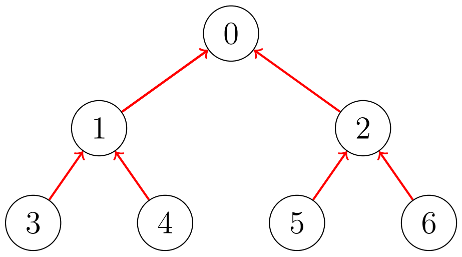
In Figure 4.11, phase three of epoch t is presented. In this phase nodes send id values and their measurements. This case is when we have "Select Star" query. Nodes, starting from the leaves, transmit their values. In the array of this Figure, there are *RecId* and *RecValue* which represent the received ids and values from the children of the corresponding node. Node 2, which has duplicate value in id 3, calculates the average value of measurements of this id that receives from its children. According to the array, node 0 receives the value of 10.5 for id 3, which is the average value of 14 and 7 of nodes 5 and 6.

Figure 4.12 represents phase three, when we dont have "Select Star" as a query. As previously mentioned, nodes execute the query, if the id values that they observe, do not belong to vector *receivedDups*. Nodes 3 and 4 send only their measurements, because their id does not belong to *receivedDups*. Nodes 5 and 6, which their id is duplicate, send all their tuples, id and value to their parent. Node 2, which is the highest node of the tree with duplicates in id 3, must execute the query SUM for the nodes 5 and 6. Node 0 receives all tuples of node 1 and the query result of nodes 3 and 4. After node 0 has collected all data from its children, it calculates the final result, which in this case will be equal to 83.5.



$NodeId$	Id	$Value$	$RecId$	$RecValue$
0	8	30	[5,6,2,3,2]	[10,15,6,10.5,30]
1	2	6	[5,6]	[10,15]
2	2	30	[3,3]	[7,14]
3	5	10	[-]	[-]
4	6	15	[-]	[-]
5	3	7	[-]	[-]
6	3	14	[-]	[-]

Figure 4.11: Phase Three in epoch t - "Select Star" query - Full Data method

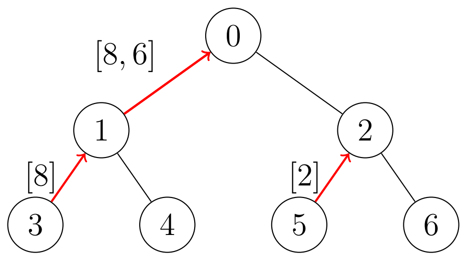


$NodeId$	Id	$Value$	$query\_result$	$RecId$	$RecValue$
0	8	30	35.5	[2,2]	[6,30]
1	2	6	25	[-]	[-]
2	2	30	-	[3,3]	[7,14]
3	5	10	-	[-]	[-]
4	6	15	-	[-]	[-]
5	3	7	-	[-]	[-]
6	3	14	-	[-]	[-]

Figure 4.12: Phase Three in epoch t - SUM query - Full Data method

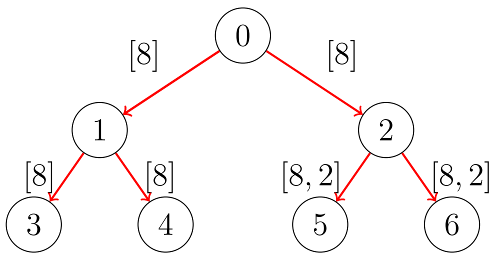
When epoch (t+1) comes and the query remains the same as the previous epoch t, the transmitting and receiving data of the tree is as shown in the following Figures. In Figure 4.13, nodes 4 and 6 haven't changed their values, so they do not send anything. Nodes 1 and 2 keep a backup of their data from previous epoch, so they use backup data for nodes 4 and 6. Node 1 sends ids 8 and 6, and forwards them to its parent, node 0. Node 2 has the same values as in previous epoch, so it does not send its values to its parent.

In phase two, as shown also in Figure 4.14, node 0 has received id 8 two times and it stores it to array *Duplicates*. Nodes 1 and 2 receive duplicates of their parent and uniting them with their own duplicate values, they forward them to their children. It



$NodeId$	Id	$Value$	$RecId$
0	8	30	[8,6,2,3]
1	8	8	[8]
2	2	30	[2]
3	8	2	[-]
4	6	15	[-]
5	2	4	[-]
6	3	14	[-]

Figure 4.13: Phase One in epoch t+1 - Full Data method



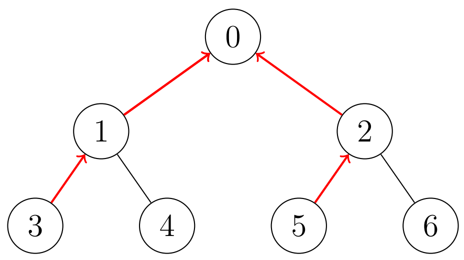
$NodeId$	Id	$Value$	$Duplicates$
0	8	30	[8]
1	8	8	[-]
2	2	30	[2]
3	8	2	[-]
4	6	15	[-]
5	2	4	[-]
6	3	14	[-]

Figure 4.14: Phase Two in epoch t+1 - Full Data method

should be mentioned, that if node 0 had still duplicate value in id equal to 2, then it would not send any message to its children, so they would assume that it is the same duplicate ids as in previous epoch.

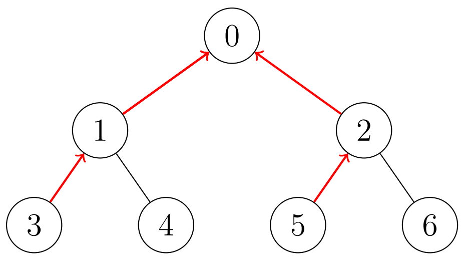
In Figure 4.15, phase three of epoch t+1 is presented. Nodes 4 and 6 haven't changed their values from previous epoch, so they do not send their data also in this phase. The rest of the nodes send their data. As previously stated, nodes keep average value of measurements of ids that belong to array *Duplicates*. Node 0 has gathered all data as it is also shown in arrays *RecId* and *RecValue*.

In Figure 4.16, phase three in epoch t+1 is presented when we have a SUM query. In this phase, the same things as the other phases of this epoch apply. Nodes 4 and 6 do



$NodeId$	Id	$Value$	$RecId$	$RecValue$
0	8	30	[8,6,8,2,3]	[2,15,8,17,14]
1	8	8	[8]	[2]
2	2	30	[2]	[4]
3	8	2	[-]	[-]
4	6	15	[-]	[-]
5	2	4	[-]	[-]
6	3	14	[-]	[-]

Figure 4.15: Phase Three in epoch t+1 - “Select Star” query - Full Data method



$NodeId$	Id	$Value$	$query\_result$	$RecId$	$RecValue$
0	8	30	46	[8,8]	[2,8]
1	8	8	-	[8]	[2]
2	2	30	-	[2]	[4]
3	8	2	-	[-]	[-]
4	6	15	-	[-]	[-]
5	2	4	-	[-]	[-]
6	3	14	-	[-]	[-]

Figure 4.16: Phase Three in epoch t+1 - SUM query - Full Data method

not send data, because they have the same value as the previous epoch. Node 0 receives all data from its children and the measurements of duplicate id 8. The final query result, in the end, will be 59.33.

Chapter 5

Saving Packet Payload

5.1 Introduction

Battery-powered embedded systems carefully manage energy consumption for maximizing system lifetime. Wireless sensor networks, made up of many mote devices, are ,most of the times, designed to operate for months without intervention. Sensor networks are typically used to monitor an environment and may be deployed in remote or hazardous locations. WSNs can consist of thousands of motes, and cover wide areas. As a result, mote software and hardware must consider energy consumption at every level.

Aggregation, a widely researched field for reducing data transmissions by combining data on motes, reduces energy use by spending additional energy on computation to save a greater amount of energy on the power-hungry radio. Increasing on-mote processing complexity will require additional computational hardware, demanding more energy. As sensor networks grow and generate larger data sets, these energy costs will keep up rising.

This thesis explores the use of two different methods of minimizing transmitting data and having as a consequence energy consumption. The following section represents the implementation of Delta method and next is Bloom Filter method. Opening and closing radio of sensors is another way of minimizing energy consumption, but it isn't enough. As it is previously stated, the application of each method depends on $Makefile$, the starting file, and the value of parameters Bloom_En and DELTA_MODE, which are defined at this file.

5.2 Delta Data Transmission

The main focus of this section is how to reduce the number of sensor data transmission, while maintaining the knowledge of existing duplicates. This proposed method could be used in sensor networks, where data payload of transmission should be minimal.

This method is called "delta", because nodes send the changes of data from the previous epoch. Nodes keep a backup of the data, that was sent in a previous epoch for each phase. When the next epoch comes, the nodes have calculated the data that needs to be sent. Then, it sends only the plus or minus values of a variable. Delta method is used only in the first and second phases, where nodes send ids or duplicate values. At the first phase, nodes send ids of objects that are new to the network or ids of objects that are no longer observed by sensors. Subsequently, at phase two,where they send duplicate values to their children, with this method nodes send the differences of the duplicate values from the previous epoch. For each phase, there is a different type of message. Below we can see the different types of messages, considering the phase that we are in.

5.2.1 Delta Method in Action

In this subsection will be shown the structs that are being transmitted in first and second phase. The first struct in Figure 5.1, represents the type of message that is sent at phase one, when delta method is enabled and it is called *SS_Message_delta*. The field **header** defines the type of the message. In this case its value is equal to five. **Plus_pt** and **minus_pt** are variables that define the number of elements that are at the arrays **plus**[**MAX_VLENGTH**] and **minus**[**MAX_VLENGTH**]. **SenderID** represents the id of the node that has sent this message. In arrays **plus**[**MAX_VLENGTH**] and **minus**[**MAX_VLENGTH**] are stored the values of ids that are new or have left the network according to the previous epoch.

The second struct in Figure 5.2, represents the type of message that is sent at phase two, when delta method is enabled and it is called *ForwardingMsg_delta*. The field **header** defines the type of the message. In this case its value is equal to six. **Plus_pt** and **minus_pt** are variables that define the number of elements that are at the arrays plus[MAX_VLENGTH] and minus[MAX_VLENGTH]. In arrays plus[MAX_VLENGTH] and minus[MAX_VLENGTH] are stored the values of ids that are new or have left the network according to the previous epoch.

```
typedef nx_struct SS_Message_delta
{
        nx_uint8_t header;
        nx_uint8_t plus_pt;
        nx_uint8_t minus_pt;
        nx_uint16_t senderID;
        nx_uint8_t plus[MAX_VLENGTH];
        nx_uint8_t minus[MAX_VLENGTH];
} SS_Message_delta;
```

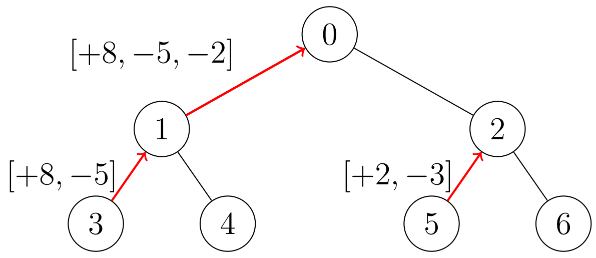
Figure 5.1: Definition of SS_Message_delta

```
typedef nx_struct ForwardingMsg_delta
{
        nx_uint8_t header;
        nx_uint8_t plus_pt;
        nx_uint8_t minus_pt;
        nx_uint8_t plus[MAX_VLENGTH];
        nx_uint8_t minus[MAX_VLENGTH];
} ForwardingMsg_delta;
```

Figure 5.2: Definition of ForwardingMsg_delta

This method only represents the delta values of ids and not delta values of bloom filter implementation. It can also be implemented in bloom filters and the only change is a new type of message. When a node receives a message of delta values of bloom filter, it can figure that the received list concerns the bloom filter of a previous epoch and not a full message.

In order to fully understand how delta method works, here is an example. As previously stated, delta method is used only in phases one and two. When we are in epoch t, and t equals to 1, which means that it is the first epoch that our algorithm is executed, the result is the same as shown in Figures 4.9, 4.10, 4.11 and 4.12. The only difference is that when using delta method, there is a plus and a minus array. In the first epoch, all values are entered in array plus.



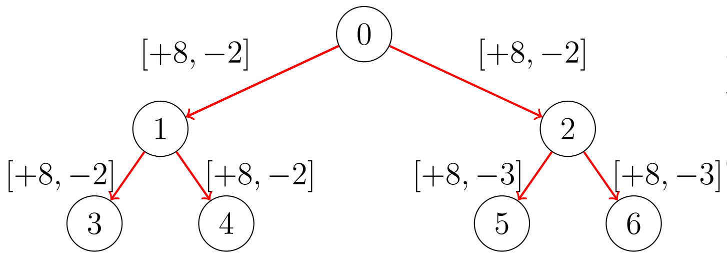
NodeId	*Id*	*Value*	*Plus*	*Minus*
0	8	30	[8]	[5,2]
1	8	8	[8]	[5,2]
2	2	30	[2]	[3]
3	8	2	[-]	[-]
4	6	15	[-]	[-]
5	2	4	[-]	[-]
6	3	14	[-]	[-]

Figure 5.3: Phase One in epoch t+1 - Delta method

When epoch (t+1) comes and the query remains the same as the previous epoch t, the transmitting and receiving data of the tree is as shown in the following Figures. In Figure 5.3, nodes send the changes of the previous epoch. Node 3 sends the new id that observes, which is 8, and removes the previous id which is 5. Node 1 understands that node 3 does not observe anymore object with id equal to 5. Node 4 does not change its values so it does not send any message. Node 1 calculates its differences from previous epoch, and informs its parent about the changes of its subtree. The same thing applies also in the right subtree of node 0. Node 2 receives the changes of node 5 and forwards them to its parent. In the array of this Figure we can see columns *Plus* and *Minus*, where they show the received ids, which are added or removed from the children of the corresponding nodes.

In phase two, as shown also in Figure 5.4, starting from node 0, it sends the different values from previous epoch. In previous epoch node 0 had duplicate in value 2, but now it has in value 8. It has to inform its children of the change, so it sends in *Plus* array value 8 and in *Minus* array value 2. Nodes that receive it, they forward it. Node 2 has also duplicate value in id equal to 2. This node in previous epoch had received duplicate in value 2 and the only difference from previous epoch is that there is no duplicate in value 3. Then, it sends in *Plus* array duplicate id 8 and in *Minus* array duplicate id 3. This means that nodes 3 and 6 deem that there are no longer duplicate values in id 3, but there are in ids 8 and 2.

Phase three is the same as Figure 4.15 and 4.16, because in phase three delta method



NodeId	*Id*	*Value*	*Plus*	*Minus*	*Duplicates*
0	8	30	[-]	[-]	[8]
1	8	8	[8]	[2]	[-]
2	2	30	[8]	[2]	[2]
3	8	2	[8]	[2]	[-]
4	6	15	[8]	[2]	[-]
5	2	4	[8]	[3]	[-]
6	3	14	[8]	[3]	[-]

Figure 5.4: Phase Two in epoch t+1 - Delta method

is not applied.

5.3 Bloom Filter method Transmission

In this thesis, the use of Bloom Filters is very lucrative, because it compresses data of dissemination minimizing data payload of the tree. Nodes create their bloom filter locally after collecting all data from their children and then they forward their own bloom filter to their parents or children depending on the phase.

Specifically, in this implementation the size of Bloom Filter at each node is defined from the variable $BFlength$. As it is previously mentioned, it's a binary tree and this means that each node must have maximum two children. Therefore, when we have 100 nodes in our network, nodes in depth one must collect in phase one, on average, 25 id values. This leads to the conclusion that we cannot have 2 hash functions in that small Bloom filter, because the probability of having a bit equal to zero is very small. That's why there is only one hash function used. However, we can also use more than one hash functions by changing the variable $hashes$ into the number of hash functions we want to use.

Bloom Filter is used only in phases one and two. In phase one, the node sends to its parent the union of the bloom filters that has received including its own. From the initial file $Makefile$, also shown in Algorithm 1, the user can define, if bloom filters should be included in the execution and this can be done by defining variable $BLOOM\_EN$.

When there is only Bloom filter enabled in network and not Full Data or Delta method, the algorithm in phase one is as follows:

- Leaves create their bloom filter, adding their ids that observe and send them to their parents.
- Intermediate nodes receive bloom filters of their children and create their own bloom filter by uniting all received bloom filters together. They also add their own ids to their filter, that observe, and forward it to their parent
- This is continued recursively until all data arrives at base station.

In Algorithm 9 the pseudocode of phase one, when using bloom filters is presented. In line 1, the current node sets bits of $BFilter$ to value 1, located at the result of hash functions, when inserting the ids that this node observes. Node, in lines 2 - 4, is in state STATE_RECEIVE, where it receives from its children their bloom filters. While receiving those messages, current node unites received filters with its own bloom filter. If a child did not send any message, then current node fetches the bloom filter that was sent at the previous epoch and unites it with current $BFilter$. In lines 6 to 11, node examines if $BFilter$ is the same as the bloom filter, which was sent in the previous epoch. If bloom filter is the same, it does not transmit to its parent(line 7), otherwise, it stores the new bloom filter to $previous\_BloomSS$ and disseminates it.

When $mytimer$ fires, an event occurs and we are at phase two. While all bloom filters are gathered in base station, the algorithm goes as follows:

- Nodes start comparing with logic AND all receiving bloom filters including their own. When logic AND returns 1 for a bit in position i, then root sets bit equal to 1 in position i of the array $DupBloomMine$.
- When all comparisons finish, root disseminates array $DupBloomMine$ to its children.
- Intermediate nodes receive $DupBloomMine$ of their parent and pass its bits to $DupBloom$ array. They compare their own $DupBloomMine$ with the received $DupBloom$. If a bit from array $DupBloomMine$ is equal to the corresponding bit of $DupBloom$, then we set this bit of $DupBloomMine$ to zero. When this is

Algorithm 9 Phase1 with Bloom Filters

```
1: Map result of hash functions of current node's ids and insert into $BFilter$
2: while ($synch\_Phase1 == STATE\_RECEIVE$) do
3:     Unite received bloom filter with $BFilter$
4: end while
5: For each child that didn't send message, unite the bloom filter of the previous epoch
   to $BFilter$
6: if $BFilter == previous\_BloomSS$ then
7:     Does not transmit $BFilter$ to parent
8: else
9:     $previous\_BloomSS = BFilter$
10:    Send $BFilter$ to parent
11: end if
12: sleepUntilNextPhase()
```

completed, we unite with logic *OR* the two arrays. Node sends *DupBloom* to its children.

- This happens recursively until all nodes receive a message including leaf nodes.

In Algorithm 10 the pseudocode of phase two is presented, when using bloom filters. In line 1, current node creates *DupBloomMine* by comparing receiving bloom filters and its own from the previous epoch. We search at each individual bit of all the filters and if there are at least 2 bits set to 1 in the same position, then at the same position we set bit of *DupBloomMine* to 1. This means that there is an id, which is probably duplicate. Vector *DupBloomMine* does the same thing as *Duplicates* does for Full Data method. *DupBloomMine* restores the bloom filter, which includes the duplicate values of this node.In lines 2 - 4, is in state STATE_RECEIVE, where the node receives from its parent pointers of its bloom filter that has duplicates. While receiving the message, current node sets to 1 bits of *DupBloom* according to received bloom filter. This means that *DupBloom* includes all the duplicate values that exist in higher hierarchy of the tree (as was *receivedDups* in Full Data mode). If current node is root, then it only needs to transmit *DupBloomMine*, only if it is different from the previous epoch(line 10).Otherwise, it does not transmit anything. If node is not root, it must check if the

```
typedef nx_struct SS_Message_bloom
{
        nx_uint8_t header;
        nx_uint16_t senderID;
        nx_uint8_t BloomFilter[BFlength];
} SS_Message_bloom;
```

Figure 5.5: Definition of SS_Message_bloom

upper tree nodes have duplicate ids at the same id, that current node has stored into *DupBloomMine*. This can be accomplished by comparing the two vectors (lines 14-15) bit by bit. If vector *DupBloomMine* has a bit equal to 1 and not *DupBloom*, then this means that current node has a duplicate value that the other node higher in the tree does not have. In order to include this value in the transmitting data, we set bit of *DupBloom* equal to 1 in the same position as it is in *DupBloomMine*. On the other hand, if bits of *DupBloom* and *DupBloomMine* in the same index are equal, then current node must remove this id from its bloom filter. This can be done by setting this bit to zero(line 15). In lines 20 to 26, node examines if *DupBloom* is the same as the bloom filter, which was sent at the previous epoch. If the bloom filter is the same, it does not transmit to its parent(line 21), otherwise, it stores the new bloom filter to *previous_BloomM* and disseminates it.

5.3.1 Bloom Filters in Action

In phase one the type of message that is sent is named as SS_Message_bloom. This message includes a variable that defines the type of message *header*. This variable is set to seven. There is also *senderID*, where it is equal to TOS_NODE_ID of node that sends the message. *BloomFilter* is the bitvector, which contains ids of objects that nodes and their subtree observe. In Figure 5.5, we can see the struct of this type of message.

In phase two the type of message that is sent is named as ForwardingMsgbloom. This message includes a variable that defines the type of message *header*. This variable is set to eight. *Value* is the bloom filter, which contains the duplicate values that exist in the tree. In figure, 5.6, we can see the struct of this type of message.

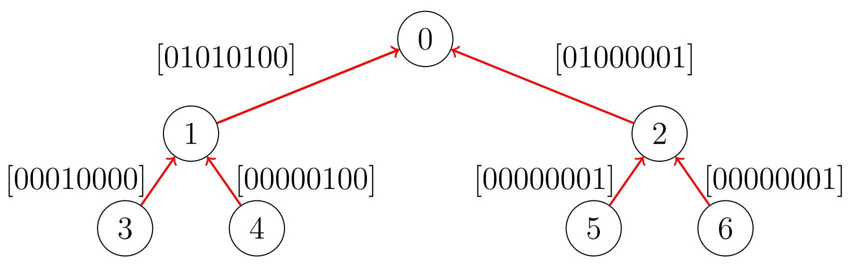
Algorithm 10 Phase2 with Bloom Filters

```
1: Create $DupBloomMine$, its bits are set from bloom filters of its children and its own
2: while ($synch\_Phase2 == STATE\_RECEIVE$) do
3:    Parse array $value$ and store values to $DupBloom$
4: end while
5: if ( is root ) then
6:    if $DupBloomMine == previous\_BloomMine$ then
7:       Does not transmit $DupBloomMine$ to children
8:    else
9:       $previous\_BloomMine = DupBloomMine$
10:      Sends $DupBloomMine$ to children
11:   end if
12: else
13:   for $i = 0; i < BFlength; i++$ do
14:      if ($DupBloom[i] == DupBloomMine[i]$) then
15:         DupBloomMine[i] = 0;
16:      else if ( $DupBloomMine[i] == 1 \&\& DupBloom[i] == 0$ ) then
17:         DupBloom[i] = 1;
18:      end if
19:   end for
20:   if $DupBloom == previous\_BloomM$ then
21:      Does not transmit $DupBloom$ to children
22:   else
23:      $previous\_BloomM = DupBloom$
24:      Sends $DupBloom$ to children
25:   end if
26: end if
27: sleepUntilNextPhase()
```

In order to fully understand how Bloom Filter method works, here is an example. As previously stated, bloom filters are used only in phases one and two. In epoch t and in phase one nodes send their ids through bloom filters. This can be seen in Figure 5.7, where nodes unite their own bloom filters and disseminate them to their parents. Nodes

```
typedef nx_struct ForwardingMsgbloom
{
        nx_uint8_t header;
        nx_uint8_t value[BFlength];
} ForwardingMsgbloom;
```

Figure 5.6: Definition of ForwardingMsgbloom



$NodeId$	Id	$Value$	$BloomFilter$
0	8	30	[11010101]
1	2	6	[01010100]
2	2	30	[01000001]
3	5	10	[00010000]
4	6	15	[00000100]
5	3	7	[00000001]
6	3	14	[00000001]

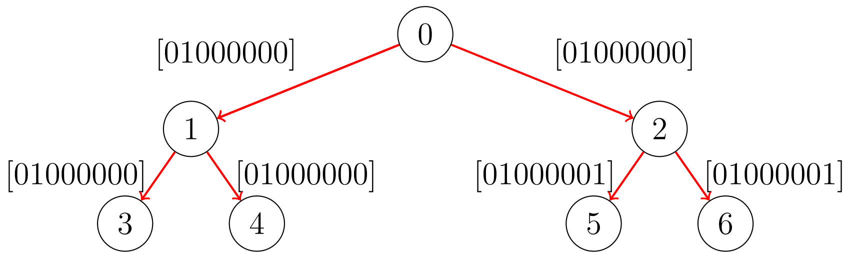
Figure 5.7: Phase One in epoch t - Bloom Filter method

3 and 4 send their filters to node 1 and then node 1 unites them with its own. Nodes 5 and 6 observe the same object, so they have the same id and by extension the same bloom filter. Node 2 unites the received bloom filters with its own and forwards it to its parent. Column $BloomFilter$, which is shown in Figure, represents the united bloom filter of each node.

Phase two can be seen in Figure 5.8, where nodes disseminate bloom filters with duplicate ids. Node 0 has a duplicate id, of which bloom filter is equal to [01000000]. It disseminates this bloom filter to its children. Node 2 has also duplicate value, with bloom filter equal to [00000001]. When this node receives from its parent the value [01000000], it unites it with its own $DupBloomMine$ and forwards it to its children.

Phase three is the same as Figure 4.11 and 4.12, because in this phase we do not apply this method.

When epoch (t+1) comes and the query remains the same as the previous epoch t, the transmitting and receiving data of the tree is as shown in the following Figures. In



NodeId	*Id*	*Value*	*DupBloomMine*
0	8	30	[01000000]
1	2	6	[-]
2	2	30	[00000001]
3	5	10	[-]
4	6	15	[-]
5	3	7	[-]
6	3	14	[-]

Figure 5.8: Phase Two in epoch t - Bloom Filter method

Figure 5.9, nodes 4 and 6 haven't changed their values, so they don't send anything. Nodes 1 and 2 keep a backup of their data from previous epoch, so they use backup data for nodes 4 and 6. Node 1 sends ids 8 and 6, and forwards them to its parent, node 0. Node 2 has the same bloom filter from previous epoch, so it does not transmit it.

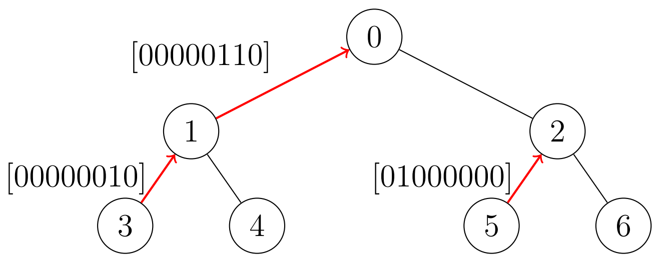
In phase two, as shown also in Figure 5.10, node 0 has a duplicate value in id of which bloom filter equals to [01000000] and stores it to *DupBloomMine* array. Nodes 1 and 2 receive bloom filter of their parent containing its duplicate values and uniting them with their own *DupBloomMine* filter, they forward it to their children. It should be mentioned, that if node 0 had still duplicate value in id of which bloom filter is equal to [01000000], then node wouldn't send a message to its children, so they would assume that it is the same duplicate value as in previous epoch.

Phase three is the same as Figures 4.11 and 4.12, because in this phase we do not apply this method.

5.4 Combination of methods

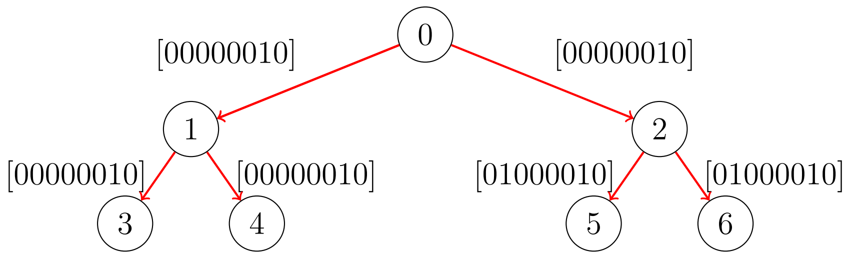
Until now the three methods have been presented to be used separately at the execution. In file *Makefile* in Algorithm 1, we can set variables FULL_DATA, BLOOM_EN and DELTA_MODE and combine the methods as we want.

When we have applied the methods that will be used, nodes have to choose which method should they use best at each phase. This can be solved by checking the data payload of each method. Nodes choose the method, which uses the least bits. When



NodeId	*Id*	*Value*	*BloomFilter*
0	8	30	[01000111]
1	8	8	[00000110]
2	2	30	[01000001]
3	8	2	[00000010]
4	6	15	[00000100]
5	2	4	[01000000]
6	3	14	[00000001]

Figure 5.9: Phase One in epoch t+1 - Bloom Filter method



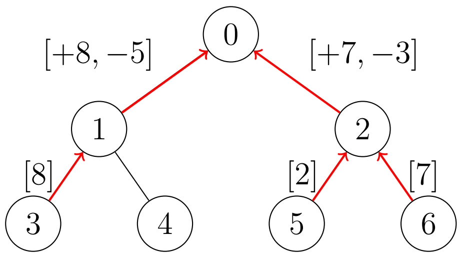
NodeId	*Id*	*Value*	*DupBloomMine*
0	8	30	[00000010]
1	8	8	[-]
2	2	30	[01000010]
3	8	2	[-]
4	6	15	[-]
5	2	4	[-]
6	3	14	[-]

Figure 5.10: Phase Two in epoch t+1 - Bloom Filter method

there is a tie, there is a priority in Full Data method, because it is more secure and nodes will not lead to false assumptions.

In order to consider how a combination of at least two methods works, it can be seen through an example. The first epoch $t = 1$, when combining Delta with Full Delta methods, is the same as Figure 4.9, because there is no previous epoch to send the differences.

When epoch t+1 arrives, Figure 5.11 presents the transmitted data. Nodes 3, 5 and 6 choose to send their ids through Full Data method, because in this way they send less data. If they chose delta method, node 3 for example had to send $[+8, -5]$. Node 1 decides also to use Full Data method, because it uses less bits. Node 2 decides to use delta method, because it only sends $[+7, -3]$ value to its parent. If node 2 chose to use



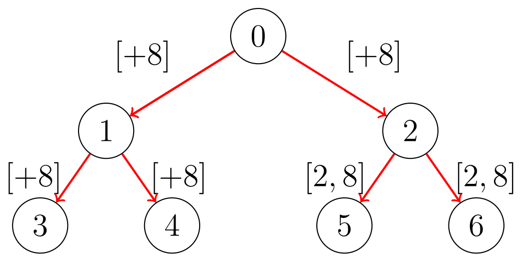
NodeId	*Id*	*Value*	*RecId*	*Plus*	*Minus*
0	8	30	[-]	[8,7]	[5,3]
1	2	8	[8]	[-]	[-]
2	2	30	[2,7]	[-]	[-]
3	8	2	[-]	[-]	[-]
4	6	15	[-]	[-]	[-]
5	2	4	[-]	[-]	[-]
6	7	1	[-]	[-]	[-]

Figure 5.11: Phase One in epoch t+1 - Full Data and Delta methods

Full Data method it had to send $[2, 7]$ data, which is one byte more in transmitting data.

Phase two, when using these two methods will be as shown in Figure 5.12. Node 0 has also duplicate values in id 8. In previous epoch, it had duplicate values in id 2. In this phase node 0 uses delta method to send only the change from previous epoch, by adding id 8 to *Duplicates* array and sending [+8] value. Node 1 and its subtree does not have any other duplicate value, so they only forward message of node 0. Node 2, in addition, in previous epoch had duplicate in id 3. In this epoch, though, it does not. If node 2 used Full Data method, it should send $[2, 8]$, otherwise, it should send $[+8, -3]$. As it is previously mentioned, when there is a tie, Full Data method prevails. In this way, node 2 chooses Full Data method.

Phase three is as shown in Figure 4.15 and 4.16, because in this phase Delta or Bloom Filter methods are not used.



$NodeId$	Id	$Value$	$Duplicates$	$Plus$	$Minus$
0	8	30	[2,8]	[-]	[-]
1	2	8	[-]	[+8]	[-]
2	2	30	[-]	[+8]	[-]
3	8	2	[-]	[+8]	[-]
4	6	15	[-]	[+8]	[-]
5	2	4	[-]	[-]	[-]
6	7	1	[-]	[-]	[-]

Figure 5.12: Phase Two in epoch t+1 - Full Data and Delta methods

Chapter 6

Experiments

Up to here, we have discussed the three methods only in theory. But let's see some real action. In the first section, the outcome of each method and the combination with each other is presented, when there is exact topology defined. In the second section, the results of methods and their combinations are shown, when the signal strength decays. In both experiments we used "*Select Star*" query, because this is the query, which sends most bits over the network.

6.1 Results with defined topology

At this experiment, all nodes have equal value of gain, which is $-50.0\ dB$. It is also defined, which nodes communicate with others. The tree is binary so all nodes have maximum two children. Through TOSSIM, there was a simulation over various number of nodes thus different topologies.

Each method has its drawbacks. This experiment has as a target to reveal these drawbacks in order to decide which method is best to use. Nodes are defined to observe only one object, thereafter, each one has one id and one measurement. Ids and measurements change their value at each epoch with a probability equal to 0.4.

In Figure 6.1 a table is presented, where in first column there is a range of values that correspond to the ids of nodes. In the second column there is a range of values that correspond to the ids of objects that each node observes. In other words, a node can observe only the objects that are at its range of values. At the last line, nodes that their TOS_NODE_ID belongs to ranges 20-39 and 90-100, it is shown that they observe objects

Figure 6.1: Range of the Values of Objects observed

TOS_NODE_ID	Object's id observed
0 – 19	0 – 9
40 – 49	20 – 29
50 – 59	20 – 29
60 – 79	30 – 39
80 – 89	40 – 49
20 – 39, 90 – 100	*TOS_NODE_ID*

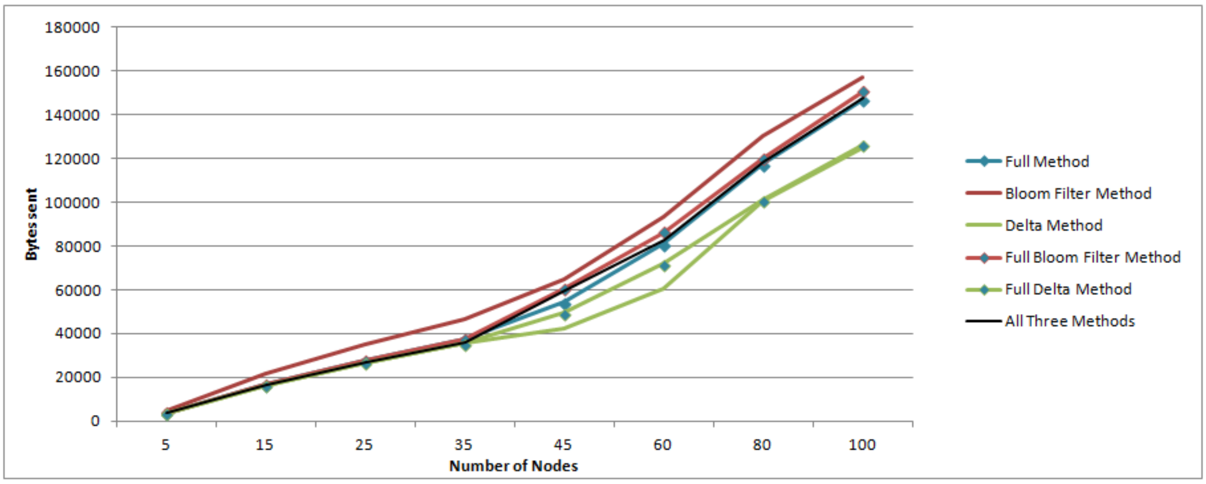


Figure 6.2: Mean number of bits sent at different topologies

that have their own TOS_NODE_ID as object's id. This table is necessary in order to define, that there will be nodes that observe the same id and share these duplicates through all the tree.

As is previously mentioned, each method uses different type of messages. This means that nodes send different number of bits according to the method they use. In order to have a sufficient image of the total number of bits that are being sent, when using each method, we repeated the experiment for 10 times and computed the mean number of bit values. Each run lasted for 16 epochs.

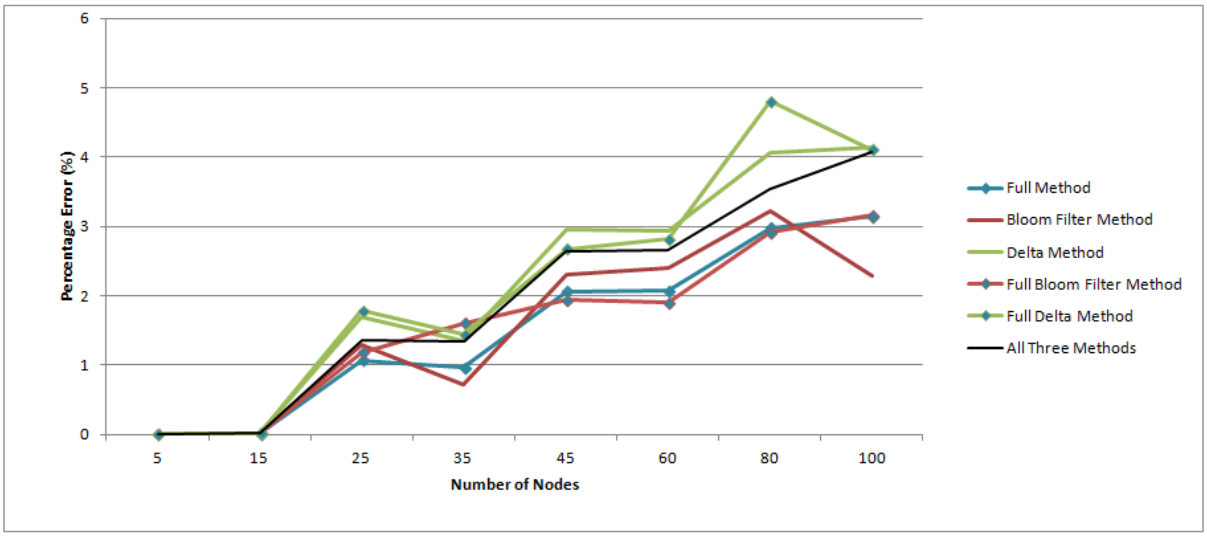


Figure 6.3: Percentage Error of final Data

Figure 6.2 shows the results of this experiment. As we can see in the x-axis there are the number of nodes, which are in the network. Y-axis shows the total number of bits sent in 16 epochs in all three phases. Delta and Full Delta methods are the two most lowest lines in the graph. Consequently, these two methods transmit the minimum number of bits. Bloom Filter is the method that sends more bits than any other combination of methods. This means that nodes will have greater energy consumption. On the contrary, if Bloom Filter method is combined with Full method and these two with Delta method, the results are equivalent to Full method. In the topology, where we have 35 nodes inside the network, we can observe that Full Bloom Delta method starts to use only Bloom Filter, because with this method, less bits are transmitted. After 100 nodes inside a network, Full method will be sending more bits than Full Bloom Delta method and probably more than Bloom Filter method.

A naive outcome is that Delta and Full Delta methods are those that send less bits, so they are the ideal methods to use. Figure 6.3, comes at this point to verify and examine this choice.

In Figure 6.3, we compute the percentage error of final data, that root has gathered. Percentage error is computed by the following formula:

$$Percentage\ Error = mean(\frac{|ideal - algorithmic|}{ideal}) \tag{6.1}$$

Ideal symbolizes the ideal value of measurements that are expected to be received. Variable algorithmic represents the summation of measurements of a specific object-id. By this formula we can find how much algorithmic value deflects from the ideal value.

This Figure shows that Delta and Full Delta methods have the highest deviation of all other methods, in other words, it is not a good choice to use only these methods, just because they send less data. Full Bloom Delta method is the third one with the highest deviation and this is because it uses Delta except from Bloom Filter and Full Data methods. From this Figure, one can notice, that Full Bloom method has the lowest deviation in more topologies than Full Data method. This means that, it is more vulnerable to use this method in some topologies, although it sends some bits more than Full Data method.

When there are 25 and 35 nodes in the network, we can see that the percentage error is falling in all methods. This is caused by the number of nodes that are in the network and the number of duplicates that are in. As previously mentioned, in Figure 6.1, nodes with id 20-39 do not have duplicate ids of objects they observe. In that case there are more unique ids(except from when there are over 60 nodes in the network) in topologies, where we have 25 and 35 nodes and it is expected deviation to decrease. In topologies, where we have 45 and 60 nodes, the range of duplicate values has not changed, (see in Figure 6.1), and no other unique ids are added in the network. In this Figure, the percentage error retains its value, which is also expected. For 60 to 80 nodes in the network, percentage error increases. This is due to the fact that, according to the previous table, there are more duplicate values, and the probability of a false result is increasing. Finally, in topologies 90-100, the number of unique values increases and duplicate ids remain the same. This leads to the decrease of percentage error like it happened previously.

6.2 Results when weakening strength of signal

Wireless sensors, like Iris, are being placed in specific environments according to the exact behavior of a wireless link. An example of a specific environment can be the aisle of a building or a parking structure. First of all, there should be defined the models of radio and wireless channel and their interaction.

RF signal propagation, reflection and scattering conditions can influence signal strength by two ways. The first one is the decay of the signal. It decays exponentially with respect to distance. The second one is, for a given distance d, the signal strength, which is random and log-normally distributed about the mean distance - dependent value. One of the most common and most used radio propagation models is the log-normal shadowing path loss model:

$$PL(d) = PL(d_0) + 10nlog_{10}(\frac{d}{d_0}) + X_\sigma \tag{6.2}$$

In Equation 6.2, d is the transmitter-receiver distance, n is the path loss exponent, which is the rate at which signal decays. X_σ is a zero-mean Gaussian random variable (in dB) with standard deviation σ. Values of n and σ are obtained from experimental values. $PL(d_0)$ is the power decay for the reference distance d_0.

According to this model, the received power (P_r) in dB is given by:

$$P_r(d) = P_t - PL(d), \tag{6.3}$$

which means, that the received signal strength P_r at a distance d is the output power of the transmitter P_t minus $PL(d)$ (all powers in dB).

Once the radio has decided on how to encode the bits of sending data, it has to decide how to send data over the wireless channel. The options are modifying amplitude, frequency or phase of the carrier frequency also called modulation. Encoding and modulation is very necessary in Wireless Sensor Network. The probability of a bit error P_e is given by:

$$P_e = \frac{1}{2}exp^{-\frac{\gamma}{2}}, \tag{6.4}$$

where γ is the signal to noise ratio (SNR) in the presence of additive white gaussian noise (AWGN). For a frame being received correctly, there needs to be all bits received correctly. Hence, for a frame of length f the probability of successfully receiving a packet is:

$$p = (1 - P_e)^8 f \tag{6.5}$$

Another important aspect that influences link behavior is noise floor, which depends on both the radio and the environment. Noise floor represents the noise that can exist in an environment. Temperature of an environment or even interfering signals can influence noise floor. When the receiver and the antenna have the same ambient temperature the noise is given by:

$$P_n = (F + 1)kT_0B, \tag{6.6}$$

where F is the noise figure, k the Boltzmann's constant, T_0 the ambient temperature and B the equivalent bandwidth. There is also link asymmetry because of hardware variance in the noise floor and output power.

To sum up, the specific behavior of the wireless link depends on two elements: the radio, and the environment (channel) where they are placed. Hence, in order to obtain better simulations, the characteristics of both elements should be provided. The model, which we used and is proposed by the ANRF group at USC, is a more general link-layer model and it is valid for static and low-dynamic environments. This model simulates hardware variance through a joint gaussian process:

$$\begin{pmatrix} T \\ R \end{pmatrix} \sim N\left(\begin{pmatrix} P_t \\ P_n \end{pmatrix}, \begin{pmatrix} S_T & S_{TR} \\ S_{RT} & S_R \end{pmatrix}\right) \tag{6.7}$$

Where the covariance matrix S shows the variance of the output power, noise floor and the correlation between them.

For the execution of this experiment, we used the suggested values that were used for MICA2 motes. Nominal values for these motes are $-20dBm < P_t < 5dBm$, $P_n = -105dBm$ and can be changed from the covariance matrix. For these values, covariance matrix is:

$$S = \begin{pmatrix} 3.7 & -3.3 \\ -3.3 & 6.0 \end{pmatrix} \tag{6.8}$$

Next we obtained values for the channel characteristics path loss exponent, shadowing standard deviation and d_0 from other studies like [12] and the suggested values from [13].

Figure 6.4: Values of channel, radio and topology parameters

Channel Parameters	**Value**
PATH_LOSS_EXPONENT	3.3
SHADOWING_STANDARD_DEVIATION	5.5
D0	1.0
PL_D0	$32.5 - 43.0$
Radio Parameters	**Value**
NOISE_FLOOR	-105.0
S11	3.7
S12	-3.3
S21	-3.3
S22	6.0
Topology Parameters	**Value**
TOPOLOGY	2
NUMBER_OF_NODES	36
TERRAIN_DIMENSIONS_X	100.0
TERRAIN_DIMENSIONS_Y	100.0

For the channel, radio and topology parameters we used the values that are presented in Figure 6.4.

Topology parameters allows us to test different deployments. The available type of deployments are:

- Grid (1): nodes are placed on a square grid topology and the number of nodes inside the network has to be a square of an integer.
- Uniform (2): based on the number of nodes, the physical terrain is divided into a number of cells. Within each cell, a node is placed randomly.
- Random (3): nodes are placed randomly within a terrain.
- File (4): position of nodes is read from an input topology file from user.

Changing channel parameter $PL(d_0)$, there is a different output file when executing this model. This output file contains the link gains for each link and the noise floor for each node. For this experiment, we tried different values of power decay $PL(d_0)$ in dB for the reference distance d_0. When increasing the value of power decay, path loss increases too. This means that for a specific number of nodes in a network, topology changes and more packets are lost. This experiment was repeated for 10 times and each run lasted for 16 epochs.

As we can see in Figure 6.4, there are 36 nodes inside our network in a $100 * 100$ terrain. Figure 6.5 represents the mean number of bits that are sent, when there are constantly 36 nodes, but topology changes. As power decay increases, the depth of the tree increases. When power decay $PL(d_0)$ is equal to 32.5, tree depth has reached value 2. On the other hand, when power decay is equal to 43, tree depth has reached value 6. From the last mentioned Figure, one can tell, that there is a very small increase in the number of transmitted bits. This is happening, because, while increasing power decay, nodes diverge from root- base station. By this, nodes transmit data of their children, which means more transmitting bits. As power decay increases, path loss increases too. In other words, the probability of losing a packet is bigger than when path loss is smaller. This explains why the mean number of bits does not increase a lot, when tree depth increases.

Concerning the comparison of the methods and their combinations, we can tell, that bloom filter method transmits, again, more bits than the other. Delta and Full Delta methods, but also the combination of the three methods send less data than Full Data and Full Bloom Filter method, which is expected as in previous experiment. The combination of the three methods (which is the black line) is the same as Full Delta method. This happens, because there are many messages lost and nodes do not choose to use Bloom Filter method in first and second phase. Full Bloom and Full Data methods send almost the same mean number of bits.

In order to decide, which method works better, when path loss increases, we must notice, also, Figure 6.6. This Figure represents the percentage error of final data, when signal power decay increases. Percentage error is computer, as in previous experiment, through this Equation 6.1.

From Figure 6.6, we can observe that Full Delta and Delta methods and the combination of all three methods have the highest percentage error as signal power decay increases. The rest of methods have similar percentage error. When $PL(d_0)$ reaches

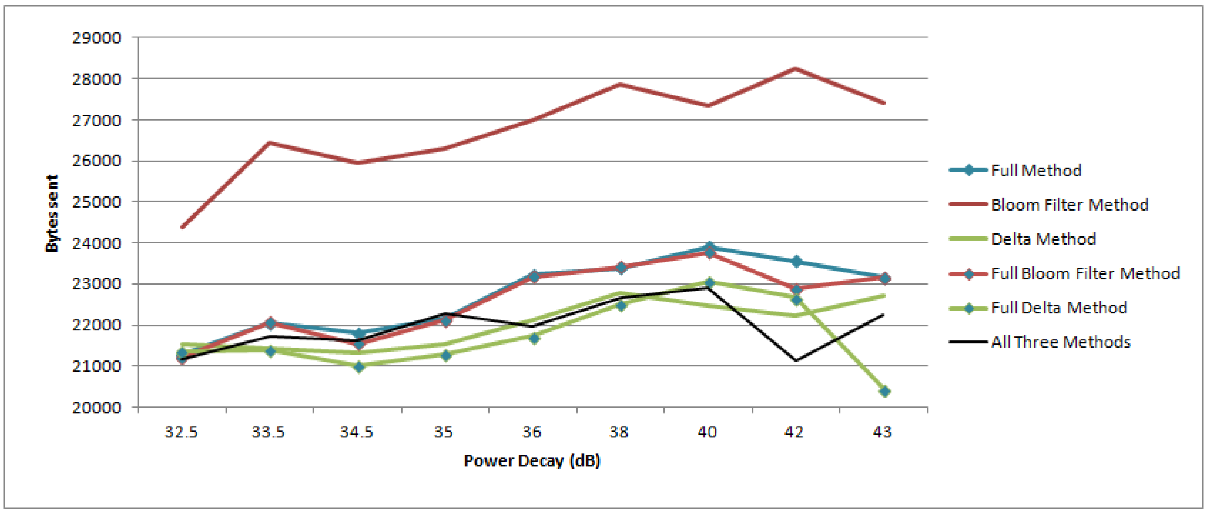


Figure 6.5: Mean number of bits sent when increasing Signal Power Decay

value of 35, tree depth is equal to 2. When power decay is between 35 and 40, the number of nodes, that are in level 3 increases, but the maximum tree depth remains 3. This means that nodes are being shared inside the network and message loss is being reduced. When power decay is equal to 40, 42 and 43, maximum tree depth changes to 4,5 and 6 respectively. This is expected, because at these values path loss is very high and the probability of message loss is bigger than before.

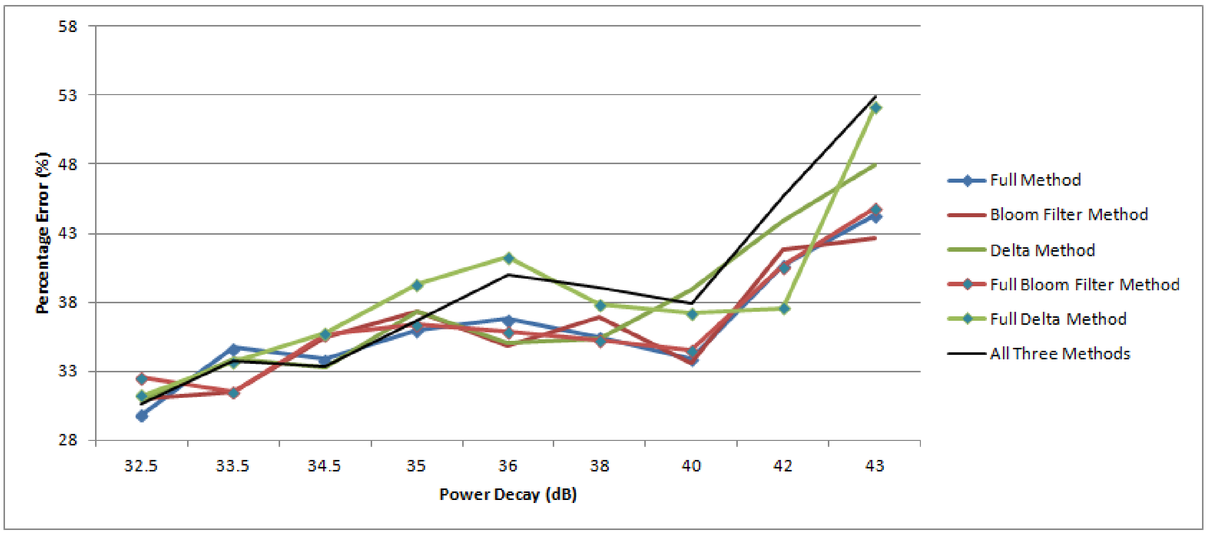


Figure 6.6: Percentage Error of final Data when increasing Signal Power Decay

Chapter 7

Conclusion

7.1 Conclusion

In this thesis, we used the combination of three methods, so as to implement aggregate queries, which are sensitive in duplicate values. Our goal was to combine and compare different methods in order to detect duplicate values and execute aggregate queries. We have proven that, it is not ideal the nodes to send all their data to their parents or children in phases one and two, respectively. We, also, showed that, it is necessary to examine the accuracy of final data, that base station gathers and not to choose a method, which transmits less bits.

Nodes might have obstacles between them and, with great possibility, messages are lost. Consequently, there is a high message loss rate leading messages to get lost and not reach their destination.

When using delta method in this environment, nodes can lead to false conjecture of what exactly sender has transmitted. For example, a node, at epoch 1, sends to its parent, a delta message of ids at the first phase, then its parent creates a list of ids according to the receiving message. At epoch 2, the same node sends a new delta message to its parent, but the message does not reach its destination. The parent of this node now deems that its child has not changed its ids from the previous epoch. Having epoch 3, the node sends another delta message with other plus and minus values. When its parent receives the message, it makes the new list of ids, which is not correct. The same problem can be created, also, at phase two. This is a drawback of delta method and its combination and the impact is also shown especially in experiment two, where pathloss is increasing.

What there is concluded from experiments, is that, it is preferable to use the Full Bloom Filter method instead of the Full Data method. By the Full Bloom Filter method, nodes transmit less bits and the error percentage is lower than the Full Data method. When the path loss increases, these two methods are equally satisfactory, but, by the Full Bloom Filter method nodes send less bits (performance of Full Bloom Filter will be greater in bigger networks with more than 100 nodes), which means, by extension, less computation and less energy consumption. The Delta and the Full Delta methods transmit much less bits than the other methods and can be used, when the accuracy of final data is not very essential.

Moreover, with the option of nodes not sending their current measurements, when being equal to the measurements of the previous epoch, the data payload of network becomes less. This can be interpreted as, that nodes do not transmit data, so energy consumption is becoming less.

We have concluded, that detection of duplicate values is vital in WSNs, when they can influence our final data. It is mandatory to eliminate duplicates for our results in order to be accurate and reliable. All other existing applications do not take into consideration of duplicate values inside a network and assume that they do not exist. Nevertheless, by this algorithm, we are able to detect in which area there is a duplicate value, and isolate it.

7.2 Future Work

The work performed in the context of this thesis implementation can be extended and neutralize message loss rate. This can be succeeded by having acknowledgments for each packet transmitted. In other words, each node, that receives a packet, transmits an acknowledgment to sender to let it know that it received the message. There will be defined a timeout, by the time it expires, sender will re-transmit its packet, if it has not received an acknowledgment for it.

Last but not least, with the ability to transmit measurements to more than one parents, by dividing them into equal pieces according to the number of parents each node has, we can manage to break large values into smaller pieces and limit the number of packets transmitted inside the network. By this, intermediate nodes will have less data to calculate and execute queries. When we have more than one parents and use the Bloom Filter method, nodes can send their bloom filter to all their parents aiming to minimize, again, false results at the two first phases of our algorithm.

References

[1] Stern, M., Buchmann, E., BG'Ahm, K.: Towards efficient processing of general-purpose joins in sensor networks. In Ioannidis, Y.E., Lee, D.L., Ng, R.T., eds.: ICDE, IEEE (2009) 126–137 2, 15, 16, 17, 18

[2] MEMSIC: Iris datasheet Only available online: www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf. 4

[3] Levis, P., Gay, D.: TinyOS Programming. 1st edn. Cambridge University Press, New York, NY, USA (2009) 6

[4] Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tag: a tiny aggregation service for ad-hoc sensor networks. SIGOPS Oper. Syst. Rev. **36**(SI) (December 2002) 131–146 10, 30

[5] Burri, N., Flury, R., Nellen, S., Sigg, B., Sommer, P., Wattenhofer, R.: Yeti: an eclipse plug-in for tinyos 2.1. In Culler, D.E., Liu, J., Welsh, M., eds.: SenSys, ACM (2009) 295–296 13

[6] Considine, J., Li, F., Kollios, G., Byers, J.: Approximate Aggregation Techniques for Sensor Databases. In: International Conference on Data Engineering, ICDE '04, IEEE (2004) 449 18, 19

[7] Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. **31**(2) (September 1985) 182–209 19

[8] Broder, A., Mitzenmacher, M.: Network Applications of Bloom Filters: A Survey. Internet Mathematics **1**(4) (2003) 485–509 20

[9] Almeida, P.S., Baquero, C., Preguiça, N., Hutchison, D.: Scalable bloom filters. Inf. Process. Lett. **101**(6) (March 2007) 255–261 22

[10] Deng, F.: Approximately detecting duplicates for streaming data using stable bloom filters. In: In SIGMOD b••06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM Press (2006) 25–36 22

[11] Maróti, M., Kusy, B., Simon, G., Lédeczi, A.: The flooding time synchronization protocol. In: Proceedings of the 2nd international conference on Embedded networked sensor systems. SenSys '04, New York, NY, USA, ACM (2004) 39–49 38

[12] Sohrabi, K., Manriquez, B., Pottie, G.J.: Near ground wideband channel measurement in 800-1000 MHz. In: Proceedings of the 49th IEEE Vehicular Technology Conference, 1999. Volume 1. (1999) 571–574 66

[13] Zuniga, M.: Building a network topology for tossim Only available online: `http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/usc-topologies.html`. 66